beginner

COLLABORATORS			
	TITLE : beginner		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 5, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

beginner

1.1 beginner.guide

GUIDA ALL'USO DI AMIGA E
****

Questa Guida dá un'introduzione al linguaggio di programmazione Amiga E e anche alcune nozioni sulla programmazione in generale.

TRADOTTO IN ITALIANO DA
AGI

1.2 beginner.guide/Introduzione ad Amiga E

1.1 Introduzione ad Amiga E
****

Per interagire con il nostro Amiga, abbiamo bisogno di usare un linguaggio idoneo per il nostro computer. Fortunatamente, abbiamo un'ampia scelta di tali linguaggi, ognuno dei quali adatto ad una particolare necessitá. Per esempio, il linguaggio BASIC (nelle sue varie versioni) è semplice e facile da imparare, di conseguenza è ideale per i principianti. L'assembly, d'altra parte, richiede sforzi notevoli ed è abbastanza tedioso, ma si possono produrre i programmi più veloci, pertanto è generalmente usato dai programmatori commerciali. Questi linguaggi sono praticamente i due estremi ma di mezzo troviamo linguaggi come il C o il Pascal/Modula-2, che cercano di trovare un buon compromesso tra semplicitá e velocitá.

I programmi E somigliano molto a quelli Pascal o Modula-2, ma il linguaggio E è basato più strettamente sul C. Chiunque abbia famigliaritá con questi linguaggi, imparerá facilmente l'E, basterá studiare quelle che sono le caratteristiche proprie di questo linguaggio e saperle distinguere da quelle che invece sono prese in prestito da altri linguaggi. Questa guida è rivolta ai principianti, potrebbe sembrare banale a dei programmatori esperti, che potrebbero trovare più adeguato il 'E Reference Manual'. (Sebbene alcune delle successive sezioni offrono differenti spiegazioni del Reference Manual che possono tornare utili).

Il capitolo 1 (questo capitolo) prende in esame parte delle basi del linguaggio E e della programmazione in generale. Il capitolo 2 approfondisce meglio gli argomenti, trattando gli argomenti più complessi e quelle che sono le caratteristiche uniche dell'E. Il capitolo 3 tratta alcuni programmi di esempio, un po' più complessi di quelli usati nei capitoli precedenti. Il capitolo 4 contiene le Appendici, dove potremo trovare qualche informazione aggiuntiva.

1.3 beginner.guide/Un semplice programma

1.1.1 Un semplice programma

Se stai ancora leggendo probabilmente sei disperato, vuoi fare qualcosa in E ma non sai come iniziare. Iniziamo con un piccolo esempio. Prima comunque devi sapere come usare un Text-Editor e la Shell/CLI.

1.4 beginner.guide/Programma

1.1.1.1 Programma

Inserire le seguenti linee di codice in un Text-Editor e salvarlo con un nome terminante in .e, (semplice.e), accertandoti di copiare ogni linea accuratamente e di premere il tasto RETURN alla fine di ognuna.

```
PROC main()
  WriteF('Il mio primo programma')
```

```
ENDPROC
```

Non provare a fare niente di diverso al codice, in quanto il Maiuscolo / Minuscolo delle lettere, in ogni parola, ha un significato e i caratteri di punteggiatura sono importanti. Se tu sei un vero principiante potresti avere difficoltá con il carattere '. Su tastiere Europee e USA lo puoi trovare due tasti a destra del tasto L, vicino al tasto ;.

1.5 beginner.guide/Compilazione

1.1.1.2 Compilazione

Una volta che il file è stato salvato (preferibilmente in RAM: finchè si tratta di un piccolo file), si può usare il compilatore E per trasformare il file salvato in un eseguibile. Quello che serve è avere il file ec nel cassetto c: (in questo caso non serve avere l'assegnazione ad Emodules, in quanto non si stá usando nessun modulo). Fatto questo, basta aprire una Shell/CLI, cambiare il path in quello dove il nuovo file è stato salvato e digitare:

```
ec semplice
```

Se tutto è giusto dovrebbero apparire alcune righe di presentazione del compilatore E. Se niente va male, verrá fatto un doppio controllo dei contenuti del file semplice.e .

1.6 beginner.guide/Esecuzione

1.1.1.3 Esecuzione

Adesso puoi eseguire il tuo primo programma, inserendo alla richista CLI:

```
semplice
```

Come ulteriore aiuto, di seguito c'è tutto quello che dovrebbe succedere nella finestra CLI:

```
1.Workbench3.0:> cd ram:
1.Ram Disk:> ec semplice
Amiga E Compiler/Assembler/Linker/PP v3.1a registered (c) 91-95 Wouter
lexical analysing ...
parsing and compiling ...
no errors
1.Ram Disk:> semplice
Il mio primo programma1.Ram Disk:>
```

La tua schermata dovrebbe essere qualcosa di simile. La richiesta CLI si trova nella stessa linea dell'output del programma (l'ultima linea). Questo difetto verrá presto eliminato.

1.7 beginner.guide/Comprensione del programma semplice

```
                1.2 Comprensione del programma semplice
    **********************************
```

Per comprendere il programma di esempio, abbiamo bisogno di capire alcune cose. Avrai notato che tutto quello che fa 'semplice.e' è di stampare un messaggio, e quel messaggio fa parte di una linea che noi abbiamo scritto nel programma. La prima cosa da fare è vedere come cambiare questo messaggio.

1.8 beginner.guide/Modifica del messaggio

```
                1.2.1 Modifica del messaggio
      ======================
```

Editare le stesse linee del file 'semplice' cambiando il messaggio fra ' e poi usare la stessa procedura descritta prima per la compilazione. Se tutto è andato bene dovresti avere, quando esegui il programma, un altro tipo di messaggio. Se qualcosa è andata male raffronta il nuovo file con l'esempio originale e controlla che l'unica differenza sia nel codice scritto fra ' '

1.9 beginner.guide/Breve rassegna

```
1.2.1.1 Breve rassegna
        --------------
```

Noi vedremo in dettaglio tutte le parti importanti del programma, nelle seguenti sezioni, ma prima abbiamo bisogno di un'occhiata di insieme. Qui c'è una breve descrizione di alcuni concetti fondamentali:

 * Procedure: abbiamo definito una procedura chiamata main (principale) e usato la procedura Built-in (interna) WriteF. Una procedura

può essere paraganata ad un piccolo programma con un nome.

* Parametri: Il messaggio fra parentesi dopo WriteF nel programma è il parametro di WriteF. Questo è il dato che la procedura dovrebbe usare.

* Stringhe : Il messaggio contenuto fra i caratteri ' ' . Questa è una stringa.

1.10 beginner.guide/Procedure

1.2.2 Procedure
=========

Come giá detto, una procedura può essere considerata come un piccolo programmino con un nome. In realtá, quando un programma E vá in esecuzione viene eseguita la procedura chiamata main. Quindi se il programma E che viene eseguito non produce risultati bisognerá definire una procedura main. Tutte le altre procedure Built-in o definite dall'utente devono essere chiamate da questa procedura main. Nell'esempio precedente era la procedura WriteF. Per chiarezza: se la procedura Fred chiama la procedura Barney il codice associato a quest'ultima viene eseguito. A sua volta la procedura Barney può chiamare altre procedure, alla fine quando il codice della procedura Barney è stato tutto eseguito l'E passa ad eseguire il prossimo pezzo di codice della procedura Fred. Quando viene raggiunto l'ultimo pezzo di codice della procedura principale (main) l'esecuzione del programma ha termine. Naturalmente tra l'inizio e la fine di una procedura possono succedere delle cose strane allora il programma potrebbe entrare in un loop infinito oppure andare in crash.

1.11 beginner.guide/Definizione di una procedura

1.2.2.1 Definizione di una Procedura (PROC e ENDPROC)

Le procedure si definiscono usando la keyword PROC seguita dal nome della nuova procedura (in minuscolo), una descrizione dei suoi parametri tra parentesi, una serie qualsiasi di linee di codice ed infine la keyword ENDPROC. Osservare l'esempio precedente per identificare le varie parti. Vedi

sez. 1.1.1.1

.

1.12 beginner.guide/Eseguire una procedura

```
                1.2.2.2 Eseguire una procedura
          ----------------------
```

Le Procedure possono essere chiamate (o eseguite) dall'interno di un'altra procedura. Basta dare il nome della procedura seguita da alcuni dati fra parentesi. Vedi

sez. 1.1.1.1

.

1.13 beginner.guide/Un esempio più completo

```
1.2.2.3 Un esempio piú completo
          -----------------------
```

Ecco il programma precedente modificato per definire un'altra procedura:

```
   PROC main()
       WriteF('Il mio primo programma')
       fred()
     ENDPROC

     PROC fred()
       WriteF('...leggermente sviluppato')
     ENDPROC
```

Questo esempio potrebbe sembrare complicato, in realtá è tutto molto semplice: la procedura principale chiama la procedura WriteF (built-in), scrive il messaggio contenuto, quindi chiama la procedura Fred, scrive il messaggio di questa procedura, quindi torna alla procedura originale.

1.14 beginner.guide/Parametri

```
                1.2.3 Parametri
      =========
```

Noi solitamente usiamo le procedure per lavorare con particolari tipi di dati, così nell'esempio precedente la procedura WriteF conteneva fra parentesi, quindi come parametro, il messaggio che volevamo apparrisse a video; tuttavia quando abbiamo chiamato la procedura fred, abbiamo lasciato le parentesi vuote, questo perchè in tal modo, quando andiamo a definire la procedura fred, potremo inserirci qualsiasi altra cosa. In questo modo possiamo avere le cosiddette variabili. Vedi

sez. 1.3

.

1.15 beginner.guide/Stringhe

```
1.2.4 Stringhe
      ========
```

Qualsiasi carattere compreso fra ’ ’ è una stringa sebbene \ e ’ abbiano un significato speciale. Per esempio: un comando di a capo in una stringa si formula usando: "\n". Usiamo sempre il nostro esempio per esemplificare:

```
   PROC main()
       WriteF(’Il mio primo programma\n’)
       fred()
     ENDPROC

     PROC fred()
       WriteF(’...leggermente sviluppato\n’)
     ENDPROC
```

1.16 beginner.guide/Stile, praticitá e leggibilitá

```
1.2.5 Stile, praticitá e leggibilitá
      ============================
```

Il nostro esempio è andato crescendo, abbiamo ora due procedure, main() e fred(), tuttavia lo stesso risultato lo avremmo ottenuto con una sola procedura:

```
    PROC main()
       WriteF(’Il mio primo programma\n’)
       WriteF(’...leggermente sviluppato\n’)
     ENDPROC
```

Abbiamo in pratica sostituito la chiamata alla procedura fred con il contenuto della stessa, questa operazione si chiama inlining. In realtá tutti i programmi possono essere riscritti, ma a volte dividere un programma in varie procedure lo rende piú leggibile. Anche il nome stesso della procedura puó essere autoesplicativo per es.: la procedura fred avrebbe potuto chiamarsi "messaggio" o simile. Un programma ben scritto in questo stile puó essere letto quasi come un linguaggio parlato.

Un’altra ragione per usare le procedure è di poterle usare più volte senza dover riscrivere quella parte di codice ogni volta che ci serve. Immagina di dover usare spesso uno stesso messaggio e anche piuttosto lungo nel tuo programma o lo riscrivi ogni volta che ti serve o lo scrivi una volta in una procedura e chiami quest’ultima quando serve. Usando la procedura hai anche il beneficio di avere solo una copia del messaggio da cambiare se mai ci fosse il bisogno di cambiarlo.

1.17 beginner.guide/Il programma semplice

```
1.2.6 Il programma semplice
      ====================
```

Il programma semplice dovrebbe ora (speriamo) sembrare facile. La procedura BUILT-IN WriteF è la sola che non è stata spiegata. Il linguaggio E ha molte procedure BUILT-IN (interne) e in seguito ne spiegheremo alcune nei dettagli. La prima cosa che dobbiamo fare tuttavia è manipolare dati. Questo è davvero quello che fa in continuazione il computer, esso accetta dati da qualche sorgente (possibilmente l'utente), li manipola in qualche modo (possibilmente memorizzandoli da qualche parte, anche) e invia nuovi dati (normalmente ad uno scermo o ad una stampante). Con il programma di esempio semplice è successo questo, fatta eccezione per le prime due frasi piuttosto banali. Tu hai detto al computer di eseguire il programma compilato (input utente) e i veri dati (il messaggio da stampare) vengono usati dal programma. Questi dati vengono manipolati passando come parametro per WriteF, che fa alcune cose intelligenti come stampare essi sullo schermo. Per riuscire noi a manipolare i nostri dati dobbiamo imparare qualcosa sulle variabili e le espressioni.

1.18 beginner.guide/Variabili ed Espressioni

```
                1.3 Variabili ed Espressioni
    ************************
```

Chiunque abbia fatto un po' di algebra a scuola saprá sicuramente che variabile non è altro che un nome che rappresenta dei dati. In algebra normalmente i dati sono dei numeri, ma in E possono essere qualsiasi cosa (per esempio una stringa). La manipolazione di dati come l'addizione di due numeri è conosciuta come espressione. Il risultato di una espressione può essere usato per costruire espressioni più grandi. Per esempio, 1+2 è un'espressione, così come 6-(1+2). E' molto utile usare variabili al posto dei dati in una espressione, se usi x per rappresentare il numero 1 e y per rappresentare il numero 5, allora l'espressione y-x rappresenta il numero 4. Nelle prossime due sezioni noi vedremo che tipo di variabili puoi definire e i diversi tipi di espressioni che esistono.

```
                 Variabili

                 Espressioni
```

1.19 beginner.guide/Variabili

```
                1.3.1 Variabili
      =========
```

In E le variabili possono contenere tanti differenti tipi di dati chiamati

types. Tuttavia, prima di poter essere usata una variabile deve essere definita cioè dichiarata. Con la dichiarazione di variabile si decide anche se la variabile è disponibile per l'intero programma o soltanto durante il codice di una procedura (ossia se la variabile è global o local). Finalmente i dati conservati in una variabile possono essere cambiati usando gli assignments (assegnazioni). Le seguenti sezioni discutono un po' più in dettaglio questi argomenti.

1.20 beginner.guide/Tipi di variabili

1.3.1.1 Tipi di variabili

In E una variabile è come un magazzino per posizionare dati (e questo magazzino è parte dell'Amiga's RAM). Differenti tipi di dati possono richiedere differenti dimensioni di magazzino. Tuttavia i dati possono essere raggruppati insieme in tipi, e due pezzi di dati dello stesso tipo richiedono la stessa quantitá di immagazzinaggio. Ogni variabile ha un tipo associato e questo impone di usare sempre il magazzino più grande. Più comunemente le variabili in E conservano dati di tipo LONG. Questo tipo contiene gli interi da -2,147,483,648 a 2,147,483,647 tale capacitá di immagazzinaggio normalmente è più che sufficente. Ci sono altri tipi, come INT e LIST e cose più complesse che si possono fare con i tipi, ma per ora quello che sappiamo delle variabili di tipo LONG è abbastanza.

1.21 beginner.guide/Dichiarazione di variabile

1.3.1.2 Dichiarazione di variabile (DEF)

Le variabili devono essere dichiarate prima di essere usate. Esse si dichiarano usando la parola chiave DEF seguita dalla lista dei nomi delle variabili da dichiarare usando la , per separare i nomi. Queste variabili saranno tutte di tipo LONG (in seguito vedremo come dichiarare variabili di altro tipo). Speriamo di chiarire il tutto con alcuni esempi:

```
DEF x

DEF a,b,c
```

La prima linea dichiara la singola variabile x, mentre la seconda dichiara le variabili a,b,c tutte in una volta.

1.22 beginner.guide/Assegnazione

1.3.1.3 Assegnazione

I dati conservati dalle variabili possono essere cambiati e questo si fa normalmente con le assegnazioni. Un'assegnazione è formata dal nome della variabile e un'espressione che evidenzia i nuovi dati da conservare. Il simbolo := separa la variabile dall'espressione. Per esempio il seguente codice conserva il numero due nella variabile x. A sinistra di := c'è il nome della variabile interessata, x in questo caso e a destra l'espressione che evidenzia il nuovo valore (semplicemente il numero due in questo caso).

```
x := 2
```

Il seguente esempio, più complesso, usa il valore conservato nella variabile dalla prima assegnazione come parte dell'espressione per i nuovi dati. Il valore dell'espressione sulla destra di := è il valore conservato nella variabile più uno. Questo valore è il nuovo valore di x, i precedenti dati vengono sostituiti dai nuovi. (Così l'effetto complessivo è che x è incrementato).

```
x := x + 1
```

Questa situazione può essere più chiara nel prossimo esempio che non cambia i dati conservati in x. In realtá questo pezzo di codice è saltanto uno spreco di tempo per la CPU in quanto non si fa altro che riconservare lo stesso valore!

```
x := x
```

1.23 beginner.guide/Variabili globali e locali

1.3.1.4 Variabili globali e locali (parametri per procedure)

Ci sono due modi per dichiarare le variabili: global e local. I dati conservati da variabili globali possono essere letti e cambiati da tutte le procedure, i dati conservati da variabili locali possono essere usati solo dalla procedura in cui sono stati dichiarati. Le variabili globali devono essere dichiarate prima della definizione della prima procedura. Mentre le variabili locali sono dichiarate all'interno della procedura che li deve utilizzare (cioè tra PROC ed ENDPROC). Per esempio il seguente codice dichiara una variabile globale w, e le variabili locali x ed y:

```
DEF w

PROC main()
  DEF x
```

```
  x:=2
  w:=1
  fred()
ENDPROC

PROC fred()
  DEF y
  y:=3
  w:=2
ENDPROC
```

La variabile x è locale alla procedura main, ed y è locale alla procedura fred. Le procedure main e fred possono leggere e modificare il valore della variabile globale w, ma fred non può leggere o modificare il valore di x (almeno fin quando quella variabile è locale a main). Similmente, main non può leggere o modificare y. Le variabili locali di una procedura sono pertanto completamente indipendenti dalle variabili locali di un'altra procedura. Per questa ragione le procedure possono usare variabili locali con lo stesso nome senza possibilitá di confusione da parte del compilatore. Pertanto nel precedente esempio, la variabile locale y in fred avrebbe potuto benissimo chiamarsi x ed il programma avrebbe fatto precisamente la stessa cosa. Es.:

```
PROC main()
  DEF x
  x:=2
  w:=1
  fred()
ENDPROC

PROC fred()
  DEF x
  x:=3
  w:=2
ENDPROC
```

Questo codice funziona perchè l'assegnazione di x in fred fa in modo che la x di fred possa essere usata solo in questa procedura (è locale a fred appunto) e la x di main è locale a main e non può essere usata da fred.

Ma se una variabile locale di una procedura ha lo stesso nome di una variabile globale allora la procedura fará riferimento solo alla variabile locale. Pertanto la variabile globale non può essere usata dalla procedura, questa situazione è chiamata descoping della variabile globale.

I parametri di una procedura sono variabili locali per quella procedura. Noi abbiamo visto come passare dei valori come parametri quando una procedura è chiamata (l'uso di WriteF nell'esempio), ma per ora noi non siamo in grado di definire una procedura che usa parametri. Comunque adesso ne sappiamo un po' di più sulle variabili e possiamo continuare:

```
DEF y

PROC onemore(x)
  y:=x+1
ENDPROC
```

Questo non è un programma completo pertanto non provare a compilarlo. Fondamentalmente noi abbiamo dichiarato una variabile y (che sará di tipo LONG) e una procedura di nome onemore. La procedura è definita con un parametro x, e questo è come se fosse una dichiarazione (locale) di variabile. Quando chiamiamo la procedura onemore dobbiamo fornire un parametro, e questo valore è conservato nella variabile locale x alla prima esecuzione del codice onemore. Il codice conserva il valore di x più uno nella variabile (globale) y. Seguono alcuni esempi di chiamata di onemore:

```
onemore(120)
onemore(52+34)
onemore(y)
```

Una procedura può essere definita in modo che accetti qualsiasi numero di parametri. Nell'esempio seguente la procedura addthem è definita per accettare due parametri, a ed b, pertanto deve essere chiamata con due parametri. Nota che i valori conservati dalle variabili usate come parametri a ed b possono essere modificati all'interno del codice della procedura:

```
DEF y

PROC addthem(a, b)
  a:=a+2
  y:=a*b
ENDPROC
```

Seguono alcuni esempi di chiamata ad addthem:

```
addthem(120,-20)
addthem(52,34)
addthem(y,y)
```

1.24 beginner.guide/Modifica dell'esempio

1.3.1.5 Modifica dell'esempio

Prima di cambiare l'esempio dobbiamo imparare qualcosa su WriteF. Giá sappiamo che i caratteri \n in una stringa hanno il significato di un linefeed (ritorno a capo). Ma ci sono molte altre importanti combinazioni di caratteri in una stringa, alcune sono specifiche di WriteF. Una tale combinazione è \d, che è più facile da descrivere dopo aver visto l'esempio cambiato:

```
PROC main()
  WriteF('Il mio primo programma\n')
  fred()
ENDPROC

PROC fred()
  WriteF('...portato a te dal numero \d\n', 236)
ENDPROC
```

Tu potresti essere in grado di capire cosa avverrá, comunque compilalo lo

stesso e provalo. Se hai fatto tutto bene, dovresti vedere che il secondo messaggio, stampa il numero passato come secondo parametro a WriteF e lo stampa nel posto in cui è stata posizionata la combinazione \d nella stringa. Vediamo l'esempio di output che il codice dovrebbe generare:

```
Il mio primo programma
...portato a te dal numero 236
```

Prova questo cambiamento:

```
PROC main()
  WriteF('Il mio primo programma\n')
  fred()
ENDPROC

PROC fred()
  WriteF('...il numero \d è abbastanza bello\n', 16)
ENDPROC
```

Questo esempio è molto simile al precedente e mostra che \d in effetti marca il posto dove il numero deve essere stampato. L'output generato dovrebbe essere questo:

```
Il mio primo programma
...il numero 16 è abbastanza bello
```

Proveremo ora a stampare due numeri:

```
PROC main()
  WriteF('Il mio primo programma\n')
  fred()
ENDPROC

PROC fred()
  WriteF('...portato a te dai numeri \d e \d\n', 16, 236)
ENDPROC
```

Poichè stiamo stampando due numeri avremo bisogno di due \d e dovremo fornire anche due numeri come parametri nell'ordine in cui noi vogliamo che essi vengano stampati. Il numero 16 perciò sará stampato prima della parola 'e' e prima del numero 236. Ecco l'output:

```
Il mio primo programma
...portato a te dai numeri 16 e 236
```

Adesso possiamo fare un grosso passo in avanti e passare i numeri come parametri alla procedura fred. Guarda semplicemente le differenze fra questo esempio e il precedente:

```
PROC main()
  WriteF('Il mio primo programma\n')
  fred(16, 236)
ENDPROC

PROC fred(a,b)
  WriteF('...portato a te dai numeri \d e \d\n', a,b)
ENDPROC
```

Abbiamo passato le variabili (locali) a e b ad WriteF. Ed è precisamente la stessa cosa di passare direttamente i valori che a ed b conservano (che in pratica è quello che è stato fatto nell'esempio precedente), di conseguenza l'output sará lo stesso. Nella prossima sezione noi manipoleremo le variabili facendo qualche operazione con a ed b, usando WriteF per stampare i risultati.

1.25 beginner.guide/Espressioni

1.3.2 Espressioni
==========

Il linguaggio E include i normali operatori logici e matematici. Questi operatori sono combinati con valori (normalmente in variabili), formano espressioni che producono nuovi valori. Le seguenti sezioni discutono quest'argomento più in dettaglio.

Matematica

Logica e comparazione (TRUE e FALSE - AND e OR)

Precedenze e raggruppamenti

1.26 beginner.guide/Matematica

1.3.2.1 Matematica

Tutti gli operatori matematici standard sono supportati in E. Si possono fare addizioni,sottrazioni,moltiplicazioni e divisioni. Altre funzioni come seno e radici quadrate sono supportate dalle librerie di sistema di amiga, per il momento comunque abbiamo bisogno di sapere qualcosa sulla matematica semplice. Il segno + è usato per l'addizione, - per la sottrazione, * per la moltiplicazione e / per la divisione (fare attenzione a non confondere \ usato nelle stringhe, con / usato per le divisioni). Seguono esempi di espressioni:

```
1+2+3+4
15-5
5*2
330/33
-10+20
3*3+1
```

Ognuna di queste espressioni produce dieci come risultato. L'ultimo esempio è stato scritto molto attentamente per ottenere lo stesso risultato. Vedi

sez. 1.3.2.3

.

Tutte le suddette espressioni usano operatori di interi, pertanto essi manipolano interi, dando degli interi come risultato. Anche i numeri in virgola mobile (Floating-Point) sono supportati in E, ma usarli è abbastanza complicato. Vedi
sez. 2.10
. (I numeri in virgola mobile possono
rappresentare sia frazioni molto piccole e sia interi molto grandi, ma hanno una precisione limitata, cioè, un numero limitato di cifre significanti.

1.27 beginner.guide/Logica e comparazione

1.3.2.2 Logica e comparazione (TRUE e FALSE - AND e OR)

La logica sta molto a cuore al computer. Esso raramente sa cosa fare; infatti fa affidamento su ordini rigorosi e precisi. Pensa alla password di protezione sulla maggior parte dei giochi. il computer deve decidere se tu hai inserito la parola o il numero corretto prima che ti permetta di eseguire il gioco. Quando stai giocando il computer prende decisioni costantemente: La tua arma, quanti alieni rimangono, quante vite, ecc... La logica controlla l'operazione di un programma.

In E, le costanti TRUE e FALSE rappresentano rispettivamente i valori di vero e falso, e gli operatori AND e OR sono gli operatori logici standard. Gli operatori di comparazione sono = (uguale a), > (più grande di), < (minore di), >= (più grande di o uguale a), <= (minore di o uguale a), e <> (non uguale a). Tutte le seguenti espressioni sono vere:

```
TRUE
TRUE AND TRUE
TRUE OR FALSE
1=1
2>1
3<>0
```

Queste sono tutte false:

```
FALSE
TRUE AND FALSE
FALSE OR FALSE
0=2
2<1
(2<1) AND (-1=0)
```

L'ultimo esempio deve usare le parentesi. Vedremo nella prossima sezione perchè (per situazioni di precedenza comunque).

I valori di veritá TRUE e FALSE sono in effetti dei numeri. Così il sistema di logica lavora in E. TRUE corrisponde al numero -1 e FALSE a 0. Gli operatori logici AND e OR attendono tali numeri come loro parametri. In realtá di AND e OR possiamo dire che sono degli operatori intelligenti che

operano sui bit (Bitwise=Bit-intelligenti Vedi
sez. 2.5.4.3
), la maggior
parte delle volte qualsiasi numero diverso da 0 è preso per TRUE. Qualche volta può essere conveniente fare affidamento sulla conoscenza di questa situazione anche se è sempre preferibile (e più leggibile) usare una sintassi più esplicita. Infatti per queste situazioni si possono causare dei sottili problemi come vedremo nella prossima sezione.

1.28 beginner.guide/Precedenze e raggruppamenti

1.3.2.3 Precedenze e raggruppamenti

A scuola la maggior parte di noi ha appreso che le moltiplicazioni hanno la precedenza sulle addizioni in una espressione. In E non è così, non c'è nessun operatore che ha la precedenza su un altro. Questo significa che una espressione come 1+3*3 non da il risultato che un matematico si attenderebbe. In realtá 1+3*3 in E rappresenta il numero 12. Questo perchè l'addizione 1+3 viene eseguita per prima visto che nell'ordine capita prima della moltiplicazione. Infatti se scrivessimo per prima la moltiplicazione verrebbe eseguita questa per prima (e avremmo il risultato che normalmente ci attenderemmo). Perciò 3*3+1 rappresenta il numero 10 sia nella matematica di scuola che in quella di E.

Per superare questa differenza comunque, possiamo usare le parentesi per creare dei gruppi di precedenza nelle espressioni. Infatti se noi avessimo scritto 1+(3*3) il risultato sarebbe stato 10. Ciò perchè noi abbiamo forzato E ad eseguire la moltiplicazione prima di tutto. Sebbene questa procedura per creare le precedenze può sembrare fastidiosa inizialmente, in realtá è molto meglio che imparare molte regole per decidere quale operatore deve essere eseguito per primo (in C per esempio è molto faticoso studiare tali regole e normalmente si finisce per usare le parentesi soltanto per essere sicuri!).

Gli esempi di logica precedenti contenevano l'espressione:

```
(2<1) AND (-1=0)
```

Di questa espressione abbiamo detto che era falsa. Ma se noi non mettevamo le parentesi, E avrebbe visto essa così:

```
((2<1) AND -1) = 0
```

Ora il numero -1 non dovrebbe davvero essere usato per rappresentare un valore di veritá con AND, ma noi sappiamo che TRUE è il numero -1 così E terrá presente questo e il compilatore non dará messaggi di errore. Presto vedremo il reale uso di AND e OR (Vedi
sez. 2.5.4.3
), per ora l'importante
è vedere come E si sarebbe comportato con tale espressione:

1. Due non è minore di uno così 2<1 può essere sostituito da FALSE

```
            (FALSE AND -1) = 0

  2. TRUE è -1 così noi possiamo sostituire -1 con TRUE

            (FALSE AND TRUE) = 0

  3. FALSE and TRUE è FALSE

            (FALSE) = 0

  4. FALSE è davvero il numero 0, così possiamo sostituirlo con 0

            0 = 0

  5. Zero è uguale a zero, così l'espressione è TRUE.

            TRUE
```

Così E ha calcolato vera l'espressione. Ma l'espressione originale (con parentesi) era falsa. Mettere fra parentesi è perciò molto importante! Ed è anche molto facile farlo correttamente.

1.29 beginner.guide/Flusso di controllo del programma

```
                1.4 Flusso di controllo del programma
    **********************************
```

Un programma per computer spesso ha bisogno di eseguire ripetutamente una serie di dichiarazioni o eseguire differenti dichiarazioni conformemente al risultato di qualche decisione. Per esempio un programma che deve stampare tutti i numeri da uno a mille sarebbe molto lungo e tedioso se ogni dichiarazione di stampa dovesse essere scritta individualmente, è molto meglio usare una variabile e ripetutamente stampare il suo valore incrementato di 1. Capita anche che delle cose a volte vanno male e il programma deve decidere se continuare o stampare un messaggio di errore e fermarsi, questa parte di programma è un tipico esempio di blocco condizionale.

```
                Blocco condizionale

                Loops
```

1.30 beginner.guide/Blocco condizionale

```
                1.4.1 Blocco condizionale
      ===================
```

Ci sono due tipi di blocco condizionale: IF e SELECT. Gli esempi per questi blocchi sono frammenti di codice E (cioè non sono dei programmi E completi quindi non compilare).

```
IF x>0
  x:=x+1
  WriteF('Incrementa: x è ora \d\n', x)
ELSEIF x<0
  x:=x-1
  WriteF('Decrementa: x è ora \d\n', x)
ELSE
  WriteF('Zero: x è 0\n')
ENDIF
```

Nel precedente blocco IF, la prima parte controlla se il valore di x è maggiore di 0, se sì, x viene incrementato e il nuovo valore viene stampato (con un messaggio che avvisa di questo incremento). Il programma salterá allora il resto del blocco e eseguirá le dichiarazioni dopo ENDIF. Se x non è più grande di 0, viene controllata la parte ELSEIF, così se x è minore di 0 sará decrementata e stampata e il resto del blocco saltato. Se x non è maggiore di 0 e neanche minore, verranno eseguite le dichiarazioni ELSE, così un messagio dirá che x è zero. La condizione IF è descritta più in dettaglio in seguito.

Blocco IF

Espressione IF

```
      SELECT x
  CASE 0
    WriteF('x è zero\n')
  CASE 10
    WriteF('x è dieci\n')
  CASE -2
    WriteF('x is -2\n')
  DEFAULT
    WriteF('x non è zero, dieci o -2\n')
ENDSELECT
```

Il blocco SELECT è simile al blocco IF, questi si comporta differentemente in base al valore di x. Tuttavia x è controllata solo con valori specifici inseriti nella serie di dichiarazioni CASE. Se queste dichiarazioni non vengono soddisfatte allora la parte dopo DEFAULT viene eseguita.

Esiste una variazione del blocco SELECT (conosciuta come SELECT..OF) che funziona con range di valori ed è abbastanza veloce. I due tipi di blocco SELECT sono descritti più in dettaglio in seguito.

Blocco SELECT

Blocco SELECT..OF

1.31 beginner.guide/Blocco IF

1.4.1.1 Blocco IF

Il blocco IF ha la seguente sintassi (al posto di espressione, si inserisce la descrizione del genere di codice E, permesso in quel punto, quindi espressioneA non fa parte del comando):

```
IF espressioneA
  dichiarazioneA
ELSEIF espressioneB
  dichiarazioneB
ELSE
  dichiarazioneC
ENDIF
```

Questo blocco significa:

* Se espressioneA è vera (cioè rappresenta TRUE o qualsiasi numero diverso da 0) il codice rappresentato dalla dichiarazioneA viene eseguito.

* Se espressioneA è falsa (cioè rappresenta FALSE o zero) e espressioneB è vera la dichiarazioneB viene eseguita.

* Se entrambe le espressioni A e B sono false viene eseguita la dichiarazioneC.

La parte ELSE non è obbligatoria ma se è presente deve essere l'ultima parte (immediatamente prima ENDIF). Possono esserci qualsiasi numero di parti ELSEIF tra le parti IF e ELSE.

Un modo alternativo a questa sintassi verticale (dove ogni parte è su linea separata) è la sintassi orizzontale:

```
IF espressione THEN dichiarazioneA ELSE dichiarazioneB
```

Lo svantaggio di questa sintassi è di non poter avere parti ELSEIF e dover far entrare tutto su una singola linea. Notare la presenza della parola chiave THEN che separa l'espressione dalla dichiarazione. Questa sintassi orizzontale è strettamente attinente alla espressione IF descritta in seguito. Vedi
sez. 1.4.1.2
.

Per aiutare a chiarire le cose seguono dei frammenti di codice E che illustrano l'uso consentito dei blocchi IF:

```
IF x>0 THEN x:=x+1 ELSE x:=0

IF x>0
  x:=x+1
ELSE
  x:=0
ENDIF

IF x=0 THEN WriteF('x è zero\n')
```

```
IF x=0
  WriteF('x è zero\n')
ENDIF

IF x<0
  Write('x negativa\n')
ELSIF x>2000
  Write('x troppo grande\n')
ELSIF (x=2000) OR (x=0)
  Write('Attenzione x\n')
ENDIF

IF x>0
  IF x>2000
    WriteF('Grande x\n')
  ELSE
    WriteF('OK x\n')
  ENDIF
ELSE
  IF x<-800 THEN WriteF('Piccola x\n') ELSE WriteF('Negativa OK x')
ENDIF
```

NOTA: non confondere <- con <=, con x<-800 vogliamo semplicemente dire: x < di -800 (stiamo quindi semplicemente usando un numero negativo)

Nell'ultimo esempio ci sono dei blocchi IF annidati (cioè un blocco IF in un altro blocco IF). Non c'è nessuna ambiguitá per stabilire a che blocco IF le parti ELSE o ELSEIF appartengono, in quanto l'inizio e la fine dei blocchi IF sono chiaramente marcati. Per esempio la prima linea ELSE può solo essere interpretata come fosse parte del blocco IF più interno.

Come se fosse una questione di stile le condizioni delle parti IF e ELSEIF non dovrebbero sovrapporsi (cioè usare condizioni vere ad esempio, per tutte e due). Se tuttavia questo avviene, la prima prenderá la precedenza. Ed è per questo che i seguenti due frammenti di codice E fanno la stessa cosa:

```
IF x>0
  WriteF('x è più grande di zero\n')
ELSEIF x>200
  WriteF('x è più grande di 200\n')
ELSE
  WriteF('x è troppo piccolo\n')
ENDIF

IF x>0
  WriteF('x è più grande di zero\n')
ELSE
  WriteF('x è troppo piccolo\n')
ENDIF
```

La parte ELSEIF del primo frammento controlla se x è più grande di 200. Ma se così è, anche la parte IF sarebbe stata vera (x è certamente più grande di zero, se è più grande di 200), di conseguenza solo il codice della parte IF viene eseguito. L'intero blocco IF funziona come se ELSEIF non ci fosse.

1.32 beginner.guide/Espressione IF

1.4.1.2 Espressione IF

IF è un'istruzione talmente usata che c'è anche una sintassi di IF detta espressione. Il blocco IF è una dichiarazione e quindi controlla quali linee di codice eseguire, mentre l'espressione IF è un'espressione e quindi controlla il proprio valore. Per esempio il blocco IF seguente:

```
IF x>0
  y:=x+1
ELSE
  y:=0
ENDIF
```

può essere scritto in maniera più succinta usando un'espressione IF:

```
y:=(IF x>0 THEN x+1 ELSE 0)
```

Le parentesi sono inutili ma aiutano a rendere più leggibile l'esempio. Fintanto che il blocco IF deve scegliere fra due assegnazioni per y, non è che ci sia molta differenza fra i due codici (sono entrambi assegnazioni), piuttosto è come sono assegnati i valori ad y che cambia. L'espressione IF rende il tutto molto più pulito, essa sceglie il valore da assegnare solo nello stesso modo in cui il blocco IF sceglie l'assegnazione.

Come hai potuto notare, le espressioni IF sono scritte con la sintassi orizzontale del blocco IF. Comunque deve esserci una parte ELSE e nessuna parte ELSEIF. Questo significa che l'espressione avrá sempre un valore e che non sará affollata da parecchi casi.

Non preoccuparti troppo per le espressioni IF, ci saranno utili in pochi casi e possono sempre essere riscritte come blocchi IF in quanto abbiamo detto che sono equivalenti, sono solo più eleganti e leggibili.

1.33 beginner.guide/Blocco SELECT

1.4.1.3 Blocco SELECT

Il blocco SELECT ha la seguente sintassi:

```
SELECT variabile
CASE espressioneA
  dichiarazioneA
CASE espressioneB
  dichiarazioneB
DEFAULT
  dichiarazioneC
ENDSELECT
```

Il valore della variabile di selezione (evidenziata da variabile nella parte SELECT) è comparata a turno con il risultato delle espressioni in ciascuna parte CASE. Se la variabile si accorda con una parte CASE allora verrá eseguita la dichiarazione di tale parte CASE. Possono esserci qualsiasi numero di parti CASE tra le parti SELECT e DEFAULT. Se non ci sono parti CASE che soddisfano la parte SELECT allora saranno eseguite le dichiarazioni della parte DEFAULT. La parte DEFAULT non è obbligatoria ma se c'è deve essere l'utima parte (immediatamente prima ENDSELECT).

Dovrebbe essere chiaro che i blocchi SELECT possono essere riscritti come blocchi IF grazie ai controlli abbastanza simili delle parti IF e ELSEIF. Per esempio, i seguenti frammenti di codice sono equivalenti:

```
SELECT x
CASE 22
  WriteF('x è 22\n')
CASE (y+z)/2
  WriteF('x è (y+x)/2\n')
DEFAULT
  WriteF('x non è niente di significante\n')
ENDSELECT

IF x=22
  WriteF('x è 22\n')
ELSEIF x=(y+z)/2
  WriteF('x è (y+x)/2\n')
ELSE
  WriteF('x non è niente di significante\n')
ENDIF
```

Notare la somiglianza fra le parti IF e ELSEIF e le parti CASE e poi quella fra la parte ELSE e la parte DEFAULT, l'ordine delle parti è uguale. Usare il blocco SELECT è vantaggioso in quanto è molto più facile da leggere, il valore di x viene controllato in continuazione e poi non dobbiamo scrivere x= nei controlli.

1.34 beginner.guide/Blocco SELECT..OF

1.4.1.4 Blocco SELECT..OF

Il blocco SELECT..OF è un po' più complicato del normale blocco SELECT, ma può essere molto utile. Esso ha la seguente sintassi:

```
SELECT maxrange OF espressione
CASE constA
  dichiarazioniA
CASE constB1 TO constB2
  dichiarazioniB
CASE range1, range2
  dichiarazioniC
DEFAULT
  dichiarazioniD
ENDSELECT
```

Il valore della variabile di selezione è dato da espressione, che può essere una qualsiasi espressione, non soltanto una semplice variabile come nel normale blocco SELECT. Comunque, maxrange, constA, constB1 e constB2 devono tutti essere numeri espliciti ossia delle costanti. Vedi
sez. 2.3
.
Il parametro maxrange deve essere una costante positiva, mentre le altre costanti deveno essere comprese tra zero e maxrange (includendo zero, ma escludendo maxrange. Ossia, se maxrange è 20, gli altri valori possono andare da 0 a 19).

I valori CASE da accordare con espressione sono specificati usando dei ranges di valori. Un semplice range è una costante singola (il primo CASE, nella sintassi qui sopra). Un range più proprio lo possiamo vedere nel secondo CASE con l'uso della keyword TO (constB2 deve essere più grande di constB1). Con un normale CASE nel blocco SELECT..OF si possono specificare tanti valori e ranges da accordare con espressione, purchè separati da una virgola come nel terzo CASE nella sintassi qui sopra. Per esempio le seguenti linee CASE sono equivalenti e possono essere usate per accordarsi con qualsiasi numero in range da uno a cinque (compreso):

```
CASE 1 TO 5

CASE 1, 2, 3, 4, 5

CASE 1 TO 3, 3 TO 5

CASE 1, 2 TO 3, 4, 5

CASE 1, 5, 2, 4, 3

CASE 2 TO 3, 5, 1, 4
```

Se il valore di espressione è minore di zero, più grande di o uguale a maxrange, oppure non si accorda con nessun valore dei ranges di CASE, allora verranno eseguite le dichiarazioni nella parte DEFAULT. Se così non è, verranno eseguite le dichiarazioni della prima parte CASE che si accorda con l'espressione. Come nel normale blocco SELECT, la parte DEFAULT non è obbligatoria.

Il seguente blocco SELECT..OF stampa il giorno (numerico) del mese, elegantemente:

```
SELECT 32 OF day
CASE 1, 21, 31
  WriteF('The \dst day of the month\n', day)
CASE 2, 22
  WriteF('The \dnd day of the month\n', day)
CASE 3, 23
  WriteF('The \drd day of the month\n', day)
CASE 4 TO 20, 24 TO 30
  WriteF('The \dth day of the month\n', day)
DEFAULT
  WriteF('Error: invalid day=\d\n', day)
ENDSELECT
```

(Tradurre in italiano questo esempio non avrebbe senso, in quanto per espressione = 1, la frase stampata sarebbe: Il 1mo giorno del mese. Non abbiamo tali forme come in inglese per 1st (primo), 2nd, 3rd, ecc. - nota del traduttore).

Il maxrange in questo blocco è 32, quindi 31 è il massimo dei valori usati nelle parti CASE. Se, per esempio, il valore di day fosse 100, verrebbero eseguite le dichiarazioni della parte DEFAULT, che segnalano un giorno non valido.

Questo esempio può essere riscritto come blocco IF:

```
IF (day=1) OR (day=21) OR (day=31)
  WriteF('The \dst day of the month\n', day)
ELSEIF (day=2) OR (day=22)
  WriteF('The \dnd day of the month\n', day)
ELSEIF (day=3) OR (day=23)
  WriteF('The \drd day of the month\n', day)
ELSEIF ((4<=day) AND (day<=20))  OR  ((24<=day) AND (day<=30))
  WriteF('The \dth day of the month\n', day)
ELSE
  WriteF('Error: invalid day=\d\n', day)
ENDIF
```

La virgola che separava i ranges delle parti CASE è stata sostituita da un OR e il TO è stato sostituito da un AND (bisogna notare il corretto uso delle parentesi nelle espressioni risultanti.)

Chiaramente, il blocco SELEC..OF è molto più leggibile dell'equivalente blocco IF. Esso è anche molto più veloce, principalmente perchè nessuno dei paragoni presenti nel blocco IF deve essere usato nella versione del blocco SELECT..OF. Al contrario il valore di espressione viene usato immediatamente per localizzare la corretta parte CASE. Comunque qualche lato negativo c'è: il valore maxrange influisce direttamente sulla dimensione del compilato eseguibile, pertanto si consiglia di usare i blocchi SELECT..OF solo con valori piccoli per maxrange. Vedi il 'Reference Manual' per maggiori dettagli.

1.35 beginner.guide/Loops

1.4.2 Loops

=====

I loops in un programma sono tutto ciò che fa eseguire una serie di dichiarazioni ripetutamente. Probabilmente il loop più facile da capire è il loop FOR. Ci sono altri tipi di loops, ma sicuramente saranno di più facile comprensione dopo aver capito il loop FOR.

Loop FOR

Loop WHILE

Loop REPEAT..UNTIL

1.36 beginner.guide/Loop FOR

1.4.2.1 Loop FOR

Se vuoi scrivere un programma che stampa i numeri da 1 a 100 puoi scrivere un'istruzione idonea per ogni numero oppure puoi usare una singola variabile e un piccolo loop FOR. Prova a compilare questo programma E (lo spazio dopo \d nella stringa è necessario per separare i numeri stampati):

```
PROC main()
  DEF x
  FOR x:=1 TO 100
    WriteF('\d ', x)
  ENDFOR
  WriteF('\n')
ENDPROC
```

Quando esegui questo programma vedrai stampati tutti i numeri da 1 a 100 proprio come noi volevamo. Funziona usando la variabile (locale) x che contiene il numero da stampare. Il loop FOR inizia settando il valore di x a 1 (la parte che assomiglia ad una assegnazione). Poi vengono eseguite le dichiarazioni fra le linee FOR e ENDFOR (così x prende il valore da stampare). Quando il programma raggiunge la linea ENDFOR incrementa x e controlla se questa è più grande di 100 (il limite che assegnamo con la parte TO). Se il programma verifica questa situazione il loop è terminato e prosegue con le dichiarazioni successive a ENDFOR altrimenti le dichiarazioni tra il FOR e ENDFOR vengono eseguite tutte nuovamente e questo ovviamente fino a quando x non supera 100 (x è incrementato di 1 per ogni ciclo in questo caso). In realtá questo programma fa la stessa cosa del seguente (i ... non sono codice E rapresentano le altre 97 dichiarazioni WriteF mancanti per completare la stampa, rigo per rigo, dei numeri da 1 a 100):

```
PROC main()
  WriteF('\d ', 1)
  WriteF('\d ', 2)
  ...
  WriteF('\d ', 100)
  WriteF('\n')
ENDPROC
```

La sintassi generale del loop FOR è la seguente:

```
  FOR var := espressioneA TO espressioneB STEP numero
    dichiarazioni
  ENDFOR
```

La parte var rappresenta la variabile di loop (nell'esempio precedente abbiamo usato la x). La parte espressioneA dá il valore di partenza per la variabile di loop e la parte espressioneB setta l'ultimo valore consentito. La parte STEP permette di specificare il valore (dato da numero) che deve

essere aggiunto alla variabile di loop per ogni ciclo, se non specificato di default viene aggiunto 1, come nell'esempio. Diversamente dai valori che si possono dare come partenza e fine ciclo (che possono essere anche delle espressioni proprie) il valore STEP deve essere sempre un numero esplicito, cioè una costante. Vedi
sez. 2.3
. Valori STEP negativi sono permessi, ma in questo caso il controllo usato alla fine di ogni loop è se la variabile di loop è minore del valore della parte TO. Lo zero non è permesso come valore di STEP.

Come per il blocco IF anche per il loop FOR c'è una sintassi orizzontale:

```
FOR var := espA TO espB STEP espC DO dichiarazione
```

1.37 beginner.guide/Loop WHILE

1.4.2.2 Loop WHILE

Il loop FOR usa una variabile di loop e controlla se questa oltrepassa il suo limite. Il loop WHILE invece ti permette di specificare il controllo del loop. Per esempio il seguente programma fa lo stesso lavoro del precedente:

```
PROC main()
  DEF x
  x:=1
  WHILE x<=100
    WriteF('\d ', x)
    x:=x+1
  ENDWHILE
  WriteF('\n')
ENDPROC
```

Come si può vedere abbiamo sostituito al loop FOR sia una inizializzazione di x (x:=1) e sia un loop WHILE con una dichiarazione di controllo del loop (x<=100) e una dichiarazione extra di incremento di x (x:=x+1). In questo modo possiamo vedere il lavoro nascosto del loop FOR che in effetti lavora proprio in questo modo.

E' importante sapere che il nostro controllo (x<=100) deve essere fatto prima che le dichiarazioni di loop vengano eseguite. Questo significa che le dichiarazioni di loop potrebbero non essere eseguite nemmeno una volta. Infatti se noi ponessimo come controllo x>=100 al posto di x<=100 avremmo una dichiarazione che risulterebbe falsa all'inizio del loop, almeno fintanto che l'assegnazione di x prima del loop (x:=1) rimane inferiore a 100. Pertanto il loop avrebbe terminato immediatamente la sua esecuzione passando il controllo alle dichiarazioni dopo ENDWHILE. E' facile quindi capire che con il loop WHILE possiamo fare in modo che un ciclo ripetitivo venga eseguito solo se determinate variabili hanno o meno un certo valore.

Vediamo ora un esempio più complicato:

```
PROC main()
  DEF x,y
  x:=1
  y:=2
  WHILE (x<10) AND (y<10)
    WriteF('x è \d e y è \d\n', x, y)
    x:=x+2
    y:=y+2
  ENDWHILE
ENDPROC
```

Questa volta abbiamo usato due variabili (locali). Non appena una di esse è uguale o maggiore di 10 si esce dal loop. Se leggiamo un po' il codice notiamo che x è inizializzata ad 1 e viene incrementata di due, di conseguenza conterrá sempre un numero dispari. Similmente la y invece conterrá un numero pari. Il controllo di WHILE mostra che non stamperá dei numeri più grandi o uguali a 10. Dal fatto che x inizia con 1 e y con 2, possiamo decidere che l'ultimo paio di numeri stampati saranno 7 e 8. L'esecuzione del programma conferma questa situazione. Infatti dovrebbe produrre il seguente output:

```
x è 1 e y è 2
x è 3 e y è 4
x è 5 e y è 6
x è 7 e y è 8
```

Questo perchè anche se la x è 9 e quindi un valore consentito dal loop la y però in quel momento sará 10 e in questo caso come abbiamo detto in precedenza il loop sará immediatamente interrotto, inpedendo così la stampa di x è 9.

Come per il loop FOR, c'è una sintassi orizzontale anche per il loop WHILE:

```
  WHILE espressione DO dichiarazione
```

la conclusione di un loop è sempre un grosso problema. I loop FOR garantiscono la loro fine quando raggiungono il loro limite (se non si commettono errori con il valore della variabile di loop). Mentre il loop WHILE (e tutti gli altri loop) possono continuare all'infinito. Per esempio se noi come controllo usiamo 1<2, è ovvio capire che tale controllo risulterebbe sempre vero, il loop non può farci niente per impedire questo! Bisogna perciò accertarsi che i loop terminino in qualche modo se si vuol raggiungere la fine del programma. Esiste un modo particolare di terminare i loops usando la dichiarazione JUMP, ma per il momento la ignoriamo.

1.38 beginner.guide/Loop REPEAT..UNTIL

```
1.4.2.3 Loop REPEAT..UNTIL
        ------------------
```

Un loop REPEAT..UNTIL è molto simile al loop WHILE. La sola differenza è dove si specifica il controllo di loop e come e quando esso viene eseguito. Per spiegare questo, c'è un programma delle precedenti due sezioni riscritto usando il loop REPEAT..UNTIL (prova a localizzare le sottili differenze):

```
PROC main()
  DEF x
  x:=1
  REPEAT
    WriteF('\d ', x)
    x:=x+1
  UNTIL x>100
  WriteF('\n')
ENDPROC
```

Proprio come nel loop WHILE noi abbiamo una inizializzazione di x e una dichiarazione extra nel loop per incrementare x. Però il controllo del loop è specificato alla fine del loop stesso (nella parte UNTIL), il controllo quindi viene eseguito alla fine di ogni loop. Grazie a questa differenza avremo che il codice in un loop REPEAT..UNTIL sará eseguito almeno una volta, al contrario del codice di un loop WHILE che come abbiamo visto può anche non essere eseguito mai se la variabile di controllo non viene soddisfatta.

1.39 beginner.guide/Sommario

```
1.5 Sommario
    ********
```

Siamo arrivati alla fine del Capitolo 1, che speriamo sia stata abbastanza chiara per farti appassionare al linguagggio E. Se hai afferrato i concetti principali puoi passare alla parte seconda che spiega il linguaggio E più in dettaglio.

Segue un piccolo test che riassume alcune caratterestiche del linguaggio E spiegate fin ora. Il seguente test usa un esempio di loop WHILE. Per semplificare il riferimento alle linee del test, ugnuna di queste è stata numerata (non provare a compilare con i numeri di linea).

```
 1.  PROC main()
 2.    DEF x,y
 3.    x:=1
 4.    y:=2
 5.    WHILE (x<10) AND (y<10)
 6.      WriteF('x è \d e y è \d\n', x, y)
 7.      x:=x+2
 8.      y:=y+2
 9.    ENDWHILE
10.  ENDPROC
```

Si spera che tu sia in grado di riconoscere tutte le caratteristiche elencate nella seguente tavola. Se non ci riesci allora potresti aver bisogno di rivedere le precedenti sezioni oppure di trovare una guida alla programmazione E migliore di questa!

```
    Osservazione delle linee del test
    ---------------------------------
      1-10    La definizione di procedura.
```

1 La dichiarazione della procedura main, senza parametri.

2 La dichiarazione di variabili locali x ed y.

3, 4 Inizializzazione di x ed y usando le dichiarazioni di assegnazione.

5-9 Il loop WHILE.

5 Il controllo del loop WHILE usando l'operatore logico AND, l'operatore di paragone < e le parentesi per raggruppare l'espressione.

6 La chiamata alla procedura (BUILT-IN) WriteF usando i parametri. Notare nella stringa, il posto esatto per i numeri con \d, e il linefeed con \n.

7, 8 Assegnazioni di x ed y, aggiungendo due ai loro valori.

9 Il marcatore per la fine del loop WHILE.

10 Il marcatore per la fine della procedura.

1.40 beginner.guide/Sintassi e schema

2.1 Sintassi e schema
****

In questa sezione prenderemo in esame le regole che governano la sintassi e lo schema del codice E. Nel capitolo precedente abbiamo visto esempi di codice E abbastanza ben identati in modo da rendere facilmente visibile la struttura del programma. Comunque abbiamo usato solo delle convenzioni, il linguaggio E non ci limita a scrivere il codice solo in quel modo, anche se ci sono delle regole che devono essere seguite. (Questa sezione fa riferimento ad alcuni concetti e parti del linguaggio E non esaminati precedentemente. Sará una buona idea rileggersi questa sezione quando quelle parti che tratteremo adesso saranno trattate anche nelle successive sezioni).

Identificatori

Dichiarazioni

Spazi e separatori

Commenti

1.41 beginner.guide/Identificatori

2.1.1 Identificatori
==========

Un identificatore è una parola che il computer deve interpretare piuttosto che trattare letteralmente. Per esempio, una variabile è un identificatore così come lo è una keyword (ad esempio IF), ma niente in una stringa lo è (per esempio, fred nella stringa 'fred e wilma' non è un identificatore). Gli identificatori possono essere composti da lettere maiuscole o minuscole, da numeri e dal carattere sottolinea (_). Ci sono solo due limitazioni:

1. Il primo carattere non può essere un numero (ci sarebbe confusione con le costanti numeriche altrimenti).

2. Bisogna prestare attenzione al maiuscolo e minuscolo dei primi caratteri degli identificatori.

Per keyword (es.: ENDPROC), costanti (es.: TRUE) e assembly mnemonics (es.: MOVE), almeno i primi due caratteri devono essere maiuscoli. Per le procedure proprie (BUILT-IN) dell'E o per le procedure di sistema Amiga, il primo carattere deve essere maiuscolo e il secondo minuscolo. Per tutti gli altri identificatori (cioè variabili locali, globali, e di parametri per le procedure, nomi di oggetti (object) e di elementi, nomi di procedure e labels) almeno il primo carattere deve essere minuscolo.

Queste sono le uniche limitazioni, per il resto possiamo scrivere gli identificatori come vogliamo anche se è prassi scrivere le varibili con caratteri tutti minuscoli, le keyword e le costanti tutti maiuscoli e i nomi di procedure con il primo maiuscolo e i seguenti minuscoli.

1.42 beginner.guide/Dichiarazioni

2.1.2 Dichiarazioni
==========

Una dichiarazione normalmente è una singola linea di ordini. Ogni dichiarazione normalmente occupa una singola linea. Se una procedura è pensata come fosse un paragrafo allora quella dichiarazione diventa un elenco, e variabili , espressioni e keyword sono le parole che lo compongono.

Finora nei nostri esempi abbiamo incontrato solo due tipi di dichiarazione: la dichiarazione fatta su una singola linea e quella multilinea. Delle assegnazioni abbiamo visto quelle su singola linea. La sintassi verticale del blocco IF è una dichiarazione multilinea. La sintassi orizzontale del blocco IF è una dichiarazione a linea singola. Nota che le dichiarazioni possono essere formate da un insieme di dichiarazioni, come nel caso dei blocchi IF. Le parti di codice tra le linee IF, ELSEIF, ELSE, e ENDIF sono sequenze di dichiarazioni.

Le dichiarazioni singole di linea possono essere a volte molto brevi e

possiamo quindi scriverle su una sola linea senza che essa sia molto lunga. Basta usare un ; come separatore per ogni singola dichiarazione della linea. Per esempio i seguenti frammenti di codice sono equivalenti:

```
fred(y,z)
y:=x
x:=z+1

fred(y,z); y:=x; x:=z+1
```

Ma potremmo anche volere il contrario e cioè dividere una lunga dichiarazione in più linee. Il compilatore ovviamente ha bisogno di capire che una determinata dichiarazione non è finita quando arriva al termine di una linea, ed è per questo che possiamo interrompere una linea solo in determinati punti. Il punto più comune è quello dopo una virgola che fa parte della dichiarazione (immaginiamo la chiamata ad una procedura con molti parametri), ma possiamo dividere la linea anche dopo gli operatori binari e ovunque fra una apertura e una chiusura di parentesi. I seguenti esempi sono un po' sciocchi ma mostrano qualche divisione di linea consentita:

```
fred(a, b, c,
     d, e, f)   /* Dopo una virgola */

x:=x+
   y+
   z            /* Dopo un operatore binario */

x:=(1+2
    +3)         /* Fra un apertura... e chiusura di parentesi */

list:= [ 1,2,
         [3,4],
        ]       /* Fra un apertura... e chiusura di parentesi */
```

La regola è semplice ed è questa: se una linea completa può essere interpretata come una dichiarazione, allora lo sará, altrimenti sará interpretata come parte di una dichiarazione che continua nelle linee seguenti.

Anche una stringa può essere lunga e quindi avere la necessitá di dividerla in più linee, in questo caso dobbiamo usare il carattere + al termine di ogni linea. Infatti se una linea finisce con + e subito prima di questo carattere c'è una stringa allora il compilatore E considera la stringa seguente come una continuazione della precedente. Le seguenti chiamate a WriteF stampano lo stesso messaggio:

```
WriteF('Questa lunga stringa può essere divisa in molte linee.\n')

WriteF('Questa lunga stringa ' +
       'può essere divisa in molte linee.\n')

WriteF('Questa lunga' +
       ' stringa può essere ' +
       'divisa in molte ' +
       'linee.\n')
```

1.43 beginner.guide/Spazi e separatori

```
2.1.3 Spazi e separatori
      ==================
```

Gli esempi che abbiamo visto finora hanno usato una rigida convenzione di identazione concepita per rendere più leggibile la struttura del programma. Ma lo abbiamo fatto solo per convenzione, il linguaggio E in effetti non da nessuna limitazione sulla quantitá di spazi bianchi (spazi, tabs e linefeeds) che usiamo fra le dichiarazioni. Tuttavia nella dichiarazione bisogna usare un certo numero di spazi per renderla più leggebile. Questo significa che dobbiamo mettere uno spazio bianco fra identificatori adiacenti che iniziano o finiscono con una lettera, numero o il carattere sottolinea (in questo modo il compilatore non lo considera come un lungo identificatore!). In pratica dobbiamo usare uno spazio dopo una keyword in modo che questa non venga confusa con una variabile o nome di procedura. Altre volte (come nelle espressioni) gli identificatori sono separati da caratteri che non sono identificatori (una virgola, una parentesi o altri simboli).

1.44 beginner.guide/Commenti

```
2.1.4 Commenti
      ========
```

Un commento è qualcosa che il compilatore E ignora e serve solo per aiutare chi legge il programma. Dopo un po' di tempo dalla realizzazione del programma può essere difficile decifrarlo senza dei buoni commenti ed è per questo che essi in un programma diventano abbastanza importanti.

Possiamo scrivere i commenti ovunque ci siano spazi bianchi nel programma che non facciano parte di una stringa. Ci sono due tipi di commento: un tipo usa /* per segnalarne l'inizio e */ per segnalarne la fine, l'altro tipo usa -> per segnalarne l'inizio e poi continuando con il testo del commento sino a fine linea (senza una chiusura). Bisogna stare attenti a non usare */ o -> come parte del testo di commento, a meno che non vengano usati per segnalare un commento annidato. In pratica è meglio inserire un commento o da solo su una linea o dopo la fine del codice della linea.

```
    /* Questa linea è un commento */
    x:=1  /* Questa linea contiene un'assegnazione e un commento */
 /* y:=2  /* Questa intera linea è un commento annidato */*/

    x:=1  -> Questa linea contiene un'assegnazione e un commento
 -> y:=2  /* Questa intera linea è un commento annidato */
```

1.45 beginner.guide/Procedure e Funzioni

```
                2.2 Procedure e Funzioni
    ********************
```

Una funzione è una procedura che ritorna un valore. Questo valore può essere una qualsiasi espressione, pertanto il valore può dipendere dai parametri con cui la funzione viene chiamata. Per esempio, l'operatore di addizione + può essere parte di una funzione che ritorna la somma dei suoi due parametri.

1.46 beginner.guide/Funzioni

```
                2.2.1 Funzioni (RETURN)
      =================
```

Possiamo definire la nostra funzione di addizione, add, in un modo molto simile alla definizione di una procedura. (La sola differenza è che una funzione ritora un valore.)

```
PROC main()
  DEF sum
  sum:=12+79
  WriteF('Usando +, la somma è \d\n', sum)
  sum:=add(12,79)
  WriteF('Usando add, la somma è \d\n', sum)
ENDPROC

PROC add(x, y)
  DEF s
  s:=x+y
ENDPROC s
```

Tale codice dovrebbe generare il seguente output:

```
Usando +, la somma è 91
Usando add, la somma è 91
```

La procedura add ritorna il valore s qualificando con s la keyword ENDPROC. Il valore ritornato da add può essere usato nelle espressioni proprio come un qualsiasi altro valore. Possiamo usare tale valore scrivendo la chiamata alla procedura esattamente dove tale valore ci serve. Nel precedente esempio volevamo che il valore ritornato fosse assegnato a sum, così

abbiamo scritto la chiamata ad add sulla destra dell'assegnazione. Notare le somiglianze tra l'uso di + e quello di add. In generale add(a,b) può essere usato negli stessi posti dove ci servirebbe usare a+b (più precisamente possiamo usarlo ovunque un a+b può essere usato).

La keyword RETURN può anche essere usata per qualificare i valori di ritorno di una procedura. Se usiamo il metodo ENDPROC allora il valore è ritornato quando la procedura raggiunge la fine del suo codice. Se invece usiamo il metodo RETURN allora il valore è ritornato immediatamente in quel momento e non dopo che tutto il codice della procedura è stato eseguito. Rivediamo l'esempio precedente usando RETURN:

```
PROC add(x, y)
  DEF s
  s:=x+y
  RETURN s
ENDPROC
```

La sola differenza è che possiamo scrivere RETURN ovunque nella parte di codice di una procedura e questa termina la sua escuzione in quel momento (piuttosto che arrivare a terminarla con ENDPROC). Infatti possiamo anche usare RETURN nella procedura main (quella principale) per far finire subito l'esecuzione di un programma se occorre.

Vediamo un uso leggermente più complicato di RETURN:

```
PROC limitiamoadd(x,y)
  IF x>10000
    RETURN 10000
  ELSEIF x<-10000
    RETURN -10000
  ELSE
    RETURN x+y
  ENDIF
  /* Le seguenti linee sono inutili */
  x:=1
  IF x=1 THEN RETURN 9999 ELSE RETURN -9999
ENDPROC
```

Questa funzione controlla se x è più grande di 10000 o minore di -10000, se così è, viene ritornato un valore da noi limitato (e quindi, generalmente, non è la somma corretta!). Se x è compresa fra -10000 e 10000 viene ritornata una risposta corretta. Le linee dopo il primo blocco IF non verranno mai eseguite in quanto l'esecuzione sará terminata a una delle linee RETURN. Quelle linee pertanto sono solo uno spreco di tempo per il compilatore e possiamo ometterle con assoluta sicurezza (come il commento suggerisce).

Se non qualifichiamo nessun valore con le keyword ENDPROC o RETURN il valore di ritorno sará zero. Perciò, tutte le procedure sono in effetti delle funzioni (e i termini procedura e funzione tenderanno a essere usati così come capita dato che parliamo della stessa cosa). Ci potremmo chiedere allora cosa ne succede del valore di una procedura quando scriviamo la sola chiamata ad essa in una linea e non in una espressione. Bene, come noi vedremo, il valore è semplicemente scartato. Questo è ciò che è accaduto nei precedenti esempi quando abbiamo chiamato le procedure fred e WriteF. Vedi

```
sez. 2.5.1
.
```

1.47 beginner.guide/One-Line Functions

```
2.2.2 Funzioni su una linea (one-line)
      ================================
```

Così come il blocco IF e il loop FOR hanno la sintassi orizzontale (cioè su una sola linea), anche la definizione di una procedura ha questa sintassi. La sintassi generale è:

```
PROC nome (arg1, arg2, ...) IS espressione
```

In alternativa, possiamo usare la keyword RETURN:

```
PROC name (arg1, arg2, ...) RETURN espressione
```

A prima vista si potrebbe pensare che una tale sintassi non è utile, invece per funzioni molto semplici trova la sua utilitá e la funzione add che abbiamo costruito nella precedente sezione ne è un buon esempio. Infatti se studiamo bene quell'esempio ci accorgiamo che la variabile locale s, della funzione add, non è strettamente necessaria, quindi possiamo scrivere la funzione add anche con la sintassi di singola linea, così:

```
PROC add(x,y) IS x+y
```

1.48 beginner.guide/Argomenti di Default

```
2.2.3 Argomenti di Default
      ====================
```

Potrebbe capitare, qualche volta, di dover chiamare una procedura (o funzione) ripetutamente con un particolare valore (costante), per uno dei suoi parametri, e questo potrebbe essere normale se non dovessimo riempire questo valore in continuazione. Fortunatamente l'E ci permette di definire degli argomenti di default per i parametri di procedura, quando definiamo la stessa. Di conseguenza, possiamo semplicemente omettere quel parametro, quando chiamiamo la procedura, in quanto tale parametro sará passato di default con il valore da noi definito. Segue un semplice esempio:

```
PROC play(track=1)
  WriteF('Inizia a suonare dalla traccia \d\n', track)
  /* Resto del codice... */
ENDPROC

PROC main()
  play(1)  -> Inizia a suonare dalla traccia 1
```

```
  play(6)  -> Inizia a suonare dalla traccia 6
  play()   -> Inizia a suonare dalla traccia 1
ENDPROC
```

Questo è un frammento di programma per il controllo di qualcosa come un CD player. La procedura play ha un parametro, track, che rappresenta la prima traccia che dovrebbe essere eseguita. Spesso, tuttavia, possiamo usare semplicemente play del CD player senza specificare una particolare traccia. Questo è esattamente quello che succede nell'esempio qui sopra: il parametro track ha il valore 1 definitogli per default (=1 nella definizione della procedura play), e la terza chiamata a play, in main, non specifica un valore per track, quindi viene usato il valore di default.

Esistono due limitazioni sull'uso degli argomenti di default:

1. Qualsiasi numero di parametri di procedura possono essere definiti con valori di default, ma questi deveno trovarsi tutti a destra. Questo significa che una procedura con tre parametri, può avere il secondo parametro con la definizione di default, se anche il terzo la ha, pertanto il primo parametro la potrá avere, se anche gli altri due la hanno. Questo non dovrebbe essere un grosso problema, perchè possiamo sempre riordinare i parametri nella definizione di procedura.

   I seguenti esempi mostrano le definizioni ammesse di procedura con argomenti di default:

   ```
   PROC fred(x, y, z) IS x+y+z          -> Nessun defaults

   PROC fred(x, y, z=1) IS x+y+z        -> z di defaults = 1

   PROC fred(x, y=23, z=1) IS x+y+z     -> y e z sono di defaults

   PROC fred(x=9, y=23, z=1) IS x+y+z   -> Sono tutte di defaults
   ```

   Mentre queste definizioni sono tutte non ammesse:

   ```
   PROC fred(x, y=23, z) IS x+y+z   -> Illegale: z non è di default

   PROC fred(x=9, y, z=1) IS x+y+z  -> Illegale: y non è di default
   ```

2. Quando chiamiamo una procedura che ha degli argomenti di default, possiamo omettere solo i parametri dell'estrema destra. Questo significa che una procedura con tre parametri e tutti con valori di default, possiamo chiamarla omettendo il secondo parametro solo omettendo il terzo parametro. Il primo parametro può non essere usato solo se anche gli altri due non vengono usati.

   Il seguente esempio mostra quali parametri sono considerati di default:

   ```
   PROC fred(x, y=23, z=1)
     WriteF('x è \d, y è \d, z è \d\n', x, y, z)
   ENDPROC

   PROC main()
     fred(2, 3, 4) -> Nessun defaults usato
     fred(2, 3)    -> z di default è 1
   ```

```
        fred(2)       -> y e z di default
        fred()        -> Illegale: x non è di default
      ENDPROC
```

In questo esempio non possiamo omettere il parametro y in una chiamata a fred senza omettere anche il parametro z. Per fare in modo che y rimanga con il suo valore di default e z con qualche valore diverso da quello suo di default, quando chiamiamo la procedura dobbiamo fornire esplicitamente il valore di y:

```
fred(2, 23, 9) -> C'è bisogno di fornire 23 per y
```

Queste limitazioni sono necessarie per non rendere ambigue le chiamate alle procedure. Consideriamo una procedura con tre paremetri, due dei quali con valori di default. Se tale procedura viene chiamata con due parametri soltanto, senza queste limitazioni, essa non saprebbe a quali dei tre parametri vogliamo riferirli. Al contrario, se la procedura viene definita e chiamata, conformemente a queste limitazioni, allora essa saprá che è il terzo parametro che deve essere settato con il valore di default (e che i due parametri con valori di default devono essere gli ultimi due).

1.49 beginner.guide/Valori Multipli di ritorno

```
2.2.4 Valori Multipli di ritorno
      ==========================
```

Finora abbiamo visto solo funzioni che ritornano un solo valore e ciò è cumune a molti linguaggi di programmazione. l'E invece ci permette di ritornare fino a tre valori da una funzione. Per ottenere questo, elenchiamo i valori separati da virgole, dopo le keywords ENDPROC, RETURN o IS, dove normalmente avremmo specificato solo un valore. Un buon esempio è una funzione che manipola una coordinata di schermo, che è data da due valori: le coordinate x- e y-.

```
PROC movediag(x, y) IS x+8, y+4
```

Tutto quello che fa questa funzione è di aggiungere 8 alla coordinata x e 4 all coordinata y. Per ottenere valori di ritorno diversi da questi dobbiamo usare una dichiarazione con multipla assegnazione:

```
PROC main()
  DEF a, b
  a, b:=movediag(10, 3)
  /* Ora a dovrebbe essere 10+8, e b dovrebbe essere 3+4 */
  WriteF('a è \d, b è \d\n', a, b)
ENDPROC
```

ad a è assegnato il primo valore di ritorno, ad b il secondo. Potremmo non avere bisogno di assegnare tutti i valori di ritorno da una funzione, quindi l'assegnazione dell'esempio qui sopra potrebbe assegnare un valore solo ad a (in tal caso non sarebbe una assegnazione multipla). Una assegnazione multipla ha senso solo se la parte destra (quella dopo :=) è una chiamata alla funzione, pertanto non sarebbe corretto scrivere un'assegnazione, come quella del seguente esempio, per assegnare b

correttamente:

```
       a,b:=6+movediag(10,3)  -> Nessun ovvio valore per b
```

Se usiamo una funzione con più di un valore di ritorno in qualsiasi altra espressione (cioè, qualcosa che non è la parte destra di un'assegnazione), allora solo il primo valore di ritorno viene usato. Per questa ragione i valori di ritorno di una funzione hanno nomi speciali: il primo valore di ritorno è chiamato regular (regolare) valore della funzione, e gli altri valori sono chiamati optional (facoltativi) valori.

```
PROC main()
  DEF a, b
  /* Le prossime due linee ignorano il secondo valore di ritorno */
  a:=movediag(10, 3)
  WriteF('x-coord di movediag(21, 4) è \d\n', movediag(21,4))
ENDPROC
```

1.50 beginner.guide/Costanti

```
                2.3 Costanti
    ********
```

Una costante è un valore che non cambia. Un numero come 121 è un buon esempio di costante, il suo valore è sempre 121. Noi abbiamo giá incontrato un altro tipo di costante: costanti stringa. Vedi
sez. 1.2.4
. Come si può
facilmente intuire le costanti sono piuttosto importanti.

Costanti numeriche

Costanti stringa, sequenze di caratteri speciali

Nomi di costanti (CONST)

Enumerazioni (ENUM)

Costanti con SET

1.51 beginner.guide/Costanti numeriche

```
2.3.1 Costanti numeriche
      ==================
```

Abbiamo incontrato parecchi numeri negli esempi precedenti. E tecnicamente parlando quei numeri erano delle costanti numeriche (costanti perchè essi non cambiano il loro valore come potrebbe fare una variabile). I numeri

usati erano tutti numeri decimali, ma noi possiamo usare anche numeri esadecimali e binari. Esiste un modo per rappresentare un numero usando dei caratteri al loro posto. Per specificare un numero esadecimale si usa il carattere $ subito prima delle cifre (e dopo il segno - se il numero è negativo). Per specificare un numero binario si usa invece il carattere %.

Rappresentare i numeri usando dei caratteri è più complicato, perchè la base di questo sistema è 256 (la base dei numeri decimali è dieci, quella dei numeri esadecimali è sedici e quella dei numeri binari è due). Le cifre sono incluse in doppie virgolette (carattere "), e possono essercene al massimo quattro. Ogni cifra è un carattere che rappresenta il suo valore ASCII. Perciò, il carattere A rappresenta 65 e il carattere 0 (zero) rappresenta 48. Perciò, in E, per usare il carattere A come valore numerico ASCII bisogna scrivere "A", e dato che la base di questo sistema di rappresentazione numerica come abbiamo detto è 256, scrivere "0z" equivale all'espressione ("0" * 256) + "z" e cioè (48 * 256) + 122, il risultato sará 12410. Tuttavia utilizzare questo metodo per ottenere dei valori numerici è abbastanza complicato, nella maggior parte dei casi sará utilizzato per ottenere solo il valore ASCII di un carattere.

La seguente tavola mostra il valore decimale di molte costanti numeriche. Nota che mentre il maiuscolo minuscolo delle lettere delle costanti esadecimali non è importante per il risultato finale, così non è per gli altri casi (in quanto il valore ASCII di A è ovviamente diverso da quello di a):

Numeri -	Valore decimale
21	21
-143	-143
$1a	26
-$B1	-177
%1110	14
-%1010	-10
"z"	122
"Je"	19,045
-"A"	-65

1.52 beginner.guide/Costanti stringa, sequenze di caratteri speciali

2.3.2 Costanti Stringa: Sequenze di Caratteri Speciali

Noi abbiamo visto che la sequenza di caratteri \n in una stringa, equivalgono a un linefeed. Vedi
sez. 1.2.4
. Ci sono molte altre sequenze
speciali, molto utili, che rappresentano caratteri che non possono essere scritti in una stringa. La seguente tavola mostra tali sequenze. Nota che esistono sequenze simili che sono usate per controllare il risultato fornito da procedure BUILT-IN come WriteF. Tali sequenze comunque sono listate dove le procedure WriteF e simili, sono descritte.
Vedi
sez. 2.6.3.1

.

```
Sequenza       Significato
--------------------------------------
   \0     Un null (ASCII zero)
   \a     Un apostrofo ’
   \b     Un ritorno carrello (ASCII 13)
   \e     Un escape (ASCII 27)
   \n     Un linefeed (ASCII 10)
   \t     Un tab (ASCII 9)
   \     Un backslash \
```

L’apostrofo può essere ottenuto anche scrivendone due consecutivi in una stringa, usandoli possibilmente nel mezzo di una stringa dove sono più logici e non causano confusione:

```
WriteF(’Un\aapostrofo.\n’)       /* Usando \a */

WriteF(’Un’’altro apostrofo.\n’)  /* Usando ’’ */
```

1.53 beginner.guide/Nomi di costanti

2.3.3 Nomi di costanti (CONST)

Spesso è utile poter dare dei nomi a certe costanti. Per esempio, come abbiamo visto in precedenza, il valore di veritá TRUE in effetti rappresenta il valore -1, e FALSE rappresenta 0. Vedi
sez. 1.3.2.2
. Questi
sono stati i nostri primi esempi di nomi di costanti. Per definire una nostra costante, dobbiamo usare la keyword CONST come segue:

```
CONST UNO=1, LINEFEED=10, GRANDE_NUM=999999
```

In questo modo abbiamo definito la costante UNO che rappresenta 1, la LINEFEED che rappresenta 10 e la GRANDE_NUM che rappresenta 999999. I nomi di costanti devono iniziare, come giá detto, con due lettere maiuscole. Vedi
sez. 2.1.1
.

Possiamo comunque anche cambiare il valore di una costante giá definita, ma in questo caso la nuava definizione deve stare da sola su una sola linea:

```
CONST ZERO=0
CONST UNO=ZERO+1
CONST DUE=UNO+1
```

L’espressione usata per definire il valore di una costante può usare solo semplici operatori (nessuna chiamata a funzione) e costanti.

1.54 beginner.guide/Enumerazioni

```
2.3.4 Enumerazioni (ENUM)
      ===================
```

Spesso vorremmo poter definire un'intera quantitá di costanti e aver bisogno che esse abbiano solo dei valori diversi, non importa quali, allora tali costanti possiamo definirle a parte in modo molto semplice. Ad esempio se vogliamo definire alcune costanti che rappresentino alcune cittá famose e vogliamo solo distinguerle le une dalle altre, allora possiamo usare l'enumerazione così:

```
ENUM LONDRA, MOSCA, NEW_YORK, PARIGI, ROMA, TOKYO
```

La keyword ENUM definisce le costanti esattamente come CONST solo che assegna loro automaticamente un valore che si incrementa di uno. I veri valori delle costanti iniziano da 0 sino ad arrivare a 5. In effetti potremmo anche scrivere così:

```
CONST LONDRA=0, MOSCA=1, NEW_YORK=2, PARIGI=3, ROMA=4, TOKYO=5
```

Tuttavia l'enumerazione non deve iniziare da 0. Possiamo cambiare il valore di partenza in qualsiasi punto specificando un valore per una costante enumerata. Ad esempio le seguenti definizioni di costanti sono equivalenti:

```
ENUM MELA, ARANCIO, GATTO=55, CANE, PESCE, FRED=-2,
     BARNEY, WILMA, BETTY

CONST MELA=0, ARANCIO=1, GATTO=55, CANE=56, PESCE=57,
      FRED=-2, BARNEY=-1, WILMA=0, BETTY=1
```

1.55 beginner.guide/Sets

```
           2.3.5 Costanti con SET
      ================
```

Un altro tipo ancora di definizione di costante si ottiene con SET. Questa è utile per definire il settaggio dei flags. Ossia possiamo trovarci un certo numero di opzioni da attivare o disattivare. Questa definizione è come una semplice enumerazione, ma usando la keyword SET i valori iniziano da uno e aumentano con potenze di due (così avremo come valori: 1, 2, 4, 8 e così via). Pertanto, le seguenti definizioni sono equivalenti:

```
SET INGLESE, FRANCESE, TEDESCO, ITALIANO, RUSSO

CONST INGLESE=1, FRANCESE=2, TEDESCO=4, ITALIANO=8, RUSSO=16
```

Tuttavia, l'espressione dei valori è meglio rappresentata usando costanti binarie:

```
CONST INGLESE=%00001, FRANCESE=%00010, TEDESCO=%00100,
      ITALIANO=%01000, RUSSO=%10000
```

Se una persona parla soltanto Italiano allora possiamo usare la costante ITALIANO. Se parla anche Inglese e vogliamo rappresentare questa situazione con un solo valore, allora ci servirebbe una nuova costante (qualcosa come ITA_ENG). In realtá avremmo bisogno di una nuova costante per ogni combinazione di linguaggi che la persona potrebbe conoscere. Ed ecco il vantaggio di SET, infatti con la precedente definizione e usando OR, possiamo unire i valori di INGLESE e ITALIANO ed avere un nuovo valore, 01001, questo nuovo numero rappresenta una assegnazione che comprende sia INGLESE che ITALIANO. D'altra parte per scoprire se una persona parla anche il Francese noi possiamo usare AND per aggiungere ai linguaggi che egli giá conosce il valore %00010 (o la costante FRANCESE). Come avrai giá capito, AND e OR sono davvero degli operatori intelligenti di bit (bitwise), e non solo quindi, semplici operatori logici. Vedi
sez. 2.5.4.3
.

Studia questo frammento di programma:

```
parla:=GERMAN OR ENGLISH OR RUSSIAN  /* parla alcune di queste */
IF parla AND ITALIANO
  WriteF('Può parlare in Italiano\n')
ELSE
  WriteF('Non può parlare in Italiano\n')
ENDIF
IF parla AND (GERMAN OR FRENCH)
  WriteF('Può parlare in Tedesco o in Francese\n')
ELSE
  WriteF('Non può parlare in Tedesco o in Francese\n')
ENDIF
```

L'assegnazione setta parla, in modo da evidenziare che la persona può parlare in Tedesco, Inglese o Russo. Il primo blocco IF verifica se la persona può parlare in Italiano, mentre il secondo verifica se la persona può parlare in Tedesco o in Francese.

Nel momento in cui si usa SET bisogna prestare attenzione, non dobbiamo essere tentati di ottenere con OR una addizione di valori invece di una unione. Infatti noi possiamo addizionare due differenti costanti definite con lo stesso SET ed avremmo lo stesso risultato che le avessimo unite con OR, ma non è così se addizioniamo una costante a se stessa (il risultato dell'addizione sará diverso da quello dell'unione con OR). Es.: INGLESE +ITALIANO da lo stesso risultato di INGLESE OR ITALIANO ma ITALIANO+ITALIANO non da lo stesso risultato di ITALIANO OR ITALIANO, Questo non è il solo caso in cui non avremmo lo stesso risultato, ma è il più evidente. Comunque se usiamo OR solo per unire non avremo problemi.

1.56 beginner.guide/I Tipi

2.4 I Tipi
******

Noi abbiamo giá incontrato il tipo LONG e sappiamo che normalmente quando definiamo una variabile, questa viene vista automaticamente di tipo LONG. Vedi
sez. 1.3.1.1
. Abbiamo anche accennato ai tipi INT e LIST. Imparare a scrivere un programma in modo efficace e leggibile è molto importante. Il tipo di una variabile (e anche il suo nome) può dare al lettore una idea più precisa del come e per cosa, essa viene usata. Ci sono ovviamente anche ragioni più importanti per avere bisogno dei tipi, per esempio per raggruppare logicamente dei dati quando usiamo gli OBJECT.
Vedi
sez. 2.4.4
.

Questo è un capitolo molto complesso e si consiglia di apprenderlo lentamente. Una delle cose più importanti è arrivare ad avere una buona padronanza nell'uso dei puntatori. Bisogna concentrarsi nel tentare di capire l'uso dei puntatori in quanto si usano molto per programmare con qualsiasi tipo di funzione di sistema.

Tipo LONG

Tipo PTR

Tipo ARRAY (Matrice)

Tipo OBJECT

Tipi LIST e STRING

Liste linked

1.57 beginner.guide/Tipo LONG

2.4.1 Tipo LONG
=========

Il tipo LONG è il tipo più importante perchè è quello di default ed è di gran lunga il tipo più usato. Il tipo LONG può essere usato per conservare molte varietá di dati, includendo gli indirizzi di memoria, come vedremo.

Tipo di default

Indirizzi di memoria

1.58 beginner.guide/Tipo di default

```
2.4.1.1 Tipo di default
        ---------------
```

LONG è il tipo di default delle variabili. E' un tipo a 32-bit, questo significa che 32-bit di memoria (RAM) sono usati per conservare i dati, ad ogni variabile di questo tipo possiamo assegnare qualsiasi valore (intero) compreso nella gamma da -2,147,483,648 a 2,147,483,647. Le variabili per default sono LONG, ma possono anche essere dichiarate in maniera esplicita di tipo LONG:

```
DEF x:LONG, y

PROC fred(p:LONG, q, r:LONG)
  DEF zed:LONG
  dichiarazioni
ENDPROC
```

La variabile locale x, i parametri di procedura p ed r, la variabile locale zed, sono state dichiarate in maniera esplicita come LONG. Le dichiarazioni sono molto simili a quelle viste finora, eccetto che le variabili hanno un :LONG dopo il loro nome nella dichiarazione. Questo è il modo per definire il tipo di una variabile. Nota che la variabile globale y e il parametro di procedura q sono sempre di tipo LONG, almeno fino a quando non specificheremo un tipo diverso per loro, LONG infatti, come giá detto, è il tipo che viene assegnato per default alle variabili.

1.59 beginner.guide/Indirizzi di memoria

```
2.4.1.2 Indirizzi di memoria
        --------------------
```

C'è un'ottima ragione per usare LONG come tipo di default. Un valore (intero) a 32 bit può essere usato come un indirizzo di memoria. Perciò noi possiamo usare la varibile per conservare l'indirizzo (o posizione) dei dati (la variabile in questo caso è chiamata puntatore). Infatti la variabile, in questo caso, non conterrebbe il valore dei dati, ma piuttosto un modo per trovarli. Una volta che conosciamo la posizione dei dati, questi possono essere letti o anche modificati! La prossima sezione spiega più in dettaglio i puntatori e gli indirizzi.

1.60 beginner.guide/Tipo PTR

```
            2.4.2 Tipo PTR
      ========
```

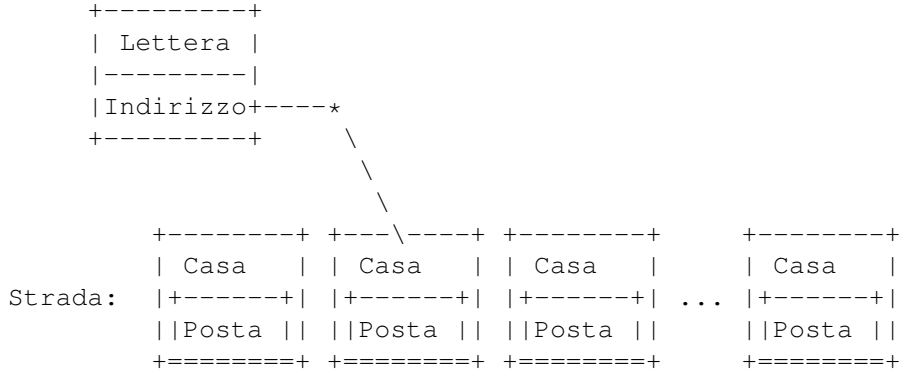
Il tipo PTR è usato per conservare gli indirizzi di memoria. Le variabili che hanno un tipo PTR sono chiamate puntatori (fin quando esse conservano un indirizzo di memoria, come abbiamo detto nella precedente sezione). Questa sezione descrive in dettaglio gli indirizzi, i puntatori e il tipo PTR.

1.61 beginner.guide/Indirizzi

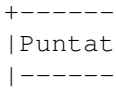
2.4.2.1 Indirizzi

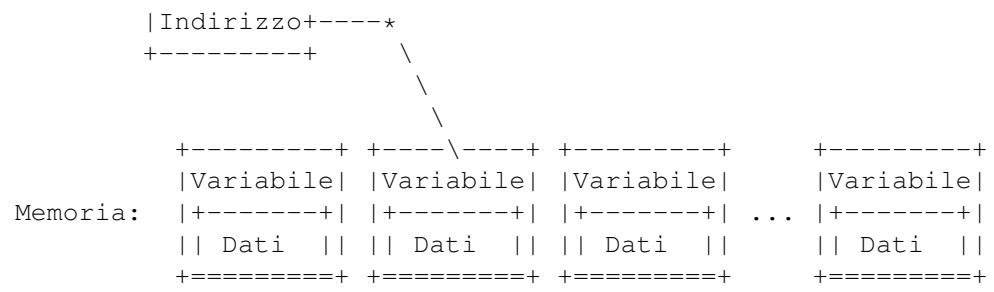
Per capire meglio gli indirizzi, immaginiamo la memoria come una lunga strada, ogni posizione di memoria come una cassetta delle lettere in una casa, e ogni frammento di dati come una lettera. Se tu sei il postino, hai bisogno di sapere dove devi consegnare le lettere, questa informazione è data dall'indirizzo della cassetta delle lettere. Dopo un po' di tempo, ogni cassetta sará riempita con differenti lettere. Questo è come cambiare il valore in una posizione di memoria (o variabile). Per cambiare il numero di lettere conservate nella tua cassetta, tu comunica il tuo indirizzo ai tuoi amici ed essi potranno spedirti delle lettere che andranno a riempire la tua cassetta. Questo è come lasciare che qualche programma cambi i tuoi dati dando ad esso il loro indirizzo.

I prossimi due tracciati illustrano questa analogia. Una lettera contiene un indirizzo che punta ad una particolare casa (o cassetta delle lettere) su una strada.



Un puntatore contiene un indirizzo che punta ad una variabile (o dati) in memoria.





1.62 beginner.guide/Puntatori

2.4.2.2 Puntatori

Le variabili che contengono gli indirizzi di memoria sono chiamate puntatori. Come abbiamo visto nella precedente sezione, possiamo conservare gli indirizzi di memoria in variabili LONG. Comunque ancora non sappiamo il tipo dei dati conservati a quegli indirizzi. Se è importante (o utile) avere questa informazione, allora il tipo PTR (o più precisamente, uno dei molti tipi PTR) dovrebbe essere usato:

```
DEF p:PTR TO LONG, i:PTR TO INT,
    cptr:PTR TO CHAR, gptr:PTR TO gadget
```

I valori conservati in p, i, cptr e gptr sono LONG fin quando essi sono degli indirizzi di memoria. Tuttavia i dati conservati all'indirizzo p sono presi come LONG (valore a 32 bit), quelli all'indirizzo i come INT (valore a 16 bit), quelli all'indirizzo cptr come CHAR (valore a 8 bit) e quelli all'indirizzo gptr come gadget che è un oggetto. Vedi
sez. 2.4.4
.

1.63 beginner.guide/Tipi indiretti

2.4.2.3 Tipi indiretti

Nel precedente esempio abbiamo visto INT e CHAR usati come tipi di destinazione dei puntatori, e questi sono rispettivamente gli equivalenti a 16 e 8 bit del tipo LONG. Tuttavia, diversamente da LONG questi tipi non possono essere usati direttamente per dichiarare variabili globali, locali o paremetri di procedura. Possono essere usati solo per costruire tipi (per esempio con PTR). Le seguenti dichiarazioni sono perciò illegali, potrebbe essere utile provare a compilare un piccolo programma con una tale dichiarazione, soltanto per vedere che messaggio di errore da il compilatore E.

```
/* Questo frammento di programma contiene dichiarazioni illegali */
DEF c:CHAR, i:INT
```

```
/* Questo frammento di programma contiene dichiarazioni illegali */
 PROC fred(a:INT, b:CHAR)
   DEF x:INT
   dichiarazioni
 ENDPROC
```

Questa non è una grossa limitazione in quanto possiamo conservare i valori INT o CHAR In variabili LONG se proprio ne abbiamo bisogno. Comunque questo significa che c'è una buona e semplice regola: ogni valore diretto in E è una dimensione di 32 bit o LONG o un puntatore. In effetti LONG è un modo breve per intendere PTR TO CHAR, così possiamo usare valori LONG come se in realtá fossero valori PTR TO CHAR.

1.64 beginner.guide/Trovare indirizzi (costruire puntatori)

2.4.2.4 Trovare indirizzi (costruire puntatori)

Se un programma conosce l'indirizzo di una variabile può leggere o modificare direttamente il valore della variabile. Per ottenere l'indirizzo di una variabile semplice dobbiamo racchiudere il nome della variabile fra parentesi graffe {}. L'indirizzo di variabili complesse (per esempio oggetti e matrici) lo possiamo trovare più facilmente (vedi le appropriate sezioni). In realtá useremo {var} molto raramente, comunque se apprendiamo bene come ottenere dei puntatori con {var} e il loro uso per arrivare ai dati, allora capirai l'uso dei puntatori per tipi complessi, molto più velocemente.

Gli indirizzi possono essere conservati in una variabile, passati ad una procedura o altro (essi sono soltanto valori a 32 bit). Proviamo il seguente programma:

```
DEF x

PROC main()
  fred(2)
ENDPROC

PROC fred(y)
  DEF z
  WriteF('x è all''indirizzo \d\n', {x})
  WriteF('y è all''indirizzo \d\n', {y})
  WriteF('z è all''indirizzo \d\n', {z})
  WriteF('fred è all''indirizzo \d\n', {fred})
ENDPROC
```

Notare che possiamo trovare anche l'indirizzo di una procedura usando la coppia di parentesi graffe {}. Tale indirizzo è la posizione di memoria del codice che la procedura rappresenta. Qui c'è l'output che l'esecuzione di questo programma da: (non aspettarti esattamente lo stesso output, tuttavia):

```
x è all'indirizzo 3758280
y è all'indirizzo 3758264
z è all'indirizzo 3758252
```

```
fred è all'indirizzo 3732878
```

Questo è un programma interessante, prova ad eseguirlo diverse volte in circostanze diverse. Vedrai che alcune volte i numeri degli indirizzi cambiano. Eseguendo il programma quando un altro è in multi-tasking (in esecuzione e quindi occupando memoria) dovresti avere i cambiamenti più vistosi, mentre eseguendo il programma consecutivamente da cli dovresti avere (se ce ne sono) cambiamenti lievi. Questo ti deve dare un'idea del complesso uso della memoria da parte di Amiga e del compilatore E.

1.65 beginner.guide/Estrazione dei dati (dereferencing i puntatori)

2.4.2.5 Estrazione dei dati (dereferencing i puntatori)

Se abbiamo un indirizzo conservato in una variabile (cioè un puntatore), possiamo estrarne i dati usando l'operatore ^. Questa azione di estrazione dati con un puntatore è chiamata dereferencing il puntatore. Questo operatore deve essere usato solo quando una {var} è stata usata per ottenere un indirizzo. A tal fine i valori LONG sono letti e scritti quando si dereferencing un puntatore in questo modo. Per i puntatori a tipi complessi (per esempio oggetti e matrici), dereferencing è ottenuto in modo molto più leggibile (per i dettagli vedere le appropriate sezioni) e tale operatore (^) non è usato. In realtá ^{var} è usato raramente nei programmi, ma è utile per capire come lavorano i puntatori, specialmente in congiunzione con {var}.

Usando i puntatori possiamo rimuovere la restrizione che abbiamo sulle variabili locali, ossia possiamo modificarle anche al di fuori della precedura a cui appartengono. Generalmente non consigliamo di fare questo, ma usiamo tale possibiltá perchè è un ottimo esempio per mostrare la potenza dei puntatori. Per esempio, il seguente programma cambia il valore della variabile locale x della procedura fred dall'interno della procedura barney:

```
PROC main()
  fred()
ENDPROC

PROC fred()
  DEF x, p:PTR TO LONG
  x:=33
  p:={x}     /* assegnamo a p l'indirizzo di x */
  barney(p) /* chiamata della procedura barney passando l'indirizzo*/
  WriteF('x è ora \d\n', x)
ENDPROC

PROC barney(indirizzodix:PTR TO LONG)
  DEF val
  val:=^indirizzodix
  ^indirizzodix:=val-6
ENDPROC
```

Questo è l'output, se hai capito, che devi aspettarti:

```
x è ora 27
```

Come si vede l'operatore ^ (cioè dereferencing) è abbastanza versatile. Nella prima assegnazione della procedura barney è usato (con il puntatore indirizzodix) per prendere il valore conservato nella variabile x, nella seconda è usato per cambiare il valore di questa variabile. Nell'uno o nell'altro caso, dereferencing fa funzionare il puntatore precisamente come se avessimo scritto la variabile a cui punta al suo posto. Per evidenziare tale affermazione, possiamo rimuovere la procedura barney, come abbiamo fatto in precedenza (Vedi

sez. 1.2.5

):

```
PROC main()
  fred()
ENDPROC

PROC fred()
  DEF x, p:PTR TO LONG, val
  x:=33
  p:={x}     /* assegnamo a p l'indirizzo di x */
  val:=x    /* x al posto di ^indirizzodix    */
  x:=val-6 /* x al posto di ^indirizzodix    */
  WriteF('x è ora \d\n', x)
ENDPROC
```

Come si può notare, dove prima nella procedura barney, abbiamo usato ^indirizzodix, ora abbiamo usato direttamente x (questo perchè ora siamo nella procedura per cui x è locale). Abbiamo anche eliminato la variabile indirizzodix (il parametro della procedura barney), perchè veniva usata solo con l'operatore ^.

Per rendere più leggibili le procedure fred e barney l'esempio è stato deliberatamente prolisso. le variabili val e p sono inutili, come è inutile definire i puntatori in modo che agiscano su valori LONG, perchè è il tipo che viene utilizzato di default come è stato spiegato in precedenza. Vedi

sez. 2.4.1

. Questa è la forma compatta dell'esempio:

```
PROC main()
  fred()
ENDPROC

PROC fred()
  DEF x
  x:=33
  barney({x})
  WriteF('x è ora \d\n', x)
ENDPROC

PROC barney(indirizzodix)
  ^indirizzodix:=^indirizzodix-6
ENDPROC
```

L'uso più comune dei puntatori è di gran lunga quello di indirizzare (o fare riferimento a) grandi strutture di dati. Sarebbe estremamente oneroso

(in termini di tempo per la CPU) passare grandi quantitá di dati da procedura a procedura, passiamo così solo gli indirizzi (che sappiamo essere solo valori a 32 bit). Le funzioni di sistema di Amiga (come quelle per creare finestre) richiedono parecchi dati strutturati, così se progettiamo seriamente un programma dobbiamo capire e usare i puntatori.

1.66 beginner.guide/Parametri di procedura

2.4.2.6 Parametri di procedura

Solo le variabili globali e locali hanno il lusso di poter scegliere fra una grande quantitá di tipi. I parametri di procedura possono essere solo di tipo LONG o PTR TO. E anche questa non è una grande limitazione come vedremo nelle successive sezioni.

1.67 beginner.guide/Tipo ARRAY

2.4.3 Tipo ARRAY (Matrice)
==========

Molto spesso i dati di un programma hanno bisogno di essere ordinati in qualche modo, principalmente per essere reperiti facilmente. Il linguaggio E fornisce un modo semplice per ottenere tale ordinamento: il tipo ARRAY. Questo tipo (nelle sue varie forme) è comune a molti linguaggi per computer.

1.68 beginner.guide/Tavole di dati

2.4.3.1 Tavole di dati

I dati possono essere raggruppati insieme in molti modi differenti, ma probabilmente il modo più comune e semplice è costruire una tavola. In una tavola i dati sono organizzati o verticalmente o orizzontalmente, ma la cosa importante è il posizionamento relativo degli elementi. In E questo

tipo di ordinamento dati è gestito dal tipo ARRAY. Un array è soltanto un raccoglitore a dimensione fissa di dati ordinati. La dimensione di una matrice è importante e bisogna stabilirla quando la dichiariamo. L'esempio seguente mostra delle dichiarazioni di matrice:

```
DEF a[132]:ARRAY,
    table[21]:ARRAY OF LONG,
    ints[3]:ARRAY OF INT,
    objs[54]:ARRAY OF mioobject
```

La dimensione della matrice la diamo fra parentesi quadre []. Il tipo degli elementi nella matrice di default è CHAR, ma questo possiamo definirlo esplicitamente usando la keyword OF e il nome del tipo. Comunque i tipi permessi sono solo LONG, INT, CHAR e object (LONG può contenere i valori dei puntatori così questa non è una grossa limitazione). I tipi OBJECT sono descritti in seguito. Vedi
sez. 2.4.4
.

Come accennato prima, i parametri di procedura non possono essere matrici. Vedi
sez. 2.4.2.6
. Supereremo presto questa limitazione. Vedi
sez. 2.4.3.5
.

1.69 beginner.guide/Utilizzare i dati di un array

2.4.3.2 Utilizzare i dati di un array

Per utilizzare un particolare elemento in una matrice useremo nuovamente le parentesi quadre, specificando l'indice (o posizione) dell'elemento che ci serve. Gli indici iniziano da zero, quindi per il primo elemento della matrice l'indice sará zero, per il secondo sará uno e quindi in generale (n-1). Questo può sembrare strano inizialmente, ma è il modo usato dalla maggior parte dei linguaggi per computer! Presto ne vedremo la ragione. Vedi
sez. 2.4.3.3
.

```
DEF a[10]:ARRAY

PROC main()
  DEF i
  FOR i:=0 TO 9
    a[i]:=i*i
  ENDFOR
  WriteF('Il 7\textdegree{} elemento della matrice a è \d\n', a[6])
  a[a[2]]:=10
  WriteF('La matrice è ora:\n')
  FOR i:=0 TO 9
    WriteF(' a[\d] = \d\n', i, a[i])
  ENDFOR
```

```
ENDPROC
```

Questo codice dovrebbe essere molto semplice da capire, sebbene una delle linee sembra un po' complicata. Prova a capire cosa succede alla matrice dopo l'assegnazione fatta nel rigo immediatamente seguente il primo WriteF. In questa assegnazione viene usato come indice un valore conservato nella matrice stessa cioè 4 (a[2] = 4), quindi nella matrice a[4] troveremo il valore 10 al posto di 16 (valore della prima assegnazione). Bisogna prestare attenzione nel momento in cui facciamo delle cose complicate con le matrici: bisogna accertarsi di non leggere o scrivere dati usando indici con valori superiori al numero di elementi che queste possono contenere. Nell'esempio ci sono solo dieci elementi nella matrice a, quindi non sarebbe logico usare un undicesimo elemento. Il programma avrebbe potuto controllare se il valore conservato in a[2] fosse un numero compreso fra zero e nove prima di tentare di accedere a quell'elemento di matrice, ma in questo caso non è stato necessario. Qui c'è l'output che l'esempio dovrebbe generare:

```
Il 7\textdegree{} elemento della matrice a è 36
La matrice è ora:
 a[0] = 0
 a[1] = 1
 a[2] = 4
 a[3] = 9
 a[4] = 10
 a[5] = 25
 a[6] = 36
 a[7] = 49
 a[8] = 64
 a[9] = 81
```

Se provi a scrivere indici di matrice inesistenti, possono accadere cose strane. Quindi è praticamente sconsigliato provarci (si può daneggiare qualche altro dato), ma se sei veramente sfortunato puoi mandare in crash il tuo computer. Di conseguenza bisogna rimanere entro i limiti imposti alla matrice.

Un'abbrevazione per il primo elemento di una matrice (cioè quella con indice zero) è omettere l'indice e scrivere solo le parentesi quadre. Pertanto, a[] è l'equivalente di a[0].

1.70 beginner.guide/Puntatori agli array

2.4.3.3 Puntatori agli array

Quando dichiariamo una matrice il suo indirizzo (inizio della matrice) è dato dal nome della variabile senza parentesi quadre. Vediamo questo programma:

```
DEF a[10]:ARRAY OF INT

PROC main()
  DEF ptr:PTR TO INT, i
```

```
    FOR i:=0 TO 9
      a[i]:=i
    ENDFOR
    ptr:=a
    ptr++
    ptr[]:=22
    FOR i:=0 TO 9
      WriteF('a[\d] è \d\n', i, a[i])
    ENDFOR
  ENDPROC
```

L'output è:

```
  a[0] è 0
  a[1] è 22
  a[2] è 2
  a[3] è 3
  a[4] è 4
  a[5] è 5
  a[6] è 6
  a[7] è 7
  a[8] è 8
  a[9] è 9
```

Dovresti notare che il secondo elemento della matrice è stato cambiato usando il puntatore. La dichiarazione ptr++ (nota: ptr è un semplice nome di variabile, di puntatore in questo caso, non è una keyword dell'E, pertanto possiamo usare qualsiasi nome al suo posto, del resto è scritto in minuscolo. ++ invece è un operatore proprio dell'E che incrementa di un passo un valore) incrementa il puntatore in modo che possa puntare all'elemento successivo della matrice. E' importante che ptr venga dichiarato come PTR TO INT in quanto anche la matrice è stata dichiarata ARRAY OF INT. Le [] sono usate per cambiare il valore contenuto all'indirizzo rappresentato da ptr (anche in questo caso il nome di questa operazione è dereference, ricordate l'operatore ^ ?), quindi il nuovo valore conservato nel secondo elemento della matrice sará 22. Comunque bisogna tener presente che ptr possiamo usarlo nello stesso modo di una matrice, quindi possiamo scrivere ptr[1] per puntare al terzo elemento della matrice (si punta al terzo se l'usiamo dopo ptr++, avanziamo di un altro passo praticamente). Possiamo usare anche valori negativi come indice, pertanto se scriviamo ptr[-1] punteremmo al primo elemento della matrice dopo l'uso di ptr++ (in pratica si decrementa di un passo in questo modo).

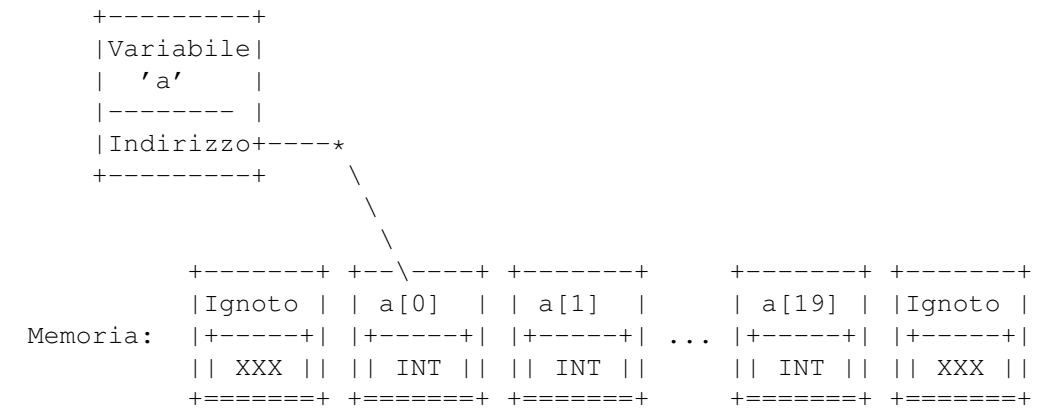
Le seguenti dichiarazioni sono identiche alle precedenti, con la sola differenza che le prime riservano un'appropriata quantitá di memoria per la matrice e le seconde fanno affidamento sul fatto che tale operazione è stata fatta in un altro punto del programma.

```
  DEF a[20]:ARRAY OF INT

  DEF a:PTR TO INT
```

Il seguente schema è simile a quelli usati prima (Vedi
sez. 2.4.2.1
)
ed è un'immagine esplicativa della matrice, a, dichiarata in modo da

contenere 20 valori di tipo INT:



Come possiamo vedere, la variabile, a, è un puntatore a quella parte di memoria che contiene gli elementi della matrice. Le parti di memoria non comprese fra a[0] e a[19] sono siglate come 'Ignoto' in quanto esse non fanno parte della matrice. Quindi non dobbiamo fare riferimento a queste parti di memoria tramite la matrice a.

1.71 beginner.guide/Puntare agli altri elementi

2.4.3.4 Puntare agli altri elementi

Abbiamo visto nella precedente sezione come incrementare un puntatore in modo da farlo puntare al successivo elemento nella matrice. Per decrementare un puntatore (cioè, per farlo puntare all'elemento precedente), si usa un sistema simile, anzichè usare la dichiarazione p++, useremo la dichiarazione p--, equivalente di p++, ma contraria. In effetti, p++ e p-- sono espressioni che rappresentano gli indirizzi del puntatore. p++ rappresenta l'indirizzo conservato in p, incrementato di un passo e p-- rappresenta l'indirizzo conservato in p, decrementato di un passo. Pertanto

```
addr:=p
p++
```

è equivalente di

```
addr:=p++
```

e

```
p--
addr:=p
```

è equivalente di

```
addr:=p--
```

La ragione per cui ++ e -- dovrebbero essere usati per incrementare o decrementare un puntatore è che valori di differente tipo (LONG, INT, ecc.)

occupano differenti numeri di posizione di memoria. In effetti, una singola posizione di memoria è un byte e questi è formato da otto bit. Pertanto valori CHAR occupano un singolo byte, mentre valori LONG occupano quattro byte (32 bit). Se p viene usato come puntatore a CHAR, allora la posizione di memoria p+1 conterrebbe il secondo elemento della matrice (e p+2 il terzo, ecc.). Ma se p viene usato come puntatore ad una matrice LONG, il secondo elemento nella matrice si troverebbe a p+4 (e il terzo a p+8). Le posizioni p, p+1, p+2, p+3, quindi servono solo per poter far riferimento a parti del valore conservato all'indirizzo p (cioè il primo elemento della matrice). Doversi ricordare tali posizioni mentre realizziamo un programma è difficoltoso e rende il programma meno leggibile al contrario di ++ e -- che puntano subito al successivo o precedente elemento della matrice, purchè ovviamente ci ricordiamo di dichiarare il tipo di valore a cui il puntatore deve far riferimento (che deve coincidere con il tipo dichiarato per la matrice).

1.72 beginner.guide/Array, parametri di procedura

2.4.3.5 Array, parametri di procedura

Ora sappiamo come ottenere l'indirizzo di una matrice e possiamo quindi vedere come passare questa come parametro di procedura utilizzando appunto il suo indirizzo (visto che in precedenza abbiamo detto che non possiamo passare direttamente la matrice come parametro). Per esempio, il seguente programma usa una procedura per riempire i dieci elementi della matrice, a, usando come valori quelli degli indici, dati dalla variabile x.

```
DEF a[10]:ARRAY OF INT

PROC main()
  DEF i
  fillin(a, 10)
  FOR i:=0 TO 9
    WriteF('a[\d] è \d\n', i, a[i])
  ENDFOR
ENDPROC

PROC fillin(ptr:PTR TO INT, x)
  DEF i
  FOR i:=0 TO x-1
    ptr[]:=i
    ptr++
  ENDFOR
ENDPROC
```

Questo è l'output che dovrebbe generare:

```
a[0] è 0
a[1] è 1
a[2] è 2
a[3] è 3
a[4] è 4
a[5] è 5
a[6] è 6
```

```
a[7] è 7
a[8] è 8
a[9] è 9
```

La matrice, a, ha solo dieci elementi, non possiamo riempirne di più. Pertanto, nell'esempio, la chiamata alla procedura fillin non può avere un numero più grande di dieci come secondo parametro. Potevamo anche trattare ptr come una matrice (e non usare ++), ma in questo caso si è rivelato leggermente meglio l'uso di ++ in quanto l'ssegnazione di ogni elemento della matrice è avvenuta a turno. La definizione alternativa della procedura fillin (senza usare ++) è:

```
PROC fillin2(ptr:PTR TO INT, x)
  DEF i
  FOR i:=0 TO x-1
    ptr[i]:=i
  ENDFOR
ENDPROC
```

Inoltre, un'altra versione ancora di fillin usa la forma di espressione ++ e la sintassi orizzontale del loop FOR, questo per avere il codice della procedura ancora più compatto:

```
PROC fillin3(ptr:PTR TO INT, x)
  DEF i
  FOR i:=0 TO x-1 DO ptr[]++:=i
ENDPROC
```

1.73 beginner.guide/Tipo OBJECT

```
            2.4.4 Tipo OBJECT
 ===========
```

Gli object in E sono l'equivalente delle strutture del C e dell'Assembly, o dei record del Pascal. Gli object sono paragonabili alle matrici, eccetto che gli elementi non sono chiamati con i numeri e possono essere di tipi differenti. Per trovare un particolare elemento in un object si usa un nome invece di un indice (numero). Gli objects sono anche la base delle caratteristiche OOP dell'E. Vedi
sez. 2.12
).

Esempio di object

Selezione e tipi degli elementi

Objects di sistema di Amiga

1.74 beginner.guide/Esempio di object

2.4.4.1 Esempio di object

Useremo direttamente un esempio, per chiarire la situazione, che definisce un oggetto e lo usa. Le definizioni di un object sono globali e quindi devono essere fatte prima delle definizioni delle procedure:

```
OBJECT rec
  tag, check
  table[8]:ARRAY
  data:LONG
ENDOBJECT

PROC main()
  DEF a:rec
  a.tag:=1
  a.check:=a
  a.data:=a.tag+(10000*a.tag)
ENDPROC
```

Questo programma non da risultati visibili, di conseguenza non c'è molto da vedere se lo compiliamo. Comunque è utile per vedere la tipica definizione di object e come i suoi elementi sono selezionati.

L'oggetto che viene definito nell'esempio è rec, e i suoi elementi sono definiti proprio come dichiarazioni di variabili (ma senza DEF). Possono esserci tutte le linee di definizioni di elementi che vogliamo tra le linee OBJECT e ENDOBJECT, e ogni linea può contenere un qualsiasi numero di elementi separati da virgole. Gli elementi dell'object rec sono tag e check (che sono di tipo LONG), table (che è una matrice di tipo CHAR con otto elementi) e data (che è di tipo LONG). Ogni tipo di variabile dell'object rec avrá uno spazio riservato idoneo. La dichiarazione della variabile (locale) a, riserva quindi la giusta memoria per l'object rec.

1.75 beginner.guide/Selezione e tipi degli elementi

2.4.4.2 Selezione e tipi degli elementi

Per selezionare gli elementi in un object obj (obj sta per il nome della variabile a cui abbiamo assegnato l'object, nell'esempio è la variabile a) bisogna usare obj.nome, dove nome è uno dei nomi degli elementi. Nell'esempio, l'elemento tag dell'object rec a, è selezionato scrivendo a.tag. Gli altri elementi sono selezionati in modo simile.

Proprio come in una dichiarazione di matrice, l'indirizzo di un object è conservato nella variabile obj, e qualsiasi puntatore di tipo PTR TO objectnome può essere usato come un object di tipo objectnome. Pertanto nel precedente esempio a è un PTR TO rec.

Come l'object dell'esempio evidenzia, gli elementi di un object possono essere di tipi diversi. Infatti gli elementi possono essere di qualsiasi tipo, includendo l'object, il puntatore all'object e all'array dell'object.

Il seguente esempio mostra come accedere ad alcuni differenti tipi di elementi:

```
OBJECT rec
  tag, check
  table[8]:ARRAY
  data:LONG
ENDOBJECT

OBJECT bigrec
  data:PTR TO LONG
  subrec:PTR TO rec
  rectable[22]:ARRAY OF rec
ENDOBJECT

PROC main()
  DEF r:rec, b:bigrec, rt:PTR TO rec
  r.table[]:="H"
  b.subrec:=r
  b.subrec.tag:=1
  b.subrec.data:=r.tag+(10000*b.subrec.tag)
  b.subrec.table[1]:="i"
  b.rectable[0].data:=r.tag
  b.rectable[0].table[0]:="A"
  rt:=b.rectable
  rt[].data++:=0
  rt[].table[]--:="B"
ENDPROC
```

Gli operatori ++ e -- sono applicati alla prima cosa nella selezione (cioè, rt in entrambe le ultime due assegnazioni dell'esempio qui sopra), e possono essere usati solo dopo tutte le selezioni. Bisogna tener presente che quella selezione dell' object e dell' array indicizzato può essere ripetuta tante volte quanto serve (ma solo come i tipi degli elementi permettono). Consideriamo la terza assegnazione come un semplice esempio:

```
  b.subrec.tag:=1
```

Questa, seleziona l'elemento subrec dall'object bigrec b, e poi assegna a 1 l'elemento tag dell'object rec. Ora consideriamo una delle assegnazioni sucessive:

```
  b.rectable[0].table[0]:="A"
```

Questa, seleziona l'elemento rectable da b, che è un array di oggetti rec, poi viene selezionato il primo elemento di questa matrice, poi l'elemento table dell'object rec e finalmente alla prima posizione di table viene assegnato il valore ASCII di A.

Come probabilmente avrai capito, è importante dare agli elementi object i tipi appropriati se vuoi fare una selezione multipla in questo modo. Tuttavia questo non sempre è possibile oppure non è il miglior modo di fare alcune cose, pertanto esiste un modo per dare un differente tipo ai puntatori (ed è chiamato explicit pointer typing-- Vedi il 'Reference Manual') per maggiori dettagli).

Segue un esempio abbastanza semplice che usa un array di object:

```
OBJECT rec
  tag, check
  table[8]:ARRAY
  data:LONG
ENDOBJECT

PROC main()
  DEF a[10]:ARRAY OF rec, p:PTR TO rec, i
  p:=a
  FOR i:=0 TO 9
    a[i].tag:=i
    p.check++:=i
  ENDFOR
  FOR i:=0 TO 9
    IF a[i].tag<>a[i].check
      WriteF('Whoops, a[\d] non riuscito...\n', i)
    ENDIF
  ENDFOR
ENDPROC
```

Se studi bene questo esempio, ti accorgerai che a[0].tag è uguale ad a.tag, questo perchè a, è un puntatore al primo elemento dell'array e gli elementi della matrice sono objects. Pertanto, a, è un puntatore ad un object (il primo object nell'array).

1.76 beginner.guide/Objects di sistema di Amiga

```
                2.4.4.3 Objects di sistema di Amiga
        ---------------------------
```

Ci sono molti differenti objects di sistema di Amiga. Per esempio, c'è un object che contiene l'informazione desiderata per avere un gadget (come il gadget di 'chiusura' sulla maggior parte delle finestre) e uno che contiene tutte le informazioni su un processo o task. Questi objects sono di importanza vitale e così sono forniti con l'E nella forma di 'moduli'. Ogni modulo è specifico ad una certa area del sistema di Amiga e contiene l'object e altre definizioni. I moduli sono spiegati più in dettaglio in seguito. Vedi

```
                sez. 2.7
                .
```

1.77 beginner.guide/Tipi LIST e STRING

```
                2.4.5 Tipi LIST e STRING
      ==================
```

Le matrici sono comuni a molti linguaggi per computer. Comunque queste possono causare qualche problema in quanto abbiamo sempre l'esigenza di verificare di non superare i limiti della stessa quando facciamo

riferimento ad essa. Ed ecco dove i tipi STRING e LIST ci vengono in aiuto. STRING è molto simile ad ARRAY OF CHAR e LIST è simile ad ARRAY OF LONG. Tuttavia il linguaggio E ha una serie di funzioni (BUILT-IN) che manipolano con sicurezza ogni varibile di questo tipo senza eccedere i loro limiti.

1.78 beginner.guide/Stringhe normali ed E-strings

2.4.5.1 Stringhe normali ed E-strings

Le stringhe normali sono comuni a più linguaggi di programmazione. Esse sono semplicemente delle matrici di caratteri, con la fine della stringa marcata da un carattere nullo (ASCII zero). Abbiamo giá incontrato le stringhe normali. Vedi

sez. 1.2.4

. Abbiamo usato stringhe fisse (costanti)

contenute fra i caratteri ’’, i quali evidenziano i puntatori alla memoria dove i dati della stringa sono conservati. Pertanto, possiamo assegnare una costante stringa ad un puntatore (a CHAR), e otteniamo una matrice con gli elementi pronti e pieni, ossia, una matrice inizializzata.

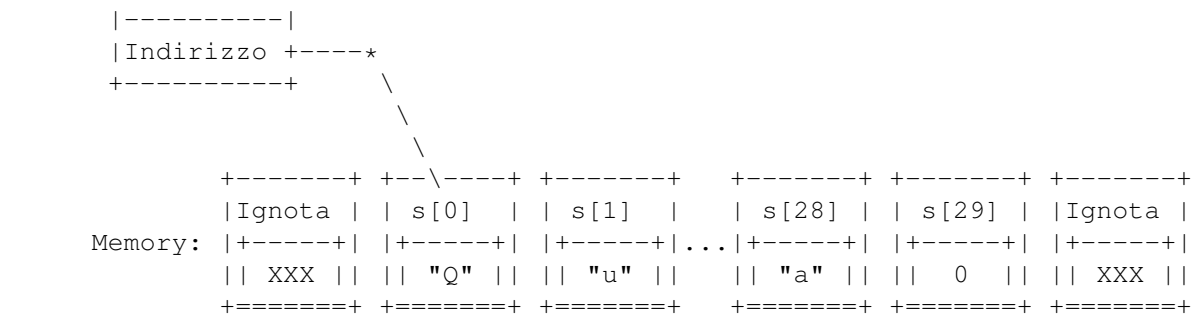
```
DEF s:PTR TO CHAR
s:=’Questa è una costante stringa’
/* Ora s[] è Q e s[2] è e */
```

Ricordiamoci che LONG è in effetti un PTR TO CHAR pertanto il seguente codice è identico al precedente:

```
DEF s
s:=’Questa è una costante stringa’
```

Lo schema seguente illustra la succitata assegnazione ad s. I primi due caratteri s[0] e s[1] sono Q e u, e l’ultimo carattere (prima del segnalatore di fine stringa null, o zero) è a. La memoria marcata come ‘Ignota’ non è parte della costante stringa.

```
+----------+
| Variable |
|    ’s’   |
```



Le stringhe E sono molto simili alle stringhe normali e in effetti possono essere usate ovunque lo possono queste ultime. Però, non è vero il contrario, così se qualcosa richiede una stringa E, al suo posto non possiamo usare una stringa normale. Un allusione alla differenza fra una stringa normale e una stringa E è stata fatta nell'introduzione a questa sezione: le stringhe E possono essere modificate con sicurezza, senza eccedere i loro limiti. Una stringa normale è solo una matrice pertanto dobbiamo prestare attenzione a non eccedere i suoi limiti. Una stringa E sa quali sono i suoi limiti e quindi una qualsiasi funzione di manipolazione delle stringhe la può modificare con sicurezza.

Una variabile di stringa E (tipo STRING) viene dichiarata come nel seguente esempio, con la massima dimensione della stringa E, dichiarata esattamente come si dichiara una matrice.

```
DEF s[30]:STRING
```

Come con una dichiarazione di matrice, la variabile s è in effetti un puntatore ai dati della stringa. Per inizializzare una stringa E, abbiamo bisogno di usare la funzione StrCopy come vedremo fra un po'.

Ci sono degli esempi completi nel Capitolo 3 (Vedi
sez. 3.2
) che mostrano
come usare le normali stringhe e le E-strings.

1.79 beginner.guide/Funzioni stringa

2.4.5.2 Funzioni stringa

Nel linguaggio E c'è un certo numero di utili funzioni BUILT-IN che manipolano le stringhe. Ricordarsi però, che se una stringa E può essere usata ovunque una stringa normale non lo può, una stringa normale non può essere usata dove è richiesta una stringa E. Se un parametro è definito come string allora possiamo passare indifferentemente con quel parametro sia una stringa normale sia una E, ma se esso è definito come un e-string allora solo una stringa E può essere usata. Alcune di queste funzioni hanno degli argomenti di default, e ciò significa che non abbiamo bisogno di specificare nessun parametro per usare valori di default. Vedi
sez. 2.2.3
.
(Ovviamente, possiamo ignorare i valori di defaults e passare sempre

tutti i parametri.)

String(maxsize)
Alloca memoria per una stringa E dalla dimensione massima definita da maxsize e ritorna un puntatore ai dati della stringa. Esso viene usato per fare spazio ad una nuova stringa E, come del resto fa una dichiarazione STRING. I seguenti frammenti di codice sono praticamente equivalenti:

```
DEF s[37]:STRING

DEF s:PTR TO CHAR
s:=String(37)
```

La leggera differenza fra i due codici è questa: quando usiamo la funzione String, potrebbe anche non esserci abbastanza memoria per contenere la striga E. In quel caso il valore speciale NIL (una costante) è ritornato. Il tuo programma deve controllare che il valore ritornato non sia NIL (niente) prima che lo usi come una stringa E (o dereference esso). La memoria per la versione di dichiarazione usata è allocata quando il programma viene eseguito, in questo modo il programma non sará eseguito se non c'è abbastanza memoria. La versione String è spesso chiamata allocazione dinamica in quanto viene fatta solo quando il programma è in esecuzione; l'allocazione per la versione di dichiarazione è fatta dal compilatore E. La memoria allocata usando String viene disallocata usando DisposeLink. Vedi
sez. 2.6.3.2
.

StrCmp(string1,string2,length)
Compara la stringa1 con la stringa2 (possono essere entrambe normali o stringhe E). La funzione ritorna TRUE se i caratteri, stabiliti dal parametro (length), delle due stringhe sono perfettamente identici anche nel maiuscolo minuscolo, altrimenti ritorna FALSE. Il parametro length può anche essere sostituito dalla speciale costante ALL, in tal caso tutti i caratteri delle due stringhe devone essere perfettamente uguali, affinchè la funzione ritorni TRUE. Per esempio, tutti i seguenti paragoni ritornano TRUE:

```
StrCmp('ABC',  'ABC')
StrCmp('ABC',  'ABC',    ALL)
StrCmp('ABCd', 'ABC',    3)
StrCmp('ABCde','ABCxxjs',3)
```

E i seguenti ritornano FALSE: (nota il ma.lo/minuscolo delle lettere)

```
StrCmp('ABC',  'ABc')
StrCmp('ABC',  'ABc', ALL)
StrCmp('ABCd', 'ABC', ALL)
```

StrCopy(e-string,string,length)
Copia il contenuto di string in e-string. Vengono copiati solo il numero di caratteri indicati da length, tuttavia la costante speciale ALL, permette di copiarli tutti (e questo è il valore speciale per length). Ricordarsi che le stringhe E sono manipolate con sicurezza, pertanto il seguente frammento di codice fará sì che in, s, avremo solo i caratteri 'Più di' fin quando la sua dimensione massima (data dalla sua

dichiarazione) sará di sei caratteri:

```
DEF s[6]:STRING
StrCopy(s, 'Più di sei caratteri', ALL)
```

Una dichiarazione usando STRING (o ARRAY) riserva una piccola parte di memoria e conserva un puntatore a tale memoria nella variabile che viene dichiarata. Così per prendere dei dati da questa memoria, dobbiamo prima copiarli usando StrCopy. Se tu hai famigliaritá con linguaggi di alto livello come il BASIC devi stare attento, in quanto potresti pensare di poter assegnare una stringa ad una matrice o una variabile Stringa E. Non dovresti fare quanto segue:

```
/* Non fare cose tipo questa! */
DEF s[80]:STRING
s:='Questa è una costante stringa'
```

Una cosa del genere è piuttosto disastrosa: il puntatore alla memoria riservata per s[80] conservato in, s, va perso e viene sostituito da un puntatore alla costante stringa. Di conseguenza s, non è più una stringa E, e non può essere modificata usando StrLen. Se vogliamo che s possa contenere la suddetta stringa, allora dobbiamo usare StrCopy:

```
DEF s[80]:STRING
StrCopy(s,'Questa è una costante stringa',ALL)
```

In definitiva, bisogna prestare attenzione a non confondersi fra l'uso di un puntatore ai dati e la necessitá di copiare dei dati. Di conseguenza bisogna ricordarsi che un'assegnazione tipo quella vista prima (s:='Questa è ...') non copia grandi matrici di dati, ma solo il puntatore ai dati, quindi se vogliamo conservare dei dati in una variabile di tipo ARRAY o STRING dobbiamo copiarli nella variabile.

StrAdd(e-string,string,length)

Questa funzione è simile a StrCopy, solo che string viene copiato a partire dalla fine di e-string, in definitiva serve ad aggiungere dati e non a sostituirli o ad inserirli ex novo. Il seguente frammento di codice chiarisce la situazione, infatti, s, dopo i seguenti ordini conterrá: Questa è una stringa e mezzo:

```
DEF s[30]:STRING
StrCopy(s, 'Questa è una stringa', ALL)
StrAdd(s,  ' e mezzo',      ALL)
```

StrLen(string)

Ritorna come valore, la lunghezza di string. Tale funzione da per scontato che la stringa termini quando incontra il carattere null (cioè, ASCII zero), e questo è vero per qualsiasi stringa costruita come stringa E o come costante stringa. Tuttavia nulla ci vieta di far sembrare breve una costante stringa inserendo il carattere null (la sequenza speciale \0) in essa. Per esempio, tutte le seguenti chiamate ritornano come valore tre:

```
StrLen('abc')
StrLen('abc\0def')
```

In realtá, la maggior parte delle funzioni di stringa assegnano

automaticamente il carattere null alla fine della stringa, quindi non dobbiamo preoccuparci di inserirlo noi a meno che ciò non sia effettivamente voluto da noi per qualche motivo.

Per le stringhe E StrLen è meno efficiente della funzione EstrLen.

EstrLen(e-string)
Ritorna come valore, la lunghezza di e-string (ricorda che in questo caso può essere usata solo una stringa E). Tale funzione è molto più efficiente di StrLen, in quanto le stringhe E giá sanno la loro lunghezza, quindi la funzione non ha bisogno di cercare nella stringa il carattere null.

StrMax(e-string)
Ritorna come valore, la lunghezza massima di e-string. Questo valore non è necessariamente la lunghezza corrente della stringa E, piuttosto esso è la dimensione usata nella dichiarazione con STRING o la chiamata a String.

RightStr(e-string1,e-string2,length)
E' identica a StrCopy, solo che copia i caratteri (indicati da length) situati all'estrema destra di e-string2 a e-string1, entrambe le stringhe devono essere stringhe E. La costante speciale ALL non può essere usata (per copiare tutta la stringa dobbiamo usare StrCopy, ovviamente). Ad esempio, un valore uno per length, significa che l'ultimo carattere di e-string2 deve essere copiato a e-string1.

MidStr(e-string,string,index,length)
Copia il contenuto di string iniziando da index (che è un indice, proprio come un indice di matrice) a e-string. Il numero di caratteri da copiare è dato da length, può essere usata la costante speciale ALL, se tutti i rimanenti caratteri di string devono essere copiati. Per esempio le seguenti due chiamate a MidStr copiano in, s, la parola tre:

```
DEF s[30]:STRING
MidStr(s, 'Soltanto tre',      9, ALL)
MidStr(s, 'Soltanto tre mele', 9, 3)
```

InStr(string1,string2,startindex)
Ritorna l'indice del primo carattere di string2 in string1 (ossia a che posto si trova tale carattere in string1), iniziando la ricerca in string1 dalla posizione data da startindex. Se string2 non viene trovata allora viene ritornato il valore -1.

TrimStr(string)
Ritorna l'indirizzo (cioè, un puntatore a) del primo carattere in string che non sia uno spazio bianco. Per esempio, il seguente codice fa in modo che s contenga solo 12345:

```
DEF s:PTR TO CHAR
s:=TrimStr('  \n \t   12345')
```

LowerStr(string)
Converte tutte le lettere maiuscole di string in minuscolo. Questo cambiamento è fatto in-place, cioè, i contenuti della stringa sono direttamente interessati. La stringa è ritornata per convenienza.

UpperStr(string)
Converte tutte le lettere minuscole di string in maiuscolo. Questo cambiamento è fatto in-place, cioè, i contenuti della stringa sono direttamente interessati e la stringa è ritornata per convenienza.

SetStr(e-string,length)
Assegna la lunghezza a e-string tramite length. Le stringhe E conoscono la loro dimensione, pertanto se noi modifichiamo una stringa E (senza usare le funzioni di stringa E) e cambiamo la sua dimensione, abbiamo bisogno di assegnare la sua lunghezza usando questa funzione prima di poterla usare nuovamente come stringa E. Per esempio, se abbiamo usato una stringa E come una matrice (possiamo farlo) e abbiamo passato ad essa dei caratteri direttamente, dobbiamo assegnare la sua lunghezza prima di poterla trattare come nient'altro che un array/string:

```
DEF s[10]:STRING
s[0]:="a"    /* Ricordare che "a" è un valore di carattere. */
s[1]:="b"
s[2]:="c"
s[3]:="d"    /* A questo punto s è soltanto una matrice di CHAR. */
SetStr(s, 4) /* Ora, s può essere usata nuovamente come una stringa E.*/
SetStr(s, 2) /* s è un po' più corta, ma è ancora una stringa E.*/
```

Nota che questa funzione può essere usata per accorciare una stringa E (ma non possiamo allungarla in questo modo).

Val(string,address)
Capire quel che fá questa funzione è semplice, ma usarla è un po' complicato. Fondamentalmente, essa converte string in un intero LONG. Spazi bianchi iniziali sono ignorati, un segno iniziale come % o $ evidenzia in string un intero binario o esadecimale (così come fanno per le costanti numeriche). Viene ritornato l'intero decodificato come valore regular (regolare) di ritorno. Vedi
sez. 2.2.4
. Il numero di
caratteri di string che vengono letti per formare l'intero è conservato in address, che normalmente è un indirizzo di variabile (dall'uso {var}), ed è ritornato come il primo valore optional (facoltativo) di ritorno. Se address è la costante speciale NIL (o zero) allora tale numero non è conservato (questo è il valore di default per address). Possiamo usare questo numero per calcolare la posizione in string che non è parte dell'intero nella stringa. Se un intero non è stato decodificato da string, allora viene ritornato il valore zero che viene conservato in address.

Segui i commenti in questo esempio e presta un'attenzione speciale all'uso del puntatore p:

```
DEF s[30]:STRING, value, chars, p:PTR TO CHAR
StrCopy(s, ' \t \n 10 \t $3F -%0101010')
value, chars:=Val('abcde 10 20')  -> Due valori di ritorno...
 /* Dopo la suddetta linea, value and chars saranno entrambi zero */
value:=Val(s, {chars})          -> Usa l'indirizzo di chars
 /* ora value sará 10, chars sará 7 */
p:=s+chars
 /* p ora punta allo spazio dopo il 10 in s */
value, chars:=Val(p)
```

```
       /* Ora value sará $3F (63), chars sará 6 */
      p:=p+chars
       /* p ora punta allo spazio dopo il $3F in s */
      value, chars:=Val(p)
       /* Ora value sará -%0101010 (-42), chars sará 10 */
```

Notare i due differenti modi di trovare il numero di caratteri letti: una multipla assegnazione e usando l'indirizzo di una variabile.

Ci sono un paio di altre funzioni stringa ReadStr e StringF che saranno discusse in seguito. Vedi
sez. 2.6.3.1
.

1.80 beginner.guide/Lists ed E-lists

2.4.5.3 Lists ed E-lists

Le liste sono identiche alle stringhe solo che gli elementi sono LONG anzichè CHAR (di conseguenza sono molto simili ad un ARRAY OF LONG). La lista equivalente di una E-string è chiamata E-list. Essa ha le stesse proprietá di una E-string, solo che gli elementi sono LONG (di conseguenza possono essere dei puntatori). Le liste normali sono più simili alle costanti stringhe, eccetto che gli elementi possono essere delle variabili anzichè delle costanti. Così come le stringhe sono diverse dalle Stringhe E, le liste normali sono diverse dalle E-list.

Le liste si scrivono usando le parentesi quadre [] per delimitare gli elementi che vengono separati da una virgola. Come le costanti stringa, una lista ritorna l'indirizzo della memoria che contiene gli elementi.

Per esempio il seguente frammento di codice:

```
DEF list:PTR TO LONG, numero
numero:=22
list:=[1,2,3,numero]
```

è equivalente a:

```
DEF list[4]:ARRAY OF LONG, numero
numero:=22
list[0]:=1
list[1]:=2
list[2]:=3
list[3]:=numero
```

Ora, quale di queste due versioni preferiresti usare? Come puoi vedere, le liste sono piuttosto utili per rendere il tuo programma più leggibile e veloce da scrivere.

Le variabili E-list sono come le variabili E-string e sono dichiarate nello stesso modo. Il seguente frammento di codice dichiara lt, in modo che possa essere una E-list di massima dimensione 30. Come al solito lt è allora un

puntatore (a LONG) e punta alla memoria allocata dalla dichiarazione.

```
DEF lt[30]:LIST
```

Le liste sono più utili per scrivere tag lists, che sono sempre più usate in importanti funzioni di sistema di Amiga. Una tag list è una lista dove gli elementi sono ideati in coppia. La prima coppia è la tag e la seconda, alcuni dati per quella tag. Vedi il 'Rom Kernel Reference Manual (Libraries)' per maggiori dettagli.

1.81 beginner.guide/Funzioni list

2.4.5.4 Funzioni list

Esistono un certo numero di funzioni list che sono molto simili alle funzioni string Vedi
sez. 2.4.5.2
. Ricorda Che le E-list sono le equivalenti, come liste, delle E-string, cioè possono essere modificate e estese con sicurezza senza eccedere i loro limiti. Come le E-string, le E-list possono sostituire le list, dove esse sono richieste, ma non il contrario. Pertanto se una funzione richiede una lista come parametro, possiamo passargli indifferentemente o una lista o una E-list. Ma se una funzione richiede una E-list, non possiamo passargli una lista al suo posto.

List(maxsize)
Alloca memoria per una E-list di massima dimensione maxsize e ritorna un puntatore ai dati della lista. Esso è usato per fare spazio ad una nuova E-list, come fá una dichiarazione LIST. I seguenti frammenti di codice sono (come con String) praticamente equivalenti:

```
DEF lt[46]:LIST

DEF lt:PTR TO LONG
lt:=List(46)
```

Ricordati che bisogna controllare che il valore di ritorno da List non sia NIL prima di usarla come una E-list. Come con String, la memoria allocata usando List viene disallocata usando DisposeLink.
Vedi
sez. 2.6.3.5
.

ListCmp(list1,list2,length)
Compara list1 con list2 (lavora sia con list che con E-list). Il funzionamento è identico a StrCmp per Le E-string, pertanto i seguenti paragoni ritornano tutti TRUE:

```
ListCmp([1,2,3,4],   [1,2,3,4], ALL)
ListCmp([1,2,3,4],   [1,2,3,7], 3)
ListCmp([1,2,3,4,5], [1,2,3],   3)
```

ListCopy(e-list,list,length)

```
   Funziona come StrCopy, il seguente esempio mostra come inizializzare una
   E-list:

            DEF lt[7]:LIST, x
            x:=4
            ListCopy(lt, [1,2,3,x], ALL)

   Come con StrCopy, una E-list non può essere riempita oltre il suo limite
   usando ListCopy.

ListAdd(e-list,list,length)
   Funziona come StrAdd, pertanto col prossimo frammento di codice la
   E-list, lt, alla fine conterrá [1,2,3,4,5,6,7,8]:

            DEF lt[30]:LIST
            ListCopy(lt, [1,2,3,4], ALL)
            ListAdd(lt, [5,6,7,8], ALL)

ListLen(list)
   Funziona come StrLen, ritorna la lunghezza della lista. Non ha nessuna
   lunghezza specifica una funzione E-list.

ListMax(e-list)
   Funziona come StrMax, ritorna la massima lunghezza della E-list.

SetList(e-list,length)
   Funziona come SetStr, setta la lunghezza della E-list con length.

ListItem(list,index)
   Ritorna l'elemento list a index. Per esempio se lt, è una E-list, allora
   ListItem(lt,n) è equivalente di lt[n]. Questa funzione è più utile
   quando la lista non è una E-list. Per esempio, i seguenti due frammenti
   di codice sono equivalenti:

            WriteF(ListItem(['Fred','Barney','Wilma','Betty'], nome))

            DEF lt:PTR TO LONG
            lt:=['Fred','Barney','Wilma','Betty']
            WriteF(lt[nome])
```

1.82 beginner.guide/Tipi complessi

```
2.4.5.5 Tipi complessi
        --------------
In E i tipi STRING e LIST sono chiamati tipi complex (complessi). Le
variabili complex possono anche essere create usando le funzioni String e
List come abbiamo visto nelle precedenti sezioni.
```

1.83 beginner.guide/Typed lists

2.4.5.6 Typed lists

Le normali liste contengono elementi LONG, così possiamo scrivere matrici inizializzate con elementi LONG. E per quanto riguarda altri tipi di matrice? Bene, abbiamo appunto le typed lists (liste specificate). Dobbiamo specificare il tipo degli elementi di una lista usando i : seguiti dal nome del tipo dopo la chiusura con la]. I tipi consentiti sono CHAR, INT, LONG e qualsiasi tipo di object. Esiste una sottile differenza fra una normale, LONG list e typed list (anche una typed list LONG): solo le liste normali possono essere usate con le funzioni di lista. Vedi
sez. 2.4.5.4
. Per questa ragione il termine ‘list’ tende a fare riferimento solo alle liste normali.

Il seguente frammento di codice usa l’object rec definito in precedenza (Vedi
sez. 2.4.4.1
) e dá un paio di esempi di typed lists:

```
DEF ints:PTR TO INT, objects:PTR TO rec, p:PTR TO CHAR
ints:=[1,2,3,4]:INT
p:='fred'
objects:=[1,2,p,4,
          300,301,'barney',303]:rec
```

Questo frammento è equivalente a:

```
DEF ints[4]:ARRAY OF INT, objects[2]:ARRAY OF rec, p:PTR TO CHAR
ints[0]:=1
ints[1]:=2
ints[2]:=3
ints[3]:=4
p:='fred'
objects[0].tag:=1
objects[0].check:=2
objects[0].table:=p
objects[0].data:=4
objects[1].table:='barney'
objects[1].tag:=300
objects[1].data:=303
objects[1].check:=301
```

L’ultimo gruppo di assegnazioni a object[1] (secondo frammento di codice) sono state deliberatamente mischiate, per mettere in evidenza che l’ordine degli elementi nella definizione dell’object rec è importante. Ciascuno degli elementi della lista corrisponde ad un elemento nell’object e l’ordine degli elementi nella lista corrisponde all’ordine degli elementi nella definizione dell’object. Nell’esempio (primo frammento di codice), la linea di assegnazione della lista (objects) è stata interrotta dopo la fine del primo oggetto (il quarto elemento) per rendere la linea stessa più leggibile. L’ultimo oggetto nella lista può anche non essere completamente definito, pertanto, ad esempio, la seconda linea dell’assegnazione poteva contenere anche tre elementi soltanto. Questo rende una lista object-typed leggermente diversa dall’array di objects, fin tanto che un array definisce sempre un numero intero di objects. Con una lista object-typed dobbiamo

prestare attenzione a non accedere agli elementi non definiti di un object in coda (ultimo nella lista), parzialmente definito.

1.84 beginner.guide/Dati statici

2.4.5.7 Dati statici

Costanti stringa (come fred), liste (come [1,2,3]) e typed lists (come [1,2,3]:INT) sono dati statici. Questo significa che l'indirizzo (inizializzazione) dei dati è stabilito quando il programma viene eseguito. Normalmente non abbiamo bisogno di preoccuparci di questo, ma, ad esempio, se vogliamo avere una serie di liste inizializzate come matrici (arrays), potremmo avere la tentazione di usare qualche tipo di loop:

```
PROC main()
  DEF i, a[10]:ARRAY OF LONG, p:PTR TO LONG
  FOR i:=0 TO 9
    a[i]:=[1, i, i*i]
     /* Questa assegnazione probabilmente non è ciò che vogliamo! */
  ENDFOR
  FOR i:=0 TO 9
    p:=a[i]
    WriteF('a[\d] è una matrice all''indirizzo \d\n', i, p)
    WriteF('  e il secondo elemento è \d\n', p[1])
  ENDFOR
ENDPROC
```

La matrice, a, è una matrice di puntatori alle matrici inizializzate (che sono tutte e tre elementi LONG). Ma, come il commento suggerisce e il programma mostra, questi probabilmente non fa quello per cui è stato concepito, e questo fin quando la lista è statica. Questo significa che l'indirizzo della lista è fisso, pertanto ogni elemento della lista prende lo stesso indirizzo (cioè, la stessa matrice). Fin quando, i, è usato nella lista, i contenuti di quella parte di memoria variano leggermente per come il primo loop FOR è concepito. Ma dopo questo loop i contenuti rimangono fissi e il secondo elemento di ciascuna delle dieci matrici è sempre nove. Segue un esempio dell'output che viene generato dal programma (i ... rappresentano le linee mancanti che sono simili):

```
a[0] è una matrice all'indirizzo 4021144
  e il secondo elemento è 9
a[1] è una matrice all'indirizzo 4021144
  e il secondo elemento è 9
...
a[9] è una matrice all'indirizzo 4021144
  e il secondo elemento è 9
```

Una soluzione è usare l'operatore di allocazione dinamica dei tipi, NEW. Vedi
sez. 2.9.4
. Un'altra soluzione è usare la funzione List e copiare la lista normale nella nuova E-list usando ListCopy:

```
PROC main()
  DEF i, a[10]:ARRAY OF LONG, p:PTR TO LONG
  FOR i:=0 TO 9
    a[i]:=List(3)
    /* Si controlla la riuscita della allocazione prima di copiare */
    IF a[i]<>NIL THEN ListCopy(a[i], [1, i, i*i], ALL)
  ENDFOR
  FOR i:=0 TO 9
    p:=a[i]
    IF p=NIL
      WriteF('Non ho potuto allocare memoria per a[\d]\n', i)
    ELSE
      WriteF('a[\d] a è una matrice all''indirizzo \d\n', i, p)
      WriteF('  e il secondo elemento è \d\n', p[1])
    ENDIF
  ENDFOR
ENDPROC
```

Il problema non è così difficoltoso con le costanti stringa, fin quando i contenuti sono fissi. Tuttavia, se modifichiamo i contenuti esplicitamente, avremo bisogno di prestare attenzione a non cadere nello stesso problema, come il seguente esempio mostra:

```
PROC main()
  DEF i, strings[10]:ARRAY OF LONG, s:PTR TO CHAR
  FOR i:=0 TO 9
    strings[i]:='Ciao Mondo\n'
    /* Questa assegnazione probabilmente non è ciò che vogliamo! */
  ENDFOR
  s:=strings[4]
  s[5]:="X"
  FOR i:=0 TO 9
    WriteF('strings[\d] è ', i)
    WriteF(strings[i])
  ENDFOR
ENDPROC
```

Segue un esempio dell'output che sarà generato (nuovamente i ... rappresentano le linee simili mancanti):

```
strings[0] è CiaoXMondo
strings[1] è CiaoXMondo
...
strings[9] è CiaoXMondo
```

Nuovamente, la soluzione è usare l'allocazione dinamica. Le funzioni String e StrCopy dovrebbero essere usate nello stesso modo in cui abbiamo usato List e ListCopy nell'esempio precedente.

1.85 beginner.guide/Linked Lists

```
            2.4.6 Liste linked
 ============
```

Le E-list e le E-string hanno un'utile estensione: esse possono essere usate per creare liste linked (collegate). Tali liste sono come le liste di cui abbiamo giá parlato, eccetto che gli elementi della lista non occupano un blocco contiguo di memoria. Infatti, ogni elemento ha un pezzo extra di dati: un puntatore al prossimo elemento nella lista. Questo significa che ogni elemento può essere ovunque in memoria. (Normalmente, l'elemento successivo di una lista è nella prossima posizione in memoria). La fine di una linked list è raggiunta quando il puntatore all'elemento successivo è il valore speciale NIL (una costante). Devi prestare molta attenzione nel controllare che il puntatore non sia NIL, in quanto se lo~è, e noi lo dereference (ricordi questa operazione? Vedi
sez. 2.4.2.5
), il programma
sicuramente andrá in crash.

Gli elementi di una linked list sono E-lists o E-strings (cioè, gli elementi sono complex typed). Pertanto possiamo linkare delle E-list ottenendone una lista linkata di liste E (o più semplicemente, una 'lista di liste'). Similmente linkando delle E-string otteniamo una linked list di E-string, o una lista di stringhe. Non possiamo collegare (linkare) questi due tipi di linked list, anche se possiamo usare una miscela di E-lists e E-strings nella stessa linked list. Per linkare un elemento complex typed a un altro dobbiamo usare la funzione Link e per trovare gli elementi successivi in una linked list dobbiamo usare le funzioni Next o Forward.

Link(complex1,complex2)
Linka complex1 a complex2. Entrambi devono essere o una E-list o una E-string, con l'eccezione che complex2 può essere la costante speciale NIL indicando così che complex1 è la fine della linked list. Il valore di complex1 è ritornato dalla funzione, e non sempre è utile in questo modo, normalmente le chiamate a Link saranno usate come dichiarazioni piuttosto che come funzioni. L'effetto di link è che complex1 punterá a complex2 considerato come il successivo elemento nella linked list (in questo modo complex1 è l'inizio della lista e complex2 ne è la fine). Inizialmente, sia per le E-lists che per le E-strings, il puntatore al prossimo elemento è NIL, pertanto avremo solo bisogno di usare Link con un parametro NIL quando vorremo fare una linked list più corta (perdendo la fine).

Next(complex)
Ritorna il puntatore al prossimo elemento nella linked list. Questi può essere la costante speciale NIL se complex è l'ultimo elemento nella linked list. Fare attenzione a controllare che il valore non sia NIL prima che noi dereference il puntatore! Segui i commenti dell'esempio seguente:

```
DEF s[23]:STRING, t[7]:STRING, lt[41]:LIST, lnk
/* Le prossime due linee assegnano la linked list "lnk" */
lnk:=Link(lt,t)  /* lnk list inizia a lt ed è lt->t    */
lnk:=Link(s,lt)  /*   Ora essa inizia a s  ed è s->lt->t */
/* Le prossime tre linee seguono i links in "lnk"  */
lnk:=Next(lnk)   /*   Ora essa inizia a lt ed è lt->t    */
lnk:=Next(lnk)   /*   Ora essa inizia a t ed è t         */
lnk:=Next(lnk)   /* Ora lnk è NIL così la lista è finita  */
```

Possiamo chiamare con sicurezza Next con un parametro NIL, e in questo

caso, ritornerá NIL.

Forward(complex,espressione)
Ritorna il puntatore all'elemento specificato da espressione (numero di collegamenti) nella linked list complex. Se espressione è uguale a uno, viene ritornato un puntatore al prossimo elemento (proprio come se usassimo Next). Se espressione è uguale a due viene ritornato un puntatore all'elemento ancora dopo.

Se espressione rappresenta un numero che è più grande del numero di collegamenti nella lista, viene ritornato il valore speciale NIL.

Fin quando il collegamento in una linked list è un puntatore al prossimo elemento, possiamo visitare la lista solo dall'inizio alla fine. Tecnicamente questa situazione è detta singly linked list (una doubly linked list avrebbe anche un puntatore al precedente elemento nella lista, abilitando la ricerca all'indietro nella lista).

Le linked list sono utili per costruire liste che possono diventare abbastanza grandi. Questo perchè è molto meglio avere parecchi pezzi di memoria che un solo grande pezzo. Tuttavia di queste cose dobbiamo preoccuparci quando abbiamo a che fare con liste abbastanza grandi (per dare un'idea, si intendono grandi le liste con più di 100000 elementi!).

1.86 beginner.guide/Dichiarazioni ed Espressioni più in dettaglio

2.5 Dichiarazioni ed Espressioni più in dettaglio
****

Questa sezione tratta varie dichiarazioni e espressioni E non menzionate nel primo capitolo e completa anche alcune descrizioni parziali.

Trasformare un'Espressione in una Dichiarazione

Dichiarazioni Inizializzate

Assegnazioni

Ancora sulle Espressioni

Ancora sulle Dichiarazioni (Statements)

Unification (Unificazione)

Espressioni Quoted (con virgoletta)

Dichiarazioni Assembly

1.87 beginner.guide/Trasformare un'Espressione in una Dichiarazione

```
            2.5.1 Trasformare un'Espressione in una Dichiarazione
      ============================================
```

L'operatore VOID converte un'espressione in una dichiarazione. VOID valuta l'espressione e ne estrae il risultato, questa operazione può sembrare inutile anche se l'abbiamo giá fatta tante volte senza accorgercene. Non abbiamo usato l'operatore VOID esplicitamente in quanto il linguaggio E quando trova un'espressione dove si aspetta una dichiarazione (normalmente quando essa occupa una linea solo per sè), compie tale operazione automaticamente. Alcune delle espressioni che abbiamo trasformato in dichiarazioni sono state le procedure di chiamata (a WriteF e fred) e l'uso di ++. Ricordati che tutte le procedure chiamate, evidenziano valori, in quanto esse in effetti sono delle vere e proprie funzioni che per default ritornano zero. Vedi
sez. 2.2.1
.

Per esempio, i seguenti frammenti di codice sono equivalenti:

```
VOID WriteF('Ciao Mondo\n')
VOID x++

WriteF('Ciao Mondo\n')
x++
```

Fin tanto che l'E usa VOID automaticamente è inutile scriverlo, sebbene ci potrebbero essere delle occasioni in cui è meglio usarlo, ad esempio se vogliamo rendere più chiaro a chi legge, un determinato passaggio. L'importante, comunque, è che in E, le espressioni possono essere usate tranquillamente come dichiarazioni.

1.88 beginner.guide/Dichiarazioni Inizializzate

```
2.5.2 Dichiarazioni Inizializzate
      ==========================
```

Alcune variabili possono essere inizializzate usando le costanti nelle loro dichiarazioni. Le variabili che non possiamo inizializzare in questo modo sono le matrici e le variabili di tipo complex (e ovviamente i parametri di procedura). Tutti gli altri tipi possono essere inizializzati se sono locali o globali. Una dichiarazione inizializzata è molto simile ad una definizione di costante, il valore segue il nome della variabile e il carattere = li unisce. Il seguente esempio mostra delle dichiarazioni inizializzate:

```
SET INGLESE, FRANCESE, TEDESCO, ITALIANO, RUSSO

CONST FREDSCELTA=INGLESE OR FRANCESE OR TEDESCO

DEF fredparla=FREDSCELTA,
```

```
    p=NIL:PTR TO LONG, q=0:PTR TO rec

PROC fred()
  DEF x=1, y=88
  /* Resto della procedura */
ENDPROC
```

Nota come la costante FREDSCELTA ha bisogno di essere definita per inizializzare la dichiarazione di fredparla a qualcosa di moderatamente complicato. Inoltre, notare la inizializzazione dei puntatori p ed q e la posizione dell'informazione di tipo.

Naturalmente, se vogliamo inizializzare delle variabili con una sola costante, senza farle complicate, possiamo usare le assegnazioni all'inizio del programma. Generalmente dovremmo sempre inizializzare le nostre variabili (usando un metodo o l'altro) in modo che ci garantiscano un valore sensato quando le usiamo. Usando il valore di una variabile che non abbiamo inizializzato, probabilmente ci troveremmo ad affrontare parecchi problemi, semplicemente perchè il valore sará qualcosa di casuale che per qualche motivo si trova nella memoria usata dalla variabile. Ci sono delle regole per come alcuni tipi di variabile vengono inizializzate dall'E (vedi il 'Reference Manual'), ma è sempre meglio inizializzare esplicitamente anche quelle e ciò renderá il nostro programma anche più leggibile.

1.89 beginner.guide/Assegnazioni

2.5.3 Assegnazioni

Abbiamo giá visto alcune assegnazioni o meglio dichiarazioni di assegnazione. Le espressioni di assegnazione sono simili eccetto (come avrai immaginato) che possono essere usate nelle espressioni. Questo perchè esse ritornano il valore sulla destra dell'assegnazione così come eseguendo l'assegnazione. Questo è utile per testare efficacemente che il valore assegnato abbia un senso. Per esempio i seguenti frammenti di codice sono equivalenti, ma il primo usa un'espressione di assegnazione invece di una normale dichiarazione di assegnazione.

```
IF (x:=y*z)=0
  WriteF('Errore: y*z è zero (e x è zero)\n')
ELSE
  WriteF('OK: y*z non è zero (e x è y*z)\n')
ENDIF

x:=y*z
IF x=0
  WriteF('Errore: y*z è zero (e x è zero)\n')
ELSE
  WriteF('OK: y*z non è zero (e x è y*z)\n')
ENDIF
```

Possiamo riconoscere facilmente l'espressione di assegnazione: essa è in parentesi e non da sola su una linea. Notare l'uso delle parentesi per raggruppare l'espressione di assegnazione. Tecnicamente, l'operatore di

assegnazione ha una precedenza molto bassa. Possiamo dire, certo non tecnicamente, che l'operatore di assegnazione assumerá il valore degli elementi di destra di ciò che è contenuto fra parentesi, pertanto abbiamo bisogno di usare le parentesi per fare in modo che x prenda il valore ((y*z)=0) (che sará TRUE o FALSE, cioè -1 o zero).

L'espressione di assegnazione, tuttavia, non permette di avere molti elementi come dichiarazioni di assegnazioni a sinistra dell'operatore. La sola cosa permessa (sulla sinistra) è un nome di variabile, mentre le sintassi permesse di dichiarazione sono:

```
var
var [ espressione ]
var . obj_elemento_nome
^ var
```

(Come con molte ripetizioni di elementi object e/o come i tipi degli elementi degli array indicizzati permettono.) ... Ognuna di queste ... E ognuna di queste può finire con ++ o --. Pertanto, le sintassi seguenti sono tutte assegnazioni valide (le ultime tre usano l'espressione di assegnazione):

```
x:=2
x--:=1
x[a*b]:=macerie
x.mela++:=3
x[22].mela:=y*z
x[].banana.basket[6]:=3+full(9)
x[].pera--:=fred(2,4)

x:=(y:=2)
x[y*z].table[1].arancio:=(IF (y:=z)=2 THEN 77 ELSE 33)
WriteF('x è ora \d\n', x:=1+(y:=(z:=fred(3,5)/2)*8))
```

Forse sei curioso di sapere su cosa ++ o -- agiscono. Bene, è molto semplice: agiscono solo su var, che è la x negli esempi precedenti. Notare che x[].pera-- è uguale a x.pera--, per le stesse ragioni menzionate in precedenza. Vedi

sez. 2.4.4.2
.

1.90 beginner.guide/Ancora sulle Espressioni

```
          2.5.4 Ancora sulle Espressioni
=======================
```

Questa sezione discute gli effetti collaterali, due nuovi operatori in dettaglio BUT e SIZEOF e completa la descrizione degli operatori AND e OR.

Effetti collaterali (Side-effects)

Espressione BUT

Bitwise AND and OR

Espressione SIZEOF

1.91 beginner.guide/Side-effects

2.5.4.1 Effetti collaterali (side-effects)

Se eseguendo un'espressione si causa il cambiamento dei contenuti delle variabili, allora di quell'espressione si dice che ha side-effects. Un'espressione di assegnazione è un semplice esempio di un'espressione con side-effects. Quelle meno ovvie, implicano chiamate a funzione con puntatori alle variabili. Generalmente, espressioni con side-effects dovrebbero essere evitate, a meno che, non sia davvero ovvio quello che fanno. Questo perchè può essere davvero difficile trovare la causa di eventuali problemi del programma se si tratta di errori nascosti in espressioni complicate. D'altra parte, espressioni con side-effects sono concise e spesso molto eleganti. Esse sono anche utili per offuscare il tuo codice (cioè, rendendolo difficile da capire - una forma di protezione dalla copia!).

1.92 beginner.guide/Espressione BUT

2.5.4.2 Espressione BUT

BUT è usato nella sequenza di due espressioni. esp1 BUT esp2 è come dire valuta esp1 ma dammi il valore di esp2. Può non sembrare una cosa utile a prima vista, ma consideriamo i seguenti frammenti di codice:

```
fred((x:=12*3) BUT x+y)
```

```
x:=12*3
fred(x+y)
```

Ovviamente sono equivalenti, con il primo frammento non facciamo altro che dire: assegna il valore 12*3 a x ma chiama la funzione fred con x+y. Ricordarsi che le parentesi che racchiudono l'espressione di assegnazione (nel primo frammento) sono necessarie per le ragioni viste in precedenza. Vedi
sez. 2.5.3
.

1.93 beginner.guide/Bitwise AND and OR

2.5.4.3 Bitwise AND e OR

Come accennato nei precedenti capitoli, gli operatori AND e OR non sono solo dei semplici operatori logici. In realtá, sono entrambi degli operatori intelligenti di bit (bitwise), dove un bit è una cifra binaria (cioè un numero che è zero o uno). Così per capire come AND e OR lavorano, dobbiamo vedere cosa succede agli zero ed agli uno.

x	y	x OR y	x AND y
1	1	1	1
1	0	1	0
0	1	1	0
0	0	0	0

Quando usiamo AND o OR fra due numeri, i bit corrispondenti (cifre binarie) dei numeri sono comparati singolarmente, in maniera conforme alla tavola qui sopra evidenziata. Pertanto se x è %0111010 e y è %1010010 allora x AND y sará %0010010 e x OR y sará %1111010:

```
  %0111010        %0111010
AND             OR
  %1010010        %1010010
   -------         -------
  %0010010        %1111010
```

I numeri (in forma binaria) sono allineati uno sopra l'altro, proprio come nelle normali addizioni (cioè, iniziando l'allineamento con i numeri di destra e forse riempendo gli spazi vuoti sulla sinistra con degli zero). I due bit di ogni colonna sono comparati con AND o OR per avere il risultato sotto la linea finale.

Così, cosa succede con TRUE, FALSE e operazioni logiche? Bene, FALSE è il numero zero, così tutti i bit di FALSE sono zeri, e TRUE è -1, che ha tutti i 32 bit a uno (questi numeri sono LONG, pertanto hanno una quantitá di 32 bit). Quindi usando AND e OR con questi valori avremmo sempre dei numeri che hanno tutti i bit a zero (cioè, FALSE) o tutti i bit a 1 (cioè TRUE). Solo quando usiamo dei numeri composti da zero e uno, possiamo pasticciare un po' con la logica. I numeri diversi da zero, uno e quattro sono (singolarmente) considerati TRUE, ma 4 AND 1 è %100 AND 001 che è zero (cioè FALSE). Quindi quando usiamo AND come operatore logico, non è rigorosamente vero che tutti i numeri diversi da zero rappresentano TRUE. Con OR non abbiamo questo problema, pertanto tutti i numeri diversi da zero sono trattati come TRUE. Esegui questo esempio per capire perchè dovresti stare attento:

```
PROC main()
  test(TRUE,          'TRUE\t\t')
  test(FALSE,         'FALSE\t\t')
  test(1,             '1\t\t')
  test(4,             '4\t\t')
  test(TRUE OR TRUE,  'TRUE OR TRUE\t')
  test(TRUE AND TRUE, 'TRUE AND TRUE\t')
  test(1 OR 4,        '1 OR 4\t\t')
  test(1 AND 4,       '1 AND 4\t\t')
ENDPROC
```

```
PROC test(x, title)
  WriteF(title)
  WriteF(IF x THEN ' è TRUE\n' ELSE ' è FALSE\n')
ENDPROC
```

Ecco l'output che si dovrebbe ottenere:

```
TRUE             è TRUE
FALSE            è FALSE
1                è TRUE
4                è TRUE
TRUE OR TRUE     è TRUE
TRUE AND TRUE    è TRUE
1 OR 4           è TRUE
1 AND 4          è FALSE
```

Quindi AND e OR sono principalmente degli operatori bit-intelligenti e possono essere usati come operatori logici in molte circostanze, con zero che rappresenta falso e tutti gli altri numeri che rappresentano vero. Bisogna prestare attenzione quando usiamo AND con alcuni numeri, in quanto come abbiamo visto il bit-wise AND non sempre dá un numero diverso da zero (o true) come risultato.

1.94 beginner.guide/Espressione SIZEOF

2.5.4.4 Espressione SIZEOF

SIZEOF ritorna la dimensione, in byte, di un oggetto o un tipo BUILT-IN (come LONG). Questo può essere utile per determinare quanta memoria allocare in caso di necessitá. Ad esempio il seguente frammento di codice stampa la dimensione dell'object rec:

```
OBJECT rec
  tag, check
  table[8]:ARRAY
  data:LONG
ENDOBJECT

PROC main()
  WriteF('La dimensione dell''oggetto rec è \d bytes\n', SIZEOF rec)
ENDPROC
```

Possiamo pensare che SIZEOF non serve molto in quanto possiamo calcolare la dimensione di un oggetto semplicemente sommando le dimensioni degli elementi. Generalmente è vero (come lo è per l'oggetto rec), ma dobbiamo stare attenti all'allineamento. E questo significa che i tipi ARRAY, INT, LONG e elementi object devono iniziare ad un indirizzo pari di memoria. Normalmente questo non è un problema, ma se abbiamo un numero dispari di typed elementi CHAR o un ARRAY OF CHAR dimensionata con un numero dispari, allora viene introdotto un byte extra nell'oggetto in modo che l'elemento seguente si trovi allineato correttamente (cioè inizi ad un indirizzo pari anzichè dispari). Questo byte aggiunto (pad byte) può essere considerato parte di un ARRAY OF CHAR, quindi in realtá le dimensioni della matrice

vengono arrotondate al numero pari più vicino. In altre circostanze i pad bytes sarebbero soltanto una parte non usabile dell'oggetto, mentre in questo caso la loro presenza potrebbe farci sbagliare il calcolo per la dimensione dell'oggetto. Prova il seguente programma:

```
OBJECT rec2
  tag, check
  table[7]:ARRAY
  data:LONG
ENDOBJECT

PROC main()
  WriteF('La dimensione dell''oggetto rec2 è \d bytes\n', SIZEOF rec2)
ENDPROC
```

La sola differenza fra gli oggetti rec e rec2 è che la dimensione della matrice in rec2 è 7. Se eseguiamo il programma ci accorgiamo però che la dimensione dell'oggetto non è cambiata, proprio come se avessimo usato una matrice di otto elementi. Se avessi fatto la somma degli elementi di rec2 anzichè usare SIZEOF forse avresti sbagliato.

1.95 beginner.guide/Ancora sulle Dichiarazioni

```
            2.5.5 Ancora sulle Dichiarazioni (Statements)
 ======================================
```

Questa sezione spiega cinque nuove dichiarazioni: INC, DEC, JUMP, EXIT e LOOP, descrive anche l'uso della label (etichetta).

1.96 beginner.guide/Dichiarazioni INC e DEC

```
2.5.5.1 Dichiarazioni INC e DEC
        ----------------------
```

INC x è uguale alla dichiarazione x:=x+1. Comunque, poichè INC non fa un'addizione è un po' più efficiente. Similmente, DEC x è uguale a x:=x-1.

Il lettore attento può pensare che INC e DEC sono gli equivalenti di ++ e --. Ma c'è un'importante differenza: INC x aumenta x sempre di uno, mentre x++ può aumentare x con passi più alti di uno e questo in base al tipo a cui x punta. Ad esempio se x è un puntatore a INT allora x++ aumenterebbe x

di due (INT sono 16 bit, vale a dire 2 byte).

1.97 beginner.guide/Labels e dichiarazione JUMP

2.5.5.2 Labels e dichiarazione JUMP

Una label nomina una posizione in un programma, questi nomi sono globali (possono essere usati in qualsiasi procedura). L'uso più comune della label è con la dichiarazione JUMP, ma possiamo usare le labels per marcare la posizione di alcuni dati. Vedi
sez. 2.5.8
. Per definire una label dobbiamo scrivere un nome seguito dai due punti, immediatamente prima della posizione che vogliamo marcare. Tale posizione deve essere esattamente prima dell'inizio di una dichiarazione cioè sulla linea precedente (solo il nome) o all'inizio della stessa linea.

La dichiarazione JUMP fa continuare l'esecuzione del programma dalla posizione contrassegnata da una label. Questa posizione deve essere nella stessa procedura, ma può trovarsi, ad esempio, fuori da un loop (e JUMP può far terminare quel loop). Per esempio, i seguenti frammenti di codice sono equivalenti:

```
  x:=1
  y:=2
  JUMP rubble
  x:=9999
  y:=1234
rubble:
  z:=88

  x:=1
  y:=2
  z:=88
```

Come possiamo vedere la dichiarazione JUMP ha causato il salto del secondo gruppo di assegnazioni a x e ad y. Un esempio più utile usa JUMP per far terminare un loop:

```
  x:=1
  y:=2
  WHILE x<10
    IF y<10
      WriteF('x è \d, y è \d\n', x, y)
    ELSE
      JUMP fine
    ENDIF
    x:=x+2
    y:=y+2
  ENDWHILE
fine:
  WriteF('Terminato!\n')
```

Questo loop termina se x non è più minore di dieci (il controllo WHILE) o

se y non è più minore di dieci (JUMP nel blocco IF). Questo esempio dovrebbe esserci famigliare, in quanto è praticamente identico a un esempio precedente. Vedi
sez. 1.4.2.2
. Infatti è equivalente a:

```
x:=1
y:=2
WHILE (x<10) AND (y<10)
  WriteF('x è \d, y è \d\n', x, y)
  x:=x+2
  y:=y+2
ENDWHILE
WriteF('Terminato!\n')
```

1.98 beginner.guide/Dichiarazione EXIT

2.5.5.3 Dichiarazione EXIT

Come avrai notato in precedenza, possiamo usare la dichiarazione JUMP con il nome di una label per uscire da un loop prima della sua fine naturale. Tuttavia esiste un modo molto più elegante per i loops WHILE e FOR: la dichiarazione EXIT. Questa dichiarazione ci fará uscire dal più vicino di questi loops (di cui essa fa parte) se l'espressione fornita viene valutata vera (ossia, un valore diverso da zero). Qualsiasi loop che usa EXIT può essere riscritto senza di esso, ma qualche volta a scapito della leggibilitá.

I seguenti frammenti di codice sono equivalenti:

```
FOR x:=1 TO 10
  y:=f(x)
EXIT y=-1
  WriteF('x=\d, f(x)=\d\n', x, y)
ENDFOR

FOR x:=1 TO 10
  y:=f(x)
  IF y=-1 THEN JUMP fine
  WriteF('x=\d, f(x)=\d\n', x, y)
ENDFOR
fine:
```

Questo esempio mostra come il codice sia più leggibile usando EXIT. Possiamo anche riscriverlo usando un loop WHILE, come il seguente, ma tale codice è ancora un po' meno chiaro.

```
going:=TRUE
x:=1
WHILE going AND (x<=10)
  y:=f(x)
  IF y=-1
    going:=FALSE
```

```
    ELSE
      WriteF('x=\d, f(x)=\d\n', x, y)
      INC x
    ENDIF
  ENDWHILE
```

1.99 beginner.guide/Blocco LOOP

```
2.5.5.4 Blocco LOOP
        -----------
```

Un blocco LOOP è una dichiarazione multi-linea. Esso è la sintassi generale dei loops simile al loop WHILE e costruisce un loop senza controllo. Pertanto, normalmente, questo tipo di loop non finirebbe mai. Tuttavia, come ora sappiamo, possiamo terminare un blocco LOOP usando la dichiarazione JUMP. Ad esempio, i seguenti due frammenti di codice sono equivalenti:

```
  x:=0
  LOOP
    IF x<100
      WriteF('x è \d\n', x++)
    ELSE
      JUMP fine
    ENDIF
  ENDLOOP
fine:
  WriteF('Terminato\n')

  x:=0
  WHILE x<100
    WriteF('x è \d\n', x++)
  ENDWHILE
  WriteF('Terminato\n')
```

1.100 beginner.guide/Unification

```
2.5.6 Unification (Unificazione)
      ==========================
```

In E unification è un modo per fare complicate assegnazioni condizionate. Esso può anche essere noto come pattern matching (pattern che si accorda con), in quanto questo è quello che fa: esso si accorda con i patterns e prova ad adattare i valori alle variabili menzionate in quei patterns. Il risultato di una unification è vero o falso in base al successo o meno dell'accordo del pattern.

La forma base di un'espressione unification è:

```
espressione <=> pattern
```

In un pattern possiamo usare solo costanti, nomi di variabile e liste di patterns. (Tecnicamente parlando, lisp-cells sono anche permesse, ma questa variante di unification è oltre la portata di questa Guida.) Il pattern è contrapposto all'espressione nei seguenti modi:

* Se pattern è una costante allora l'accordo ha successo solo se espressione è valutata con lo stesso valore. Pertanto la semplice espressione unification x<=>1 è simile ad un controllo di eguaglianza come x=1.

* Se pattern è un nome di variabile allora l'accordo ha subito successo e la variabile è assegnata con il valore di espressione. Pertanto la semplice espressione di unification 1<=>x è simile ad una assegnazione come x:=1.

* Se pattern è una lista allora espressione è presa come una lista, e ogni elemento di pattern viene preso come un pattern da contrapporre (ricorsivamente) all'elemento corrispondente (all'indice) dell'espressione list. L'accordo riesce solo se la pattern list e l'espressione list hanno la stessa lunghezza e tutti i loro elementi si accordano. (E' un errore serio di programmazione se pattern è una lista, ma espressione non lo è. In questo caso, possono succedere cose strane e il programma può andare in crash.)

Quindi quelle cose in pattern, che controllano se un accordo è valido, sono le costanti e le liste.

Se un accordo è valido, allora a tutte le variabili menzionate in pattern saranno assegnati gli appropriati valori. Tuttavia, se un accordo fallisce dovremmo controllare tutte le variabili implicate nel pattern per trovare quelle con valori non definiti (quindi possiamo aver bisogno di inizializzare di nuovo i loro valori per sicurezza). Questo perchè il vero modo in cui unification è implementato non può seguire le regole suddette nel modo ovvio, ma avrá effetto lo stesso, in caso di successo e interesserá solo le variabili menzoniate nel pattern se l'accordo fallisce.

Per esempio, il seguente programma mostra l'uso di un paio di semplici espressioni di unification:

```
PROC main()
  DEF x, lt
  x:=0
  WriteF('x è \d\n', x)
  lt:=[9,-1,7,4]

  /* La prossima linea usa unification */
  IF lt <=> [9,-1,x,4]
    WriteF('Primo accordo riuscito\n')
    WriteF('1) x è ora \d\n', x)
  ELSE
    WriteF('Primo accordo fallito\n')
    /* Per essere sicuri, resettiamo x */
    x:=0
  ENDIF

  /* La prossima linea usa unification */
  IF lt <=> [1,x,6,4]
```

```
    WriteF('Secondo accordo riuscito\n')
    WriteF('2) x è ora \d\n', x)
  ELSE
    WriteF('Secondo accordo fallito\n')
    /* Per essere sicuri, resettiamo x */
    x:=0
  ENDIF
ENDPROC
```

Il primo accordo in questo esempio riuscirá e vi sará un side-effect per assegnare sette a x. Il secondo accordo non riuscirá in quanto, per esempio, il primo elemento di lt non è uno.

Possiamo riscrivere il suddetto esempio senza usare l'operatore unification (per mostrare quanto unification sia utile). Questo codice segue le regole in un modo particolare, quindi non si garantisce che abbia lo stesso effetto della versione con unification se uno qualsiasi degli accordi fallisce.

```
PROC main()
  DEF x, lt, match
  x:=0
  WriteF('x è \d\n', x)
  lt:=[9,-1,7,4]

  /* Le prossime linee imitano: lt <=> [9,-1,x,4] */
  match:=FALSE
  IF ListLen(lt)=4
    IF ListItem(lt, 0)=9
      IF ListItem(lt, 1)=-1
        x:=ListItem(lt,2)
        IF ListItem(lt, 3)=4
          match:=TRUE
        ENDIF
      ENDIF
    ENDIF
  ENDIF
  IF match
    WriteF('Primo accordo riuscito\n')
    WriteF('1) x è ora \d\n', x)
  ELSE
    WriteF('Primo accordo fallito\n')
    /* Per essere sicuri, resettiamo x */
    x:=0
  ENDIF

  /* Le prossime linee imitano: lt <=> [1,x,6,4] */
  match:=FALSE
  IF ListLen(lt)=4
    IF ListItem(lt, 0)=1
      x:=ListItem(lt, 1)
      IF ListItem(lt, 2)=6
        IF ListItem(lt, 3)=4
          match:=TRUE
        ENDIF
      ENDIF
    ENDIF
```

```
  ENDIF
  IF match
    WriteF('Secondo accordo riuscito\n')
    WriteF('2) x è ora \d\n', x)
  ELSE
    WriteF('Secondo accordo fallito\n')
    /* Per sicurezza, resettiamo x */
    x:=0
  ENDIF
ENDPROC
```

Segue un esempio leggermente più complicato, che mostra come potremmo usare patterns fatto con liste annidate. Ricordarsi che se pattern è una lista, allora anche l'espressione da contrapporre deve essere una lista. In un'altro caso (per esempio, se l'espressione rappresenta NIL) allora il tuo programma potrá funzionare stranamente o anche mandare in crash il computer. Un simile, ma meno disastroso, problema è se capita il contrario: il pattern non è una lista, ma l'espressione da contrapporre lo è. In questo caso il puntatore (alla lista) è contrapposto alla costante pattern o assegnato alla variabile pattern.

```
PROC main()
  DEF x=10, y=-3, p=NIL:PTR TO LONG, lt, i
  WriteF('x è \d, y è \d\n', x, y)
  lt:=[[23,x],y]

  /* Queste fondamentalmente scambiano x ed y */
  IF lt <=> [[23,y],x]
    WriteF('Primo accordo riuscito\n')
    WriteF('1) Ora x è \d, y è \d\n', x, y)
  ELSE
    WriteF('Primo accordo fallito\n')
    /* Per sicurezza, resettiamo x ed y */
    x:=10;  y:=-3
  ENDIF

  /* Queste faranno puntare p alla sub-list di lt */
  IF lt <=> [p,-3]
    WriteF('Secondo accordo riuscito\n')
    WriteF('2) p è ora $\h (un puntatore ad una lista)\n', p)
    FOR i:=0 TO ListLen(p)-1
      WriteF('   L''elemento \d della lista p è \d\n', i, p[i])
    ENDFOR
  ELSE
    WriteF('Primo accordo fallito\n')
    /* Per essere sicuri, resettiamo p */
    p:=NIL
  ENDIF
ENDPROC
```

1.101 beginner.guide/Espressioni Quoted

```
            2.5.7 Espressioni Quoted (con virgoletta)
 ==================================
```

Le espressioni quoted sono una potente caratteristica del linguaggio E, richiedono una conoscenza abbastanza avanzata del linguaggio. In pratica sono tutte quelle espressioni che iniziano con il carattere ` (back-quote), prestare molta attenzione a non confonderlo con il carattere ' che viene invece usato per le stringhe. L'uso di tale virgoletta fa in modo che l'espressione non venga valutata, ma che ne venga invece ritornato l'indirizzo del suo codice. Questo indirizzo può essere poi usato come qualsiasi altro indirizzo, in tal modo possiamo, per esempio, conservarlo in una variabile e passarlo alle procedure. Naturalmente in qualche punto del programma useremo tale indirizzo per far eseguire il codice dell'espressione e ottenerne così il valore.

L'idea delle espressioni quoted è stata presa dal linguaggio di programmazione funzionale Lisp. Anche altre potenti funzioni che combinano liste con espressioni quoted derivano dal Lisp, e questo per avere delle dichiarazioni molto più concise e leggibili.

1.102 beginner.guide/Evaluation

2.5.7.1 Valutazione (Eval)

Quando definiamo un'espressione quoted, ne ricaviamo l'indirizzo del codice che calcola il valore dell'espressione. Affinchè questo calcolo avvenga dobbiamo passare l'indirizzo alla funzione Eval. Adesso sappiamo qualcosa circa il modo di calcolare il valore di un'espressione. (se hai una tastiera Europea o USA potrai ottenere il carattere ` premendo il tasto posto esattamente sotto il tasto ESC. Su una tastiera GB potrai ottenerlo premendo esattamente lo stesso tasto, ma premendo contemporaneamente il tasto ALT.)

```
PROC main()
  DEF adr, x, y
  x:=0; y:=0
  adr:=`1+(fred(x,1)*8)-y
  x:=2; y:=7
  WriteF('Il valore è \d\n', Eval(adr))
  x:=1; y:=100
  WriteF('Ora il valore è \d\n', Eval(adr))
ENDPROC

PROC fred(a,b) RETURN (a+b)*a+20
```

L'output che dovrebbe essere generato è il seguente:

```
Il valore è 202
```

```
Ora il valore è 77
```

Questo esempio ci mostra una espressione abbastanza complicata, essendo quoted. L'indirizzo dell'espressione viene conservato nella variabile adr e l'espressione è valutata usando Eval nelle chiamate a WriteF. I valori delle variabili x ed y sono irrilevanti quando un'espressione è quoted, diventano significanti solo quando si usa Eval. Infatti nell'esempio, grazie a questa caratteristica, abbiamo potuto cambiare i valori di x ed y prima della seconda chiamata a WriteF e ottenere così un risultato differente dalla prima chiamata a WriteF.

Valutare ripetutamente la stessa espressione è l'uso più ovvio delle espressioni quoted. Un altro uso comune si ha quando vogliamo fare una stessa operazione per tante varietá di espressioni differenti. Ad esempio se hai bisogno di calcolare la quantitá di tempo che occorre per calcolare i risultati di varie espressioni, sarebbe meglio usare le espressioni quoted con un codice simile a questo:

```
DEF x,y

PROC main()
  x:=999; y:=173
  time(`x+y,     'Addizione')
  time(`x*y,     'Moltiplicazione')
  time(`fred(x), 'Chiamata alla procedura')
ENDPROC

PROC time(esp, messaggio)
  WriteF(messaggio)
  /* Trova tempo corrente */
  Eval(esp)
  /* Trova nuovo tempo e calcola la differenza, t */
  WriteF(': tempo preso \d\n', t)
ENDPROC
```

Questo programma non è completo, pertanto non perdere tempo nel compilarlo. La versione completa la vedremo in seguito come applicazione pratica in un esempio. Vedi
sez. 3.3
.

1.103 beginner.guide/Espressioni quotable

2.5.7.2 Espressioni quotable

Possiamo usare la virgoletta con qualsiasi tipo di espressione senza restrizione, dobbiamo solo stare attenti alle variabili se sono accessibili (scoping, ossia se sono locali o globali). Se usiamo variabili locali in una espressione quoted, possiamo usare Eval solo all'interno della stessa procedura (in modo che le variabili siano accessibili 'in scope'). Tuttavia, se usiamo solo variabili globali possiamo usare Eval in qualsiasi procedura. Pertanto se ci accingiamo a passare una espressione quoted ad una precedura per realizzare qualcosa con essa dovremmo usare solo

variabili globali.

Un avvertimento: Eval non controlla se l'indirizzo che gli viene passato è realmente un indirizzo di un'espressione. Pertanto puoi confonderti molto se passi ad Eval l'indirizzo di una variabile usando {var}. Devi controllare sempre di aver fatto la cosa giusta ogni volta che usi Eval, in quanto il compilatore E ti permette di scrivere cose come Eval(x+9), dove probabilmente avresti voluto scrivere Eval(`x+9). Il compilatore ti permette cose come x+9, perchè magari vuoi che l'indirizzo da passare sia il risultato di un'espressione complicata. Per questo motivo puoi passare x+9 come parametro!

1.104 beginner.guide/Espressioni lists e quoted

2.5.7.3 Espressioni lists e quoted

In E ci sono molte funzioni BUILT-IN che usano espressioni lists e quoted in modo potente. Queste funzioni sono simili alla programmazione costruita in maniera funzionale, praticamente, permettono di avere un codice molto leggibile che altrimenti richiederebbe algoritmi iterativi (cioè dei loop).

MapList(address,list,e-list,quoted-exp)
Il primo parametro, address, è l'indirizzo di una variabile (ad esempio {x}), il secondo, list, è una lista o una E-list (sorgente), il terzo, e-list, è una variabile E-list (destinazione) e il quarto, quoted-exp, è l'indirizzo di una espressione che usa la variabile indirizzata {x} (per esempio `x+2). L'effetto della funzione è quello di prendere a turno un valore da list, conservarlo in address, valutare quoted-exp e conservare il risultato nella destinazione e-list. Viene ritornata anche la lista risultante (per chiarezza). Esempio:

```
MapList({y}, [1,2,3,a,99,1+c], lt, `y*y)
```

in lt avremo il seguente valore:

```
[1,4,9,a*a,9801,(1+c)*(1+c)]
```

Praticamente è facile notare che ogni valore di list è stato preso a turno (singolarmente), trasferito in {y}, l'espressione quoted `y*y ne ha calcolato il nuovo valore che è stato poi conservato in lt. Il primo elemento in {y} è stato 1 l'espressione quoted ne ha calcolato il nuovo valore 1*1 che è stato poi conservato in lt, il secondo elemento è stato 2, l'espressione 2*2, il risultato 4, e in lt si forma così la nuova lista [1,4 ecc...]

I programmatori esperti direbbero che la funzione MapList ha mappato la nuova lista con (l'espressione quoted) attraverso il (sorgente) list.

ForAll(address,list,quoted-exp)
Lavora in maniera identica a MapList eccetto che la lista risultante non viene conservata. In pratica ForAll ritorna TRUE se ogni elemento della lista risultante è TRUE (cioè diverso da zero) altrimenti ritorna FALSE. In questo modo ForAll decide, attraverso l'espressione quoted, se ogni

elemento della lista sorgente è TRUE. Nell'esempio seguente sono TRUE:

```
ForAll({x}, [1,2,-13,8,0], 'x<10)
ForAll({x}, [1,2,-13,8,0], 'x<=8)
ForAll({x}, [1,2,-13,8,0], 'x>-20)
```

i seguenti sono FALSE:

```
ForAll({x}, [1,2,-13,8,0], 'x OR x)
ForAll({x}, [1,2,-13,8,0], 'x=2)
ForAll({x}, [1,2,-13,8,0], 'x<>2)
```

Exists(address,list,quoted-exp)
Lavora in maniera identica a ForAll, eccetto che ritorna TRUE se l'espressione quoted è TRUE (cioè diverso da zero) per almeno uno degli elementi del sorgente list, altrimenti ritorna FALSE. Nell'esempio seguente sono TRUE:

```
Exists({x}, [1,2,-13,8,0], 'x<10)
Exists({x}, [1,2,-13,8,0], 'x=2)
Exists({x}, [1,2,-13,8,0], 'x>0)
```

i seguenti sono FALSE:

```
Exists({x}, [1,2,-13,8,0], 'x<-20)
Exists({x}, [1,2,-13,8,0], 'x=4)
Exists({x}, [1,2,-13,8,0], 'x>8)
```

SelectList(address,list,e-list,quoted-exp)
Lavora in maniera identica a MapList, eccetto che quoted-exp è usato per decidere quali elementi di list sono da copiare in e-list. Solo gli elementi valutati diversi da zero (cioè true) da quoted-exp, vengono copiati. Viene ritornata anche la lista risultante (per chiarezza). Esempio:

```
SelectList({y}, [99,6,1,2,7,1,1,6,6], lt, 'y>5)
```

in lt avremo il seguente valore:

```
[99,6,7,6,6]
```

1.105 beginner.guide/Assembly Statements

2.5.8 Dichiarazioni Assembly
==========

Il linguaggio E incorpora l'assembler pertanto possiamo usare delle Assembly mnemonics come dichiarazioni E. Possiamo anche scrivere programmi Assembly completi e compilarli usando il compilatore E. In maniera ancora più potente possiamo usare le variabili E come parte del mnemonics, in tal modo possiamo realmente miscelare le dichiarazioni Assembly con le normali dichiarazioni E.

Non possiamo trattare qui la programmazione Assembly, quindi se vuoi usare

questa caratteristica dell'E, dovresti procurarti un buon libro, preferibilmente sull'Assembly dell'Amiga. Ricordarsi che l'Amiga usa il Motorola 68000 come CPU. Amiga più potenti e recenti, usano CPU più avanzate (come il 68020) che ha mnemonics extra. Programmi scritti usando mnemonics solo della CPU 68000 lavoreranno su tutti gli Amiga.

Se non conosci l'Assembler del 68000 dovresti saltare questa sezione e ricordarti semplicemente che le dichiarazioni E che non riconosci sono probabilmente degli Assembly mnemonics.

1.106 beginner.guide/Assembly e Linguaggio E

2.5.8.1 Assembly e Linguaggio E

Possiamo far riferimento alle variabili E semplicemente usandole in un operando. Segui i commenti nel prossimo esempio (i commenti sono solo sulle linee che usano Assembly mnemonics, le altre linee sono solo normali dichiarazioni E):

```
PROC main()
  DEF x
  x:=1
  MOVE.L x,  D0 /* Copia il valore in x al registro D0          */
  ADD.L  D0, D0 /* Raddoppia il valore in D0                    */
  MOVE.L D0, x  /* Copia il nuovo valore in D0 alla variabile x */
  WriteF('x è ora \d\n', x)
ENDPROC
```

Possiamo usare anche le costanti ma devono essere precedute dal simbolo #:

```
CONST MELA=2

PROC main()
  DEF x
  MOVE.L #MELA, D0  /* Copia la costante MELA al registro D0        */
  ADD.L  D0, D0     /* Raddoppia il valore in D0                    */
  MOVE.L D0, x      /* Copia il nuovo valore in D0 alla variabile x */
  WriteF('x è ora \d\n', x)
ENDPROC
```

Le labels e le procedure possono essere usate similmente, ma sono PC-relative pertanto possiamo indirizzarle in questo modo, l'esempio seguente spiega ma non fa niente di utile:

```
PROC main()
  DEF x
```

```
    LEA main(PC), A0 /* Copia l'indirizzo di main al registro A0    */
    MOVE.L A0, x     /* Copia il valore in A0 alla variabile x      */
    WriteF('x è ora \d\n', x)
ENDPROC
```

Possiamo chiamare funzioni di sistema Amiga nello stesso modo che useremmo normalmente in Assembly. Dobbiamo caricare il registro A6 con l'appropriata base di libreria, caricare gli altri registri con i dati appropriati e poi JSR alla routine di sistema. Il prossimo esempio usa la variabile E BUILT-IN intuitionbase e la routine di libreria Intuition DisplayBeep. Quando lo eseguiamo lo schermo lampeggia (o, con Workbench 2.1 e superiori, potresti usare un beep o un suono campionato, in base al settaggio del Workbench).

```
PROC main()
  MOVE.L #NIL, A0
  MOVE.L intuitionbase, A6
  JSR DisplayBeep(A6)
ENDPROC
```

1.107 beginner.guide/Memoria statica

2.5.8.2 Memoria statica

I programmi Assembly riservano memoria statica per cose come le stringhe usando il mnemonics DC. Tuttavia questi non sono veri mnemonics, ma direttive per il compilatore, di conseguenza possono variare da compilatore a compilatore. Le versioni E sono LONG, INT e CHAR (i nomi di tipo), che prendono una lista di valori, riservano l'appropriata quantitá di memoria statica e la riempiono con i valori forniti. La sintassi CHAR permette di usare una lista di caratteri che si possono fornire più facilmente come una stringa. In questo caso la stringa sará allineata automaticamente ad una posizione di memoria pari, sebbene rimaniamo responsabili di dichiararne la fine con il terminatore null. Possiamo anche includere un intero file come dati statici usando INCBIN (il file chiamato usando questa dichiarazione deve giá esistere quando il programma viene compilato). Per usare i dati li marchiamo con una label.

Il prossimo esempio è un po' ricercato, ma illustra alcuni dati statici:

```
PROC main()
  DEF x:PTR TO CHAR
  LEA datatable(PC), A0
  MOVE.L A0, x
  WriteF(x)
ENDPROC

datatable:
  CHAR 'Hello world\n', 0
moredata:
  LONG 1,5,-999,0;    INT -1,222
  INCBIN 'file.data'; CHAR 0,7,-8
```

Usare l'Assembly per avere l'indirizzo della label non è proprio

necessario, pertanto l'esempio avrebbe potuto essere soltanto:

```
PROC main()
  WriteF({datatable})
ENDPROC

datatable:
  CHAR 'Hello world\n', 0
```

1.108 beginner.guide/A cosa stare attenti

2.5.8.3 A cosa stare attenti

Ci sono alcune cose a cui bisogna stare attenti quando si usa l 'Assembly con l'E:

* Tutti i mnemonics e i registri devono essere in maiuscolo, mentre, naturalmente, le variabili E ecc., seguono le normali regole dell'E.

* Molti Assembler standard usano il ; per marcare il resto della linea come un commento. In E usiamo ->, per ottenere lo stesso effetto, oppure i delimitatori /* */.

* I registri A4 e A5 sono usati internamente dall' E pertanto non dovrebbero essere confusi se stiamo miscelando l'E con codice Assembly. Potrebbero essere usati anche altri registri, specialmente se abbiamo usato la keyword REG. Fare riferimento al 'Reference Manual' per maggiori dettagli.

* Le chiamate a funzioni E come WriteF possono interessare i registri. Fare riferimento al 'Reference Manual' per ulteriori dettagli.

1.109 beginner.guide/Costanti, Variabili e Funzioni E BUILT-IN

2.6 Costanti, Variabili e Funzioni E BUILT-IN
****

Questa sezione descrive le costanti, le variabili e le funzioni che sono BUILT-IN (proprie) del linguaggio E. Si possono aggiungerne altre usando i moduli, ma questo è un argomento più avanzato. Vedi
sez. 2.7
.

Costanti Built-In

Variabili Built-In

Funzioni Built-In

1.110 beginner.guide/Costanti Built-In

```
               2.6.1 Costanti Built-In
      ==================
```

Abbiamo giá usato molte costanti BUILT-IN. Questa è la lista completa:

TRUE, FALSE
   Le costanti booleane. Come numeri, TRUE è -1 e FALSE è 0.

NIL
   L'errato valore del puntatore. Molte funzioni producono questo valore per un puntatore se capita un errore. Come numero NIL è 0.

ALL
   Viene usato con le funzioni stringa e list per indicare che tutta la stringa o la lista deve essere usata. Come numero ALL è -1

GADGETSIZE
   Il numero minimo di byte richiesti per contenere tutti i dati per un gadget. Vedi
               sez. 2.6.3.2
               .

OLDFILE, NEWFILE
   Usati con Open per aprire un file giá esistente o uno nuovo. Vedere il il 'AmigaDOS Manual' per maggiori informazioni.

STRLEN
   La lunghezza dell'ultima costante stringa usata. Ricordarsi che una costante stringa è qualcosa fra i caratteri '. Il seguente programma stampa la stringa s e poi la sua lunghezza:

```
PROC main()
  DEF s:PTR TO CHAR, len
  s:='12345678'
  len:=STRLEN
  WriteF(s)
  WriteF('\nè \d caratteri lunga\n', len)
ENDPROC
```

1.111 beginner.guide/Variabili Built-In

```
               2.6.2 Variabili Built-In
      ==================
```

Le seguenti variabili sono BUILT-IN in E, e sono dette variabili di sistema (system variables). Esse sono globali, pertanto possono essere usate da

qualsiasi procedura.

arg
E‘ una stringa che contiene la linea di comando (command line) ossia gli argomenti passati al nostro programma quando questi viene eseguito (dalla Shell o da CLI). Ad esempio se il nome del tuo programma è fred e quando lo mandi in esecuzione vuole questi parametri:

```
fred file.txt "a big file" another
```

allora arg conterrá la stringa:

```
file.txt "a big file" another
```

Se hai AmigaDOS 2.0 (o superiore), puoi usare la routine di sistema ReadArgs per analizzare la linea di comando in un modo molto più versatile. C’è un esempio completo su quest’argomento nel capitolo 3, Analisi degli Argomenti (Argument Parsing). Vedi
sez. 3.4
.

wbmessage
Questa variabile contiene NIL se il tuo programma viene fatto partire dalla Shell/CLI, altrimenti essa contiene un puntatore al messaggio Workbench che contiene informazioni sulle icone selezionate quando il programma viene mandato in esecuzione da Workbench. Pertanto se hai fatto partire il programma da Workbench, wbmessage non sará NIL, conterrá invece gli argomenti Workbench, ma se hai fatto partire il programma dalla Shell/CLI wbmessage sará NIL e gli argomenti saranno in arg (o attraverso ReadArgs). C’è un esempio completo su quest’argomento nel capitolo 3, Analisi degli Argomenti (Argument Parsing).
Vedi
sez. 3.4
.

stdin, stdout, conout
Le variabili stdin e stdout contengono l’input e l’output standard filehandle. Se il tuo programma viene fatto partire dalla Shell/CLI esse saranno filehandles sulla finestra Shell/CLI (e conout sará NIL). Tuttavia, se il tuo programma viene eseguito da Workbench esse saranno entrambe NIL, e in questo caso la prima chiamata a WriteF aprirá una finestra di output CON: e conserverá il file handle per la finestra in stdout e conout. Il file handle conservato in conout, quando il programma ha termine, sará chiuso usando Close, pertanto possiamo assegnare La nostra window CON: o file per l’uso delle funzioni di output e farla chiudere automaticamente. Vedi
sez. 2.6.3.1
.

stdrast
La porta raster usata dalle funzioni grafiche BUILT-IN dell’E come Box e Plot. Questa variabile può essere cambiata in modo che queste funzioni possano disegnare su schermi differenti ecc.
Vedi
sez. 2.6.3.3
.

dosbase, execbase, gfxbase, intuitionbase
Queste variabili sono puntatori all'appropriata base di libreria e sono inizializzate dal codice di startup dell'E, ossia le librerie Dos, Exec, Graphics e Intuition sono aperte dall'E, non hai bisogno di farlo tu. Queste librerie sono anche chiuse automaticamente dall'E, pertanto non devi neanche preoccuparti di questo. Tuttavia, devi esplicitamente aprire e chiudere tutte le altre librerie di sistema di Amiga che vuoi usare. Le altre variabili base di libreria sono definite nel modulo allegato. Vedi
sez. 2.7
. Consultare il 'Reference Manual' per maggiori dettagli circa la variabile base di libreria mathbase.

1.112 beginner.guide/Funzioni Built-In

2.6.3 Funzioni Built-In
==========

Ci sono molte funzioni BUILT-IN nell'E. Abbiamo giá visto parecchie funzioni string e list e abbiamo usato WriteF per stampare. Le rimanenti funzioni sono, generalmente, semplificazioni di funzioni di sistema complesse di Amiga o versioni E, di funzioni di supporto, prese da linguaggi come il C e il Pascal.

Per capire e usare bene le funzioni graphics e intuition dovresti prendere veramente qualcosa come il 'Rom Kernel Reference Manual (Libraries)'. Tuttavia se non vuoi fare niente di troppo complicato basta questo manuale per farti capire qualcosa.

Funzioni di input e output

Funzioni di supporto intuition

Funzioni grafiche

Funzioni matematiche e logiche

Funzioni di supporto system

1.113 beginner.guide/Funzioni di input e output

2.6.3.1 Funzioni di input e output

WriteF(string,param1,param2,...)
Scrive una stringa all'output standard e ritorna il numero di caratteri scritti. Se nella stringa vengono usati caratteri di posizione e proprietá (place-holders) allora bisogna fornire dopo la stringa

l'appropriato numero di parametri nell'ordine in cui dovranno essere stampati come parte della stringa. Finora abbiamo incontrato il \d place-holders per i numeri decimali. La lista completa è:

Place-Holder	Tipo di Parametro	Stampa
\c	Number	Carattere
\d	Number	Numero decimale
\h	Number	Numero esadecimale
\s	String	Stringa

Quindi per stampare una stringa si usa il \s place-holder nella stringa e quindi si fornisce la stringa (per esempio un PTR TO CHAR) come parametro. Provare il seguente programma (ricordarsi che \a stampa un carattere d'apostrofo):

```
PROC main()
  DEF s[30]:STRING
  StrCopy(s, 'Hello world', ALL)
  WriteF('Il terzo elemento di s è "\c"\n', s[2])
  WriteF('o \d (decimale)\n',               s[2])
  WriteF('o \h (esadecimale)\n',            s[2])
  WriteF('e s completa è \a\s\a\n',         s)
ENDPROC
```

Questo è l'output generato:

```
Il terzo elemento di s è "l"
o 108 (decimale)
o 6C (esadecimale)
e s completa è 'Hello world'
```

Possiamo controllare la formattazione del parametro nei campi \d, \h e \s usando un'altra raccolta di sequenze di caratteri speciali prima del place-holder e gli indicatori di dimensione dopo di esso. Se non viene indicata nessuna dimensione, il campo sará grande quanto richiedono i dati. Una dimensione di campo può essere indicata usando [numero] dopo il place-holder. Per le stringhe possiamo usare anche l'indicatore di dimensione (min,max) che specifica le minime e massime dimensioni del campo. Per default i dati sono giustificati a destra nel campo e la parte sinistra del campo viene riempita, se necessario, con degli spazi. Le seguenti sequenze poste prima del place-holder possono fare questi cambiamenti:

Sequenza	Significato
\l	Giustifica a sinistra nel campo
\r	Giustifica a destra nel campo
\z	Assegna e riempe con il carattere "0"

Vediamo come questi controlli di formattazione agiscono nell'esempio:

```
PROC main()
  DEF s[30]:STRING
  StrCopy(s, 'Hello world', ALL)
  WriteF('Il terzo elemento di s è "\c"\n', s[2])
  WriteF('o \d[4] (decimale)\n',            s[2])
```

```
    WriteF('o \z\h[4] (esadecimale)\n',       s[2])
    WriteF('\a\s[5]\a sono i primi cinque elementi di s \n', s)
    WriteF('e s in un campo molto grande  \a\s[20]\a\n',   s)
    WriteF('e s giustificata a sinistra   \a\l\s[20]\a\n', s)
ENDPROC
```

Qui c'è l'output che dovrebbe generare:

```
Il terzo elemento di s è "l"
o  108 (decimale)
o 006C (esadecimale)
'Hello' sono i primi cinque elementi di s
e s in un campo molto grande  '         Hello world'
e s giustificata a sinistra   'Hello world         '
```

WriteF usa l'output standard e questo file handle è conservato nella variabile stdout. Se il tuo programma viene eseguito da Workbench, entrambe le variabili conterranno NIL. In questo caso, la prima chiamata a WriteF aprirá una finestra di output speciale e metterá il file handle nelle variabili stdout e conout, come evidenziato prima.

PrintF(string,param1,param2,...)
Lavora in maniera identica a WriteF, eccetto che usa le più efficienti routine di output bufferizzate, disponibili solo se il tuo Amiga usa la versione di Kickstart 37 o superiore (ossia, AmigaDOS 2.04 e superiori).

StringF(e-string,string,arg1,arg2,...)
Identica a WriteF, eccetto che il risultato viene scritto in una e-string invece di essere stampato. Per esempio, il seguente frammento di codice assegna ad s la stringa: 00123 è un (solo parte della stringa quindi, almeno fin quando la E-string non è abbastanza lunga per contenere l'intera stringa e cioè s[17] anzichè s[10]):

```
DEF s[10]:STRING
StringF(s, '\z\d[5] è un numero', 123)
```

Out(filehandle,char)
Output di un singolo carattere, char, al file o alla finestra di console evidenziata da filehandle e ritorna -1 per indicare la riuscita dell'operazione (quindi qualsiasi altro valore di ritorno sta ad indicare che è capitato un errore). Per esempio, filehandle potrebbe esserre stdout, in tal caso il carattere viene scritto all'output standard. (Hai bisogno di accertarti che stdout non sia NIL, puoi farlo usando una chiamata a WriteF('')).

Inp(filehandle)
Legge e ritorna un singolo carattere da filehandle. Se viene ritornato -1 allora vuol dire che è stata raggiunta la fine del file (EOF) o che è stato trovato un errore.

ReadStr(filehandle,e-string)
Legge un'intera stringa da filehandle e ritorna -1 se viene raggiunto EOF o se capita un errore. I caratteri vengono letti sino ad un linefeed oppure per quanto è lunga la stringa, che non finisce mai prima di un linefeed. Pertanto, la stringa risultante potrebbe anche essere soltanto una riga parziale. Se il risultato ritornato è -1 allora o è stato raggiunto un EOF oppure è stato riscontrato un errore, nell'uno o

nell'altro caso la stringa finora è ancora valida. Pertanto hai ancora bisogno di controllare la stringa, nonostante venga ritornato -1. (Questo si verifica più comunemente con le stringhe che non terminano con un linefeed). La stringa risulterá vuota (cioè di lunghezza zero) se non viene letto nient'altro quando l'errore o EOF viene riscontrato.

Il seguente piccolo programma, legge continuamente dal suo input finchè si verifica un errore o l'utente abbondona digitando quit. Il programma ripete le linee che legge, in maiuscolo. Se viene scritta una linea più lunga di dieci caratteri, il programma leggerá un carattere in più. A causa del modo normale in cui le console windows lavorano, bisogna battere un return prima che una linea venga letta dal programma, (ma questo permette di editare una linea prima che il programma la veda). Se il programma è partito da Workbench allora stdin dovrebbe essere NIL, quindi viene usato WriteF('') per forzare stdout ad essere valido, e in questo caso sará una nuova finestra di console che può essere usata per accettare degli input! (Per far partire un programma compilato dal Workbench, hai bisogno semplicemente di associarlo ad una icona tool. Un modo veloce di fare questo, è di usare una icona tool di un qualsiasi altro programma, dandole ovviamente, lo stesso nome del tuo programma compilato, con l'aggiunta dell'estensione .info.)

```
PROC main()
  DEF s[10]:STRING, fh
  WriteF('')
  fh:=IF stdin THEN stdin ELSE stdout
  WHILE ReadStr(fh, s)<>-1
    UpperStr(s)
  EXIT StrCmp(s, 'QUIT', ALL)
    WriteF('Leggo: \a\s\a\n', s)
  ENDWHILE
  WriteF('Finito\n')
ENDPROC
```

Ci sono degli esempi specifici nel Capitolo 3 (Vedi
sez. 3.2
) che
mostrano anche come usare ReadStr.

FileLength(string)
Ritorna la lunghezza del file nominato in string oppure -1 se il file non esiste o si verifica un errore. Notare che non hai bisogno di aprire il file o avere un filehandle, semplicemente fornisci il nome del file. C'è un esempio completo nel Capitolo 3 (Vedi
sez. 3.2
) che mostra l'uso
di questa funzione.

SetStdIn(filehandle)
Ritorna il valore di stdin prima di settarlo a filehandle. Quindi i seguenti frammenti di codice sono equivalenti:

```
vecchiostdin:=SetStdIn(nuovostdin)

vecchiostdin:=stdin
stdin:=nuovostdin
```

```
SetStdOut(filehandle)
   Ritorna il valore di stdout prima di  settarlo  a  filehandle,  funziona
   come SetStdIn.
```

1.114 beginner.guide/Funzioni di supporto intuition

```
                2.6.3.2 Funzioni di supporto intuition
        ------------------------------
Le funzioni in questa sezione sono versioni  semplificate  di  funzioni  di
sistema  Amiga  (nella  libreria Intuition, come suggerisce il titolo). Per
fare un uso migliore di  tali  funzioni,  probabilmente  avrai  bisogno  di
qualcosa come il 'Rom Kernel Reference Manual (Libraries)', specialmente se
vuoi capire delle cose specifiche dell'Amiga come IDCMP e le porte raster.

Le descrizioni date qui sono in stile leggermente diverso dalle precedenti.
Tutti  i parametri di funzione possono essere espressioni che rappresentano
numeri o indirizzi,  appropriati.  Poichè  molte  funzioni  vogliono  molti
parametri, sono state chiamate in un modo piuttosto descrittivo, così ci si
può riferire ad esse più facilmente.

OpenW(x,y,wid,hgt,idcmp,wflgs,title,scrn,sflgs,gads,tags=NIL)
   Apre e ritorna un puntatore ad una finestra con le proprietá fornite. Se
   per qualche ragione la finestra non può aprirsi viene ritornato NIL.

    x,  y
      La posizione nello schermo dove si aprirá la window.

    wid,  hgt
      La larghezza e l'altezza della window.

    idcmp,  wflgs
      IDCMP e flags specifici della window.

    title
      Il titolo della window (una stringa) che appare sulla barra di titolo
      della window.

    scrn,  sflgs
      Lo schermo su cui la window dovrebbe aprirsi. Se sflgs è 1 la  window
      sará aperta sul Workbench, ignorando scrn (così esso può essere NIL).
      Se sflgs è $F (cioè, 15) la window si aprirá sullo schermo puntato da
      scrn  (che  deve  essere,  quindi,  valido). Vedi OpenS per capire come
      aprire uno schermo personale e ottenere un puntatore allo schermo.

    gads
      Un puntatore ad una gadget list oppure NIL se non vuoi nessun gadget.
      Questi  non  sono  i  gadgets standard di window, in quanto essi sono
      specificati usando i flags di window.  Una  gadget  list  può  essere
      creata usando la funzione Gadget.

    tags
      Una tag-list di altre opzioni, disponibile con la versione  Kickstart
      37  o  superiore. Questo parametro può normalmente essere omesso, fin
```

tanto che di default è NIL. Vedere il 'Rom Kernel Reference Manual (Libraries)' per i dettagli sulle tags disponibili e i loro significati.

Non c'è abbastanza spazio per descrivere tutti gli ottimi dettagli sulle windows e IDCMP (vedere il 'Rom Kernel Reference Manual (Libraries)' per dettagli completi), ma una breve descrizione in termini di flags potrebbe essere utile:

IDCMP Flag	Valore
IDCMP_NEWSIZE	$2
IDCMP_REFRESHWINDOW	$4
IDCMP_MOUSEBUTTONS	$8
IDCMP_MOUSEMOVE	$10
IDCMP_GADGETDOWN	$20
IDCMP_GADGETUP	$40
IDCMP_MENUPICK	$100
IDCMP_CLOSEWINDOW	$200
IDCMP_RAWKEY	$400
IDCMP_DISKINSERTED	$8000
IDCMP_DISKREMOVED	$10000

Segue una tavola di utili flags di window:

Window Flag	Valore
WFLG_SIZEGADGET	$1
WFLG_DRAGBAR	$2
WFLG_DEPTHGADGET	$4
WFLG_CLOSEGADGET	$8
WFLG_SIZEBRIGHT	$10
WFLG_SIZEBBOTTOM	$20
WFLG_SMART_REFRESH	0
WFLG_SIMPLE_REFRESH	$40
WFLG_SUPER_BITMAP	$80
WFLG_BACKDROP	$100
WFLG_REPORTMOUSE	$200
WFLG_GIMMEZEROZERO	$400
WFLG_BORDERLESS	$800
WFLG_ACTIVATE	$1000

Tutti questi flag sono definiti nel modulo intuition/intuition, pertanto se usiamo quel modulo, possiamo usare le costanti, piuttosto che dover scrivere direttamente il valore che è meno descrittivo. Vedi
sez. 2.7
.
Naturalmente, possiamo sempre definire delle nostre costanti per i valori che usiamo.

Useremo OR per usare contemporaneamente dei flags, in modo simile all'uso dei settaggi. Vedi
sez. 2.3.5
. Tuttavia, dovremo fornire solo
flags IDCMP come parte del parametro idcmp e solo flags window come parte del parametro wflgs. Pertanto per avere messaggi IDCMP quando un disco viene inserito e quando il gadget di chiusura viene cliccato,

dobbiamo specificare entrambi i flags IDCMP_DISKINSERTED e IDCMP_CLOSEWINDOW per il parametro idcmp, usando OR fra di essi, oppure usando la somma dei valori delle due costanti (meno leggibile) $8200.

Parte dei flags window richiedono alcuni flags IDCMP per funzionare bene, se in pratica un effetto deve essere completo. Per esempio se vogliamo che la nostra finestra abbia un gadget di chiusura (un gadget standard window) dobbiamo usare WFLG_CLOSEGADGET come uno dei flags window. Se vogliamo che quel gadget funzioni allora abbiamo bisogno di prendere un messaggio IDCMP quando il gadget viene cliccato. Pertanto abbiamo bisogno di usare IDCMP_CLOSEWINDOW come uno dei flags IDCMP. Quindi il gadget per funzionare completamente richiede entrambi i flags (un gadget è abbastanza inutile se non può comunicare quando è stato cliccato). L'esempio completo nel Capitolo 3 illustra come usare questi flags. Vedi

sez. 3.5.1
.

Se vogliamo soltanto l'output di un testo in una window (e forse qualche input da una window), sarebbe meglio usare una console window. Queste forniscono un testo base, in una window, di input e output, e vengono aperte usando la funzione Open della Dos library con l'appropriato CON: nome del file. Vedere 'AmigaDOS Manual' per maggiori dettagli sulle console windows.

CloseW(winptr)

Chiude la finistra puntata da winptr. E' sicuro assegnare NIL a winptr, ma in questo caso, naturalmente, nessuna finestra verrá chiusa! Il puntatore della finestra è normalmente un puntatore ritornato dalla chiamata a OpenW. Devi ricordarti di chiudere qualsiasi window che puoi aver aperto, prima di terminare il tuo programma.

OpenS(wid,hgt,depth,scrnres,title,tags=NIL)

Apre e ritorna un puntatore ad uno schermo personale con le proprietá fornitegli. Se per qualche ragione lo schermo non si apre viene ritornato NIL.

wid, hgt
La larghezza e altezza dello schermo.

depth
La profonditá dello schermo, cioè, il numero di bit-planes. Questo può essere un numero compreso fra 1 e 8 per macchine AGA o fra 1 e 6 per quelle precedenti. Uno schermo con profonditá 3 sará in grado di mostrare 2 elevato 3 (cioè, 8) differenti colori, per ogni colore avremo una differente penna (o registro colore) disponibile. Possiamo assegnare i colori delle penne usando la funzione SetColour.

scrnres
I flags di risoluzione dello schermo.

title
Il titolo dell schermo (una stringa) che appare sulla barra di titolo dello schermo.

tags
Una tag-list di altre opzioni, disponibile con la versione Kickstart

37 o superiore. Questo parametro può normalmente essere omesso, fin tanto che di default è NIL. Vedere il 'Rom Kernel Reference Manual (Libraries)' per i dettagli sulle tags disponibili e i loro significati.

I flags di risoluzione dello schermo controllano il modo dello schermo (screen mode). I seguenti (comuni) valori sono presi dal modulo graphics/view. Vedi
sez. 2.7
. Se vuoi puoi definire le tue costanti per i valori che usi. In un modo o nell'altro è sempre meglio usare delle costanti descrittive, piuttosto che usare direttamente i valori.

Mode Flag	Valore
V_LACE	$4
V_SUPERHIRES	$20
V_PFBA	$40
V_EXTRA_HALFBRITE	$80
V_DUALPF	$400
V_HAM	$800
V_HIRES	$8000

Pertanto per usare uno schermo in alta risoluzione interlacciata devi specificare entrambi i flags V_HIRES e V_LACE usando OR fra di essi oppure usando la somma dei valori delle due costanti (meno leggibile) $8004. C'è un esempio completo che usa questa funzione nel Capitolo 3. Vedi
sez. 3.5.4
.

CloseS(scrnptr)
Chiude lo schermo puntato da scrnptr. E' sicuro assegnare NIL a scrnptr, ma in questo caso, naturalmente, nessuno schermo verrá chiuso! Il puntatore dello schermo è normalmente un puntatore ritornato dalla chiamata a OpenS. Devi ricordarti di chiudere qualsiasi schermo che puoi aver aperto, prima di terminare il tuo programma. Devi chiudere anche tutte le window che hai aperto sul tuo schermo prima che lo possa chiudere.

Gadget(buf,glist,id,flags,x,y,width,text)
Crea un nuovo gadget con le propritá fornite e ritorna un puntatore alla prossima posizione nel buffer (memoria) che può essere usato per un gadget.

buf
Questo è il buffer di memoria, cioè, un chunk (pezzo) di memoria allocata. Il miglior modo di allocare questa memoria è dichiarare un array di dimensione n*GADGETSIZE, dove n è il numero di gadget che stanno per essere creati. La prima chiamata a Gadeget userá l'array come buffer, le chiamate successive useranno il risultato della precedente chiamata come buffer (fin quando questa funzione ritorna la successiva posizione libera nel buffer).

glist
Questo è un puntatore alla gadget list che verrá creata, cioè, l'array usato come buffer. Quando crei il primo gadget nella lista

usando un array a, questo parametro dovrebbe essere NIL. Per tutti gli altri gadget nella lista questo parametro dovrebbe essere l'array a.

id
Un numero che identifica il gadget. E' meglio dare un numero unico per ogni gadget, in modo da identificarli più facilmente. Questo numero è il solo modo che hai per identificare quale gadget è stato cliccato.

flags
Il tipo di gadget che verrá creato. Zero rappresenta un normale gadget, uno un gadget booleano (un commutatore) e tre un booleano che esce giá selezionato.

x, y
La posizione del gadget, tenendo presente l'angolo superiore sinistro della window.

width
La larghezza del gadget (in pixel, non in caratteri).

text
Il testo (una stringa) che sará centrata nel gadget, quindi width deve essere abbastanza largha da contenerlo.

Una volta che una lista di gadget è stata creata, possibilmente con molte chiamate a questa funzione, la lista può essere passata come parametro gads di OpenW. Nel Capitolo 3 c'è un esempio completo che usa questa funzione. Vedi sez. 3.5.1.

Mouse()
Ritorna lo stato del mouse (includendo il tasto centrale se hai un mouse a tre tasti). Questa è una serie di flag, i valori individuali dei flag sono:

Bottone Premuto	Valore
Left	%001
Middle	%010
Right	%100

Così se questa funzione ritorna %001 sappiamo che il tasto di sinistra viene premuto e se ritorna %110 sappiamo che il tasto centrale e quello di destra sono entrambi premuti.

MouseX(winptr)
Ritorna la coordinata x del puntatore del mouse, relativa alla finestra puntata da winptr.

MouseY(winptr)
Ritorna la coordinata y del puntatore del mouse, relativa alla finestra puntata da winptr.

Le tre funzioni mouse non sono rigorosamente precise per programmare il mouse. Si suggerisce di usare queste funzioni solo per piccoli test o programmi demo. Il modo preciso per avere informazioni mouse è usare gli

appropriati flags IDCMP per la tua finestra, attendere gli avvenimenti e decodificare l'informazione.

LeftMouse(winptr)
Ritorna TRUE se il tasto sinistro del mouse è stato cliccato nella finestra puntata dalla winptr, altrimenti ritorna FALSE. Affinchè questa funzione possa lavorare sensatamente, la window deve avere il flag IDCMP settato con IDCMP_MOUSEBUTTONS (Vedi sezioni precedenti).

Questa funzione si comporta in un modo più vicino ad Intuition, agendo correttamente, quindi è una buona alternativa alla funzione Mouse.

WaitIMessage(winptr)
Questa funzione attende un messaggio da Intuition per la window puntata da winptr e ritorna la classe del messaggio (che è un flag IDCMP). Se non hai specificato nessun flag IDCMP all'apertura della window o i messaggi specificati non hanno potuto verificarsi (per esempio se hai specificato solo messaggi gadget e non c'è nessun gadget), allora questa funzione può attendere all'infinito. Quando hai un messaggio puoi usare le funzioni MsgXXX per avere più informazioni sul messaggio. Vedere il 'Rom Kernel Reference Manual (Libraries)' per maggiori informazioni su Intuition e IDCMP. C'è un esempio completo che usa questa funzione nel Capitolo 3. Vedi
sez. 3.5.2
.

Questa funzione è fondamentalmente equivalente alla seguente funzione, eccetto che le funzioni MsgXXX possono accedere anche ai dati di messaggio contenuti nelle variabili code, qual e iaddr.

```
PROC waitimessage(win:PTR TO window)
  DEF port,msg:PTR TO intuimessage,class,code,qual,iaddr
  port:=win.userport
  IF (msg:=GetMsg(port))=NIL
    REPEAT
      WaitPort(port)
    UNTIL (msg:=GetMsg(port))<>NIL
  ENDIF
  class:=msg.class
  code:=msg.code
  qual:=msg.qualifier
  iaddr:=msg.iaddress
  ReplyMsg(msg)
ENDPROC class
```

MsgCode()
Ritorna code, parte del messaggio ritornato da WaitIMessage.

MsgIaddr()
Ritorna iaddr, parte del messaggio ritornato da WaitIMessage. C'è un esempio completo che usa questa funzione nel Capitolo 3. Vedi
sez. 3.5.2
MsgQualifier()
Ritorna qual, parte del messaggio ritornato da WaitIMessage.

WaitLeftMouse(winptr)
Questa funzione aspetta che venga premuto il tasto sinistro del mouse

nella finestra puntata dalla winptr. E' consigliabile che la window abbia il flag IDCMP settato con IDCMP_MOUSEBUTTONS. (Vedi sezioni precedenti).

Questa funzione si comporta in un modo più vicino ad Intuition, agendo correttamente, quindi è una buona alternativa alla funzione Mouse.

1.115 beginner.guide/Funzioni Grafiche

2.6.3.3 Funzioni grafiche

Le funzioni di questa sezione usano la porta raster standard, l'indirizzo di questa è contenuto nella variabile stdrast. Non dobbiamo preoccuparci molto di questo in quanto le funzioni E che aprono finestre e schermi assegnano questa variabile. Vedi
sez. 2.6.3.2
. Pertanto, per default, queste funzioni interessano l'ultima finestra o schermo aperto. Quando chiudiamo una finestra o uno schermo, stdrast diventa NIL e le chiamate a queste funzioni non hanno nessun effetto. C'è un esempio completo nel Capitolo 3 che usa queste funzioni. Vedi
sez. 3.5.3
.

Le descrizioni in questa sezione seguono lo stesso stile della precedente.

Plot(x,y,pen=1)
Disegna un singolo punto alle coordinate (x,y) nel colore specificato da pen. La posizione si calcola tenendo presente l'angolo superiore sinistro della window o dello schermo dato dalla porta raster corrente (normalmente l'ultimo schermo o window che è stata aperta). La gamma di valori di pen disponibile dipende dal settaggio dello schermo, ma nel migliore dei casi va da 0 a 255 su macchine AGA e da 0 a 31 sulle pre-AGA. Il colore di sfondo (background), normalmente è pen zero, e il colore principale di primo-piano (foreground) è pen uno (e questa è la penna di default). Possiamo assegnare i colori delle penne usando la funzione SetColour.

Line(x1,y1,x2,y2,pen=1)
Disegna una linea da (x1,y1) a (x2,y2) nel colore specificato da pen.

Box(x1,y1,x2,y2,pen=1)
Disegna un box (pieno) con vertici (x1,y1),(x2,y1),(x1,y2),(x2,y2) nel colore specificato da pen.

Colour(fore-pen,back-pen=0)
Assegna i colori di primo piano e di sfondo. Come accennato in precedenza, il colore di sfondo normalmente è zero e quello di primo piano è 1. Con questa funzione possiamo cambiare i colori di default e se la penna di sfondo ci sta bene a zero allora possiamo chiamare questa funzione con un solo parametro, cambiando così, solo la penna di primo piano.

```
TextF(x,y,format-string,arg1,arg2,...)
   Funziona come WriteF eccetto che la  stringa  risultante  viene  scritta
   alle  coordinate  (x,y)  e non bisogna usare nessun carattere line-feed,
   carriage return, tab o escape nella stringa, questi non  funzionerebbero
   come in WriteF.

SetColour(scrnptr,pen,r,g,b)
   Assegna il colore al registro colore pen, per lo schermo  puntato  dalla
   scrnptr, con l'appropriato valore RGB (cioè, r=red, g=green, b=blue). Il
   parametro pen non può superare il  valore  di  255,  in  base  al  depth
   (profonditá)  dello  schermo.  Non  tenendo conto del chip-set che viene
   usato, r,g, ed b sono compresi in un range da 0 a 255, quindi  i  colori
   sono  sempre specificati a 24-bit. In fase operativa, tuttavia, i colori
   sono scalati a colori a 12-bit, per macchine non AGA.

SetStdRast(newrast)
   Ritorna il valore di stdrast prima di settarlo con un  nuovo  valore.  I
   seguenti frammenti di codice sono equivalenti:

            vecchiostdrast:=SetStdRast(nuovostdrast)

            vecchiostdrast:=stdrast
            stdrast:=nuovostdrast

SetTopaz(size=8)
   Assegna il font di testo per la porta raster corrente  a  Topaz  con  la
   dimensione specificata da size che di default è alla dimensione standard
   otto.
```

1.116 beginner.guide/Funzioni matematiche e logiche

```
                2.6.3.4 Funzioni matematiche e logiche
        ------------------------------
Abbiamo giá trattato gli operatori aritmetici standard.  Gli  operatori  di
addizione,  +, e di sottrazione, -, usano un intero pieno a 32-bit, ma, per
una  migliore  efficienza,  gli  operatori  di  moltiplicazione,  *,  e  di
divisione, /, usano valori limitati. Possiamo usare solo * per moltiplicare
interi a 16-bit, il risultato sará un intero a 32-bit. Similmente, possiamo
usare  solo  /  per  dividere un intero a 32-bit con un intero a 16-bit, il
risultato sará un intero a 16-bit. Queste restrizioni non hanno effetto sui
calcoli, ma se abbiamo davvero bisogno di usarli tutti come interi a 32-bit
(e possiamo affrontare overflows ecc.), possiamo usare le  funzioni  Mul  e
Div. Mul(a,b) corrisponde ad a*b e Div(a,b) corrisponde ad a/b.

Abbiamo giá trattato anche gli operatori logici  AND  e  OR,  che  sappiamo
essere  davvero  degli operatori bit-wise. Possiamo anche usare le funzioni
And e Or per fare esattamente le stesse cose che possiamo fare con AND e OR
(rispettivamente).  Quindi, per esempio, And(a,b) è la stessa cosa di a AND
b. La ragione per cui esistono queste funzioni  è  che  ci  sono  anche  le
funzioni  bit-wise Not e Eor (ma non gli operatori NOT e EOR corrispondenti
a queste funzioni). Not(a) scambia il bit uno con il bit zero e  viceversa,
così,  per  esempio  Not(TRUE)  è FALSE e Not(FALSE) è TRUE. Eor(a,b) è una
versione di OR esclusivo e fa quasi la stessa cosa, tranne che  Eor(1,1)  è
```

uguale a zero, mentre Or(1,1) è uguale a 1 (e questo si estende a tutti i bit). Quindi, fondamentalmente, Eor ci dice quali bit sono differenti o logicamente, se i valori di veritá sono differenti. Pertanto Eor(TRUE,TRUE) è FALSE e Eor(TRUE,FALSE) è TRUE.

Abbiamo una raccolta di altre funzioni attinenti la matematica, la logica o i numeri in generale:

Abs(x)
Ritorna il valore assoluto dell'espressione x. Il valore assoluto di un numero è un numero reso positivo, quando necessario. Quindi, Abs(9) è 9 e Abs(-9) è sempre 9.

Sign(x)
Ritorna il segno di x, che è 1 se x è (rigorosamente) positiva, -1 se x è (rigorosamente) negativa, e zero se x è zero.

Even(x)
Ritorna TRUE se l'espressione x rappresenta un numero pari, altrimenti ritorna FALSE.

Odd(x)
Ritorna TRUE se l'espressione x rappresenta un numero dispari, altrimenti ritorna FALSE.

Max(exp1, exp2)
Ritorna il massimo di exp1 e exp2

Min(exp1, exp2)
Ritorna il minimo di exp1 e exp2

Bounds(exp, minexp, maxexp)
Ritorna il valore di exp limitato ai limiti dati da minexp (minimo bound) e maxexp (massimo bound). Ossia, se exp è compreso nei limiti allora exp viene ritornato, ma se exp è minore di minexp allora viene ritornato minexp oppure se exp è maggiore di maxexp allora viene ritornato maxexp. Questa funzione è utile per dichiarare un valore calcolato in maniera limitata, in modo che sia un valido (intero) percentuale (cioè, un valore tra zero e cento).

I seguenti frammenti di codice sono equivalenti:

```
y:=Bounds(x, min, max)

y:=IF x<min THEN min ELSE IF x>max THEN max ELSE x
```

Mod(exp1,exp2)
Ritorna il resto (o modulus) a 16-bit della divisione fra il 32-bit exp1 e il 16-bit exp2 come il regular (regolare) valore di ritorno (Vedi

sez. 2.2.4

) e il risultato a 16-bit della divisione come il primo valore di ritorno facoltativo. Per esempio, la prima assegnazione nel seguente codice assegna a con il valore di 5 (poichè 26=(7*3)+5), b con il valore di 3, c con il valore di -5 e d con il valore di -3. E' importante notare che se exp1 è negativo allora anche il resto sará negativo. Questo a causa delle regole della divisione fra interi: esse,

semplicemente, scartano parti frazionarie piuttosto che arrotondarle.

```
a,b:=Mod(26,7)
c,d:=Mod(-26,7)
```

Rnd(x)
Ritorna un numero casuale compreso tra 0 e (n-1), dove x rappresenta il valore n. Questi numeri sono pseudo-casuali, quindi sebbene ci sembra di ottenere un valore casuale ad ogni chiamata, in realtá la sequenza di numeri che otterremo, probabilmente sará sempre la stessa ogni volta che il programma verrá eseguito. Quindi prima di usare Rnd per la prima volta nel programma, dovremmo chiamarlo con un numero negativo, in modo che venga deciso il punto di partenza dei numeri pseudo-casuali.

RndQ(x)
Ritorna un valore casuale a 32-bit, basato sull'espressione x. Questa funzione è più veloce di Rnd, ma ritorna valori compresi nella gamma a 32-bit, non una gamma specificata. Il valore di x è usato per selezionare differenti sequenze di numeri pseudo-casuali, e la prima chiamata a RndQ dovrebbe usare un valore molto grande per x.

Shl(exp1,exp2)
Ritorna il valore rappresentato da exp1 spostato di exp2 bit a sinistra. Per esempio, Shl(%0001110,2) dá %0111000 e Shl(%0001011,3) dá %1011000. Generalmente spostare un numero di un bit a sinistra equivale a moltiplicarlo per 2 (sebbene questo non è vero quando spostiamo grandi valori positivi o negativi). (I nuovi bit spostati a destra sono sempre zeri.)

Shr(exp1,exp2)
Ritorna il valore rappresentato da exp1 spostato di exp2 bit a destra. Per esempio, Shr(%0001110,2) dá %0000011 e Shr(%1011010,3) dá %0001011. Generalmente spostare un numero di un bit a destra equivale a dividerlo per 2. (I nuovi bit spostati a sinistra sono sempre zeri.)

Long(addr), Int(addr), Char(addr)
Ritornano il valore LONG, INT o CHAR all'indirizzo addr. Queste funzioni dovrebbero essere usate solo nel momento in cui si sta settando un puntatore e dereferencing esso, nel modo normale il programma sarebbe più confuso e meno leggibile. L'uso di questo tipo di funzioni è spesso detto peeking memory (specialmente in dialetti del linguaggio BASIC).

PutLong(addr,exp), PutInt(addr,exp), PutChar(addr,exp)
Scrive il valore LONG, INT o CHAR rappresentato da exp, all'indirizzo addr. Nuovamente queste funzioni dovrebbero essere usate solo quando è davvero necessario. L'uso di questo tipo di funzioni è spesso detto poking memory.

1.117 beginner.guide/Funzioni di supporto system

2.6.3.5 Funzioni di supporto system

New(byte)

Ritorna un puntatore ad un chunk (pezzo) di memoria che viene nuovamente allocata, l'espressione byte rappresenta il numero di byte. Se la memoria non può essere allocata viene ritornato NIL. Ogni byte di memoria viene inizializzato a zero e viene presa da qualsiasi parte essa è disponibile (Fast o Chip, in quest'ordine di preferenza). Quando abbiamo finito di utilizzare questa memoria, possiamo usare Dispose per liberarla e utilizzarla altrove nel nostro programma. Non abbiamo bisogno di utilizzare Dispose con memoria allocata da New, in quanto il programma libererá automaticamente questa memoria al termine della sua esecuzione. Questo non avviene per la memoria allocata usando le normali routine di sistema di Amiga.

NewR(bytes)
Identica a New eccetto che se la memoria non può essere allocata, allora l'exception (eccezione) "MEM" viene ottenuta (e quindi, in questo caso, la funzione non ritorna). Vedi
sez. 2.8
.

NewM(bytes,type)
Identica a NewR eccetto che il type (tipo) di memoria (Fast or Chip) da essere allocata può essere specificata usando i flags. I flags sono definiti nel modulo exec/memory. Vedi
sez. 2.7.2
. Vedi il 'Rom Kernel
Reference Manual (Libraries)' per i dettagli sulla funzione AllocMem che usa tali flags nello stesso modo. Come utile esempio, segue un piccolo programma che alloca un po' di memoria chip pulita (cioè azzerata).

```
MODULE 'exec/memory'

PROC main()
  DEF m
  m:=NewM(20, MEMF_CHIP OR MEMF_CLEAR)
  WriteF('Allocazione riuscita, m = $\h\n', m)
EXCEPT
  IF exception="NEW" THEN WriteF('Fallita\n')
ENDPROC
```

Dispose(address)
Usato per liberare la memoria allocata con New, dovremmo aver bisogno raramente di usare questa funzione, poichè la memoria è liberata automaticamente quando il programma termina.

DisposeLink(complex)
Usato per liberare la memoria allocata da String (Vedi
sez. 2.4.5.2
) e
List. Vedi
sez. 2.4.5.4
. Nuovamente, dovremmo aver bisogno di usare raramente questa funzione, in quanto la memoria è automaticamente liberata quando il programma termina.

FastNew(bytes)
Identica a NewR eccetto che essa usa un velocissimo metodo riciclabile di allocare la memoria. La memoria allocata con FastNew è, come al

solito, disallocata automaticamente alla fine del programma e può essere disallocata allora, prima di usare FastDispose. Nota, che può essere usato solo FastDispose e che questi differisce leggermente dalle funzioni Dispose e DisposeLink (dobbiamo specificare lo stesso numero di bytes usati al momento dell'allocazione, quando disallochiamo).

FastDispose(address,bytes)
Usato per liberare la memoria allocata da FastNew. Il parametro byte deve corrispondere a quello usato con FastNew, ma il vantaggio è una allocazione e disallocazione molto più veloce e generalmente un uso più efficiente della memoria.

CleanUp(x=0)
Termina il programma in questo punto e fá le normali cose che un programma E fá quando termina. Il valore evidenziato dall'espressione x è ritornato come il codice di errore per il programma. Questa funzione sostituisce la routine AmigaDOS Exit, che non dovrebbe mai essere usata in un programma E. Usare questa funzione è l'unico modo sicuro per terminare un programma, oltre quello di raggiungere la fine (logica) della procedura main (che è di gran lunga il modo più comune!).

CtrlC()
Ritorna TRUE se control-C è stato premuto dall'ultima chiamata, altrimenti ritorna FALSE. Questa funzione è utile solo per programmi che vengono fatti partire dalla Shell/CLI.

FreeStack()
Ritorna la quantitá corrente di spazio stack libero per il programma. Solo per programmi complicati abbiamo bisogno di preoccuparci di cose come lo stack. Le recursioni sono la causa principale dell'occupazione di parecchio spazio stack.

KickVersion(x)
Ritorna TRUE se la tua revisione di Kickstart è almeno uguale al numero dato dall'espressione x, altrimenti ritorna FALSE. Per esempio KickVersion(37) se stiamo usando la versione 37 o superiore (cioè, AmigaDOS 2.04 o superiore).

1.118 beginner.guide/Moduli

2.7 Moduli

******

Un modulo è l'equivalente in E di un file header in C e di un file include in Assembly. Esso può contenere varie definizioni di oggetti e costanti, e anche offsets di funzioni di libreria e variabili base di libreria. Questa informazione è necessaria per il corretto uso di una libreria.

Uso dei Moduli

Moduli di Sistema Amiga

1.119 beginner.guide/Uso dei Moduli

2.7.1 Uso dei Moduli
==========

Per usare le definizioni di un particolare modulo, dobbiamo usare la dichiarazione MODULE all'inizio del programma (prima della definizione della prima procedura). Dopo la keyword MODULE segue una lista di stringhe separate da una virgola, ognuna delle quali è il nome del file (o il percorso se necessario) di un modulo senza l'estensione .m (ogni nome di un modulo finisce con .m). I nomi dei file (e i percorsi o paths) sono relativi al volume logico Emodules:, che viene usato usando un assign come descritto nel 'Reference Manual', a meno che il primo carattere della stringa è *. In questo caso i file sono relativi alla directory attuale del file sorgente. Per esempio la dichiarazione:

```
MODULE 'fred', 'dir/barney', '*mymod'
```

proverá a caricare i file Emodules:fred.m, Emodules:dir/barney.m e ?mymod.m. Se questi file non vengono trovati, oppure non sono dei moduli corretti, il compilatore dará dei messaggi di errore.

Tutte le definizioni presenti nei moduli inclusi in questo modo nel programma, sono disponibili ad ogni procedura nel programma. Per vedere cosa contiene un modulo, possiamo usare il programma showmodule che viene fornito con il pacchetto dell'Amiga E.

1.120 beginner.guide/Moduli di Sistema Amiga

2.7.2 Moduli di Sistema Amiga
==========

Amiga E per usare l'Amiga standard system fornisce dei file come i moduli E. I moduli AmigaDOS 2.04 sono forniti con la versione E 2.1, mentre i moduli AmigaDOS 3.0 sono forniti con la versione E 3.0. Comunque i moduli della versione 3.0 sono molto più utili. Vedi
sez. 2.7.5
. Se vogliamo
usare correttamente una qualsiasi delle librerie standard di Amiga avremo bisogno di sapere quali sono i moduli per quella libreria. I file .m nella directory principale Emodules:, contengono gli offset delle funzioni di libreria, mentre quelli nelle sotto directory di Emodules, contengono le costanti e le definizioni di object per l'appropriata libreria. Per

esempio, il modulo asl (cioè il file Emodules:asl.m) contiene gli offset delle funzioni di libreria ASL e libraries/asl (cioè il file Emodules:libraries/asl.m) contiene le costanti e gli object della libreria ASL.

Se stiamo per usare la ASL library, allora dobbiamo prima aprirla usando la funzione OpenLibrary (una funzione di sistema Amiga) altrimenti non possiamo usare nessuna delle funzioni della libreria. Dobbiamo anche definire gli offsets delle funzioni di libreria usando la dichiarazione MODULE. Comunque le librerie DOS, Exec, Graphics e Intuition non hanno bisogno di essere aperte e i loro offsets sono BUILT-IN in E. Ecco perchè non troveremo, per esempio, un file DOS.m in Emodules:. Le costanti e gli oggetti per queste librerie invece, devono essere inclusi attraverso i moduli (non sono BUILT-IN nell'E).

1.121 beginner.guide/Moduli Non-Standard

2.7.3 Moduli Non-Standard
==========

Con l'Amiga E sono forniti anche molti moduli di libreria non-standard. Per procurarci altri moduli possiamo usare i programmi pragma2module e iconvert. Questi programmi convertono i file header standard C ed i file include Assembly im moduli. I file C header dovrebbero contenere i pragmas per gli offsets di funzione e i file include Assembly dovrebbero contenere le costanti e le definizioni di struttura (le strutture Assembly saranno convertite in object). Tuttavia, se non stai tentando di realizzare qualcosa di veramente avanzato, probabilmente non avrai bisogno di preoccuparti di nulla riguardo a quanto detto.

1.122 beginner.guide/Esempio sull'uso dei Moduli

2.7.4 Esempio sull'uso dei Moduli
==========

Il programma di esempio sui gadget nel Capitolo 3 mostra come usare le costanti del modulo intuition/intuition (Vedi
sez. 3.5.1
), mentre il
programma di esempio IDCMP mostra come viene usato l'object gadget dello stesso modulo. Vedi
sez. 3.5.2
. Il seguente programma usa i moduli idonei
per la libreria Reqtools, che non è una libreria standard di sistema Amiga, ma che comunque viene comunemente usata, tali moduli sono forniti con Amiga E. Per eseguire questo programma, avrai bisogno naturalmente, della reqtools.library in Libs:.

```
MODULE 'reqtools'
```

```
PROC main()
  DEF col
  IF (reqtoolsbase:=OpenLibrary('reqtools.library',37))<>NIL
    IF (col:=RtPaletteRequestA('Seleziona un colore', 0,0))<>-1
      RtEZRequestA('Hai selezionato il colore \d',
                   'I did|I can\at remember',0,[col],0)
    ENDIF
    CloseLibrary(reqtoolsbase)
  ELSE
    WriteF('Non si può aprire la reqtools.library, version 37+\n')
  ENDIF
ENDPROC
```

La variabile reqtoolsbase è la variabile base di libreria per la Reqtools library. Questa è definita nel modulo reqtools e il risultato della chiamata a OpenLibrary deve essere conservato in questa variabile se stiamo per usare una qualsiasi funzione della Reqtools library. (Possiamo scoprire quale variabile usare per le altre librerie, usando il programma showmodule sul modulo di libreria che ci serve). Le due funzioni usate nel programma sono RtPaletteRequestA e RtEZRequestA. Senza l'inclusione del modulo reqtools e il settaggio della variabile reqtoolsbase non saremmo in grado di usare queste funzioni. Se i parametri di MODULE non sono esatti (path errato), non saremmo in grado di compilare il programma perchè il compilatore non saprebbe da dove vengono le funzioni e quindi manderebbe un messaggio d'errore.

Notare che la Reqtools library è chiusa prima del termine del programma (se essa era stata aperta con successo). Questo è sempre necessario: se riusciamo ad aprire una libreria dobbiamo anche chiuderla quando non è più necessaria.

1.123 beginner.guide/Code Modules

2.7.5 Code Modules (Codice dei Moduli)
==========

Anche noi possiamo scrivere dei moduli che contengono definizioni di procedura e alcune variabili globali, sono chiamati code modules e possono essere estremamente utili. Questa sezione descrive brevemente qualcosa sulla loro costruzione e uso. Per dettagli approfonditi vedere il 'Reference Manual'.

Per costruire un code modules, usiamo il compilatore E, come faremmo per un eseguibile solo che all' inizio del codice dobbiamo usare la dichiarazione OPT MODULE. Inoltre, tutte le definizioni che dovranno essere usate dall'esterno (da un programma che chiama il modulo), devono essere marcate con la keyword EXPORT. In alternativa, tutte le definizioni possono essere esportate usando OPT EXPORT all'inizio del codice. Quindi le definizioni incluse nel modulo, le usiamo nel nostro programma usando MODULE nel modo normale.

Segue un codice di esempio di un piccolo modulo:

```
OPT MODULE

EXPORT CONST MAX_LEN=20

EXPORT OBJECT nomecompleto
  nome, cognome
ENDOBJECT

EXPORT PROC stampanome(p:PTR TO nomecompleto)
  IF corto(p.cognome)
    WriteF('Ciao, \s \s\n', p.nome, p.cognome)
  ELSE
    WriteF('Haarg, hai un nome lungo\n')
  ENDIF
ENDPROC

PROC corto(s)
  RETURN StrLen(s)<MAX_LEN
ENDPROC
```

Tutto viene esportato tranne la procedura corto, pertanto potrá essere usata solo nel modulo. In realtá, la procedura stampanome, usa la procedura corto (piuttosto artificialmente), per controllare la lunghezza del cognome. Tale procedura non è molto usata o importante nel modulo, ecco perchè non è esportata. In effetti però, abbiamo nascosto all'utente del modulo, il fatto che stampanome usa corto. Assumendo che il suddetto codice venga compilato come modulo, posto nel disco Emodules in una directory chiamata MieiModuli col nome di nome potrebbe essere usato in un programma nel seguente modo:

```
MODULE 'MieiModuli/nome'

PROC main()
  DEF fred:PTR TO nomecompleto, nomelungo
  fred.nome:='Fred'
  fred.cognome:='Flintstone'
  stampanome(fred)
  nomelungo:=['Mario', 'Estremolunghissimopreistoriconome']
  stampanome(nomelungo)
ENDPROC
```

Le variabili globali in un modulo sono un po' più problematiche degli altri tipi di definizioni. Non possiamo inizializzarle nella dichiarazione o riservare loro chunks (pezzi) di memoria. Pertanto non possiamo avere ARRAY, OBJECT, STRING o dichiarazioni LIST, tuttavia possiamo usare i puntatori, quindi questo non è un grosso problema. La ragione di questa limitazione è che le variabili globali con lo stesso nome, esportate in un modulo e nel programma principale sono considerate come la stessa variabile e i valori vengono condivisi. Così possiamo avere una dichiarazione di array nel programma principale:

```
DEF a[80]:ARRAY OF INT
```

e l'appropriata dichiarazione di puntatore nel modulo:

```
EXPORT DEF a:PTR TO INT
```

L'array, allora, può essere usato nel modulo prendendolo dal programma principale! Per questa ragione bisogna prestare molta attenzione ai nomi delle variabili esportate, per non avere indesiderate condivisioni. Le variabili globali che non vengono esportate sono private al modulo, quindi non entrerebbero in contrasto con variabili nel programma principale o in altri moduli.

1.124 beginner.guide/Controllo delle Eccezioni

```
                2.8 Controllo delle Eccezioni (Exception Handling)
    ********************************************
```

Spesso il nostro programma deve controllare i risultati delle funzioni e prendere diverse decisioni se sono capitati degli errori. Per esempio se proviamo ad aprire una finestra (usando OpenW), potremmo ottenere NIL per il puntatore ritornato, il che vuol dire che la finestra, per qualche ragione, non si è potuta aprire. In questo caso, normalmente, bisogna compiere le operazioni di prechiusura e terminare il programma. Le operazioni di prechiusura a volte consistono nel chiudere le windows, gli schermi e le librerie, quindi può capitare che i casi di errore possano rendere il programma ingombrante e disordinato. Ecco dove le eccezioni sono utili, exception è semplicemente un caso di errore, e exception handling affronta i casi di errore. La exception handling in E separa accuratamente il codice dell'errore specifico dal vero codice del nostro programma.

1.125 beginner.guide/Procedure con Exception Handlers

```
2.8.1 Procedure con Exception Handlers
      ================================
```

Una procedura con un exception handler è simile a questa:

```
PROC fred(params...) HANDLE
  /* Principale, codice reale */
EXCEPT
  /* Codice per il controllo dell'errore */
ENDPROC
```

Questa è molto simile ad una normale procedura con la differenza delle

keywords HANDLE e EXCEPT. La keyword HANDLE segnala alla procedura che sta per avere un controllo delle eccezioni (exception handler) e la keyword EXCEPT marca la fine del normale codice e l'inizio del codice per il controllo dell'errore. La procedura si comporta proprio come una normale procedura, quando esegue il codice nella parte precedente a EXCEPT, ma quando capita un errore, possiamo passare il controllo all'exception handler (ossia, il codice posto dopo EXCEPT, viene eseguito).

1.126 beginner.guide/Ottenere una Exception

```
            2.8.2 Ottenere una Exception
 ======================
```

Quando capita un errore (e vogliamo controllarlo), possiamo ottenere un'eccezione usando le funzioni Raise o Throw. Chiamiamo la funzione Raise con un numero che identifica il tipo di errore capitato. Il codice dell'exception handler è responsabile della decodifica del numero e quindi di fare la cosa appropriata. La funzione Throw è molto simile a quella Raise e la seguente descrizione di Raise è applicabile anche a Throw. La differenza è che Throw ha un secondo argomento che può essere usato per passare un'informazione extra ad un handler (normalmente una stringa). I termini 'raising' e 'throwing' (ottenere e lanciare) una excption possono essere usati indifferentemente.

Quando Raise è chiamata, essa immediatamante interrompe l'esecuzione del codice della procedura corrente e passa il controllo all'exception handler della procedura recente che ha un handler (che può essere la procedura corrente). Questo e un po' complicato, ma noi possiamo inserire, chiamare e usare l'exception handler nella stessa procedura, come nel seguente esempio:

```
CONST BIG_AMOUNT = 100000

ENUM ERR_MEM=1

PROC main() HANDLE
  DEF block
  block:=New(BIG_AMOUNT)
  IF block=NIL THEN Raise(ERR_MEM)
  WriteF('C''è abbastanza memoria\n')
EXCEPT
  IF exception=ERR_MEM
    WriteF('Non c''è abbastanza memoria\n')
  ELSE
    WriteF('Exception sconosciuta\n')
  ENDIF
ENDPROC
```

Questo esempio usa l'exception handler per stampare il messaggio 'Non c'è abbastanza memoria' se la chiamata a New ritorna NIL. Il parametro di Raise è conservato nella variabile speciale exception nella parte exception handler del codice, così se Raise è chiamata con un numero diverso da ERR_MEM, verrá stampato il messaggio 'Exception sconosciuta'.

Prova ad eseguire questo programma assegnando davvero un grande valore a BIG_AMOUNT (grande_quantitá), in modo che New non possa allocare la memoria. Noterai che il messaggio 'C'è abbastanza memoria' non viene stampato se Raise viene chiamata. Questo perchè, quando chiamiamo Raise l'esecuzione del normale codice della procedura viene interrotto e il controllo viene passato all'appropriato exception handler. Quando viene raggiunta la fine dell'exception handler, la procedura ha termine, in questo caso ha termine anche il programma in quanto trattasi della procedura main.

Se usiamo Throw al posto di Raise, allora, nell'handler, la variabile speciale exceptioninfo conterrá il valore del secondo parametro, che potrá essere usato in congiunzione con exception per fornire l'handler di ulteriori informazioni circa l'errore. Segue l'esempio precedente riscritto con Throw:

```
CONST BIG_AMOUNT = 100000

ENUM ERR_MEM=1

PROC main() HANDLE
  DEF block
  block:=New(BIG_AMOUNT)
  IF block=NIL THEN Throw(ERR_MEM, 'Non c''è abbastanza memoria\n')
  WriteF('C''è abbastanza memoria\n')
EXCEPT
  IF exception=ERR_MEM
    WriteF(exceptioninfo)
  ELSE
    WriteF('Exception sconosciuta\n')
  ENDIF
ENDPROC
```

Una enumerazione (con ENUM) è un buon metodo per creare differenti costanti per varie exception e sempre come consiglio, è sempre bene usare delle costanti come parametro per Raise e per l'exception handler, in quanto il tutto diventa molto più leggibile: Raise(ERR_MEM) è molto più leggibile di Raise(1). L'enumerazione inizia con uno, poichè zero è una exception speciale: essa normalmente significa che non è capitato nessun errore. Questo è utile quando l'handler fá le stesse operazioni di prechiusura che sarebbero normalmente fatte quando il programma termina con esso. Per questa ragione esiste una forma speciale di EXCEPT che usa una exception zero quando il codice nella procedura termina con successo. Questa forma è EXCPET DO, con il DO si suggerisce al lettore che l'exception handler è chiamata anche se non capita nessun errore, inoltre se l'argomento della funzione Raise è omesso, di default è zero. Vedi
sez. 2.2.3
.

Quindi, che cosa succede se chiamiamo Raise in una procedura senza un exception handler? Bene, qui è dove la vera potenza del meccanismo di handling viene ad illuminarci. In questo caso, il controllo passa all'exception handler della procedura recente con un handler; se non ne viene trovato nessuno, il programma termina.

Una procedura si definisce recente, in quanto è tale per la procedura che da essa viene chiamata. Quindi, se la procedura fred chiama la barney,

allora quando la barney è in esecuzione, fred è una procedura recente per la barney. Poichè la procedura main è quella dove il programma inizia, essa è una procedura recente per ogni altra procedura nel programma. Questo significa in pratica che:

* Se definiamo fred per essere una procedura con un exception handler, allora qualsiasi procedura chiamata da fred, avrá le sue exception controllate dall'handler in fred se esse non hanno il proprio handler.

* Se definiamo main per essere una procedura con un exception handler, allora qualsiasi exception ottenuta sará sempre trattata con qualche codice di exception handler (cioè l'handler di main o qualche altra procedura.

Segue un esempio più complicato:

```
ENUM FRED=1, BARNEY

PROC main()
  WriteF('Ciao da main\n')
  fred()
  barney()
  WriteF('Arrivederci da main\n')
ENDPROC

PROC fred() HANDLE
  WriteF(' Ciao da fred\n')
  Raise(FRED)
  WriteF(' Arrivederci da fred\n')
EXCEPT
  WriteF(' Handler fred: \d\n', exception)
ENDPROC

PROC barney()
  WriteF('  Ciao da barney\n')
  Raise(BARNEY)
  WriteF('  Arrivederci da barney\n')
ENDPROC
```

Quando esegui questo programma ottieni il seguente output:

```
Ciao da main
 Ciao da fred
 Handler fred: 1
  Ciao da barney
```

Questo perchè la procedura fred è terminata dalla chiamata Raise(FRED) mentre l'intero programma è terminato dalla chiamata Raise(BARNEY) (finchè barney e main non hanno handler).

Ora proviamo questo programma:

```
ENUM FRED=1, BARNEY

PROC main()
  WriteF('Ciao da main\n')
  fred()
```

```
  WriteF('Arrivederci da main\n')
ENDPROC

PROC fred() HANDLE
  WriteF(' Ciao da fred\n')
  barney()
  Raise(FRED)
  WriteF(' Arrivederci da fred\n')
EXCEPT
  WriteF(' Handler fred: \d\n', exception)
ENDPROC

PROC barney()
  WriteF('  Ciao da barney\n')
  Raise(BARNEY)
  WriteF('  Arrivederci da barney\n')
ENDPROC
```

Quando esegui questo programma ottieni il seguente output:

```
Ciao da main
 Ciao da fred
  Ciao da barney
 Handler fred: 2
Arrivederci da main
```

Ora la procedura fred chiama la barney, così main e fred sono procedure recenti quando Raise(BARNEY) è eseguita, pertanto l'exception handler di fred è chiamato e quando questi termina, anche la chiamata a fred dalla main termina, così la procedura main viene completata e possiamo vedere il messaggio 'Arrivederci'. Nel programma precedente a questo la chiamata Raise(BARNEY) non aveva procedure recenti con handler a cui riferirsi e pertanto l'intero programma è terminato in quel punto.

1.127 beginner.guide/Exceptions automatiche

2.8.3 Exceptions automatiche
      ======================

Nella precedente sezione abbiamo visto come ottenere un'eccezione quando una chiamata a New ritorna NIL. Possiamo riscrivere quell'esempio per ottenere l'exception automaticamente:

```
CONST BIG_AMOUNT = 100000

ENUM ERR_MEM=1

RAISE ERR_MEM IF New()=NIL

PROC main() HANDLE
  DEF block
  block:=New(BIG_AMOUNT)
  WriteF('C''è abbastanza memoria\n')
```

```
EXCEPT
  IF exception=ERR_MEM
    WriteF('Non c''è abbastanza memoria\n')
  ELSE
    WriteF('Exception sconosciuta\n')
  ENDIF
ENDPROC
```

La sola differenza è la rimozione dell'IF che controllava il valore di block e l'aggiunta di una parte RAISE. Questo significa che ogni qualvolta, nel programma, viene chiamata la funzione New, e questa ritorna NIL, l'exception ERR_MEM verrá ottenuta (ossia l'exception ERR_MEM è ottenuta automaticamente). Con la rimozione di molte dichiarazioni IF di controllo errore, possiamo semplificare molto il programma.

La sintassi precisa della parte RAISE è:

```
RAISE exception  IF function()  compare  value,
      exception2 IF function2() compare2 value2 ,
      ...
```

Il parametro exception è una costante (o numero) che rappresenta l'exception da ottenere, function è un BUILT-IN E oppure una funzione di sistema da controllare automaticamente, value è il valore di ritorno da controllare e compare è il metodo usato per comparare il valore (ossia, =, <>, <, <=, > o >=). Questo meccanismo è valido solo per BUILT-IN o funzioni di libreria, in quanto, altrimenti, non avrebbero nessun modo per ottenere le exception. Le procedure definite da noi possono, naturalmente, ottenere le exception in un modo molto più flessibile.

1.128 beginner.guide/Raise all'interno dell'Exception Handler

2.8.4 Raise all'interno dell'Exception Handler

Se chiamiamo Raise all'interno di un exception handler, il controllo viene passato al prossimo handler più recente. In questo modo possiamo scrivere delle procedure che hanno un handler che esegue delle prechiusure locali e usando Raise alla fine di un codice handler, possiamo invocare un codice recente di prechiusura.

Come esempio useremo le funzioni di sistema di Amiga AllocMem e FreeMem, simili alle funzioni built-in New e Dispose, ma la memoria allocata da AllocMem deve essere disallocata (usando FreeMem) quando essa non serve più, prima della fine del programma.

```
CONST POCA=100, MOLTA=123456789

ENUM ERR_MEM=1

RAISE ERR_MEM IF AllocMem()=NIL

PROC main()
  allocare()
```

```
ENDPROC

PROC allocare() HANDLE
  DEF mem=NIL
  mem:=AllocMem(POCA, 0)
  altralloc()
  FreeMem(mem, POCA)
EXCEPT
  IF mem THEN FreeMem(mem, POCA)
  WriteF('Handler: disallocazione locale di memoria "allocare"\n')
ENDPROC

PROC altralloc() HANDLE
  DEF altra=NIL, ealtra=NIL
  altra:=AllocMem(POCA, 0)
  ealtra:=AllocMem(MOLTA, 0)
  WriteF('Allocata tutta la memoria!\n')
  FreeMem(ealtra, MOLTA)
  FreeMem(altra, POCA)
EXCEPT
  IF ealtra THEN FreeMem(ealtra, MOLTA)
  IF altra THEN FreeMem(altra, POCA)
  WriteF('Handler: disallocazione locale di memoria "altralloc"\n')
  Raise(ERR_MEM)
ENDPROC
```

Le chiamate ad AllocMem sono controllate automaticamente, se viene ritornato NIL, viene ottenuta l'exception ERR_MEM. L'handler nella procedura allocare, controlla se essa ha bisogno di liberare la memoria puntata da mem, mentre l'handler nella procedura altralloc controlla ealtra e altra. Alla fine dell'handler altralloc, c'è la chiamata a Raise(ERR_MEM), che passa il controllo all'exception handler della procedura allocare, in quanto da questa procedura viena chiamata la procedura altralloc (quindi, per altralloc la procedura recente è allocare).

Ci sono un paio di sottigliezze da notare nell'esempio. In primo luogo, le variabili di memoria sono tutte inizializzate a NIL. Questo perchè l'exception automatica da ottenere su AllocMem risulterá nelle variabili, che non vengono assegnate se la chiamata ritorna NIL (cioè, l'exception viene alzata prima che si verifichi l'assegnazione), e l'handler ha bisogno di esso per essere NIL se AllocMem fallisce. Naturalmente se AllocMem non ritorna NIL le assegnazioni lavorano normalmente.

In secondo luogo, le dichiarazioni IF negli handler, controllano che le variabili che puntano alla memoria non contengano NIL, usando i loro valori come valori di veritá. Siccome, attualmente, NIL è zero, un puntatore non-NIL sará non-zero, cioè, true (vero) nel controllo IF. Questa sintassi breve viene usata spesso, pertanto dobbiamo prestare attenzione ad essa.

E' abbastanza usuale che si voglia ottenere la stessa exception di un exception handler dopo che questi ha svolto il suo compito. La funzione ReThrow (che non ha nessun argomento) può essere usata per questo scopo. Essa riotterá l'exception, ma solo se l'exception non è zero (fin tanto che questo valore speciale stará a significare che non è capitato nessun errore). Se l'exception è zero allora questa funzione non avrá nessun effetto. In pratica i seguenti frammenti di codice (all'interno di un

handler) sono equivalenti:

```
ReThrow()

IF exception THEN Throw(exception, exceptioninfo)
```

Ci sono due esempi, nel Capitolo 3, su come usare un exception handler per rendere più leggibile un programma: in uno vengono usati i file di dati (Vedi
sez. 3.2
) e nell'altro le aperture di schermi e finestre. Vedi
sez. 3.5.4
.

1.129 beginner.guide/Allocazione di Memoria

```
2.9 Allocazione di Memoria
**********************
```

Quando un programma è in esecuzione, la memoria viene usata in vari differenti modi. Per usare qualsiasi parte di memoria, essa deve essere prima allocata, che è semplicemente un modo di marcare la memoria che fosse 'in uso'. Questo per impedire che lo stesso pezzo di memoria venga usato per un differente immagazzinaggio di dati (per esempio da differenti programmi) e avere così un aiuto per impedire la corruzione dei dati conservati lá. Ci sono due modi generali in cui la memoria può essere allocata: dinamicamente e staticamente.

1.130 beginner.guide/Allocazione Statica

```
2.9.1 Allocazione Statica
==================
```

La memoria allocata staticamente è la memoria allocata dal programma per variabili e dati statici come costanti stringa, per lists e typed lists. Vedi
sez. 2.4.5.7
. Ogni variabile in un programma, richiede un po' di

memoria in cui conservare il proprio valore. Le variabili dichiarate per essere di tipo ARRAY, LIST, STRING o qualsiasi object, richiedono due pezzi di memoria: uno per conservare il valore del puntatore e uno per conservare la grande quantitá di dati (ad esempio gli elementi in un ARRAY). In effetti, tali dichiarazioni sono semplicemente dichiarazioni di tipo PTR TO unite all'inizializzazione del puntatore all'indirizzo di qualche memoria (staticamente) allocata per contenere i dati. Il seguente esempio mostra delle dichiarazioni molto simili, la differenza è che nel secondo caso (usando PTR), viene allocata solo la memoria per contenere i valori dei puntatori. Nel primo caso viene allocata anche la memoria per contenere l'appropriata dimensione dell'array, dell'object e della E-string.

```
DEF a[20]:ARRAY,   m:myobj,        s[10]:STRING

DEF a:PTR TO CHAR, m:PTR TO myobj, s:PTR TO CHAR
```

I puntatori, nel secondo caso, non sono inizializzati dalla dichiarazione e quindi non sono dei puntatori validi. Questo significa che essi non dovrebbero essere dereferenziati (dereference) in nessun modo, finchè vengono inizializzati all'indirizzo di qualche memoria allocata e ciò implica l'allocazione dinamaca della memoria. Vedi
sez. 2.9.3
.

1.131 beginner.guide/Disallocazione della Memoria

2.9.2 Disallocazione della Memoria
==========

Quando la memoria viene allocata è marcata, concettualmente, come fosse 'in uso', quindi la stessa memoria non può essere allocata due volte, così verrá allocato un differente pezzo di memoria (se ce ne sono di disponibili) quando il programma vuole allocare ancora. In questo modo, le variabili sono allocate in differenti pezzi di memoria e pertanto i loro valori possono essere distinti. Ma c'è solo una certa quantitá di memoria disponibile e se non può essere marcata di nuovo come 'non in uso', presto verrebbe a mancare (e il programma avrebbe una fine sgradevole). Questo è quello che fa la disallocazione: marca la memoria allocata in precedenza, come fosse 'non in uso' e la rende disponibile per una nuova allocazione. Tuttavia la memoria dovrebbe essere disallocata solo quando essa è, effettivamente, non più in uso ed è qui che le cose diventano un po' complicate.

La memoria è una risorsa talmente vitale in ogni computer che diventa importante usarla bene, un po' di essa come necessita e dissallocarla ogni qualvolta è possibile. Anche perchè un linguaggio di programmazione come l'E, usa la maggior parte dell'allocazione della memoria per le variabili. La memoria allocata per le variabili viene automaticamente disallocata quando non è più possibile per il programma usare quelle variabili. Comunque tale disallocazione automatica non è utile per le variabili globali in quanto possono essere usate da qualsiasi procedura e quindi la disallocazione può avvenire solo al termine del programma. Le variabili locali di una procedura, d'altra parte, vengono allocate quando viene

chiamata la procedura, ma non possono essere usate dopo il suo ritorno. Pertanto esse vengono disallocate quando la procedura ritorna.

I puntatori, come sempre, possono causare grossi problemi. Il seguente esempio mostra perchè abbiamo bisogno di stare attenti nel momento in cui usiamo i puntatori come valore di ritorno di una procedura.

```
/* Questo è un esempio di cosa *NON* fare */
PROC fullname(first, last)
  DEF full[40]:STRING
  StrCopy(full, first)
  StrAdd(full, ' ')
  StrAdd(full, last)
ENDPROC full

PROC main()
  WriteF('Il nome è \s\n', fullname('Fred', 'Flintstone'))
ENDPROC
```

A prima vista questo codice sembra ottimo e in effetti potrebbe anche lavorare correttamente se lo eseguiamo una o due volte (ma stiamo attenti: può mandare in tilt il computer). Il problema è che la procedura fullname ritorna il valore della variabile locale full, che è un puntatore a qualche zona di memoria allocata staticamente per la E-string e questa memoria verrá disallocata quando la procedura ritorna. Questo significa che il valore di ritorno di ogni chiamata a fullname è l'indirizzo della memoria appena disallocata e quindi non è possibile dereferenziarla. Ma la chiamata a WriteF fá soltanto quello: essa dereferenzia il risultato di fullname al fine di stampare la E-string puntata da full. Questo è un problema molto comune, in quanto è molto facile caderci. Il fatto che tale situazione può essere usata in molte occasioni, la rende anche, molto più difficile da individuare. La soluzione in questo caso è usare l'allocazione dinamica. Vedi

sez. 2.9.3
.

Se sei ancora un po' scettico, allora questo è un problema reale, prova il precedente esempio sostituendo la procedura main con una di queste, ma stai nuovamente attento, in quanto, anche queste, potrebbero mandare in tilt il computer.

```
/* Questa potrebbe non stampare la corretta stringa */
PROC main()
  DEF f
  f:=fullname('Fred', 'Flintstone')
  WriteF('Il nome è \s\n', f)
ENDPROC

/* Questa stamperá definitivamente g invece di f */
PROC main()
  DEF f, g
  f:=fullname('Fred', 'Flintstone')
  g:=fullname('Barney', 'Rubble')
  WriteF('Il nome è \s\n', f)
ENDPROC
```

(Il motivo per cui le cose vanno male è stato spiegato prima, ma il perchè

di certe stampe è oltre la portata di questa Guida.)

1.132 beginner.guide/Allocazione Dinamica

2.9.3 Allocazione Dinamica
==========

La memoria allocata dinamicamente, è qualsiasi memoria che non viene allocata staticamente. Per allocare la memoria dinamicamente possiamo usare le funzioni List e String, tutti i modi di New e il versatile operatore NEW. Ma affinchè la memoria sia allocata dinamicamente, deve essere esplicitamente disallocata quando non è più necessaria. In tutti i suddetti casi, tuttavia, qualsiasi memoria che risulti ancora allocata al termine del programma, verrá disallocata automaticamente.

Un altro metodo per allocare la memoria dinamicamente è usare le funzioni di sistema di Amiga, basate su AllocMem. Tuttavia, queste funzioni richiedono che la memoria da loro allocata, sia disallocata (usando funzioni come FreeMem) prima che il programma termini, altrimenti essa non sará mai disallocata (almeno fino a quando non riavviamo il computer). Pertanto è più sicuro provare ad usare funzioni E, ogni qualvolta è possibile, per l'allocazione dinamica.

Ci sono molte ragioni perchè potremmo aver bisogno di usare l'allocazione dinamica, la mggior parte di esse implica l'inizializzazione dei puntatori. Per esempio, le dichiarazioni nella sezione sull'allocazione statica possono essere estese per inizializzare i puntatori dichiarati nella seconda linea DEF. Vedi
sez. 2.9.1
.

```
DEF a[20]:ARRAY,   m:myobj,        s[10]:STRING

DEF a:PTR TO CHAR, m:PTR TO myobj, s:PTR TO CHAR
a:=New(20)
m:=New(SIZEOF myobj)
s:=String(20)
```

Queste sono inizializzazioni alla memoria allocata dinamicamente, mentre la prima linea di dichiarazioni inizializza dei puntatori simili, alla memoria allocata staticamente. Se queste sezioni di codice fossero parte di una procedura, allora sarebbero variabili locali e qui subentra un'altra significante differenza: la memoria allocata dinamicamente non sarebbe automaticamente disallocata al ritorno della procedura, mentre la memoria allocata staticamente lo sarebbe. Questo significa che possiamo risolvere il problema della disallocazione. Vedi
sez. 2.9.2
.

```
/* Questo è il modo corretto di procedere */
PROC fullname(first, last)
  DEF full
  full:=String(40)
```

```
  StrCopy(full, first)
  StrAdd(full, ' ')
  StrAdd(full, last)
ENDPROC full

PROC main()
  DEF f, g
  WriteF('Il nome è \s\n', fullname('Fred', 'Flintstone'))
  f:=fullname('Fred', 'Flintstone')
  g:=fullname('Barney', 'Rubble')
  WriteF('Il nome è \s\n', f)
ENDPROC
```

Usando String, la memoria puntata all'E-string da full, ora è allocata dinamicamente, e non viene più disallocata sino alla fine del programma. Questo significa che diventa abbastanza corretto, sia passare il valore di full come risultato della procedura fullname, sia dereferenziare il risultato, usando WriteF per stamparlo. Tuttavia questo tipo di allocazione ha causato un ultimo problema: la memoria non viene disallocata se non alla fine del programma, pertanto, è potenzialmente sprecata, almeno fino a quando non viene usata, ad esempio, per contenere i risultati di successive chiamate alla procedura. Naturalmente, la memoria può essere disallocata, solo quando i dati che conserva, non sono più richiesti. La seguente procedura main, da sostituire alla procedente, mostra quando potremmo aver bisogno di disallocare la E-string (usando DisposeLink).

```
PROC main()
  DEF f, g
  f:=fullname('Fred', 'Flintstone')
  WriteF('Il nome è \s, f punta a $\h\n', f, f)
/* Provala con e senza la prossima linea DisposeLink */
  DisposeLink(f)
  g:=fullname('Barney', 'Rubble')
  WriteF('Il nome è \s, g punta a $\h\n', g, g)
  DisposeLink(g)
ENDPROC
```

Se eseguiamo il programma con la linea DisposeLink(f), probabilmente g punterá alla stessa memoria di f. Questo perchè, la chiamata a DisposeLink disalloca la memoria puntata da f, e quindi può essere riusata per conservare la E-string puntata da g. Se rendi tale linea come un commento (o la cancelli), allora noterai che f ed g puntano sempre ad una memoria differente.

In alcuni casi è meglio non fare nessuna disallocazione, a causa dei problemi in cui possiamo trovarci, se disallochiamo la memoria troppo presto (ossia, prima di aver terminato di usare i dati che essa contiene). Naturalmente è un operazione sicura (ma temporaneamente devestante), disallocare, nel momento sbagliato, la memoria allocata dinamicamente dalle funzioni E, mentre diventa un'operazione davvero devastante (e sbagliata) se questo succede con le funzioni di sistema di Amiga come AllocMem.

Un altro beneficio dell'allocazione dinamica è che la dimensione degli arrays, E-lists ed E-string che possiamo creare, può essere il risultato di qualsiasi espressione, quindi non è limitata a valori di costanti. (Ricorda che la dimensione data a dichiarazioni ARRAY, LIST e STRING deve essere una costante.) Questo significa che la procedura fullname può essere resa più

efficiente e si può allocare solo la memoria di cui ha veramente bisogno per la creazione della E-string.

```
PROC fullname(first, last)
  DEF full
  /* L'extra +1 è per lo spazio aggiunto */
  full:=String(StrLen(first)+StrLen(last)+1)
  StrCopy(full, first)
  StrAdd(full, ' ')
  StrAdd(full, last)
ENDPROC full
```

Comunque, può essere molto complicato o inutile calcolare la dimensione corretta. In tali casi, una veloce stima di una costante complessiva, potrebbe andar meglio.

Le varie funzioni che allocano dinamicamente la memoria hanno le corrispondenti funzioni per disallocare tale memoria. La seguente tavola mostra parte delle più comuni corrispondenze:

```
Allocazione      Disallocazione
------------------------------
New              Dispose
NewR             Dispose
List             DisposeLink
String           DisposeLink
NEW              END
FastNew          FastDispose
AllocMem         FreeMem
AllocVec         FreeVec
AllocDosObject   FreeDosObject
```

NEW ed END sono degli operatori versatili e potenti, spiegati nella seguente sezione. Le funzioni che iniziano con il prefisso Alloc- sono funzioni di sistema di Amiga e sono accoppiate con funzioni con nome uguale e con prefisso Free-. Per maggiori dettagli vedere il 'Rom Kernel Reference Manual'.

1.133 beginner.guide/Operatori NEW ed END

```
            2.9.4 Operatori NEW ed END
 ====================
```

Un aiuto ad affrontare l'allocazione dinamica della memoria e la relativa disallocazione lo abbiamo da due potenti operatori, NEW ed END. L'operatore NEW è molto versatile, è simile, come scopo, alla famiglia di funzioni built-in New. Vedi
sez. 2.6.3.5
. L'operatore END è il corrispondente deallocatore di NEW (pertanto è simile alla famiglia di funzioni built-in Dispose). La maggiore differenza tra NEW e i vari modi di New è che NEW alloca la memoria basandosi sui tipi dei suoi argomenti.

1.134 beginner.guide/Object e semplice allocazione dei tipi

2.9.4.1 Object e semplice allocazione dei tipi

Le seguenti sezioni di codice sono approssimativamente equivalenti e servono per mostrare la funzione NEW, e come sia strettamente attinente a NewR. (Il tipo può essere qualsiasi object o un tipo semplice.)

```
DEF p:PTR TO tipo
NEW p

DEF p:PTR TO tipo
p:=NewR(SIZEOF tipo)
```

Notare che l'uso di NEW non è come una chiamata a funzione, in quanto il parametro p, non è racchiuso fra parentesi. Questo perchè NEW è un operatore più che una funzione. Funziona diversamente da una funzione proprio perchè NEW ha bisogno di riconoscere i tipi dei suoi argomenti. Quindi la dichiarazione di p diventa molto importante se deve influire sulla quantitá della memoria che NEW deve allocare. La versione di codice che usa NewR, dichiara esplicitamente la quantitá di memoria che deve essere allocata (usando l'operatore SIZEOF), quindi, in tal caso, la dichiarazione del tipo di p, non è molto importante per avere una allocazione corretta.

Il prossimo esempio mostra come NEW può essere usato per inizializzare molti puntatori in una volta. La seconda sezione di codice è approssimativamente equivalente, solo che usa NewR. (Ricorda che il tipo di default di una variabile è LONG, che in effetti è PTR TO CHAR.)

```
DEF p:PTR TO LONG, q:PTR TO myobj, r
NEW p, q, r

DEF p:PTR TO LONG, q:PTR TO myobj, r
p:=NewR(SIZEOF LONG)
q:=NewR(SIZEOF myobj)
r:=NewR(SIZEOF CHAR)
```

Questi primi due esempi hanno mastrato la sintassi per la dichiarazione di NEW. Esiste una sintassi come espressione, che ha un parametro e ritorna l'indirizzo della memoria appena allocata, come pure inizializza l'argomento del puntatore a tale indirizzo.

```
DEF p:PTR TO myobj, q:PTR TO myobj
```

```
q:=NEW p

DEF p:PTR TO myobj, q:PTR TO myobj
q:=(p:=NewR(SIZEOF tipo))
```

Questa possibilitá può sembrare non necessariamente utile, ma è anche il modo in cui NEW viene usato per allocare copie di lists e typed lists. Vedi

sez. 2.9.4.3
.

Per disallocare la memoria allocata da NEW, bisogna usare END, con la dichiarazione dei puntatori che si vogliono disallocare. Per lavorare correttamente, END richiede che il tipo di ogni puntatore, sia quello usato al momento della dichiarazione con NEW. Un insuccesso dovuto a questo motivo, disallocherá una quantitá scorretta di memoria e ciò può causare molti particolari problemi in un programma. Bisogna anche stare attenti a non disallocare la memoria due volte, a tal fine i puntatori passati a END sono reinizializzati a NIL, dopo che la memoria da loro puntata, viene disallocata (è abbastanza sicuro usare END con un puntatore di valore NIL). Tuttavia, ciò non copre tutti i problemi, in quanto più di un puntatore può puntare alla stessa area di memoria, come ci mostra il seguente esempio.

```
DEF p:PTR TO LONG, q:PTR TO LONG
q:=NEW p
p[]:=-24
q[]:=613
END p
/* p ora è NIL, ma q ora non è valida anche se non è NIL */
```

La prima assegnazione inizializza q per essere una copia di p (che è inizializzata da NEW). Le due successive assegnazioni cambiano, entrambe, il valore puntato da p ed q. Poi la memoria allocata per conservare tale valore, viene disallocata usando END e questi assegna anche, p, con il valore NIL. Comunque, l'indirizzo conservato in q, non viene modificato e punta ancora alla memoria, appena disallocata. Questo significa che ora q ha un credibile, ma non valido, valore di puntatore. La sola cosa che si può fare con sicurezza, con q, è reinizializzarla. Una delle peggiori cose, invece, è usare q con END, ciò disallocherebbe di nuovo la stessa memoria e potenzialmente ci sarebbe la possibilitá di mandare in tilt il computer. Quindi, ricapitolando, non disallocare lo stesso valore di un puntatore più di una volta e tenere nota di tutte le variabili che puntano alla stessa memoria.

Così come l'uso di NEW ha un semplice (ma approssimativo) equivalente nell'uso di NewR, END ha un suo equivalente nell'uso di Dispose, come possiamo vedere nelle seguenti sezioni di codice.

```
END p

IF p
  Dispose(p)
  p:=NIL
ENDIF
```

In realtá tutto diventa un po' più complicato se allochiamo e disallochiamo gli object OOP con NEW ed END. Vedi

sez. 2.12
.

1.135 beginner.guide/Allocazione di Array

2.9.4.2 Allocazione di Array

Anche gli arrays (matrici) possono essere allocati da NEW, usandolo in un modo molto simile a quello evidenziato nella precedente sezione. La differenza è che deve essere fornita anche la dimensione dell'array, sia a NEW che ad END. Naturalmente la dimensione fornita ad END deve essere identica a quella fornita per l'appropriato uso di NEW. Grazie a questo sforzo extra, diventerai abile nel creare un array con una dimensione non costante (diversamente dalle variabili di tipo ARRAY). Questo significa che la dimensione fornita a NEW ed END può essere il risultato di una particolare espressione.

```
DEF a:PTR TO LONG, b:PTR TO myobj, s
NEW a[10]  /* Un dinamico array di LONG  */
s:=my_random(20)
NEW b[s]   /* Un dinamico array di myobj */
/* ...altro codice... */
END a[10], b[s]
```

La funzione my_random, rappresenta qualche particolare calcolo, usata per dimostrare che la dimensione dell'array non deve necessariamente essere una costante (o numero). Anche questa sintassi di NEW può essere usata come espressione, esattamente come abbiamo giá visto in precedenza.

1.136 beginner.guide/Allocazione di list e typed list

2.9.4.3 Allocazione di list e typed list

Lists e typed lists, normalmente sono dati statici, ma NEW può essere usato per creare delle versioni allocate dinamicamente. Questa sintassi di NEW può essere usata solo come espressione, NEW prende la lista (o la typed list) come suo argomento e ritorna l'indirizzo della copia della lista allocata dinamicamente. La disallocazione della memoria allocata in questo modo è un po' più complicata di quella vista prima, ma possiamo, naturalmente, lasciare che sia disallocata automaticamente, alla fine del programma.

Il seguente esempio mostra quanto sia semplice l'uso di NEW per risolvere il problema dei dati statici descritto in precedenza. Vedi
sez. 2.4.5.7
.
La differenza dall'errato programma originale è minima.

```
PROC main()
```

```
  DEF i, a[10]:ARRAY OF LONG, p:PTR TO LONG
  FOR i:=0 TO 9
    a[i]:=NEW [1, i, i*i]
      /* a[i] è ora allocata dinamicamente */
  ENDFOR
  FOR i:=0 TO 9
    p:=a[i]
    WriteF('a[\d] è una matrice all''indirizzo \d\n', i, p)
    WriteF('  e il secondo elemento è \d\n', p[1])
  ENDFOR
ENDPROC
```

Il piccolo cambiamento con l'originale, è stato, prefissare la lista con NEW, così si è resa la lista, dinamica. Questo significa che ogni a[i], ora è una differente lista, anzichè la stessa statica lista della versione originale del programma.

Le typed lists si allocano in modo simile, e il seguente esempio mostra anche come disallocarle. Fondamentalmente, abbiamo bisogno di sapere quanto è lunga la nuova matrice (cioè, di quanti elementi è composta), se se la typed list è veramente, solo, una matrice inizializzata. Con questa informazione, possiamo disallocarla con una normale matrice, ricordandoci di usare l'appropriato tipo di puntatore. Le liste object-typed sono limitate (quando usate con NEW) ad una matrice con al massimo un object, quindi NEW è utile solo per allocare un object inizializzato (non realmente una matrice). Notare, come nel seguente codice, il puntatore q, può essere trattato sia come un object, sia come una matrice di un singolo object. Vedi

sez. 2.4.4.2

.

```
OBJECT myobj
  x:INT, y:LONG, z:INT
ENDOBJECT

PROC main()
  DEF p:PTR TO INT, q:PTR TO myobj
  p:=NEW [1, 9, 3, 7, 6]:INT
  q:=NEW [1, 2]:myobj
  WriteF('L''ultimo elemento nella matrice p è \d\n', p[4])
  WriteF('L''object q è x=\d, y=\d, z=\d\n',
         q.x,   q.y,   q.z)
  WriteF('La matrice q è q[0].x=\d, q[0].y=\d, q[0].z=\d\n',
         q[].x, q[].y, q[].z)
  END p[5], q
ENDPROC
```

La versione di una lista object-typed, allocata dinamicamente, ha ancora un'altra differenza, rispetto alla versione statica: essa ha sempre la memoria allocata per un numero intero di object, quindi un object inizializzato parzialmente è riempito con degli zero. La versione statica non alloca questo riempimento extra, pertanto dobbiamo stare attenti ad accedere solo agli elementi nominati nella lista.

La disallocazione delle copie di liste normali ottenute con NEW, può avvenire, come al solito, automaticamente alla fine del programma. Se vogliamo disallocare tali liste prima, dobbiamo usare la funzione

FastDisposeList, passandogli come unico argomento, l'indirizzo della lista. Non dobbiamo usare END o qualsiasi altro metodo di disallocazione. FastDisposeList è il solo modo sicuro per disallocare le liste allocate da NEW.

1.137 beginner.guide/Allocazione di object OOP

2.9.4.4 Allocazione di object OOP

Attualmente, il solo modo per creare object OOP in E, è usare NEW, mentre l'unico modo per eliminarli è usare END. Questo probabilmente, è l'uso più comune di NEW ed END ed è descritto in dettaglio in seguito. Vedi

sez. 2.12.2
.

1.138 beginner.guide/Numeri in Virgola Mobile

2.10 Numeri in Virgola Mobile

I numeri in virgola mobile o i numeri reali, possono essere entrambi usati per rappresentare sia delle frazioni molto piccole di numeri e sia numeri molto grandi. Tuttavia, diversamente da LONG, che può rappresentare ogni intero compreso in una certo range (Vedi
sez. 1.3.1.1
), i numeri in virgola
mobile hanno una precisione limitata. Sei avvisato, anche se: in E, l'aritmetica in virgola mobile è abbastanza complicata e molti problemi possono essere risolti senza ricorrere ad essa, quindi puoi anche saltare questo capitolo, finchè non hai davvero la necessitá di usare tale aritmetica.

Valori in Virgola Mobile

Calcoli in Virgola Mobile

Funzioni in Virgola Mobile

Precisione e Range

1.139 beginner.guide/Valori in Virgola Mobile

2.10.1 Valori in Virgola Mobile
==========

I valori in virgola mobile, in E, sono scritti proprio come potresti attenderti e sono conservati in variabili LONG:

```
DEF x
x:=3.75
x:=-0.0000367
x:=275.0
```

Dobbiamo ricordarci di usare, nel numero, un punto decimale (senza spazi laterali), se vogliamo considerarlo un numero in virgola mobile, questo è il motivo dell'uso del .0 finale, nel numero dell'ultima assegnazione. Al momento, non possiamo esprimere, in questo modo, qualsiasi valore in virgola mobile vogliamo; il compilatore ci avvertirebbe che il valore non si adatta in 32-bit, se proviamo ad usare più di nove cifre all'incirca, per un solo numero. Possiamo, comunque, usare le varie funzioni per la matematica in virgola mobile, per calcolare qualsiasi valore vogliamo. Vedi

sez. 2.10.3
.

1.140 beginner.guide/Calcoli in Virgola Mobile

2.10.2 Calcoli in Virgola Mobile
==========

Se un numero in virgola mobile è conservato in una variabile LONG, normalmente sarebbe interpretato come un intero e questa interpretazione, generalmente non dará qualche altro numero come l'intenzionale numero in virgola mobile. Per usare i numeri in virgola mobile nelle espressioni, dobbiamo usare (piuttosto complicato) l'operatore di conversione in virgola mobile, che corrisponde al carattere !. Questo converte le espressioni e i normali operatori matematici e di paragone, in e da virgola mobile.

Tutte le espressioni sono, di default, espressioni di intero. Cioè, rappresentano valori LONG di interi, piuttosto che valori in virgola mobile. La prima volta che si usa ! in un'espressione, il valore dell'espressione, momentaneamente è convertito in virgola mobile e tutti gli operatori e le variabili dopo il !, sono considerati in virgola mobile. La volta successiva che usiamo !, il valore dell'espressione, momentaneamente è convertito in un intero, come vengono riconvertiti gli operatori e le variabili che seguono il !. E` stata usata la parola, momentaneamente, in quanto possiamo usare il !, tante volte, quanto necessario, entro un'espressione e ogni volta fa l'operazione contraria alla precedente. Le parti fra parentesi di un'espressione, sono trattate come espressioni separate, pertanto di default, sono espressioni di intero (ciò, include gli argomenti di chiamata a funzione).

Le conversioni intero/virgola mobile, eseguite da !, non sono semplici. Esse implicano l'arrotondamento ed anche il troncamento. Una conversione,

per esempio, da intero a virgola mobile e viceversa, generalmente, non ci ridará il valore intero originale.

Seguono alcuni esempi commentati, dove f contiene sempre un numero in virgola mobile e j contiene sempre degli interi:

```
DEF f, i, j
i:=1
f:=1.0
f:=i!  -> i convertito in virgola mobile (1.0)
f:=6.2
i:=!f! -> l'espressione f è in virgola mobile,
       -> poi convertita all'intero (6)
```

Nella prima assegnazione, il valore intero 1 è assegnato ad i. Nella seconda, il valore in virgola mobile 1.0 è assegnato a f. L'espressione sulla destra della terza assegnazione è considerata come un intero sino al momento in cui viene incontrato il ! e in quel momento viene convertita al valore in virgola mobile più vicino. Pertanto ad f viene assegnato il valore in virgola mobile di 1 (cioè, 1.0), proprio quello della seconda assegnazione. L'espressione nell'assegnazione finale, ha bisogno di iniziare come virgola mobile, per poter interpretare il valore di f all'intero più vicino (in questo caso 6).

Le assegnazioni seguenti sono più complicate, ma dovrebbero essere capite per proseguire. Nuovamente, f contiene sempre un numero in virgola mobile ed i ed j sempre degli interi.

```
f:=!f*f -> l'intera espressione è in virgola mobile,
        -> ed f è al quadrato (6.2*6.2)
f:=!f*(i!) -> l'intera espressione è in virgola mobile,
           -> i è convertito in virgola mobile e
           -> multiplicato per f
j:=!f/(i!)! -> l'intera divisione è in virgola mobile,
            -> con il risultato convertito in intero
j:=!f!/i -> il virgola mobile f è convertito in intero
         -> ed è (intero) diviso per i
IF !f<230.0 THEN RETURN 0  -> paragone < in virgola mobile
IF !f>(i!)  THEN RETURN 0  -> i convertito in virgola mobile,
                           -> poi comparato ad f
```

Se avessimo omesso il ! nella prima assegnazione, allora, non solo il valore in f sarebbe interpretato (erroneamente) come intero, ma la moltiplicazione eseguita, sarebbe una moltiplicazione di interi anzichè in virgola mobile. Nella seconda assegnazione, le parentesi che racchiudono l'espressione i, sono di importanza vitale. Senza le parentesi, il valore conservato in i, sarebbe interpretato come virgola mobile e ciò sarebbe errato, in quanto, i, in effetti, conserva un valore intero, le parentesi infatti, sono usate per avere un'espressione separata o nuova che dir si voglia (che di default è intera). Il valore di i, in questo modo, è interpretato correttamente e infine convertito in virgola mobile (da !, esattamente prima la parentesi di chiusura). La (virgola mobile) moltiplicazione, allora, viene eseguita fra due valori in virgola mobile e il risultato viene conservato in f. Nelle ultime due assegnazioni (con la divisione), ad j è assegnato approssimativamente lo stesso valore. Tuttavia l'espressione nella prima assegnazione, permette di avere una precisione maggiore con l'uso della divisione in virgola mobile. Questo significa che

il risultato, in una divisione in virgola mobile viene arrtondato, mentre il risultato, in una divisione di interi, viene troncato.

Una cosa importante da sapere sui numeri in virgola mobile in E è che le seguenti assegnazioni conservano lo stesso valore in g (nuovamente f conserva un numero in virgola mobile), questo perchè non viene eseguito nessun calcolo e nessuna conversione: il valore in f è semplicemente copiato in g. Ciò è importante, principalmente, per le chiamate a funzioni, come vedremo nella prossima sezione. In senso stretto, tuttavia, la seconda versione è migliore, in quanto mostra (al lettore del codice) che il valore in f è da intendersi in virgola mobile.

```
g:=f
g:=!f
```

1.141 beginner.guide/Funzioni in Virgola Mobile

2.10.3 Funzioni in Virgola Mobile
==========

Esistono delle funzioni per formattare dei numeri in virgola mobile, in E-strings (in modo da poterli stampare) e altre per decodificare i numeri in virgola mobile dalle stringhe. Esistono anche delle funzioni built-in in virgola mobile che elaborano parte delle funzioni matematiche meno comuni, come le varie funzioni trigonometriche.

RealVal(string)
Lavora in modo simile a Val che estrae degli interi da una stringa. Il valore decodificato in virgola mobile viene ritornato come il regolare valore di ritorno e il numero di caratteri letti in string, per formare il numero, viene ritornato come primo valore di ritorno facoltativo. Se un valore in virgola mobile non ha potuto essere decodificato da string, allora viene ritornato zero come valore di ritorno facoltativo e il valore regolare di ritorno sará zero (cioè, 0.0).

RealF(e-string,float,digits)
Converte il valore in virgola mobile float in una stringa che viene conservata in e-string. Il numero di cifre da usare dopo il punto decimale viene specificato da digits, che può avere un valore da 0 a 8. Il valore in virgola mobile è arrotondato al numero di cifre specificato. Un valore zero per digits, dá un risultato senza parte frazionaria e senza punto decimale. Da questa funzione viene ritornata una e-string, che poi è semplice da utilizzare con WriteF.

```
PROC main()
  DEF s[20]:STRING, f, i
  f:=21.60539
  FOR i:=0 TO 8
    WriteF('f è \s (usando digits=\d)\n', RealF(s, f, i), i)
  ENDFOR
ENDPROC
```

Notare che l'argomento in virgola mobile, f, in RealF, non ha bisogno del ! iniziale, in quanto stiamo semplicemente passando il suo valore,

non stiamo eseguendo un calcolo con esso. Il programma dovrebbe generare il seguente output:

```
f è 22 (usando digits=0)
f è 21.6 (usando digits=1)
f è 21.61 (usando digits=2)
f è 21.605 (usando digits=3)
f è 21.6054 (usando digits=4)
f è 21.60539 (usando digits=5)
f è 21.605390 (usando digits=6)
f è 21.6053900 (usando digits=7)
f è 21.60539000 (usando digits=8)
```

Fsin(float), Fcos(float), Ftan(float)
Calcolano (rispettivamente), il seno, il coseno e la tangente dell'angolo float fornito, che è specificato in radianti.

Fabs(float)
Ritorna il valore assoluto di float, molto simile a quello che fa Abs per gli interi.

Ffloor(float), Fceil(float)
La funzione Ffloor arrotonda un valore in virgola mobile al valore minore più vicino, mentre la funzione Fceil arrotonda un valore in virgola mobile al valore maggiore più vicino.

Fsqrt(float)
Ritorna la radice quadrata di float.

Fpow(x,y), Fexp(float)
La funzione Fpow ritorna il valore di x elevato alla potenza di y (che sono entrambi valori in virgola mobile). La funzione Fexp ritorna il valore di, e, elevato alla potenza di float, dove, e, è il valore matematico speciale (approssimativamente 2.718282). Tale valore, elevato ad una potenza è conosciuto come exponentiation.

Flog10(float), Flog(float)
La funzione Flog10 ritorna il logaritmo in base 10 di float (logaritmo comune). La funzione Flog ritorna il logaritmo in base e, di float (logaritmo naturale). Flog10 e Fpow sono linkati nel modo seguente (ignorando le inesattezze in virgola mobile):

```
x = Fpow(10.0, Flog10(x))
```

Flog e Fexp hanno una attinenza simile (Fexp può essere riutilizzato usando 2.718282 come il primo argomento al posto di 10.0).

```
x = Fexp(Flog(x))
```

Segue un piccolo programma che usa un po' delle funzioni viste e mostra come definire le funzioni che usano o ritornano valori in virgola mobile.

```
DEF f, i, s[20]:STRING

PROC print_float()
  WriteF('\tf è \s\n', RealF(s, !f, 8))
ENDPROC
```

```
PROC print_both()
  WriteF('\ti è \d, ', i)
  print_float()
ENDPROC

/* Un float al quadrato */
PROC square_float(f) IS !f*f

/* Un intero al quadrato */
PROC square_integer(i) IS i*i

/* Converte un float in un intero */
PROC convert_to_integer(f) IS Val(RealF(s, !f, 0))

/* Converte un intero in un float */
PROC convert_to_float(i) IS RealVal(StringF(s, '\d', i))

/* Questa dovrebbe essere uguale a Ftan */
PROC my_tan(f) IS !Fsin(!f)/Fcos(!f)

/* Questa dovrebbe mostrare le inesattezze float */
PROC inaccurate(f) IS Fexp(Flog(!f))

PROC main()
  WriteF('Le prossime 2 linee dovrebbero essere uguali\n')
  f:=2.7; i:=!f!
  print_both()
  f:=2.7; i:=convert_to_integer(!f)
  print_both()

  WriteF('Le prossime 2 linee dovrebbero essere uguali\n')
  i:=10;  f:=i!
  print_both()
  i:=10;  f:=convert_to_float(i)
  print_both()

  WriteF('f ed i dovrebbero essere uguali\n')
  i:=square_integer(i)
  f:=square_float(f)
  print_both()

  WriteF('Le prossime due linee dovrebbero essere uguali\n')
  f:=Ftan(.8)
  print_float()
  f:=my_tan(.8)
  print_float()

  WriteF('Le prossime 2 linee dovrebbero essere uguali\n')
  f:=.35
  print_float()
  f:=inaccurate(f)
  print_float()
ENDPROC
```

Le funzioni convert_to_integer e convert_to_float esegueno delle conversioni simili a quelle fatte da !, quando viene usato in

un'espressione. Per rendere il programma più leggibile, ci sono parecchi usi inutili di !, questi sono, quando f è passato direttamente come parametro ad una funzione (in questi casi il ! può essere omesso con sicurezza). Tutti gli esempi hanno la potenzialitá per dare risultati differenti dove invece dovrebbero dare lo stesso risultato e questo è dovuto all'imprecisione dei numeri in virgola mobile. L'ultimo esempio è stato scelto attentamente proprio per evidenziare questa situazione.

1.142 beginner.guide/Precisione e Range

```
2.10.4 Precisione e Range
       ==================
```

Un numero in virgola mobile è soltanto un altro valore a 32-bit, così da poter essere conservato in variabili LONG. E' soltanto l'interpretazione dei 32-bit che li rende differenti. Un numero in virgola mobile può essere compreso in un range che parte da numeri piccoli come 1.3E-38 per arrivare a numeri grandi quanto 3.4E+38 (si tratta di numeri molto piccoli e molto grandi, se non conosci la notazione scientifica!). Comunque non tutti i numeri in questo range possono essere rappresentati con precisione se il numero di cifre significanti è all'incirca otto.

La precisione ha un'importanza rilevante nel momento in cui si sta tentando di comparare due numeri in virgola mobile e nel momento in cui li combiniamo dopo averli divisi. Normalmente è meglio controllare che un valore in virgola mobile sia compreso in un piccolo range di valori, piuttosto che abbia solo un determinato valore. Quando si stanno combinando dei valori, invece, è meglio permettere una piccola quantitá di errore dovuto all'arrotondamento ecc. Vedere il 'Reference Manual' per maggiori dettagli sull'implementazione dei numeri in virgola mobile.

1.143 beginner.guide/Ricorsione

```
                2.11 Ricorsione (Recursion)
     **********************
```

Una funzione ricorsiva è molto simile ad una funzione che usa un loop. Fondamentalmente, una chiamata ricorsiva ad una funzione, richiama se stessa (dopo una manipolazione di dati) piuttosto che ripetere una parte di codice usando un loop. Esistono anche i tipi ricorsivi, che sono objects, con gli elementi che hanno tipi di objects (in E questi sarebbero puntatori agli objects). Abbiamo gia visto usare il modo ricorsivo: nelle liste linkate, dove ogni elemento nella lista, conteneva un puntatore all'elemento successivo. Vedi
sez. 2.4.6
.

Le definizioni ricorsive, normalmente, sono molto più comprensibili di una equivalente funzione iterativa ed è più facile manipolarne i dati. Tuttavia, la ricorsione non è certamente un argomento semplice. Leggi

quindi a tuo rischio e pericolo.

1.144 beginner.guide/Esempio Fattoriale

2.11.1 Esempio Fattoriale

Un esempio calzante per una definizione ricorsiva è la funzione fattoriale. Nella matematica scolastica il simbolo ! è usato dopo un numero, per evidenziare il fattoriale di quel numero (e solo gli interi positivi hanno i fattoriali). n! è n-fattoriale, che è definito come segue:

```
n! = n * (n-1) * (n-2) * ... * 1     (per n >= 1)
```

Quindi, 4! è 4*3*2*1, che è 24. 5! è 5*4*3*2*1, che è 120.

Segue la definizione iterativa di una funzione fattoriale (Raise è usato per ottenere un'eccezione, se il numero non è positivo, ma possiamo anche tralasciare con sicurezza questo controllo, se siamo sicuri che la funzione sará chiamata solo con numeri positivi):

```
PROC fact_iter(n)
  DEF i, result=1
  IF n<=0 THEN Raise("FACT")
  FOR i:=1 TO n
    result:=result*i
  ENDFOR
ENDPROC result
```

Abbiamo usato un loop FOR per generare i numeri da 1 a n (il parametro di fact_iter), e result per contenere i risultati parziali e quello finale. Il risultato finale viene ritornato, quindi puoi controllare che fact_iter(4) ritorni 24 e fact_iter(5) ritorni 120, aggiungendo alla procedura precedente, qualcosa che sia simile alla seguente procedura main:

```
PROC main()
  WriteF('4! è \d\n5! è\d\n', fact_iter(4), fact_iter(5))
ENDPROC
```

Se sei stato davvero attento, potresti aver notato che 5! è 5*4!, e, che in generale n! è n*(n-1)!. Questo è il nostro primo approccio ad una definizione ricorsiva, possiamo definire la funzione fattoriale negli

stessi termini. La vera definizione di fattoriale è (il motivo per cui questa è la vera definizione, è che i '...', nella definizione precedente, non sono sufficientemente precisi per una definizione matematica):

```
1! = 1
n! = n * (n-1)!    (per n > 1)
```

Notare che ci sono due casi da considerare. Il primo caso è detto caso base e dá un valore calcolato facilmente (ossia, non viene usata nessuna ricorsione). Il secondo caso è il caso ricorsivo e dá una definizione in termini di un numero più vicino al caso base (cioè, (n-1) è più vicino 1 di n, per n>1). Il problema in cui ci si imbatte facilmente quando si usa la ricorsione è la dimenticanza del caso base. Senza il caso base in un programma ricorsivo, probabilmente, il computer andrá in tilt! Vedi

sez. 2.11.4
.

Adesso possiamo definire la versione ricorsiva della funzione fact_iter (di nuovo, verrá usato Raise se il parametro non è un numero positivo):

```
PROC fact_rec(n)
  IF n=1
    RETURN 1
  ELSEIF n>=2
    RETURN n*fact_rec(n-1)
  ELSE
    Raise("FACT")
  ENDIF
ENDPROC
```

Notare come questa definizione sembra proprio come quella matematica e come sia bella e compatta. Possiamo anche usare, per la definizione, la sintassi di una sola linea della funzione (se omettiamo il controllo di positivitá sul parametro):

```
PROC fact_rec2(n) RETURN IF n=1 THEN 1 ELSE n*fact_rec2(n-1)
```

Potremmo essere tentati di omettere il caso base e scrivere qualcosa di simile:

```
/* Non fare questo! */
PROC fact_bad(n) RETURN n*fact_bad(n-1)
```

Il problema è che la ricorsione non avrá mai fine. La funzione fact_bad sará chiamata con ogni numero da n a zero e poi con tutti gli interi negativi. Un valore non sará ritornato mai e il computer dopo un po' andrá in tilt. La ragione precisa del perchè il computer va in tilt verrá spiegata in seguito. Vedi

sez. 2.11.4
.

1.145 beginner.guide/Ricorsione Reciproca

```
2.11.2 Ricorsione Reciproca (Mutual)
       ============================
```

Nella precedente sezione abbiamo visto la funzione fact_rec che chiamava se stessa. Se abbiamo due funzioni, fun1 e fun2, e fun1 chiama fun2, e fun2 chiama fun1, allora queste due funzioni sono reciprocamente (mutually) ricorsive. Questo vale per tutte le funzioni collegate in questo modo.

Segue un esempio piuttosto ricercato di due funzioni reciprocamente ricorsive.

```
PROC f(n)
  IF n=1
    RETURN 1
  ELSEIF n>=2
    RETURN n*g(n-1)
  ELSE
    Raise("F")
  ENDIF
ENDPROC

PROC g(n)
  IF n=1
    RETURN 2*1
  ELSEIF n>=2
    RETURN 2*n*f(n-1)
  ELSE
    Raise("G")
  ENDIF
ENDPROC
```

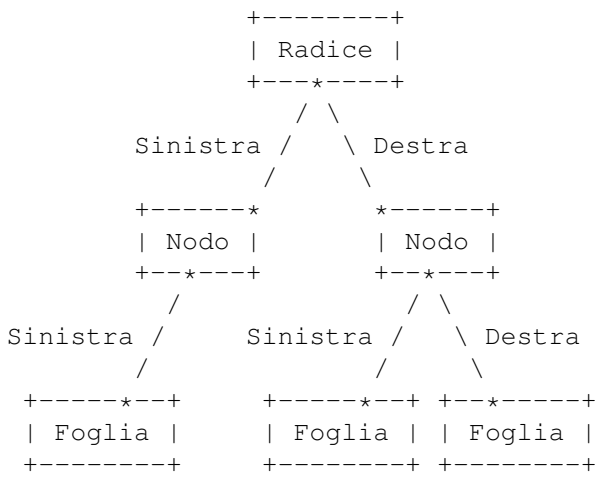
Entrambe le funzioni sono molto simili alla funzione fact_rec, ma g ritorna i valori normali raddoppiati. L'effetto complessivo è che ogni altro valore nella versione estesa della moltiplicazione è raddoppiato. Pertanto, f(n) elabora in questo modo: n*(2*(n-1))*(n-2)*(2*(n-3))*...*2, il che probabilmente non è molto interessante.

1.146 beginner.guide/Alberi Binari

```
2.11.3 Alberi Binari (Binary Trees)
       ===========================
```

Questa sezione spiega un'altro tipo ricorsivo e l'effetto che esso ha sulle funzioni che manipolano questo tipo di dati. Un albero binario è simile ad una lista linkata, ma invece di avere per ogni elemento, un solo collegamento ad un altro elemento, ci sono due collegamenti per ogni elemento di un albero binario (che puntano agli alberi più piccoli, chiamati diramazioni (branches)). I primi collegamenti puntano alla sinistra della diramazione, mentre i secondi, puntano alla destra. Ogni elemento dell'albero è chiamato nodo (node), esistono due tipi speciali di nodo: il punto di partenza, chiamato radice (root) dell'albero (come la

cima di una lista) e i nodi che non hanno diramazioni nè a sinistra, nè a destra (cioè, puntatori NIL per entrambi i collegamenti), chiamati foglia (leaves). Ogni nodo dell'albero contiene qualche tipo di dati (proprio come le liste linkate contengono, in ogni elemento, una E-string o una E-list). Il seguente schema illustra un piccolo albero.



Notare che un nodo potrebbe avere solo una diramazione (non deve necessariamente avere, sia la sinistra che la destra). Inoltre, nell'esempio, le foglie sono tutte allo stesso livello, ma questa non è una regola, una qualsiasi delle foglie, poteva facilmente essere, un nodo da cui si diramavano altri nodi.

Quindi, come può essere scritta una struttura ad albero, come un object E ? Bene, la sintassi generale è questa:

```
OBJECT tree
  data
  left:PTR TO tree, right:PTR TO tree
ENDOBJECT
```

Gli elementi left e right sono puntatori alle diramazioni di sinistra e di destra (che saranno, anche, alberi objects). L'elemento data, rappresenta alcuni dati per ogni nodo, ma poteva ben essere anche un puntatore, un ARRAY o un numero di differenti elementi di dati.

Allora, che uso si può fare di un tale albero ? Bene, un uso comune è quello di avere una raccolta ordinata di dati, che ha la necessitá di poter raggiungere velocemente i vari elementi. Ad esempio, il dato ad ogni nodo, potrebbe essere un intero, quindi un albero di questo tipo, potrebbe contenere una serie ordinata di interi. Per rendere l'albero ordinato, le limitazioni devono essere poste sulle diramazioni di sinistra e di destra di un nodo. La diramazione di sinistra, potrebbe contenere, solo dei nodi con valori più piccoli di quello contenuto nel nodo genitore, mentre quella di destra, potrebbe contenere, solo valori più grandi. Nodi con gli stessi dati, potrebbero anche essere inseriti in una delle diramazioni, ma per il nostro esempio non li useremo. Adesso siamo pronti per scrivere qualche funzione per manipolare il nostro albero.

La prima funzione inizializza una nuova serie di interi (cioè, dá inizio ad un nuovo albero). Essa dovrebbe prendere un intero come parametro e ritornare un puntatore al nodo radice del nuovo albero (con gli interi come

dati dei nodi).

```
PROC new_set(int)
  DEF root:PTR TO tree
  NEW root
  root.data:=int
ENDPROC root
```

La memoria per il nuovo elemento dell'albero deve essere allocata dinamicamente, pertanto questo esempio diventa buono anche per un altro uso di NEW. Se NEW pulisce la memoria che esso alloca, tutti gli elementi del nuovo object saranno zero. In particolare, i puntatori di sinistra e di destra saranno NIL, così il nodo radice sará anche un nodo foglia. Se NEW fallisce verrá ottenuta una eccezione "MEM", altrimenti il dato è assegnato con il valore fornito e viene ritornato un puntatore al nodo radice.

Per aggiungere un nuovo intero a un tale assegnamento, abbiamo bisogno di trovare l'appropriata posizione per inserirlo e di assegnare correttamente le diramazioni di sinistra e di destra. Ciò perchè se l'intero è assegnato a un nodo nuovo, questi sará aggiunto come un nuovo nodo foglia, e quindi uno dei nodi esistenti cambierá la sua diramazione a sinistra o a destra.

```
PROC add(i, set:PTR TO tree)
  IF set=NIL
    RETURN new_set(i)
  ELSE
    IF i<set.data
      set.left:=add(i, set.left)
    ELSEIF i>set.data
      set.right:=add(i, set.right)
    ENDIF
    RETURN set
  ENDIF
ENDPROC
```

Questa funzione ritorna un puntatore a set, a cui ha aggiunto l'intero. Se tale set inizialmente era vuoto, viene creato un nuovo set, altrimenti viene ritornato il puntatore originale. Le appropriate diramazioni sono corrette come la ricerca progredisce. Solo l'ultima assegnazione alla diramazione di sinistra o di destra è significante (tutte le altre non cambiano il valore del puntatore), se questa è proprio l'assegnazione che aggiunge un nuovo nodo foglia. Segue una versione iterativa di questa funzione:

```
PROC add_iter(i, set:PTR TO tree)
  DEF node:PTR TO tree
  IF set=NIL
    RETURN new_set(i)
  ELSE
    node:=set
    LOOP
      IF i<node.data
        IF node.left=NIL
          node.left:=new_set(i)
          RETURN set
        ELSE
          node:=node.left
```

```
          ENDIF
        ELSEIF i>node.data
          IF node.right=NIL
            node.right:=new_set(i)
            RETURN set
          ELSE
            node:=node.right
          ENDIF
        ELSE
          RETURN set
        ENDIF
      ENDLOOP
    ENDIF
  ENDPROC
```

Come possiamo notare, questa versione è abbastanza confusionaria. Le funzioni ricorsive lavorano bene con la manipolazione di tipi ricorsivi.

Quest'altro esempio stampa i contenuti di set. Esso è ingannevolmente semplice:

```
  PROC show(set:PTR TO tree)
    IF set<>NIL
      show(set.left)
      WriteF('\d ', set.data)
      show(set.right)
    ENDIF
  ENDPROC
```

Gli interi nei nodi, saranno stampati in ordine (per aggiungerli usare la funzione add). I nodi di sinistra contengono gli elementi più piccoli, quindi i dati che essi contengono saranno stampati per primi, seguiti dai dati del nodo corrente e poi da quelli dei nodi di destra. Prova a scrivere una funzione iterativa di questa funzione se vuoi risolvere un problema davvero difficile.

Per avere un programma funzionante, che raccolga in se tutti gli esempi (e cioè, OBJECT tree + new_set(...) + add(...) + show(...) + il prossimo), segue una procedura main, che può appunto essere usata per provare le suddette funzioni:

```
  PROC main() HANDLE
    DEF s, i, j
    Rnd(-999999)    /* Inizializza il seme */
    s:=new_set(10)  /* Inizializza set s per contenere il numero 10 */
    WriteF('Input:\n')
    FOR i:=1 TO 50  /* Genera 50 numeri casuali e li aggiunge a set s */
      j:=Rnd(100)
      add(j, s)
      WriteF('\d ',j)
    ENDFOR
    WriteF('\nOutput:\n')
    show(s)         /* Mostra i contenuti dell'(ordinato) set s */
    WriteF('\n')
  EXCEPT
    IF exception="NEW" THEN WriteF('Memoria finita\n')
  ENDPROC
```

1.147 beginner.guide/Stack (e Crashing)

2.11.4 Stack (e Crashing)

Quando chiamiamo una procedura, sfruttiamo un po' di stack del programma. Lo stack è usato per poter rintracciare le procedure di un programma in esecuzione e possono sorgere dei veri problemi se lo spazio stack viene a mancare. Normalmente, la quantitá di stack disponibile per ogni programma è sufficiente, se il compilatore E maneggia tutti i bit importanti abbastanza bene. Tuttavia ai programmi che fanno molto uso della ricorsione, può venire a mancare abbastanza facilmente lo stack.

Per esempio fact_rec(10) avrá bisogno di abbastanza stack per dieci chiamate a fact_rec, nove delle quali sono chiamate ricorsive. Questo perchè ogni chiamata non termina, finchè non è stato elaborato il valore di ritorno, quindi tutte le chiamate ricorsive a fact_rec(1) hanno bisogno di essere contenute nello stack finchè fact_rec(1) non ritorni uno. Quindi ogni procedura potrá essere scalata di un posto verso l'alto fin quando ci sará spazio nello stack. Se proviamo a computare fact_rec(40000), non solo questa porterá via una gran quantitá di tempo, ma probabilmente andrá fuori dallo spazio riservato allo stack, il computer in questo caso andrá in tilt oppure si comporterá in modo strano. La versione iterativa fact_iter non ha di questi problemi in quanto si tratta di contenere solo una chiamata a procedura per calcolare un fattoriale usando tale funzione.

Se c'è la possibilitá che lo spazio stack venga a mancare, possiamo usare una chiamata alla funzione (built-in) FreeStack. Vedi

sez. 2.6.3.5

. Tale funzione ritorna la quantitá di spazio stack libero. Se esso scende, all'incirca, sotto 1KB, allora potremmo decidere di fermare la ricorsione o qualsiasi altra cosa stia sfruttando lo stack. Inoltre, possiamo specificare la quantitá di stack da fornire al nostro programma (specificare quello che il compilatore potrebbe decidere è più appropriato) usando l'opzione OPT STACK. Vedere il 'Reference Manual' per maggiori dettagli sull'organizzazione E dello stack.

1.148 beginner.guide/Stack ed Exceptions

2.11.5 Stack ed Exceptions

Il concetto 'recente' usato in precedenza è collegato con lo stack. Vedi

sez. 2.8.2

. Una procedura recente è quella che si trova in cima allo stack, essendo la più recente è la procedura corrente. Così, quando viene

chiamata Raise, questa cerca nello stack finchè non trova una procedura con un exception handler. Quell'handler, allora, verrá usato e tutte le procedure precedenti a questa verranno scalate di un posto verso l'alto nello stack.

Pertanto, una funzione ricorsiva con un exception handler può usare Raise nell'handler per chiamare l'handler nella precedente (ricorsiva) chiamata della funzione. Così, qualcosa che sia stato allocato ricorsivamente può essere 'ricorsivamente' disallocato dagli exception handlers. Questa è una caratteristica molto potente e importante degli exception handlers.

1.149 beginner.guide/Object Orientato all'E

```
              2.12 Object Orientato all'E
     **********************
```

In questo capitolo tratteremo gli aspetti in E, della programmazione orientata agli oggetti (OOP). Non preoccuparti se non conosci parole particolari come 'object' (oggetto), 'method' (metodo) e 'inheritance' (ereditá): questi termini sono spiegati nell'introduzione alla OOP qui di seguito. (Per qualche ragione, la scienza del computer usa parole strane per velare di segretezza dei concetti semplici.)

1.150 beginner.guide/Introduzione alla OOP

```
               2.12.1 Introduzione alla OOP
       ====================
```

'Programmazione Orientata all'Object' è il nome dato ad una raccolta di tecniche di programmazione, studiate per velocizzare lo sviluppo e rendere più facile la manutenzione di grandi programmi. Queste tecniche esistono da parecchio tempo, ma è solo recentemente che i linguaggi che le supportano sono diventati popolari. Comunque, anche se il linguaggio supporta la OOP, non abbiamo la necessitá di programmare in modo orientato all'object, solo che tutto diventa un po' più semplice se lo facciamo!

1.151 beginner.guide/Classi e Metodi

2.12.1.1 Classi e Metodi

Il cuore della OOP è l'approccio alla programmazione 'Black Box'. Il genere di black box in questione è quello dove i contenuti sono ignoti, ma esistono un certo numero di fili che dall'esterno ci danno la possibilitá, in qualche modo, di interagire con le cose all'interno. I black boxes della OOP, sono in effetti, raccolte di dati (proprio come il concetto di variabile che abbiamo giá visto) e sono chiamati objects (questo è il termine generale, che è casualmente collegato con il tipo OBJECT dell'E). Gli objects possono essere raggruppati insieme in classes (classi), come i tipi per le variabili, eccetto che una classe definisce anche quali differenti tipi di fili protendere dal black box. Questo pezzo extra (i fili) è conosciuto come interface (interfaccia) all'oggetto ed è costituita da un certo numero di methods (metodi) (quindi un metodo equivale ad un filo). Ogni metodo, in effetti, è proprio come una procedura. Con un reale black box, i fili sono il solo modo di interagire con il box, quindi i metodi di un oggetto dovrebbero essere il solo modo di creare e usare l'oggetto. Naturalmente, i metodi stessi, normalmente hanno bisogno di conoscere i lavori interni dell'oggetto, cioè in che modo normalmente i fili sono collegati a qualcosa nel black box.

Esistono due tipi speciali di metodi: constructors (costruttori) e destructors (distruttori). Il metodo constructor è usato per inizializzare i dati in un oggetto, una classe può avere molti differenti constructors (permettendo diversi generi di inizializzazione), oppure può non averne nessuno se non è neccessaria nessuna particolare inizializzazione. I constructors sono normalmente usati per allocare delle risorse (come la memoria) di cui un oggetto ha bisogno. La disallocazione di tali risorse è fatta dal destructor, ne esiste uno per la maggior parte di ogni classe.

Il modo per proteggere i contenuti di un oggetto nel 'black box' è conosciuto come data-hiding (i dati nell'oggetto sono visibili solo ai suoi metodi) e il solo modo che permette di manipolare i contenuti di un oggetto attraverso la sua interfaccia è conosciuto come data abstraction. Usando questo approccio, solo i metodi conoscono la struttura dei dati in un oggetto e quindi questa struttura può essere cambiata, senza influire sull'intero programma: solo i metodi, potenzialmente, avrebbero bisogno di essere cambiati. Anche tu, adesso, dovresti essere in grado di dire che, questo semplifica abbastanza notevolmente la manutenzione.

1.152 beginner.guide/Esempio di classe

2.12.1.2 Esempio di classe

Un buon esempio di una classe è la nozione matematica di un (set) insieme (di interi). Un particolare oggetto da questa classe rappresenterebbe una particolare serie di interi. L'interfaccia per la classe, probabilmente, includerebbe i seguenti metodi:

1. Add -- aggiunge un intero ad un oggetto set.

2. Member -- Verifica l'appartenenza di un intero in un oggetto set.

3. Empty -- Verifica se l'oggetto set è vuoto.

4. Union -- Unisce un oggetto set con un oggetto set.

Una classe più completa conterrebbe anche metodi per rimuovere elementi, intersecare gli insiemi ecc. La cosa importante da notare è che per usare questa classe, abbiamo bisogno di sapere solo come usare i metodi. L'approccio al black box significa che non sappiamo come la classe set è in effetti implementata, cioè, come i dati sono strutturati all'interno di un oggetto set. Solo i metodi stessi hanno bisogno di sapere come manipolare i dati che rappresentano un oggetto set.

Il beneficio della OOP, lo abbiamo in effetti, quando usiamo le classi, quindi supponiamo di implementare questa classe set e poi di usarla nel codice di qualche programma di database. Se troviamo che l'implementazione set è un po' inefficiente (in termini di memoria o velocitá), allora avendo programmato in modo OOP, non dovremmo modificare l'intero programma di database, ma solo la classe set! Possiamo cambiare il modo in cui i dati set sono strutturati in un oggetto, quante volte ci pare, purchè ogni implementazione abbia la stessa interfaccia (e dia gli stessi risultati!).

1.153 beginner.guide/Inheritance

2.12.1.3 Inheritance (Ereditá)

Il rimanente concetto OOP, di interesse, è inheritance (ereditá). Un nome veramente appropriato per un modo di costruire sulle classi che abilita la classe derivata (cioè più grande) per essere usata come se i suoi oggetti fossero davvero membri della classe ereditata o base. Per esempio, supponiamo che la classe D venga ricavata dalla classe B, allora D sarebbe la classe derivata e B la classe base. In questo caso, la classe D eredita la struttura di dati della classe B, che può aggiungere alla D dati extra. La D eredita anche tutti i metodi della classe B e quindi gli oggetti della classe D possono essere trattati come se fossero davvero oggetti della classe B.

Naturalmente, un metodo ereditato non può influire sui dati extra in classe D, ma solo sui dati ereditati. Per influire sui dati extra, la classe D può avere metodi definiti in modo extra o può creare nuove definizioni per i metodi ereditati. Il secondo approccio è davvero utile, solo se la nuova definizione di un metodo ereditato, è abbastanza simile al metodo ereditato, differente da quest'ultimo quindi, solo per influire sui dati

extra nella classe D. Questo controllo dei metodi non influenza i metodi della classe B (nè quelli di altre classi ricavate da B, ma solo quelli della classe D e le classi ricavate da D.

1.154 beginner.guide/Oggetti in E

```
                2.12.2 Oggetti in E
       ============
```

Le classi sono definite usando OBJECT nello stesso modo visto nelle precedenti sezioni. Vedi
sez. 2.4.4
. Quindi in E i termini 'object
declaration' e 'class' possono essere usati interscambiabilmente. Tuttavia, riferendosi a un tipo OBJECT come una 'classe', segnalare la presenza dei metodi in un oggetto.

Il seguente esempio OBJECT è la base di una classe set, come descritto in precedenza. Vedi
sez. 2.12.1.2
. Questa implementazione di set è abbastanza
semplice, si limita ad un massimo di 100 elementi (elts = elements = elementi).

```
     OBJECT set
       elts[100]:ARRAY OF LONG
       size
     ENDOBJECT
```

Attualmente il solo modo per allocare un oggetto OOP è usare NEW con un appropriato tipo di puntatore. Le seguenti sezioni di codice, allocano tutte, la memoria per i dati di set, ma solo l'ultima alloca in modo OOP l'oggetto set. Tutte possono usare e accedere ai dati di set, ma solo l'ultima può chiamare i metodi di set.

```
       DEF s:set

       DEF s:PTR TO set
       s:=NewR(SIZEOF set)

       DEF s:PTR TO set
       s:=NEW s
```

Gli oggetti OOP, naturalmente, possono essere disallocati usando END, in tal caso è chiamato anche il destructor per la classe corrispondente. Lasciare che un oggetto OOP venga disallocato automaticamente alla fine di un programma, non è molto sicuro, nè normale, in quanto in questo caso, il destructors non sará chiamato. Constructors e destructors sono descritti più in dettaglio in seguito.

1.155 beginner.guide/Metodi in E

```
                2.12.3 Metodi in E
      ===========
```

I metodi dell'E sono molto simili alle normali procedure, ma esiste una grande differenza: un metodo è parte di una classe, quindi deve essere identificato, in qualche modo, con le altre parti della classe. In E questa identificazione è fatta mettendo in relazione tutti i metodi con il corrispondente tipo OBJECT per la classe, usando la keyword OF, dopo la descrizione dei parametri del metodo. Quindi, i metodi della semplice classe set, verrebbero definiti come evidenziato qui di seguito (naturalmente in questi esempi è stato omesso il codice dei metodi).

```
PROC add(x) OF set
  /* codice per il metodo Add */
ENDPROC

PROC member(x) OF set
  /* codice per il metodo Member */
ENDPROC

PROC empty() OF set
  /* codice per il metodo Empty */
ENDPROC

PROC union(s:PTR TO set) OF set
  /* codice per il metodo Union */
ENDPROC
```

A prima vista potrebbe sembrare che al particolare oggetto set, che verrebbe manipolato da questi metodi, mancano dei parametri. Per esempio sembra che il metodo Empty dovrebbe aver bisogno del parametro extra PTR TO set e che sia l'oggetto set stesso a controllare se è vuoto. Tuttavia, i metodi sono chiamati in modo leggermente differente dalle normali procedure. Un metodo è una parte di una classe ed è chiamato in modo simile a quello per accedere ai dati elements della classe. Cioè il metodo è selezionato usando il . e agisce (implicitamente) sull'oggetto scelto (cioè, s.add, dove s rappresenta l'oggetto set e add il metodo). Il seguente esempio mostra l'allocazione di un oggetto set e l'uso di parte dei metodi citati in precedenza.

```
DEF s:PTR TO set
NEW s  -> Allocato un oggetto OOP
s.add(17)
s.add(-34)
IF s.empty()
  WriteF('Errore: il set s non dovrebbe essere vuoto!\n')
ELSE
  WriteF('OK: non è vuoto\n')
ENDIF
IF s.member(0)
  WriteF('Errore: come è entrato 0 lá dentro ?\n')
ELSE
  WriteF('OK: 0 non è un membro\n')
ENDIF
```

```
IF s.member(-34)
  WriteF('OK: -34 è un membro\n')
ELSE
  WriteF('Errore: dove è andato -34 ?\n')
ENDIF
END s  -> Terminato ora con s
```

Ecco il perchè i metodi non usano quell'argomento extra PTR TO set, infatti, se chiamiamo un metodo, questi viene scelto per un oggetto appropriato, che deve essere l'oggetto su cui agire (Es.: s.add). Il metodo un po' complicato è Union, che aggiunge un altro oggetto set da unire con il primo. In questo caso, l'argomento per il metodo è un PTR TO set, ma questi è il set da aggiungere, non il primo set che viene espanso.

Quindi come facciamo a consultare l'oggetto che ci interessa ? In altre parole come lo troviamo ? Bene, questa è la differenza rimanente dalle normali procedure: ogni metodo ha una speciale variabile locale, self, che è di tipo PTR TO class e viene inizializzata per puntare all'oggetto scelto dal metodo. Usando questa variabile, possiamo avere accesso e usare normalmente i dati e i metodi dell'oggetto. Per esempio il metodo Empty ha una variabile locale self di tipo PTR TO set e può essere definita come descritto qui di seguito:

```
PROC empty() OF set IS self.size=0
```

I constructor sono semplicemente i metodi che inizializzano i dati di un oggetto. Per questa ragione, essi dovrebbero essere chiamati, normalmente, solo quando l'oggetto è allocato. L'operatore NEW permette agli oggetti OOP di chiamare un costructor al punto in cui questi sono allocati, ciò per rendere il tutto più semplice e più chiaro. Il constructor sará chiamato dopo che NEW ha allocato la memoria per l'oggetto. Una cosa intelligente, è dare ai constructor dei nomi appropriati, come create (creare) e copy (copiare) o lo stesso nome della classe. I seguenti constructors potrebbero essere definiti per la classe set:

```
/* Crea empty set */
PROC create() OF set
  self.size=0
ENDPROC

/* Copia l'esistente set */
PROC copy(oldset:PTR TO set) OF set
  DEF i
  FOR i:=0 TO oldset.size-1
    self.elements[i]:=oldset.elements[i]
  ENDFOR
  self.size:=oldset.size
ENDPROC
```

Questi constructor potrebbero essere usati come vedremo nel codice seguente. Notare che il constructor create è, superfluo, in questo caso, questo fin quando NEW inizializzerá i dati elements a zero. Se NEW fá una inizializzazione sufficiente, allora non dobbiamo definire nessun constructor e anche se li abbiamo, non dobbiamo usarli se stiamo allocando gli oggetti.

```
DEF s:PTR TO set, t:PTR TO set, u:PTR TO set
```

```
  NEW s.create()
  IF s.empty THEN WriteF('s è vuoto\n')
  END s
  NEW t  /* usare t o create è la stessa cosa */
  IF t.empty THEN WriteF('t è vuoto\n')
  t.add(10)
  NEW u.copy(t)
  IF u.member(10) THEN WriteF('10 è in u\n')
  END t, u
```

Per ogni classe esiste, al massimo, un destructor e questi è responsabile della pulizia e della disallocazione delle risorse. Se vogliamo usare il destructor, dobbiamo chiamarlo con end, e (come il nome potrebbe far intuire) ciò avviene automaticamente, quando un oggetto OOP è disallocato usando END. Quindi, per gli oggetti OOP con un destructor, il codice equivalente (approssimativamente) di END usando Dispose è un po' differente. Bisogna ricordarsi, anche, che il destructor non viene chiamato, se non usiamo END, per disallocare un oggetto OOP (cioè se lasciamo che la disallocazione venga fatta automaticamente alla fine del programma).

```
  END p

  IF p
    p.end()  -> Chiama il destructor
    Dispose(p)
    p:=NIL
  ENDIF
```

La semplice implementazione della classe set non ha bisogno di nessun destructor. Se tuttavia gli elementi dei dati fossero dei puntatori (a LONG) e l'array fosse allocato sulla base di qualche parametro di dimensione ad un constructor, allora un destructor sarebbe utile. In questo caso, la classe set avrebbe bisogno anche come dato di un elemento maxsize, che registri la massima dimensione allocata dell'array elements.

```
OBJECT set
  elements:PTR TO LONG
  size
  maxsize
ENDOBJECT

PROC create(sz=100) OF set  -> Default a 100
  DEF p:PTR TO LONG
  self.maxsize:=IF (sz>0) AND (sz<100000) THEN sz ELSE 100
  self.elements:=NEW p[self.maxsize]
ENDPROC

PROC end() OF set
  DEF p:PTR TO LONG
  IF self.maxsize=0
    WriteF('Errore: non create() il set\n')
  ELSE
    p:=self.elements
    END p[self.maxsize]
  ENDIF
ENDPROC
```

Senza il destructor end, la memoria allocata per elements, non verrebbe disallocata quando viene usato END, sebbene sarebbe disallocata alla fine del programma (in questo caso). Tuttavia, se fosse usato AllocMem al posto di NEW per allocare l'array, allora la memoria dovrebbe essere disallocata usando FreeMem, in modo simile a quello visto prima per Dispose. (Se usiamo AllocMem, la memoria non verrebbe disallocata automaticamente alla fine del programma.) Un'altra soluzione a questo tipo di problema sarebbe avere un metodo speciale che chiama FreeMem e poi ricordarsi di chiamare questo metodo esattamente prima di disallocare uno di questi oggetti, come si può capire, l'iterazione di END con i destructors è molto utile.

La precedente redifinizione di set, giá inizia a mostrare la potenza della OOP. La vera implementazione della classe set è molto differente, ma l'interfaccia può rimanere la stessa. Il codice per i metodi avrebbe bisogno di essere cambiato in modo che possa prendere in considerazione il nuovo elemento maxsize (nella posizione precedente a quella dove viene fissata la dimensione 'size' 100) e bisogna anche affrontare la possibilitá che il constructor create non venga usato (in questo caso elements sarebbe NIL e maxsize zero). Il codice che usa la classe set non dovrebbe aver bisogno di essere cambiato, eccetto forse allocare più sensatamente le dimensioni sets!

In precedenza abbiamo visto una differente implementazione di set. Vedi

sez. 2.11.3

. In effetti, veramente pochi cambiamenti, sarebbero necessari per convertire il codice di quella sezione in un'altra implementazione della classe set. La procedura new_set è come un constructor set che inizializza set per essere un singleton (cioè, per contenere un elemento) e la procedura add è identica al metodo add della classe set. Il solo piccolo problema è che i sets vuoti (empty) non sono previsti dall'implementazione dell'albero binario e per questo motivo, essa non sarebbe un'implementazione completa. Sarebbe facile rendere completa questa particolare implentazione (ma eccessivamente complicato a questo punto) aggiungendo un supporto per i sets empty.

1.156 beginner.guide/Ereditá in E

2.12.4 Ereditá in E
==========

Una classe viene ricavata da un'altra, usando la keyword OF nella definizione OBJECT della classe ricavata, in un modo simile all'uso di OF con i metodi. Per esempio, il seguente codice mostra come definire la classe d in modo che venga ricavata dalla classe b. La classe b è allora la classe che viene ereditata dalla classe d.

```
OBJECT b
  b_data
ENDOBJECT

OBJECT d OF b
```

```
  extra_d_data
ENDOBJECT
```

I nomi b ed d sono stati scelti in quanto hanno dei riferimenti particolari, infatti la classe che viene eredita (cioè, b) è conosciuta come classe base, mentre la classe ereditaria (cioè, d) è conosciuta come classe derivata.

La definizione di d è uguale alla seguente definizione di duff, eccetto per una differenza: con la suddetta derivazione i metodi di b sono ereditati anche da d e essi diventano i metodi della classe d. La definizione di duff non si riferisce in nessun modo ad b, se non accidentalmente, nel migliore dei casi (fin tanto che qualsiasi cambiamento ad b non influisce su duff, mentre invece influiscono su d).

```
OBJECT duff
  b_data
  extra_d_data
ENDOBJECT
```

Una proprietá di questa derivazione si applica ai dati (data) registrati, costruiti con OBJECT e anche alle classi OOP. I dati registrati di tipo d o duff, possono essere usati ovunque un dato registrato di tipo b venga richiesto (per esempio l'argomento a qualche procedura), in effetti i dati registrati di tipo d o duff sono indistinguibili dai dati registrati di tipo b. Sebbene, nel caso cambiassimo la definizione di b (per esempio il nome dell'elemento b_data) i dati registrati di tipo duff non sarebbero usabili in questo modo, mentre quelli di tipo d lo sarebbero ancora. Pertanto è intelligente usare l'ereditá per mostrare le relazioni fra le classi o i dati dei tipi OBJECT. Il seguente esempio mostra come la procedura print_b_data possa essere chiamata, correttamente, in molti modi, dando le definizioni di b, d e duff, viste prima.

```
PROC print_b_data(p:PTR TO b)
  WriteF('b_data = \d\n', p.b_data)
ENDPROC

PROC main()
  DEF p_b:PTR TO b, p_d:PTR TO d, p_duff:PTR TO duff
  NEW p_b, p_d, p_duff
  p_b.b_data:=11
  p_d.b_data:=-3
  p_duff.b_data:=27
  WriteF('Stampa di p_b: ')
  print_b_data(p_b)
  WriteF('Stampa di p_d: ')
  print_b_data(p_d)
  WriteF('Stampa di p_duff: ')
  print_b_data(p_duff)
ENDPROC
```

Finora, nessun metodo è stato definito per b, questo significa che esso è soltanto un tipo OBJECT. La procedura print_b_data, suggerisce un metodo utile di b, che sará chiamato print.

```
PROC print() OF b
  WriteF('b_data = \d\n', self.b_data)
```

```
ENDPROC
```

Questa definizione definirebbe anche un metodo print per d, in quanto d è derivato da b eriditando così tutti i metodi di b. Pertanto, duff sarebbe ancora, solo un tipo OBJECT, sebbene abbia potuto avere un metodo print simile, esplicitamente definito per esso (cioè se fosse una classe), quindi i dati registrati di tipo duff, non possono essere usati come se fossero oggetti della classe b e non è consigliabile provare! In questo caso, solo gli oggetti della classe derivata d, possono essere usati in questa maniera. (Se b è una classe, allora anche d è una classe, a causa dell'ereditá.)

```
PROC main()
  DEF p_b:PTR TO b, p_d:PTR TO d, p_duff:PTR TO duff
  NEW p_b, p_d, p_duff
  p_b.b_data:=11
  p_d.b_data:=-3;   p_d.extra_d_data:=3
  p_duff.b_data:=7; p_duff.extra_d_data:=-7
  WriteF('Stampa di p_b: ')
/* b esplicitamente ha il metodo print */
  p_b.print()
  WriteF('Stampa di p_d: ')
/* d eredita il metodo print da b */
  p_d.print()
  WriteF('Nessun metodo print per p_duff\n')
/* Non provare a stampare p_duff in questo modo */
/*  p_duff.print() */
ENDPROC
```

Sfortunatamente, il metodo print ereditato da d, stampa solo l'elemento b_data (poichè è davvero un metodo di b, non può accedere ai dati extra aggiunti in d). Tuttavia, qualsiasi metodo ereditato può essere sovrascritto, definendolo nuovamente, adesso vien fatto per la classe derivata.

```
PROC print() OF d
  WriteF('extra_d_data = \d, ', self.extra_d_data)
  WriteF('b_data = \d\n', self.b_data)
ENDPROC
```

Con questa definizione extra, la stessa procedura main, vista prima, stamperebbe ora tutti i dati di d, ma solo l'elemento b_data di b. Ciò perchè la nuova definizione di print, interessa solo la classe d (e le classi derivate da d).

I metodi ereditati, spesso sono sovrascritti soltanto per aggiungere delle funzionalitá extra, come nel caso visto prima, dove abbiamo voluto che i dati extra e i dati derivati da b fossero stampati. Per questo scopo, possiamo usare l'operatore SUPER su una chiamata di metodo, per forzare l'uso del metodo della classe base, dove normalmente sarebbe usato il metodo della classe derivata. Così, la definizione del metodo print per la classe d, potrebbe chiamare il metodo print della classe b.

```
PROC print() OF d
  WriteF('extra_d_data = \d, ', self.extra_d_data)
  SUPER self.print()
ENDPROC
```

Bisogna stare attenti, però, perchè senza l'operatore SUPER, ne deriverebbe una chiamata ricorsiva al metodo print della classe d, anzichè una chiamata al metodo della classe base.

Proprio come i dati registrati di tipo d potrebbero essere usati al posto dei dati registrati di tipo b, ovunque questi venissero richiesti, gli oggetti della classe d possono essere usati al posto degli oggetti della classe b. La procedura seguente, stampa un messaggio e i dati dell'oggetto, usando il metodo print di b. (Naturalmente, solo i metodi chiamati dalla classe b possono essere usati in una tale procedura, finchè il puntatore p è di tipo PTR TO b.)

```
PROC msg_print(msg, p:PTR TO b)
  WriteF('Stampa di \s: ', msg)
  p.print()
ENDPROC

PROC main()
  DEF p_b:PTR TO b, p_d:PTR TO d
  NEW p_b, p_d
  p_b.b_data:=11
  p_d.b_data:=-3; p_d.extra_d_data:=3
  msg_print('p_b', p_b)
  msg_print('p_d', p_d)
ENDPROC
```

Non possiamo usare duff ora, finchè esso non è una classe mentre b lo è e msg_print attende un puntatore alla classe b. I soli altri oggetti che possono essere passati a msg_print sono quelli con le classi derivate da b e questo perchè p_d può essere stampato usando msg_print. Se uniamo i vari pezzi del codice e lo eseguiamo, vedremo cha la chiamata a print in msg_print, usa il metodo sovrascritto print, quando msg_print è chiamato con p_d come parametro. Cioè, il metodo corretto è chiamato anche se il puntatore non è di tipo PTR TO d. Questa proprietá è chiamata polymorphism (polimorfismo): differenti implementazioni di print possono essere chiamate in base al reale tipo dinamico di p. Segue quello che dovrebbe essere stampato:

```
Stampa di p_b: b_data = 11
Stampa di p_d: extra_d_data = 3, b_data = -3
```

L'ereditá non è limitata a una sola stratificazione: possiamo ricavare altre classi da b, altre classi da d, e così via. Per esempio, se la classe e, è derivata dalla classe d, allora essa erediterebbe tutti i dati di d, e tutti i metodi di d. Ciò significa che e, erediterebbe una versione più completa di print e potrebbe anche sovrascriverla di nuovo. In questo caso la classe, e, avrebbe due classi base, b ed d, ma sarebbero derivate direttamente da d (e indirettamente da b, attraverso d). Pertanto la classe d sarebbe conusciuta come la super classe di e, se questa è derivata direttamente da d. (la classe super di d è solo la sua classe base, la b.) Quindi l'operatore SUPER, in effetti, è usato per chiamare i metodi della classe super. In questo esempio, l'operatore SUPER può essere usato nei metodi di e, per chiamare i metodi di d.

L'implementazione di albero binario precedente (Vedi
sez. 2.11.3

```
              )
```

suggerisce un buon esempio per una classe gerarchica (una raccolta di classi attinenti l'ereditá). Una struttura base ad albero può essere incapsulata in una definizione di classe base e poi da questa possono essere ricavati i specifici tipi di albero (con differenti dati nei nodi). In realtá, la classe base tree (albero) definita qui di seguito è utile solo per ereditare, in quanto un albero è abbastanza inutile senza qualche dato per i nodi. Siccome è molto probabile che l'oggetto di classe tree non sará mai utile (ma gli oggetti derivati da tree lo sono), la classe tree è detta, una classe astratta (abstract).

```
OBJECT tree
  left:PTR TO tree, right:PTR TO tree
ENDOBJECT

PROC nodes() OF tree
  DEF tot=1
  IF self.left  THEN tot:=tot+self.left.nodes()
  IF self.right THEN tot:=tot+self.right.nodes()
ENDPROC tot

PROC leaves(show=FALSE) OF tree
  DEF tot=0
  IF self.left
    tot:=tot+self.left.leaves(show)
  ENDIF
  IF self.right
    tot:=tot+self.right.leaves(show)
  ELSEIF self.left=NIL
    IF show THEN self.print_node()
    tot++
  ENDIF
ENDPROC tot

PROC print_node() OF tree
  WriteF('<NULL> ')
ENDPROC

PROC print() OF tree
  IF self.left  THEN self.left.print()
  self.print_node()
  IF self.right THEN self.right.print()
ENDPROC
```

I metodi nodes e leaves ritornano rispettivamente il numero dei nodi e delle foglie dell'albero, con il metodo leaves che prevede un flag (segnalatore di vero o falso) per specificare se le foglie devono anche essere stampate. Questi metodi non dovrebbero mai aver bisogno di sovrascrivere in una classe derivata da tree, nè dovrebbero stampare quel che riguarda tree, stampando i nodi da sinistra a destra. Tuttavia il metodo print_node dovrebbe essere sovrascritto, così come avviene nell'albero di interi definito qui di seguito.

```
OBJECT integer_tree OF tree
  int
ENDOBJECT
```

```
PROC create(i) OF integer_tree
  self.int:=i
ENDPROC

PROC add(i) OF integer_tree
  DEF p:PTR TO integer_tree
  IF i < self.int
    IF self.left
      p:=self.left
      p.add(i)
    ELSE
      self.left:=NEW p.create(i)
    ENDIF
  ELSEIF i > self.int
    IF self.right
      p:=self.right
      p.add(i)
    ELSE
      self.right:=NEW p.create(i)
    ENDIF
  ENDIF
ENDPROC

PROC print_node() OF integer_tree
  WriteF('\d ', self.int)
ENDPROC
```

Questo è un buon esempio dell'uso del polimorfismo: possiamo implementare un albero che lavora con gli interi, semplicemente definendo i metodi appropriati. Il metodo leaves (della classe tree) quindi chiamerá automaticamente la versione integer_tree di print_node ogni qualvolta gli passiamo un oggetto integer_tree. Le definizioni di tree e integer_tree possono anche trovarsi in moduli differenti (Vedi sez. 2.12.5) e usando queste tecniche OOP, il modulo che contiene tree non avrebbe bisogno di essere ricompilato anche se una classe come integer_tree viene aggiunta o modificata. Questo ci dovrebbe far capire perchè la OOP è ottima per estendere e riutilizzare il codice: con tecniche tradizionali di programmazione, dovremmo adattare le funzioni di albero binario per giustificare gli interi e questo per ogni nuovo datatype.

Notare che l'uso ricorsivo del nuovo metodo add, deve essere chiamato per mezzo di un puntatore ausiliario p, della classe derivata. Questo perchè gli elementi left e right di tree, sono puntatori ad oggetti tree, e add non è un metodo di tree (il compilatore non accetterebbe il codice, dando un errore di sintassi, se provassimo ad accedere direttamente ad add in queste circostanze). Naturalmente se la classe tree avesse un metodo add, non ci sarebbe questo problema, ma che codice sarebbe con un tale metodo ?

Un metodo add non ha davvero senso per tree, ma se quasi tutte le classi derivate da tree avessero necessitá di un tale metodo, potrebbe essere comodo includerlo nella classe tree di base. Questo è lo scopo dei metodi astratti. Un metodo astratto può esistere solamente in una classe base in modo da poter essere sovrascritto in qualche classe derivata. Normalmente, tali metodi non hanno nessuna definizione sensata nella classe base, pertanto esiste una keyword speciale, EMPTY, che può essere usata per

definirli. Per esempio, il metodo add in tree sarebbe definito come segue:

```
PROC add(x) OF tree IS EMPTY
```

Con questa definizione il codice del metodo add per la classe integer_tree può essere semplificato. (Il puntatore ausiliario ,p, è ancora necessario per l'uso con NEW finchè self.left non è una variabile del puntatore.)

```
PROC add(i) OF integer_tree
  DEF p:PTR TO integer_tree
  IF i < self.int
    IF self.left
      self.left.add(i)
    ELSE
      self.left:=NEW p.create(i)
    ENDIF
  ELSEIF i > self.int
    IF self.right
      self.right.add(i)
    ELSE
      self.right:=NEW p.create(i)
    ENDIF
  ENDIF
ENDPROC
```

Questo esempio, tuttavia, non è l'esempio migliore di un metodo astratto, in quanto il metodo add, in ogni classe derivata da tree, ora deve prendere un singolo valore LONG come parametro, per essere compatibile. In generale, comunque, una classe che rappresenta un albero con i dati dei nodi di tipo t, vorrebbe davvero un metodo add per prendere un singolo parametro di tipo t. Il fatto che un valore LONG può rappresentare un puntatore a qualsiasi tipo, qui viene utile. Questo significa che la definizione di add potrebbe non essere così limitata, dopo tutto.

Il metodo print_node, ovviamente, è molto più adatto per essere un metodo astratto. Tale definizione stampa qualcosa di sciocco, in quanto in quel punto della spiegazione non sapevamo ancora nulla sui metodi astratti e a noi serviva definirlo nella classe base. Una migliore definizione renderebbe print_node astratto.

```
PROC print_node() OF tree IS EMPTY
```

E' abbastanza sicuro chiamare i metodi astratti, anche per gli oggetti di classe tree. Se un metodo è ancora astratto in qualsiasi classe (cioè, non è stato sovrascritto), allora chiamarlo su oggetti di quella classe, ha lo stesso effetto che chiamare una funzione che ritorna semplicemente zero (cioè diventa piccolissimo!).

La classe integer_tree può essere usata in questo modo:

```
PROC main()
  DEF t:PTR TO integer_tree
  NEW t.create(10)
  t.add(-10)
  t.add(3)
  t.add(5)
  t.add(-1)
```

```
  t.add(1)
  WriteF('t ha \d nodi, con \d foglie: ',
         t.nodes(), t.leaves())
  t.leaves(TRUE)
  WriteF('\n')
  WriteF('Contenuto di t: ')
  t.print()
  WriteF('\n')
  END t
ENDPROC
```

1.157 beginner.guide/Dati Nascosti in E

```
2.12.5 Dati Nascosti in E (Data-Hiding)
       ================================
```

Data-hiding (nascondere i dati) viene realizzato in E con i moduli. Questo significa che è efficace e intelligente definire le classi in moduli separati (o almeno, usare insieme, solo le classi strettamente attinenti in un modulo), accertandoci di esportare (EXPORT) solo le definizioni di cui abbiamo bisogno. Possiamo usare anche la keyword PRIVATE, nella definizione di qualsiasi OBJECT, per nascondere tutti gli elementi seguenti la keyword, al codice che usa il modulo (sebbene questo non ha effetto per il codice all'interno del modulo). La keyword PUBLIC può essere usata in modo simile per rendere gli elementi, che segueno la keyword, visibili (cioè, accessibili) di nuovo, come sarebbero per default. Per esempio, la seguente definizione OBJECT rende x, y, a, ed b privati (quindi visibili solo al codice all'interno dello stesso modulo) mentre p, q ed r pubblici (quindi visibili anche al codice esterno al modulo).

```
OBJECT rec
  p:INT
PRIVATE
  x:INT
  y
PUBLIC
  q
  r:PTR TO LONG
PRIVATE
  a:PTR TO LONG, b
ENDOBJECT
```

Per la classe set, probabilmente vorremmo rendere tutti i dati privati e tutti i metodi pubblici. In questo modo forzeremmo tutti i programmi che usassero questo modulo ad usare l'interfaccia fornita, piuttosto che a imbrogliarsi con le strutture di dati. Il seguente esempio è il codice completo per una semplice, inefficiente classe set e può essere compilato come modulo.

```
OPT MODULE  -> Definisce la classe 'set' come modulo
OPT EXPORT  -> Esporta tutto

/* I dati per la classe */
```

```
OBJECT set PRIVATE  -> Rende tutti i dati privati
  elements:PTR TO LONG
  maxsize, size
ENDOBJECT

/* Creazione del constructor */
/* Dimensione minima 1, massima 100000, default 100 */
PROC create(sz=100) OF set
  DEF p:PTR TO LONG
  self.maxsize:=IF (sz>0) AND (sz<100000) THEN sz ELSE 100 -> Controlla la ↩
     dimensione
  self.elements:=NEW p[self.maxsize]
ENDPROC

/* Constructor copy */
PROC copy(oldset:PTR TO set) OF set
  DEF i
  self.create(oldset.maxsize)  -> Chiama il metodo create!
  FOR i:=0 TO oldset.size-1  -> Copia elements
    self.elements[i]:=oldset.elements[i]
  ENDFOR
  self.size:=oldset.size
ENDPROC

/* Destructor */
PROC end() OF set
  DEF p:PTR TO LONG
  IF self.maxsize<>0  -> Controlla l'avvenuta allocazione
    p:=self.elements
    END p[self.maxsize]
  ENDIF
ENDPROC

/* Aggiunge un elemento */
PROC add(x) OF set
  IF self.member(x)=FALSE  -> Is it new? (Chiama il metodo member!)
    IF self.size=self.maxsize
      Raise("full")  -> Il set è giá pieno
    ELSE
      self.elements[self.size]:=x
      self.size:=self.size+1
    ENDIF
  ENDIF
ENDPROC

/* Verifica l'appartenenza */
PROC member(x) OF set
  DEF i
  FOR i:=0 TO self.size-1
    IF self.elements[i]=x THEN RETURN TRUE
  ENDFOR
ENDPROC FALSE

/* Verifica se è vuoto */
PROC empty() OF set IS self.size=0

/* Unisce (aggiunge) un altro set */
```

```
PROC union(other:PTR TO set) OF set
  DEF i
  FOR i:=0 TO other.size-1
    self.add(other.elements[i])  -> Chiama il metodo add!
  ENDFOR
ENDPROC

/* Stampa i contenuti */
PROC print() OF set
  DEF i
  WriteF('{ ')
  FOR i:=0 TO self.size-1
    WriteF('\d ', self.elements[i])
  ENDFOR
  WriteF('}')
ENDPROC
```

Questa classe può essere usata in un altro modulo o in programma nel modo seguente:

```
MODULE '*set'

PROC main() HANDLE
  DEF s=NIL:PTR TO set
  NEW s.create(20)
  s.add(1)
  s.add(-13)
  s.add(91)
  s.add(42)
  s.add(-76)
  IF s.member(1) THEN WriteF('1 è un member\n')
  IF s.member(11) THEN WriteF('11 è un member\n')
  WriteF('s = ')
  s.print()
  WriteF('\n')
EXCEPT DO
  END s
  SELECT exception
  CASE "NEW"
    WriteF('Fuori memoria\n')
  CASE "full"
    WriteF('Set è pieno\n')
  ENDSELECT
ENDPROC
```

1.158 beginner.guide/Introduzione agli Esempi

```
              3.1 Introduzione agli Esempi
    ************************
```

In questo capitolo tratteremo alcuni esempi un po' più complessi di quelli visti finora. Comunque, nessuno di essi è veramente complicato, quindi non dovrebbero essere difficili da capire. In ogni esempio, le parti degne di nota sono spiegate, un aiuto verrá anche da alcuni commenti inseriti nel

codice, anzi i programmi più complicati fanno largo uso di un commento descrittivo inserito al punto giusto.

Tutti gli esempi girano su ogni Amiga, tranne quello che usa ReadArgs che è una funzione dell'AmigaDOS 2.0. E` davvero il caso di upgradare il tuo sistema all'AmigaDOS 2.0 (o superiore) se stai usando ancora le versioni precedenti. L'esempio ReadArgs può dare solo un'idea della potenza e della amichevolezza delle funzioni di sistema più recenti. Se sei tanto fortunato da possedere un A4000 o una macchina accelerata, allora l'esempio timing (temporizzazione) dará i migliori risultati (ossia i più veloci).

Fornita con questa Guida dovrebbe esserci una directory con i sorgenti della maggior parte degli esempi. Segue la lista completa:

```
simple.e (semplice.e)
     Il programma semplice dell'introduzione. Vedi
                sez. 1.1.1
                while.e
     Il loop WHILE. Vedi
                sez. 1.4.2.2
                address.e
     Il programma che stampa gli indirizzi di alcune variabili.
     Vedi
                sez. 2.4.2.4
                static.e
     Il problema dei dati statici. Vedi
                sez. 2.4.5.7
                static2.e
     La prima soluzione al problema dei dati statici. Vedi
                sez. 2.4.5.7
                except.e
     Un esempio di exception handler. Vedi
                sez. 2.8.2
                except2.e
     Un'altro esempio di exception handler. Vedi
                sez. 2.8.2
                static3.e
     La seconda soluzione al problema dei dati statici usando NEW.
     Vedi
                sez. 2.9.4.3
                float.e
     Il programma di esempio dei numeri in virgola mobile. Vedi
                sez. 2.10.3
                bintree.e
     L'esempio di albero binario. Vedi
                sez. 2.11.3
                tree.e
     Le classi tree ed integer_tree, come modulo. Vedi
                sez. 2.12.4
                tree-use.e
     Un programma per usare la classe integer_tree. Vedi
                sez. 2.12.4
                set.e
     La semplice, inefficiente classe set, come modulo. Vedi
                sez. 2.12.5
                set-use.e
     Un programma per usare la classe set. Vedi
```

```
            sez. 2.12.5
            csv-estr.e
     Il programma di lettura CSV, usando le E-strings. Vedi
            sez. 3.2
            csv-norm.e
     Il programma di lettura CSV, usando le stringhe normali. Vedi
            sez. 3.2
            csv-buff.e
     Il programma di lettura CSV, usando le stringhe normali e un grande
     buffer. Vedi
            sez. 3.2
            csv.e
     Il programma di lettura CSV, usando le stringhe normali, un grande
     buffer e un exception handler. Vedi
            sez. 3.2
            timing.e
     L'esempio di temporizzazione. Vedi
            sez. 3.3
            args.e
     L'esempio di analisi dell'argomento, per qualsiasi AmigaDOS.
     Vedi
            sez. 3.4.1
            args20.e
     L'esempio di analisi dell'argomento, per qualsiasi AmigaDOS 2.0 e
     superiore. Vedi
            sez. 3.4.2
            gadgets.e
     L'esempio dei gadgets. Vedi
            sez. 3.5.1
            idcmp.e
     Esempio di IDCMP e gadgets. Vedi
            sez. 3.5.2
            graphics.e
     L'esempio di grafica. Vedi
            sez. 3.5.3
            screens.e
     L'esempio degli schermi, senza un exception handler. Vedi
            sez. 3.5.4
            screens2.e
     L'esempio degli schermi con un exception handler. Vedi
            sez. 3.5.4
            dragon.e
     L'esempio di ricorsione, curva del drago. Vedi
            sez. 3.6
```

1.159 beginner.guide/String Handling e I-O

```
3.2 String Handling e I/O
    **********************
```

Questa sezione mostra come usare le normali stringhe e le E-string ed anche come leggere i dati da un file. I programmi usano alcune funzioni stringa e fanno un corretto (ma differente) uso della memoria dove possibile. I punti importanti da capire sono:

* La differenza fra le stringhe normali e le E-strings.

* L'allocazione di memoria necessaria per le E-strings.

* L'inutile, ma consigliata, disallocazione della memoria riservata alle E-string, quando non è più necessaria. La disallocazione potrebbe avvenire automaticamente alla fine del programma, ma si potrebbe sprecare molta memoria nel frattempo e se i dati da inserire sono molti, allora la si potrebbe esaurire facilmente.

* Il modo in cui parti di una E-string possono essere facilmente trasformate in stringhe normali (e viceversa).

* Come si usano gli exception handler in un programma.

Il problema da risolvere è quello di leggere un file CSV (comma separated variables - variabili separate da una virgola), che è un file di formato standard per i database e i spreadsheets. Il formato è molto semplice: ogni record è una linea (ossia con un line-feed finale) e ogni campo in un record è separato da una virgola. Per rendere tale esempio molto più semplice, non useremo campi che contengano virgole (normalmente ciò richiederebbe che il campo sia fra virgolette). Di conseguenza, un tipico file di input sarebbe simile al seguente:

```
Campo1,Campo2,Campo3
10,19,-3
fred,barney,wilma
,,ultimo
primo,,
```

In questo esempio tutti i records hanno tre campi, come è ben evidenziato dalla prima linea (cioè, il primo record). Gli ultimi due records possono sembrare un po' strani, ma così mostriamo semplicemente come i campi possono essere vuoti. Nell'ultimo record tutti i campi tranne primo, sono vuoti, mentre nel penultimo record tutti i campi tranne ultimo, sono vuoti.

Adesso conosciamo il formato del file da leggere. Per poter operare su di un file, dobbiamo prima aprirlo, usando la funzione Open (della dos.library), per leggerne le linee, useremo la funzione ReadStr (built-in). Ci sono quattro versioni del programma che legge un file CSV: due di queste leggono i dati linea per linea, le altre due leggono il file tutto in una volta. Delle due versioni che leggono il file linea per linea, una manipola le linee lette, come E-strings, mentre l'altra le manipola come stringhe normali. L'uso delle stringhe normali è (discutibilmente) più avanzato dell'uso delle E-strings, se vengono utilizzati dei trucchetti per un uso corretto della memoria. Comunque tali programmi non sono stati ideati per mostrare cosa sia meglio fra le E-strings e le normali stringhe, ma piuttosto intendono mostrare il loro corretto uso.

```
/* Una sufficiente dimensione per il buffer del record */
CONST BUFFERSIZE=512

PROC main()
  DEF filehandle, status, buffer[BUFFERSIZE]:STRING, filename
  filename:='datafile'
  IF filehandle:=Open(filename, OLDFILE)
```

```
        REPEAT
          status:=ReadStr(filehandle, buffer)
/*Questo è il modo per controllare se ReadStr() in effetti legge qualcosa*/
          IF buffer[] OR (status<>-1) THEN process_record(buffer)
        UNTIL status=-1
        /* Se Open() riesce allora dobbiamo Close() il file */
        Close(filehandle)
      ELSE
        WriteF('Errore: Apertura fallita di "\s"\n', filename)
      ENDIF
    ENDPROC

    PROC process_record(line)
      DEF i=1, start=0, end, len, s
      /* Mostra l'intera linea da processare */
      WriteF('Processo del record: "\s"\n', line)
      REPEAT
   /* Trova l'indice di una virgola dopo l'indice start */
        end:=InStr(line, ',', start)
        /* La lunghezza è l'indice finale meno l'indice start */
        len:=(IF end<>-1 THEN end ELSE EstrLen(line))-start
        IF len>0
          /* Alloca la lunghezza corretta della E-string */
          IF s:=String(len)
            /* Copia parte della linea nella E-string s */
            MidStr(s, line, start, len)
            /* A questo punto possiamo fare qualcosa di utile... */
            WriteF('\t\d) "\s"\n', i, s)
            /* Abbiamo finito con la E-string quindi la disallochiamo */
            DisposeLink(s)
          ELSE
          /* Non è un errore fatale se la chiamata a String() fallisce */
            WriteF('\t\d) Memoria esaurita! (len=\d)\n', len)
          ENDIF
        ELSE
          WriteF('\t\d) Campo Vuoto\n', i)
        ENDIF
        /* Il nuovo start è dopo la end che abbiamo trovato */
        start:=end+1
        INC i
      /* Quando la virgola non viene trovata abbiamo finito */
      UNTIL end=-1
    ENDPROC
```

Ci sono alcune cose da notare in questo programma:

* Una capiente E-string, buffer, viene usata per contenere ogni linea prima di essere processata. Se un record eccede la dimensione della E-string, allora ReadStr leggerá il record solo parzialmente la prima volta, il resto verrá letto al secondo passaggio da ReadStr (ReadStr è posto in un ciclo REPEAT..UNTIL). Tuttavia, per il programma, ogni chiamata a ReadStr corrisponde alla lettura di un intero record, quindi in un caso come quello appena descritto, i records verrebbero letti in modo errato. Questa situazione diventa una limitazione del programma pertanto si renderebbe necessario avvisare gli utenti del programma in modo che non eccedano con la lunghezza delle linee nei datafiles.

* Il nome del file è datafile in quanto così sappiamo subito con che genere di file abbiamo a che fare. Un programma più flessibile permetterebbe di passare tale nome come argomento (Vedi sez. 3.4).

* ReadStr può ritornare -1 per indicare un errore (normalmente quando viene raggiunta la fine del file), ma così ancora non sappiamo se la E-string ha letto qualcosa o meno. Il controllo sulla E-string e sul valore dell'errore è il modo corretto per decidere se ReadStr in effetti non ha letto nulla dal file.

* Studiare attentamente la manipolazione degli indici di stringa start ed end ed anche il calcolo della lunghezza di una parte della stringa.

* MidStr è usato per copiare un campo da un record, quindi una E-string deve essere usata per contenerlo.

* La E-string s è valida solo tra l'allocazione riuscita come stringa e il DisposeLink. Non sarebbe corretto provare, per esempio, a stampare la stringa in qualsiasi altro punto del programma. D'altra parte, un programma più complicato potrebbe avere la necessitá di conservare tutti i dati e di conseguenza potrebbe essere inappropriato disallocare la E-string in tale punto. In tal caso, potrebbe essere conservato un puntatore alla E-string e questi potrebbe essere valido per il resto del programma.

* L'allocazione fatta da String è seguita da vicino dalla disallocazione fatta da DisposeLink. Ciò ci fa capire che una singola E-string ha potuto essere allocata e usata ripetutamente (come lo è buffer) a causa della semplice natura di questo esempio.

Per cambiare l'esempio in modo che possa usare le normali stringhe (usando la memoria in modo corretto), dobbiamo modificare solo la procedura process_record. Le differenze degne di nota sono:

* Piccole parti della E-string, buffer, sono trasformate in normali stringhe terminandole con NIL, quando necessario. Ciò implica il cambiamento della virgola quando viene trovata.

* Nessuna nuova memoria viene allocata, ma piuttosto il, buffer, di memoria viene riutilizzato (come appena descritto). Questo va bene per questo esempio, sebbene si avverte la necessitá di copiare i campi dopo che un record viene processato se i contenuti di buffer sono cambiati da ReadStr.

```
PROC process_record(line)
  DEF i=1, start=0, end, s
  /* Mostra l'intera linea da processare */
  WriteF('Processo del record: "\s"\n', line)
  REPEAT
    /* Trova l'indice di una virgola dopo l'indice start */
    end:=InStr(line, ',', start)
    /* Se viene trovata una virgola allora terminare con NIL */
    IF end<>-1 THEN line[end]:=NIL
    /* Punta all'inizio del campo */
    s:=line+start
    IF s[]
```

```
        /* A questo punto possiamo fare qualcosa di utile... */
        WriteF('\t\d) "\s"\n', i, s)
      ELSE
        WriteF('\t\d) Campo Vuoto\n', i)
      ENDIF
      /* Il nuovo start è dopo la end che abbiamo trovato */
      start:=end+1
      INC i
    /* Quando la virgola non viene trovata abbiamo finito */
    UNTIL end=-1
  ENDPROC
```

Le prossime due versioni del programma, fondamentalmente, sono uguali fra loro: entrambe leggono l'intero file in grande buffer, allocato dinamicamente e poi processano i dati. La seconda delle due versioni fa uso anche delle exceptions per rendere il programma molto più leggibile. Le differenze con le versioni precedenti sono:

* La procedura main calcola la lunghezza dei dati nel file e poi usa New per allocare dinamicamente qualche zona di memoria per contenerli.

* I dati letti vengono terminati con un NIL, di conseguenza possono essere processati con sicurezza come una stringa normale (molto lunga).

* La procedura process_buffer, divide i dati letti in stringhe normali, una per ogni linea di dati.

```
  PROC main()
    DEF buffer, filehandle, len, filename
    filename:='datafile'
    /* Legge quanto è lungo il file */
    IF 0<(len:=FileLength(filename))
 /* Alloca solo lo spazio necessario per i dati + un NIL finale */
      IF buffer:=New(len+1)
        IF filehandle:=Open(filename, OLDFILE)
       /* Legge l'intero file, controllando la quantitá da leggere */
          IF len=Read(filehandle, buffer, len)
            /* Terminare buffer con un NIL solo in caso... */
            buffer[len]:=NIL
            process_buffer(buffer, len)
          ELSE
            WriteF('Errore: Errore leggendo il file\n')
          ENDIF
          /* Se Open() riesce allora dobbiamo Close() il file */
          Close(filehandle)
        ELSE
          WriteF('Errore: Apertura fallita di "\s"\n', filename)
        ENDIF
      /* Disallochiamo il buffer (non necessario in questo esempio) */
        Dispose(buffer)
      ELSE
        WriteF('Errore: Insufficiente memoria per caricare il file\n')
      ENDIF
    ELSE
      WriteF('Errore: "\s" è un file vuoto\n', filename)
    ENDIF
```

```
ENDPROC

/* buffer è visto come una normale stringa se è terminato con NIL */
PROC process_buffer(buffer, len)
  DEF start=0, end
  REPEAT
    /* Trova l'indice di un linefeed dopo l'indice di partenza */
    end:=InStr(buffer, '\n', start)
    /* Se un linefeed viene trovato allora terminare con un NIL */
    IF end<>-1 THEN buffer[end]:=NIL
    process_record(buffer+start)
    start:=end+1
 /* Abbiamo finito se siamo alla fine o non ci sono più linefeeds */
  UNTIL (start>=len) OR (end=-1)
ENDPROC

PROC process_record(line)
  DEF i=1, start=0, end, s
  /* Mostra l'intera linea da processare */
  WriteF('Processing record: "\s"\n', line)
  REPEAT
    /* Trova l'indice di una virgola dopo l'indice start */
    end:=InStr(line, ',', start)
    /* Se viene trovata una virgola allora terminare con NIL */
    IF end<>-1 THEN line[end]:=NIL
    /* Punta all'inizio del campo */
    s:=line+start
    IF s[]
      /* A questo punto possiamo fare qualcosa di utile... */
      WriteF('\t\d) "\s"\n', i, s)
    ELSE
      WriteF('\t\d) Campo Vuoto\n', i)
    ENDIF
    /* Il nuovo start è dopo la end che abbiamo trovato */
    start:=end+1
    INC i
  /* Quando la virgola non viene trovata abbiamo finito */
  UNTIL end=-1
ENDPROC
```

Il programma adesso è abbastanza disordinato, con molti casi di errore nella procedura main. Possiamo però rimediare facilmente a questa situazione usando un exception handler e alcune exception automatiche.

```
/* Alcune costanti per le exceptions (ERR_NONE è zero: no errore) */
ENUM ERR_NONE, ERR_LEN, ERR_NEW, ERR_OPEN, ERR_READ

/* Rendiamo automatica qualche exceptions */
RAISE ERR_LEN  IF FileLength()<=0,
      ERR_NEW  IF New()=NIL,
      ERR_OPEN IF Open()=NIL

PROC main() HANDLE
  /* Nota la prudente inizializzazione di buffer e filehandle */
  DEF buffer=NIL, filehandle=NIL, len, filename
  filename:='datafile'
```

```
      /* Legge quanto è lungo il file */
      len:=FileLength(filename)
      /* Alloca solo lo spazio necessario per i dati + un NIL finale */
      buffer:=New(len+1)
      filehandle:=Open(filename, OLDFILE)
      /* Legge l'intero file, controllando la quantitá da leggere */
      IF len<>Read(filehandle, buffer, len) THEN Raise(ERR_READ)
      /* Terminare buffer con un NIL solo in caso... */
      buffer[len]:=NIL
      process_buffer(buffer, len)
  EXCEPT DO
    /* Entrambe queste sono sicure grazie alla inizializzazione */
    IF buffer THEN Dispose(buffer)
    IF filehandle THEN Close(filehandle)
    /* Rapporto sull'errore (se ne capita uno) */
    SELECT exception
    CASE ERR_LEN;   WriteF('Errore: "\s" è un file vuoto\n', filename)
    CASE ERR_NEW;   WriteF('Errore: Insufficiente memoria per caricare il file\n')
    CASE ERR_OPEN;  WriteF('Errore: Apertura fallita di "\s"\n', filename)
    CASE ERR_READ;  WriteF('Errore: Errore leggendo il file\n')
    ENDSELECT
    ENDPROC
```

Il codice ora è molto più chiaro e la maggior parte degli errori vengono gestiti automaticamente. Nota che l'exception handler viene usato anche se non ci sono errori (grazie al DO dopo EXCEPT). Questo perchè quando il programma termina, bisogna in ogni caso disallocare le risorse che sono state allocate (errori o no), così questo codice ha lo stesso effetto del precedente. La disallocazione condizionale (di buffer, per esempio) è resa sicura da una appropriata inizializzazione.

Se te la senti, un piccolo esercizio sarebbe di provare a scrivere un programma simile, ma usando il modulo tools/file che viene fornito con la distribuzione standard dell'Amiga E. Naturalmente, prima avrai bisogno di leggere la documentazione allegata, ma dovresti accorgerti che tale modulo interagisce molto semplicemente con i file.

1.160 beginner.guide/Espressioni di Temporizzazione

```
                3.3 Espressioni di Temporizzazione
    ******************************
```

Dovresti ricordare una procedura di temporizzazione incompleta nel Capitolo 2. Vedi

```
                sez. 2.5.7.1
```

. Questa sezione mostra la versione completa di quell'esempio. La parte che mancava nell'esempio, era come determinare il tempo di sistema e usarlo per calcolare il tempo usato dalle chiamate a Eval. Quindi le parti da notare in questo esempio sono:

* L'uso della funzione di sistema di Amiga DateStamp (della dos.library) (Si ha davvero la necessitá del 'Rom Kernel Reference Manuals' e del 'AmigaDOS Manual' per capire le funzioni di sistema.)

* L'uso del modulo dos/dos per includere le definizioni dell'object datestamp e della costante TICKS_PER_SECOND. (Ci sono cinquanta ticks al secondo.)

* L'uso della procedura repeat per usare Eval più volte per ogni espressione (in modo che venga usato più tempo dalle chiamate!).

* La temporizzazzione della valutazione di 0, per calcolare il sovraccarico delle procedure di chiamata e di loop. Tale valore viene conservato nella variabile offset la prima volta che la procedura test è chiamata. L'espressione 0 dovrebbe usare una quantitá trascurabile di tempo, pertanto il numero temporeggiato di ticks è effettivamente il tempo usato dalla procedura di chiamata e dai calcoli nei loop. Sottraendo questo tempo dagli altri tempi abbiamo il tempo esatto usato dalle espressioni, relativo da una all'altra. (Un grazie a Wouter per l'idea dell'offset.)

* L'uso di Forbid e Permit per disabilitare temporaneamente il multi-tasking, in modo che la CPU possa calcolare solo le espressioni (piuttosto che far calcoli anche per l'output di schermo, per altri programmi, ecc.).

* L'uso di CtrlC e CleanUp per permettere all'utente di fermare il programma se vuole.

* L'uso dell'opzione LARGE (usando OPT) per ottenere un eseguibile che usa grandi dati e un codice modello. Ciò sembra che aiuti le temporizzazioni ad essere meno suscettibili alle variazioni dovute, per esempio, alle ottimizzazioni, ottenendo così dei paragoni più validi. Vedi il 'Reference Manual' per maggiori dettagli.

Seguono anche alcuni outputs d'esempio. Il primo è stato ottenuto da un A1200 con 2MB di Chip RAM e 4MB di Fast RAM. Il secondo è stato ottenuto da un A500Plus con 2MB Chip RAM. Entrambi sono stati ottenuti con la costante LOTS_OF_TIMES uguale a 500,000, ma si potrebbe aver bisogno di aumentare questo numero per paragonare, ad esempio, un A4000/040 ad un A4000/030. Tuttavia, 500,000 dá un'attesa piuttosto lunga per ottenere i risultati con un A500.

```
MODULE 'dos/dos'

CONST TICKS_PER_MINUTE=TICKS_PER_SECOND*60, LOTS_OF_TIMES=500000

DEF x, y, offset

PROC fred(n)
  DEF i
  i:=n+x
ENDPROC

/* Ripete la valutazione di un'espressione */
PROC repeat(exp)
  DEF i
  FOR i:=0 TO LOTS_OF_TIMES
    Eval(exp)  /* Valuta l'espressione */
  ENDFOR
ENDPROC
```

```
  /* Misura un'espressione e assegna l'offset se non è stato giá fatto */
     PROC test(exp, message)
       DEF t
       IF offset=0 THEN offset:=time(`0)  /* Calcola l'offset */
       t:=time(exp)
       WriteF('\s:\t\d ticks\n', message, t-offset)
     ENDPROC

     /* Misura le chiamate ripetute, e calcola il numero di ticks */
     PROC time(x)
       DEF ds1:datestamp, ds2:datestamp
       Forbid()
       DateStamp(ds1)
       repeat(x)
       DateStamp(ds2)
       Permit()
       IF CtrlC() THEN CleanUp(1)
     ENDPROC ((ds2.minute-ds1.minute)*TICKS_PER_MINUTE)+ds2.tick-ds1.tick

     PROC main()
       x:=9999
       y:=1717
       test(`x+y,     'Addizione')
       test(`y-x,     'Sottrazione')
       test(`x*y,     'Moltiplicazione')
       test(`x/y,     'Divisione')
       test(`x OR y,  'Bitwise OR')
       test(`x AND y, 'Bitwise AND')
       test(`x=y,     'Uguaglianza')
       test(`x<y,     'Minore di')
       test(`x<=y,    'Minore di o uguale a')
       test(`y:=1,    'Assegnazione di 1')
       test(`y:=x,    'Assegnazione di x')
       test(`y++,     'Incremento')
       test(`IF FALSE THEN y ELSE x, 'IF FALSE')
       test(`IF TRUE THEN y ELSE x,  'IF TRUE')
       test(`IF x THEN y ELSE x,     'IF x')
       test(`fred(2),  'fred(2)')
     ENDPROC
```

Questo è l'output dell'A1200:

```
     Addizione: 22 ticks
     Sottrazione:       22 ticks
     Moltiplicazione:   69 ticks
     Divisione: 123 ticks
     Bitwise OR:        33 ticks
     Bitwise AND:       27 ticks
     Uguaglianza:       44 ticks
     Minore di: 43 ticks
     Minore di o uguale a:      70 ticks
     Assegnazione di 1: 9 ticks
     Assegnazione di x: 38 ticks
     Incremento:        23 ticks
     IF FALSE:  27 ticks
     IF TRUE:   38 ticks
```

```
     IF x:      44 ticks
     fred(2):   121 ticks

Paragona questo all'output dell'A500Plus:

     Addizione: 118 ticks
     Sottrazione:       117 ticks
     Moltiplicazione:   297 ticks
     Divisione: 643 ticks
     Bitwise OR:        118 ticks
     Bitwise AND:       117 ticks
     Uguaglianza:       164 ticks
     Minore di: 164 ticks
     Minore di o uguale a:      164 ticks
     Assegnazione di 1: 60 ticks
     Assegnazione di x: 102 ticks
     Incremento:        134 ticks
     IF FALSE:  118 ticks
     IF TRUE:   164 ticks
     IF x:      193 ticks
     fred(2):   523 ticks

Si evidenzia, ammesso che fosse necessario, che l'A1200 è
approssimativamente cinque volte più veloce di un A500 e che non si stanno
usando le istruzioni speciali della CPU 68020!
```

1.161 beginner.guide/Analisi degli Argomenti

```
                3.4 Analisi degli Argomenti
    ***********************

In questa sezione mostreremo due esempi. Uno è per qualsiasi AmigaDOS e
l'altro è per AmigaDOS 2.0 e superiori. Entrambi spiegano come analizzare
gli argomenti di un programma. Se il tuo programma viene lanciato dalla
Shell/CLI, gli argomenti seguono il nome del comando sulla linea di
comando, ma se viene lanciato da Workbench (cioè cliccando due volte su
un'icona per il programma) allora gli argomenti sono quelle icone che
vengono selezionate contemporaneamente (Vedi il manuale Workbench per
maggiori dettagli).


                 Per ogni Amiga DOS

                 AmigaDOS 2.0 (e superiore)
```

1.162 beginner.guide/Any AmigaDOS

3.4.1 Per ogni AmigaDOS

Questo primo esempio funziona con qualsiasi AmigaDOS. La prima cosa fatta è l'assegnazione di wbmessage a un corretto tipo di puntatore. Nello stesso tempo possiamo controllare se esso è NIL (cioè se il programma è stato lanciato da Workbench o meno). Se il programma non viene lanciato da Workbench, vengono stampati gli argomenti in arg, altrimenti dobbiamo fare affidamento sul fatto che wbmessage è realmente un puntatore ad un object wbstartup (definito nel modulo workbench/startup), avendo accesso così alla lista degli argomenti. Poi per ogni argomento nella lista, dobbiamo controllare il percorso (lock) fornito con l'argomento. Il percorso per essere corretto deve indicare dove trovare il file argomento. Il nome nell'argomento è solo un nome di file, non un percorso completo, pertanto per leggere tale file dobbiamo cambiare la directory corrente alla directory del file stesso. Quando abbiamo il percorso corretto e abbiamo cambiato la directory a quella che ci interessa, possiamo ottenere la lunghezza del file (usando FileLength) e stamparla.

```
MODULE 'workbench/startup'

PROC main()
  DEF startup:PTR TO wbstartup, args:PTR TO wbarg, i, oldlock, len
  IF (startup:=wbmessage)=NIL
    WriteF('Lanciato dalla Shell/CLI\n   Argomenti: "\s"\n', arg)
  ELSE
    WriteF('Lanciato dal Workbench\n')
    args:=startup.arglist
    FOR i:=1 TO startup.numargs  /* Loop del numero di argomenti */
      IF args[].lock=NIL
        WriteF('  Argomenti \d: "\s" (no lock)\n', i, args[].name)
      ELSE
        oldlock:=CurrentDir(args[].lock)
        len:=FileLength(args[].name)  /* Fa qualcosa con il file */
        IF len=-1
          WriteF('  Argomenti \d: "\s" (il file non esiste)\n',
                 i, args[].name)
        ELSE
          WriteF('  Argomento \d: "\s", la lunghezza del file è \d bytes\n',
                 i, args[].name, len)
        ENDIF
        CurrentDir(oldlock) /* Importante: ripristinare dir corrente */
      ENDIF
      args++
    ENDFOR
  ENDIF
ENDPROC
```

Quando eseguirai questo programma noterai una piccola differenza tra arg e il messaggio Workbench: arg non contiene il nome del programma, soltanto gli argomenti, mentre il primo argomento nella lista degli argomenti Workbench è il nome del programma. Se vuoi, puoi semplicemente ignorare il primo argomento nella lista Workbench.

1.163 beginner.guide/AmigaDOS 2.0 (e superiore)

```
3.4.2 AmigaDOS 2.0 (e superiore)
      ==========================
```

Questo secondo programma può essere usato come la parte Shell/CLI del precedente programma per ottenere un'analisi migliore della linea di comando. Può essere usato solo con AmigaDOS 2.0 e superiore (ossia, con OSVERSION 37 o maggiore). Il template FILE/M usato con ReadArgs da un'analisi della linea di comando simile all'array del C argv. Il template può essere molto più interessante di questo, ma per maggiori dettagli bisogna consultare il 'AmigaDOS Manual'.

```
OPT OSVERSION=37

PROC main()
  DEF templ, rdargs, args=NIL:PTR TO LONG, i
  IF wbmessage=NIL
    WriteF('Lanciato dalla Shell/CLI\n')
    templ:='FILE/M'
    rdargs:=ReadArgs(templ,{args},NIL)
    IF rdargs
      IF args
        i:=0
        WHILE args[i]  /* Loop del numero di argomenti */
          WriteF('   Argomento \d: "\s"\n', i, args[i])
          i++
        ENDWHILE
      ENDIF
      FreeArgs(rdargs)
    ENDIF
  ENDIF
ENDPROC
```

Come possiamo vedere la chiamata a ReadArgs con questo template è un array di nomi di file. Lo speciale uso delle virgolette con i nomi dei files serve a trattare tali nomi correttamente (cioè quando usiamo le " per racchiudere un nome di file che contiene degli spazi) e abbiamo bisogno di farlo se usiamo il metodo arg.

1.164 beginner.guide/Gadgets IDCMP e Grafica

```
                3.5 Gadgets IDCMP e Grafica
    ************************
```

Questa sezione mostra tre esempi. Il primo spiega come aprire una finestra con alcuni gadgets. Il secondo spiega come decifrare i messaggi Intuition che arrivano attraverso IDCMP. Il terzo spiega le funzioni grafiche, disegnando qualcosa.

```
                 Gadgets
```

Messaggi IDCMP

Grafica

Schermi

1.165 beginner.guide/Gadgets

3.5.1 Gadgets
=======

Il seguente programma spiega come creare una lista gadget e come utilizzarla:

```
MODULE 'intuition/intuition'

CONST GADGETBUFSIZE = 4 * GADGETSIZE

PROC main()
  DEF buf[GADGETBUFSIZE]:ARRAY, next, wptr
  next:=Gadget(buf,  NIL, 1, 0, 10, 30, 50, 'Ciao')
  next:=Gadget(next, buf, 2, 3, 70, 30, 50, 'Gente')
  next:=Gadget(next, buf, 3, 1, 10, 50, 50, 'da')
  next:=Gadget(next, buf, 4, 0, 70, 50, 70, 'gadgets')
  wptr:=OpenW(20,50,200,100, 0, WFLG_ACTIVATE,
              'Gadgets in una window',NIL,1,buf)
  IF wptr         /* Controlla se abbiamo aperto una window */
    Delay(500)    /* Aspetta un po' */
    CloseW(wptr)  /* Chiude la window */
  ELSE
    WriteF('Errore -- non posso aprire la window!')
  ENDIF
ENDPROC
```

Sono stati creati quattro gadgets usando un'appropriata dimensione di array come buffer. Questi gadgets sono passati a OpenW (l'ultimo parametro). Se la finestra è stata aperta, viene usato un piccolo ritardo per farla rimanere un po' a video prima di chiuderla e terminare il programma. Delay è una funzione di sitema di Amiga della DOS library, Delay(n) aspetta n/50, pertanto la finestra rimane per 10 secondi, che è un tempo sufficiente per provare i gadgets e vedere di che tipo sono. Il prossimo esempio spiegherá un modo migliore per decidere quando terminare il programma (usando il gadget di chiusura standard).

1.166 beginner.guide/Messaggi IDCMP

3.5.2 Messaggi IDCMP
==========

Questo programma mostra come usare WaitIMessage con un gadgets.

```
MODULE 'intuition/intuition'

CONST GADGETBUFSIZE = GADGETSIZE, OURGADGET = 1

PROC main()
  DEF buf[GADGETBUFSIZE]:ARRAY, wptr, class, gad:PTR TO gadget
  Gadget(buf, NIL, OURGADGET, 1, 10, 30, 100, 'Premi Me')
  wptr:=OpenW(20,50,200,100,
              IDCMP_CLOSEWINDOW OR IDCMP_GADGETUP,
              WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
              'Messagi gadget nella window',NIL,1,buf)
  IF wptr              /* Controlla se abbiamo aperto una window */
    WHILE (class:=WaitIMessage(wptr))<>IDCMP_CLOSEWINDOW
      gad:=MsgIaddr()  /* Il nostro gadget è stato cliccato? */
      IF (class=IDCMP_GADGETUP) AND (gad.userdata=OURGADGET)
        TextF(10,60,
              IF gad.flags=0 THEN 'Gadget off ' ELSE 'Gadget on   ')
      ENDIF
    ENDWHILE
    CloseW(wptr)       /* Chiude la window */
  ELSE
    WriteF('Errore -- non posso aprire la window!')
  ENDIF
ENDPROC
```

Il gadget ritorna il suo stato, quando lo clicchiamo, usando la funzione TestF Vedi
sez. 2.6.3.3
. Il solo modo per abbandonare il programma è usare il gadget di chiusura della finestra. L'object gadget è definito nel modulo intuition/intuition e la parte iaddr del messaggio IDCMP è un puntatore al nostro gadget se il messaggio è un messaggio gadget. L'elemento userdata del gadget, identifica il gadget cliccato, l'elemento flags è zero se il gadget booleano è spento (non selezionato) oppure diverso da zero se il gadget booleano è acceso (selezionato).

1.167 beginner.guide/Grafica

3.5.3 Grafica
=======

Il seguente programma spiega come usare le varie funzioni grafiche.

```
MODULE 'intuition/intuition'

PROC main()
  DEF wptr, i
  wptr:=OpenW(20,50,200,100,IDCMP_CLOSEWINDOW,
              WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
              'Finestra demo per la grafica',NIL,1,NIL)
  IF wptr  /* Controlla se abbiamo aperto una window */
```

```
        Colour(1,3)
        TextF(20,30,'Hello World')
        SetTopaz(11)
        TextF(20,60,'Hello World')
        FOR i:=10 TO 150 STEP 8  /* Traccia alcuni punti */
          Plot(i,40,2)
        ENDFOR
        Line(160,40,160,70,3)
        Line(160,70,170,40,2)
        Box(10,75,160,85,1)
        WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
        ENDWHILE
        CloseW(wptr)
      ELSE
        WriteF('Errore -- non posso aprire la window!\n')
      ENDIF
    ENDPROC
```

Innanzitutto viene aperta una piccola finestra con un gadget di chiusura attivando la window (in modo che la finestra sia selezionata). Se premiamo sul gadget di chiusura, IDCMP lo comunicherá e questo è il solo modo per abbondonare il programma. Le funzioni grafiche sono usate nel modo seguente:

* Colour è usato per assegnare il colore di primo piano alla penna uno e il colore di sfondo alla penna tre. Questo per rendere il testo ben evidenziato.

* Il primo testo viene mostrato nel font standard.

* Il font viene assegnato a Topaz 11.

* Il successivo testo prababilmente sará mostrato in differente font.

* Il loop FOR disegna una linea tratteggiata in penna due.

* Viene disegnata una linea verticale in penna tre.

* Viene disegnata una linea diagonale in penna due, che unita alla precedente linea forma una v.

* Viene disegnato un box pieno in penna uno.

1.168 beginner.guide/Schermi

```
3.5.4 Schermi
      =======
```

Il seguente programma usa parti del precedente esempio, ma apre anche una schermo personale. Fondamentalmente, vengono disegnate delle linee colorate e dei boxes in una grande window aperta su uno schermo a 16 colori, in alta risoluzione.

```
    MODULE 'intuition/intuition', 'graphics/view'
```

```
PROC main()
  DEF sptr=NIL, wptr=NIL, i
  sptr:=OpenS(640,200,4,V_HIRES,'Screen demo')
  IF sptr
    wptr:=OpenW(0,20,640,180,IDCMP_CLOSEWINDOW,
                WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
                'Finestra demo per la grafica',sptr,$F,NIL)
    IF wptr
      TextF(20,20,'Hello World')
      FOR i:=0 TO 15  /* Disegna una linea e un box per ogni colore */
        Line(20,30,620,30+(7*i),i)
        Box(10+(40*i),140,30+(40*i),170,1)
        Box(11+(40*i),141,29+(40*i),169,i)
      ENDFOR
      WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
      ENDWHILE
      WriteF('Programmma terminato con successo\n')
    ELSE
      WriteF('Non posso aprire la window\n')
    ENDIF
  ELSE
    WriteF('Non posso aprire lo schermo\n')
  ENDIF
  IF wptr THEN CloseW(wptr)
  IF sptr THEN CloseS(sptr)
ENDPROC
```

Come puoi notare, le verifiche degli errori con i blocchi IF, possono rendere il programma difficile da leggere. Segue lo stesso esempio scritto con un exception handler:

```
MODULE 'intuition/intuition', 'graphics/view'

ENUM WIN=1, SCRN

RAISE WIN  IF OpenW()=NIL,
      SCRN IF OpenS()=NIL

PROC main() HANDLE
  DEF sptr=NIL, wptr=NIL, i
  sptr:=OpenS(640,200,4,V_HIRES,'Screen demo')
  wptr:=OpenW(0,20,640,180,IDCMP_CLOSEWINDOW,
              WFLG_CLOSEGADGET OR WFLG_ACTIVATE,
              'Finestra demo per la grafica',sptr,$F,NIL)
  TextF(20,20,'Hello World')
  FOR i:=0 TO 15  /* Disegna una linea e un box per ogni colore */
    Line(20,30,620,30+(7*i),i)
    Box(10+(40*i),140,30+(40*i),170,1)
    Box(11+(40*i),141,29+(40*i),169,i)
  ENDFOR
  WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
  ENDWHILE
EXCEPT DO
  IF wptr THEN CloseW(wptr)
  IF sptr THEN CloseS(sptr)
  SELECT exception
```

```
    CASE 0
      WriteF('Programma terminato con successo\n')
    CASE WIN
      WriteF('Non posso aprire la finestra\n')
    CASE SCRN
      WriteF('Non posso aprire lo schermo\n')
    ENDSELECT
  ENDPROC
```

Adesso il programma è più leggibile. La parte importante del programma (il pezzo prima di EXCEPT) non è più ingombrato con le verifiche degli errori ed è più facile notare cosa succede se capita un errore. Nota che se il programma termina con successo, ancora non sono stati chiusi correttamente lo schermo e la window, pertanto è sensato usare EXCEPT DO per ottenere una exception zero e far compiere così tali operazioni nell'handler.

1.169 beginner.guide/Esempio di Ricorsione

```
3.6 Esempio di Ricorsione
    ********************
```

Il prossimo esempio usa un paio di procedure reciprocamente (mutually) ricorsive per disegnare una cosa conosciuta come dragon curve (curva del drago, un simpatico riempimento di spazi pattern).

```
  MODULE 'intuition/intuition', 'graphics/view'

  /* Dimensione schermo, usa SIZEY=512 per uno schermo PAL */
  CONST SIZEX=640, SIZEY=400

  /* Valori exception */
  ENUM WIN=1, SCRN, STK, BRK

  /* Direzioni (DIRECTIONS dá un numero alle direzioni) */
  ENUM NORTH, EAST, SOUTH, WEST, DIRECTIONS

  RAISE WIN  IF OpenW()=NIL,
        SCRN IF OpenS()=NIL

  /* Inizia a puntare a WEST */
  DEF state=WEST, x, y, t

  /* Fronte sinistro */
  PROC left()
    state:=Mod(state-1+DIRECTIONS, DIRECTIONS)
  ENDPROC

  /* Muove a destra, cambiando state */
  PROC right()
    state:=Mod(state+1, DIRECTIONS)
  ENDPROC

  /* Muove nella direzione del fronte attuale */
  PROC move()
```

```
  SELECT state
  CASE NORTH; draw(0,t)
  CASE EAST;  draw(t,0)
  CASE SOUTH; draw(0,-t)
  CASE WEST;  draw(-t,0)
  ENDSELECT
ENDPROC

/* Disegna e muove nella relativa posizione specificata */
PROC draw(dx, dy)
  /* Controlla che la linea venga disegnata nei limiti della window */
  IF (x>=Abs(dx)) AND (x<=SIZEX-Abs(dx)) AND
     (y>=Abs(dy)) AND (y<=SIZEY-10-Abs(dy))
    Line(x, y, x+dx, y+dy, 2)
  ENDIF
  x:=x+dx
  y:=y+dy
ENDPROC

PROC main() HANDLE
  DEF sptr=NIL, wptr=NIL, i, m
  /* Legge gli argomenti   :        [m [t [x  [y]]]] */
  /* così possiamo scrivere: dragon  16              */
  /*                       o: dragon  16 1           */
  /*                       o: dragon  16 1 450       */
  /*                       o: dragon  16 1 450 100   */
  /* m è il depth del dragon, t è la lunghezza delle linee */
  /* (x,y) è la posizione di partenza */
  m:=Val(arg, {i})
  t:=Val(arg:=arg+i, {i})
  x:=Val(arg:=arg+i, {i})
  y:=Val(arg:=arg+i, {i})
  /* Se m o t è zero usa un default più logico */
  IF m=0 THEN m:=5
  IF t=0 THEN t:=5
  sptr:=OpenS(SIZEX,SIZEY,4,V_HIRES OR V_LACE,'Dragon Curve Screen')
  wptr:=OpenW(0,10,SIZEX,SIZEY-10,
              IDCMP_CLOSEWINDOW,WFLG_CLOSEGADGET,
              'Dragon Curve Window',sptr,$F,NIL)
  /* Disegna la dragon curve */
  dragon(m)
  WHILE WaitIMessage(wptr)<>IDCMP_CLOSEWINDOW
  ENDWHILE
EXCEPT DO
  IF wptr THEN CloseW(wptr)
  IF sptr THEN CloseS(sptr)
  SELECT exception
  CASE 0
    WriteF('Programma terminato con successo\n')
  CASE WIN
    WriteF('Non posso aprire la window\n')
  CASE SCRN
    WriteF('Non posso aprire lo schermo\n')
  CASE STK
    WriteF('Stack non sufficiente per la ricorsione\n')
  CASE BRK
    WriteF('L''utente abbandona\n')
```

```
    ENDSELECT
  ENDPROC

  /* Disegna la dragon curve (da sinistra) */
  PROC dragon(m)
    /* Controlla lo stack e ctrl-C prima della ricorsione */
    IF FreeStack()<1000 THEN Raise(STK)
    IF CtrlC() THEN Raise(BRK)
    IF m>0
      dragon(m-1)
      left()
      nogard(m-1)
    ELSE
      move()
    ENDIF
  ENDPROC

  /* Disegna la dragon curve (da destra) */
  PROC nogard(m)
    IF m>0
      dragon(m-1)
      right()
      nogard(m-1)
    ELSE
      move()
    ENDIF
  ENDPROC
```

Se chiami questo file dragon.e e lo compili, allora con l'eseguibile dragon alcune cose belle da provare sono:

```
dragon 5 9 300 100
dragon 10 4 250 250
dragon 11 3 250 250
dragon 15 1 300 100
dragon 16 1 400 150
```

Se vuoi capire come funziona il programma, devi studiare le parti ricorsive. Segue una rassegna del programma evidanziando gli aspetti più importanti:

* Le costanti SIZEX e SIZEY sono rispettivamente la larghezza e l'altezza dello schermo personale (e della window). Come il commento suggerisce, puoi cambiare SIZEY a 512 per uno schermo più grande se hai un Amiga PAL.

* La variabile state contiene la direzione corrente (nord, sud, est o ovest).

* Le procedure left e right spostano la direzione corrente a sinistra e a destra (rispettivamente) usando qualche trucco aritmetico con Mod.

* La procedura move usa la procedura draw per disegnare una linea (di lunghezza t) nella direzione corrente, dal punto corrente (conservato in x ed y).

* La procedura draw disegna una linea tenendo presente il punto corrente

corrente, ma solo se esso è compreso nei limiti della window. Il punto corrente viene spostato alla fine della linea (anche se essa non viene disegnata).

* La procedura main legge gli argomenti della linea di comando nelle variabili m, t, x, y. La depth/size del dragon è data da m (il primo argomento) e la lunghezza di ogni linea che forma il dragon è data da t (il secondo argomento). Il punto di partenza è dato da x, y (gli ultimi due argomenti). I valori di default sono cinque per m, t e zero per x, y.

* La procedura main apre anche uno schermo e una window e assegna il disegno del dragon.

* Le procedure dragon e nogard sono molto simili e il loro compito è di creare la dragon curve chiamando le procedure left, right e move.

* La procedura dragon contiene un paio di controlli, uno per l'uso dei tasti Control-C da parte dell'utente, l'altro se al programma viene a mancare spazio stack, ottenendo un'appropriata exception se necessario. Queste exceptions sono controllate dalla procedura main.

Nota l'uso di Val e dell'exception handling. Inoltre il caso base importante della ricorsione è quando, m, raggiunge lo zero (o diventa negativo, ma non dovrebbe accadere). Se gli argomenti che dai quando esegui dragon sono molto grandi, puoi bloccare elegantemente il programma con i tasti Control-C. Se il programma termina il disegno dobbiamo semplicemente cliccare il gadget di chiusura della window.

1.170 beginner.guide/Problemi Comuni

4.1 Problemi Comuni
****

Se sei un principiante della programmazione o del linguaggio E, allora potresti apprezzare se ti viene dato qualche aiuto per localizzare i problemi (o difetti) nei tuoi programmi. Seguono alcuni dei più comuni errori che si possono fare.

Valori con segno e senza segno

1.171 beginner.guide/Assegnare e Copiare

4.1.1 Assegnare e Copiare
==========

Questo probabilmente è il problema più comune in cui si imbatte chi è abituato a linguaggi come il BASIC. Stringhe, liste, matrici e oggetti non possono essere inizializzati usando una dichiarazione di assegnazione: i dati devono essere copiati. Diversamente dal BASIC, questo tipo di dati è rappresentato da un puntatore Vedi
sez. 2.4.2
, quindi solo il puntatore
dovrebbe essere copiato da una dichiarazione di assegnazione, non i dati a cui esso punta. I seguenti esempi copiano tutti un puntatore e non i dati, pertanto la memoria per i dati è condivisa (e ciò probabilmente è quello che non è stato recepito).

```
  DEF s[30]:STRING, t[30]:STRING,
      l[10]:LIST, m[10]:LIST,
      x:myobj, y:myobj,
      a[25]:ARRAY OF INT, b[25]:ARRAY OF INT

/* Probabilmente non vuoi fare alcune di queste cose */
  s:='Un testo qualsiasi nella stringa'
  l:=[-6,4,-9]
  x:=[1,2,3]:myobj
  a:=[1,-3,8,7]:INT

  t:=s
  m:=l
  y:=x
  b:=a
```

Tutte le dichiarazioni allocano l'appropriata memoria per i dati. Le prime quattro assegnazioni sostituiscono i puntatori a questa memoria con dei puntatori a qualche zona di memoria allocata staticamente. La memoria allocata dalle dichiarazioni ora probabilmente è irraggiungibile in quanto i soli puntatori ad essa sono stati sovrascritti. I programmatori BASIC potrebbero aspettarsi che l'assegnazione di s abbia copiato la stringa nella memoria allocata per s dalla sua dichiarazione, ma non è così (solo il puntatore della stringa viene copiato).

Per la E-string, s, e la E-list, l, esiste un altro disastroso side-effect. L'assegnazione di s, per esempio, fará in modo che, s, punti ad una stringa normale e non ad una E-string. Quindi, s, non potrá essere usata con nessuna delle funzioni E-string. Lo stesso dicasi per la E-list, l.

Anche le ultime quattro assegnazioni copiano solo puntatori. Questo significa che s ed t punteranno precisamente alla stessa stringa e qualsiasi cambiamento a uno di essi (con StrAdd, per esempio) cambierá entrambi (naturalmente solo un pezzo di memoria viene modificato, ma

esistono due riferimenti ad essa). Tale situazione è chiamata memoria condivisa, il solo problema è se non hai capito questo!

Per ottenere il risultato che un programmatore BASIC si aspetta, abbiamo bisogno di copiare gli appropriati dati. Per le E-strings e le E-lists le funzioni da usare sono rispettivamente StrCopy e ListCopy. Tutti gli altri dati devono essere copiati usando una funzione come CopyMem (una funzione di sistema di Amiga della Exec library). (Le stringhe normali possono essere copiate usando AstrCopy una funzione BUILT-IN, vedi il 'Reference Manual'.) Seguono le sintassi corrette delle assegnazioni precedenti:

```
DEF s[30]:STRING, t[30]:STRING,
    l[10]:LIST, m[10]:LIST,
    x:myobj, y:myobj,
    a[25]:ARRAY OF INT, b[25]:ARRAY OF INT

StrCopy(s, 'Un testo qualsiasi nella stringa') /* ALL di defaults */
ListCopy(l, [-6,4,-9])                         /* ALL di defaults */
CopyMem([1,2,3]:myobj, x, SIZEOF myobj)
CopyMem([1,-3,8,7]:INT, a, 4*SIZEOF INT)

StrCopy(t, s)   /* ALL di defaults */
ListCopy(m, l)  /* ALL di defaults */
CopyMem(x, y, SIZEOF myobj)
CopyMem(a, b, 4*SIZEOF INT)
```

Nota che abbiamo bisogno di fornire la dimensione (in byte) dei dati da copiare quando usiamo CopyMem. I parametri sono dati anche in ordine inverso rispetto alle funzioni di copia delle E-strings e delle E-lists (ossia, il sorgente deve essere il primo parametro e la destinazione il secondo). La funzione CopyMem copia byte per byte cioè fa qualcosa di simile:

```
PROC copymem(src, dest, size)
  DEF i
  FOR i:=1 TO size DO dest[]++:=src[]++
ENDPROC
```

Naturalmente puoi usare costanti stringa e list per inizializzare le matrici, ma in questo caso dobbiamo inizializzare un appropriato tipo di puntatore. Dobbiamo stare attenti anche a non incorrere nel problema dei dati statici Vedi
sez. 2.4.5.7
.

```
DEF s:PTR TO CHAR, l:PTR TO LONG, x:PTR TO myobj, a:PTR TO INT
s:='Un testo qualsiasi nella stringa'
l:=[-6,4,-9]
x:=[1,2,3]:myobj
a:=[1,-3,8,7]:INT
```

1.172 beginner.guide/Puntatori ed Allocazione di Memoria

```
            4.1.2 Puntatori ed Allocazione di Memoria
      ===================================
```

Un altro errore comune è dichiarare un puntatore (normalmente un puntatore ad un oggetto) e poi utilizzarlo senza che la memoria per i dati puntati venga allocata.

```
     /* Non possiamo fare questo */
       DEF p:PTR TO object
       p.element:=99
```

Ci sono due modi per correggere questa situazione: o allocare dinamicamente la memoria usando NEW o più semplicemente lasciare che un'appropriata dichiarazione la allochi. Vedi

```
            sez. 2.9
```

```
              DEF p:PTR TO object
       NEW p
       p.element:=99

       DEF p:object
       p.element:=99
```

1.173 beginner.guide/Uso Errato di Stringhe e Liste

```
            4.1.3 Uso Errato di Stringhe e Liste
      ==============================
```

Parte delle funzioni stringa possono essere usate solo con E-strings e generalmente sono quelle che possono estendere la stringa, se usiamo una stringa normale invece, possiamo incorrere in alcuni seri (ma subdoli) problemi. Comunemente le funzioni usate in modo errato sono ReadStr, MidStr e RightStr. Problemi sinili possono sorgere quando usiamo una lista dove invece è richiesta una E-list da una funzione list.

Le costanti stringa e le liste normali sono dati statici, pertanto non dovremmo provare a modificare i loro contenuti a meno di non sapere quello che stiamo facendo Vedi

```
            sez. 2.4.5.7
            .
```

1.174 beginner.guide/Inizializzare i Dati

```
4.1.4 Inizializzare i Dati
      ====================
```

Probabilmente uno degli sbagli più comuni che anche programmatori con più esperienza fanno è dimenticare di inizializzare le variabili (specialmente

i puntatori). Le regole nel 'Reference Manual' stabiliscono quali dichiarazioni inizializzano le variabili con valore zero, ma spesso è sempre meglio farlo esplicitamente (usando dichiarazioni inizializzate). Le inizializzazioni delle variabili diventano anche più importanti se dobbiamo usare le exceptions automatiche.

1.175 beginner.guide/Liberare le Risorse

```
4.1.5 Liberare le Risorse
      ===================
```

Diversamente da un sistema operativo Unix, il sistema operativo di Amiga richiede al programmatore di rilasciare o liberare qualsiasi risorsa usata da un programma. In pratica, questo significa che tutte le window, gli schermi, le librerie, ecc., aperte con successo, devono essere chiuse prima che il programma termini. Sebbene Amiga E fornisca qualche aiuto: Le quattro librerie usate più comunemente ((Dos, Exec, Graphics and Intuition) sono aperte prima dell'avvio di un programma E e sono chiuse alla fine (o quando viene chiamata CleanUp). Inoltre la memoria allocata usando New, List e String viene liberata automaticamente al termine del programma.

1.176 beginner.guide/Puntatori e Dereferencing

```
4.1.6 Puntatori e Dereferencing
      ========================
```

I programmatori C possono pensare che le espressioni ^var e {var} sono le dirette equivalenti delle espressioni del C &var e *var. Tuttavia in E la dereferencing è normalmente ottenuta usando la selezione dell'elemento di una matrice o di un object e i puntatori a grandi quantitá di dati (come E-string o objects) sono ottenuti da dichiarazioni. Questo significa che le espressioni ^var e {var} sono usate raramente, mentre var[] è molto comune.

1.177 beginner.guide/Funzioni Matematiche

```
                4.1.7 Funzioni Matematiche
      ====================
```

Gli operatori matematici standard / e * non usano l'intero valore a 32-bit nei loro calcoli, come abbiamo visto in precedenza Vedi
sez. 2.6.3.4
. Un
comune problema è dimenticarlo e usare tali operatori dove i valori eccederanno il limite dei 16-bit. Vedi
sez. 4.1.8

1.178 beginner.guide/Valori con segno e senza segno

4.1.8 Valori con segno e senza segno
==========

Questo è un argomento abbastanza avanzato, ma potrebbe essere la causa di alcuni strani difetti nei tuoi programmi. Fondamentalmente l'E non ha un modo per differenziare valori con segno da valori senza segno. Prendiamo il tipo LONG, cioè tutti i valori a 32-bit, questi sono considerati come valori con segno, quindi la gamma di valori permessi va da -2,147,483,648 a 2,147,483,647. Se i valori di questo tipo venissero presi come valori senza segno, allora non sarebbe permesso nessun valore negativo, ma si avrebbero più valori positivi da utilizzare (cioè, la gamma di valori sarebbe da zero a 4,294,967,295). Questa distinzione interesserebbe anche gli operatori matematici.

In pratica, comunque, non è il tipo LONG che può causare problemi, ma il tipo INT che è a 16-bit ed è considerato con segno. Questo significa che la gamma di valori è compresa fra -32,768 a 32767. Tuttavia, gli object di sistema di Amiga contengono numeri a 16-bit, elementi INT che sono in effetti interpretati come senza segno, ossia che si estendono da zero a 65,535. Un esempio interessante è il gadget proporzionale che forma una parte della barra di scroll su una window (per esempio una drawer window sul Workbench). Tale gadget lavora con valori a 16-bit senza segno che sono in contrasto con il tipo INT dell'E. Questi valori sono comunemente usati nei calcoli per determinare la posizione di qualcosa che viene mostrata in una window e se il tipo INT viene usato senza prendere in consiserazione il problema con segno /senza segno i risultati ottenuti potrebbero essere abbastanza sbagliati. Fortunatamente è abbastanza semplice convertire i valori INT con segno in valori senza segno se fanno parte di qualche espressione e se il valore di qualsiasi espressione viene preso dal tipo LONG (e i valori INT senza segno si adattano bene anche all'interno della gamma di valori LONG con segno).

```
PROC unsigned_int(x) IS x AND $FFFF
```

La funzione unsigned_int è specifica al modo in cui l'Amiga maneggia i valori internamente, quindi il capire come funziona è oltre la portata di questa Guida. Essa dovrebbe essere usata ovunque un valore a 16-bit senza segno è conservato in un elemento INT, diciamo, un object di sistema di Amiga. Per esempio, la posizione iniziale di un (verticale) gadget proporzionale con una percentuale della sua dimensione (da zero a cento) può essere calcolata così:

```
/* propinfo è nel modulo 'intuition/intuition' */
DEF gad:PTR TO propinfo, pct
/* Assegnare gad... */
/* Calcola la percentuale (MAXPOT è in 'intuition/intuition') */
pct:=Div(Mul(100,unsigned_int(gad.vertpot)),MAXPOT)
```

Nota che le funzioni a 32-bit pieni Div e Mul devono essere usate fin quando il calcolo può star bene sui normali 16-bit usati negli operatori / e *.

Il rimanente tipo CHAR, in pratica, non è un problema. Esso è il solo tipo senza segno, con una gamma di valori da zero a 255. Esiste un modo molto

semplice per convertire tali valori a valori con segno (e di nuovo questo è il modo particolare con cui l'Amiga conserva internamente i valori). Un buon esempio di un valore CHAR con segno è il valore di prioritá associato con un nodo di un'Amiga list (per esempio, l'elemento pri di un object ln del modulo exec/nodes).

```
PROC signed_char(x) IS IF x<128 THEN x ELSE x-256
```

1.179 beginner.guide/Altre Informazioni

```
              4.2 Altre Informazioni
    ******************
```

Questa appendice contiene alcune utili informazioni.

1.180 beginner.guide/Versioni Amiga E

```
4.2.1 Versione di Amiga E
      ===================
```

Nel momento in cui scrivo, la versione corrente di Amiga E è la 3.1a (che è più aggiornata della v3.0e), questa edizione della Guida è basata principalmente su tale versione, ma la maggior parte delle cose si può ancora applicare alle versioni più vecchie, includendo l'ultima versione di pubblico dominio (v2.1b). La versione 3.2 è imminente e questa guida speriamo sia inclusa in tale aggiornamento. Vedi il 'Reference Manual' per i dettagli delle nuove caratteristiche e cambiamenti.

Per favore nota che dalla v3.0a, Amiga E è un prodotto commerciale, pertanto devi pagare una certa cifra per avere una versione completa del compilatore (che sará registrato a te). La distribuzione di pubblico dominio contiene solo una versione dimostrativa del compilatore, con funzionalitá limitata. Vedi il 'Reference Manual' per maggiori dettagli.

1.181 beginner.guide/Ulteriori Manuali

4.2.2 Ulteriori Manuali
=======

'Amiga E Language Reference'
Noto come il 'Reference Manual' in questa Guida, è uno dei manuali che viene fornito con la confezione dell'Amiga E, ed è essenziale leggerlo in quanto è stato scritto da Wouter (l'autore di Amiga E) e contiene molte altre informazioni.

'Rom Kernel Reference Manual' (Addison-Wesley)
Questo manuale è la documentazione ufficiale Commodore sulle funzioni di sistema di Amiga ed è d'obbligo se vuoi usare tali funzioni correttamente. Nel momento in cui scrivo l'edizione più recente è la Terza e copre le funzioni di sistema di Amiga superiori alla Versione 2 (cioè, AmigaDOS 2.04 e KickStart 37). Poichè le informazioni da fornire sono molte, il manuale è composto da tre libri separati: 'Libraries', 'Includes and Autodocs', e 'Devices'. Il libro 'Libraries' è probabilmente il più utile in quanto contiene molti esempi e tutorial. Tuttavia gli esempi sono scritti principalmente in C (gli altri in Assembly). Per sopperire a questo problema ho iniziato a ricodificarli in E e parte di questo sforzo dovrebbe essere disponibile nello stesso posto dove hai trovato questa Guida (il nome dell'archivio sará qualcosa come JRH-RKRM-1).

'The AmigaDOS Manual' (Bantam Books)
questo libro è il compagno ideale del 'Rom Kernel Reference Manual' ed è il libro ufficiale Commodore sull'AmigaDOS (contiene programmi AmigaDOS e funzioni di libreria DOS). La Terza edizione è la più recente.

Sorgenti di esempio
Amiga E viene fornito con una grande raccolta di programmi d'esempio. Quando avrai famigliaritá con il linguaggio E, dovresti essere in grado di apprendere qualcosa in più dagli esempi. Ci sono parecchi piccoli programmi tutorial e alcuni programmi più complicati.

1.182 beginner.guide/L'autore di Amiga E

4.2.3 L'autore di Amiga E
=======

Nel caso non sapessi chi è l'autore e creatore di Amiga E, indico il suo nome e il suo indirizzo. Il suo nome è Wouter van Oortmerssen (o $#%!) e lo puoi raggiungere con la normale posta al seguente indirizzo:

Wouter van Oortmerssen ($#%!)
Levendaal 87
2311 JG Leiden
HOLLAND

Tuttavia egli preferisce molto corrispondere con la Posta elettronica e lo puoi raggiungere ai seguenti indirizzi:

```
     Wouter@alf.let.uva.nl (Supporto alla programmazione E)
     Wouter@mars.let.uva.nl (personale)
     Oortmers@gene.fwi.uva.nl (altro)
```

Ancora meglio se il tuo problema o informazione è di interesse generale per gli utenti dell'Amiga E, puoi trovare utile inserirti nella lista indirizzi di Amiga E. Wouter contribuisce regolarmente a tale lista e ci sono anche dei buoni programmatori a portata di mano per aiutare o discutere problemi. Per inserirti, spedire un messaggio a:

```
     amigae-richiedere@bkhouse.cts.com
```

Dopo che ti abboni, riceverai una copia di ogni messaggio spedito alla lista. Riceverai anche un messaggio d'invito a contribuire (cioè, ponendo domande!).

1.183 beginner.guide/L'Autore della Guida

```
4.2.4 L'Autore della Guida
      ====================
```

Questa guida è stata scritta da Jason Hulance, aiutato e guidato molto da Wouter. L'intenzione iniziale era di creare qualcosa che poteva fungere per i principianti come un'utile introduzione all'Amiga E, in modo che il linguaggio potesse essere (giustamente) maggiormente diffuso, ma lo scopo nascosto era di liberare Wouter da un tale impegno, in modo che potesse concentrare i suoi sforzi nello sviluppare Amiga E.

Puoi raggiungermi facilmente con la posta normale al seguente indirizzo (di lavoro):

```
     Jason R. Hulance
     Formal Systems (Europe) Ltd.
     3 Alfred Street
     Oxford
     OX1 4EH
     ENGLAND
```

In alternativa, puoi trovarmi nella lista di indirizzi Amiga E o con la Posta elettronica, direttamente ad uno dei seguenti indirizzi:

```
     jason@fsel.com
     m88jrh@uk.ac.oxford.ecs
```

Se hai da proporre cambiamenti o aggiunte che vorresti vedere, sarei molto felice di considerarli. La critica del testo è anche benvenuta, specialmente se puoi suggerire un modo migliore per spiegare le cose, sarei anche entusiasta di sentire delle persone che possono evidenziare delle aree della Guida che sono particolarmente confuse o malamente spiegate!.

Inoltre con una piccola cifra puoi avere una versione stampabile di questa Guida in formato DVI o PostScript che include un enorme indice, molte immagini e molte belle tavole. Costa soltanto 5 sterline per residenti UK e 8 sterline per i non residenti UK (i prezzi includono un disco e il costo

della tariffa postale). Posso fare anche delle versioni stampate (includendo una buona rilegatura se richiesto) ovviamente con qualcosa in più nel prezzo. Puoi contattarmi liberamente con la posta elettronica o con quella normale, agli indirizzi visti prima, se vuoi maggiori informazioni.

1.184 beginner.guide/Traduzione in Italiano

```
4.2.5 Traduzione in Italiano
      ======================
```

Questa guida è stata tradotta in italiano da Amendolagine GIanni, con la collaborazione di Milella Amedeo. Tutti gli utenti italiani del linguaggio E che hanno trovato di loro interesse questo lavoro, possono inviarci tramite cartolina il loro apprezzamento e anche i loro consigli ed incoraggiamento a continuare nella traduzione di altri documenti quali ad esempio il 'Amiga E Language Reference' e altro, contenuto nel pacchetto dell'Amiga E e/o eventualmente documenti avuti direttamente dall'autore. Ovviamente per queste ulteriori traduzioni ed eventuali versioni stampabili e/o giá stampate si richiederá un contributo spese.

Potrete contattarci con posta normale al seguente indirizzo:

```
     Amendolagine Gianni
     Via Napoli 19/A
     70050 S.Spirito - BARI
```

e con posta elettronica ai seguenti:

```
     (Amedeo Milella)

E-mail:     milella@teseo.it
FidoNET:        2:335/704.22
```

Se questa nostra iniziativa di diffusione ITALIANA di questo (finalmente ottimo per tutti) linguaggio specifico per gli amanti del mondo Amiga, avrá un buon riscontro, risponderemo a tutti coloro che ci avranno scritto inviando in risposta informazioni più precise su ciò che sará disponibile e con quali richieste.

1.185 beginner.guide/Indice Completo delle Sezioni

```
                5.1 Indice Completo delle Sezioni
    ****************************
```

Questo indice elenca tutte le sezioni delle Guida.

1.1
Introduzione ad Amiga E
1.1.1
Un semplice programma
1.1.1.1
Programma
1.1.1.2
Compilazione
1.1.1.3
Esecuzione
1.2
Comprensione del programma semplice
1.2.1
Modifica del messaggio
1.2.1.1
Breve rassegna
1.2.2
Procedure
1.2.2.1
Definizione di una procedura (PROC e ENDPROC)
1.2.2.2
Eseguire una procedura
1.2.2.3
Un esempio più completo
1.2.3
Parametri
1.2.4
Stringhe
1.2.5
Stile, praticitá e leggibilitá
1.2.6
Il programma semplice
1.3
Variabili ed Espressioni
1.3.1
Variabili
1.3.1.1
Tipi di variabili
1.3.1.2
Dichiarazione di variabile (DEF)
1.3.1.3
Assegnazione
1.3.1.4
Variabili globali e locali (Parametri per procedure)
1.3.1.5
Modifica dell'esempio
1.3.2
Espressioni
1.3.2.1
Matematica
1.3.2.2
Logica e comparazione (TRUE e FALSE - AND e OR)
1.3.2.3
Precedenze e raggruppamenti
1.4
Flusso di controllo del programma

1.186 beginner.guide/Indice del Linguaggio E

5.2 Indice del Linguaggio E
****

Questo indice dovrebbe essere usato per trovare informazioni precise sulle keyword, funzioni, variabili e costanti che fanno parte del linguaggio Amiga E. Esiste un indice separato per i concetti ecc. Vedi
sez. 5.3
.

Simbolo, close curly brace

Trovare indirizzi (costruire puntatori)

Simbolo, double-quote
Costanti numeriche

Simbolo, open curly brace
Trovare indirizzi (costruire puntatori)

Simbolo, !
Calcoli in Virgola Mobile

Simbolo, $
Costanti numeriche

Simbolo, %
Costanti numeriche

Simbolo, ' .. ' (string)
Stringhe normali ed E-strings

Simbolo, *
Matematica

Simbolo, +
Matematica

Simbolo, + (strings)
Dichiarazioni

Simbolo, ++
Puntare agli altri elementi

Simbolo, -
Matematica

Simbolo, --
Puntare agli altri elementi

Simbolo, ->
Commenti

Simbolo, /
Matematica

Simbolo, /* .. */
Commenti

Simbolo, :
Labels e dichiarazione JUMP

Simbolo, :=
Assegnazione

Simbolo, ;
Dichiarazioni

Simbolo, <

```
                              Logica e comparazione

Simbolo, <=
                             Logica e comparazione

Simbolo, <=>
                            Unification (Unificazione)

Simbolo, <>
                             Logica e comparazione

Simbolo, =
                              Logica e comparazione

Simbolo, >
                              Logica e comparazione

Simbolo, >=
                             Logica e comparazione

Simbolo, [ .. , .. ] (list)
          Lists ed E-lists

Simbolo, [ .. , .. ]:type (typed list)
Typed lists

Simbolo, [ .. ] (array)
              Tavole di dati

Simbolo, [] (array)
                   Utilizzare i dati di un array

Simbolo, \0
                             Costanti stringa, sequenze di caratteri ↩
                                 speciali

Simbolo, \a
                             Costanti stringa, sequenze di caratteri ↩
                                 speciali

Simbolo, \b
                             Costanti stringa, sequenze di caratteri ↩
                                 speciali

Simbolo, \c
                             Funzioni di input e output

Simbolo, \d
                             Funzioni di input e output

Simbolo, \d
                             Modifica dell'esempio

Simbolo, \e
                             Costanti stringa, sequenze di caratteri ↩
                                 speciali
```

Simbolo, \h
Funzioni di input e output

Simbolo, \l
Funzioni di input e output

Simbolo, \n
Stringhe

Simbolo, \n
Costanti stringa, sequenze di caratteri ↩ speciali

Simbolo, \q
Costanti stringa, sequenze di caratteri ↩ speciali

Simbolo, \r
Funzioni di input e output

Simbolo, \s
Funzioni di input e output

Simbolo, \t
Costanti stringa, sequenze di caratteri ↩ speciali

Simbolo, \z
Funzioni di input e output

Simbolo, \\
Costanti stringa, sequenze di caratteri ↩ speciali

Simbolo, ^
Estrazione dei dati (Dereferencing i ↩ puntatori)

Simbolo, ` (backquote)
Espressioni Quoted

Abs
Funzioni matematiche e logiche

ALL
Costanti Built-In

AND
Bitwise AND e OR

And
Funzioni matematiche e logiche

arg
Variabili Built-In

ARRAY

Dichiarazione di variabile (DEF)

DEFAULT
Blocco SELECT

DEFAULT
Blocco SELECT..OF

Dispose
Funzioni di supporto system

DisposeLink
Funzioni di supporto system

Div
Funzioni matematiche e logiche

DO, (FOR loop)
Loop FOR

DO, (WHILE loop)
Loop WHILE

dosbase
Variabili Built-In

ELSE
Blocco IF

ELSEIF
Blocco IF

EMPTY
Ereditá in E

end
Metodi in E

END
Operatori NEW ed END

ENDFOR
Loop FOR

ENDIF
Blocco IF

ENDLOOP
Blocco LOOP

ENDOBJECT
Esempio di object

ENDPROC
Definizione di una procedura (PROC e ↩
ENDPROC)

FastDispose
Funzioni di supporto system

FastDisposeList
Allocazione di list e typed list

FastNew
Funzioni di supporto system

Fceil
Funzioni in Virgola Mobile

Fcos
Funzioni in Virgola Mobile

Fexp
Funzioni in Virgola Mobile

Ffloor
Funzioni in Virgola Mobile

FileLength
Funzioni di input e output

Flog
Funzioni in Virgola Mobile

Flog10
Funzioni in Virgola Mobile

FOR
Loop FOR

ForAll
Espressioni lists e quoted

Forward
Liste linked

Fpow
Funzioni in Virgola Mobile

FreeStack
Funzioni di supporto system

Fsin
Funzioni in Virgola Mobile

Fsqrt
Funzioni in Virgola Mobile

Ftan
Funzioni in Virgola Mobile

Gadget
Funzioni di supporto intuition

OpenS
Funzioni di supporto intuition

OpenW
Funzioni di supporto intuition

Or
Funzioni matematiche e logiche

OR
Bitwise AND e OR

Out
Funzioni di input e output

Plot
Funzioni grafiche

PrintF
Funzioni di input e output

PRIVATE
Dati Nascosti in E (Data-Hiding)

PROC
Definizione di una procedura (↩
PROC e ENDPROC)

PROC..OF
Metodi in E

PTR TO type
Tipo PTR

PUBLIC
Dati Nascosti in E (Data-Hiding)

PutChar
Funzioni matematiche e logiche

PutInt
Funzioni matematiche e logiche

PutLong
Funzioni matematiche e logiche

RAISE
Exception automatiche

Raise
Ottenere una Exception

ReadStr
Funzioni di input e output

RealF
Funzioni in Virgola Mobile

RealVal
Funzioni in Virgola Mobile

REPEAT
Loop REPEAT..UNTIL

RETURN
Funzioni (RETURN)

RightStr
Funzioni stringa

Rnd
Funzioni matematiche e logiche

RndQ
Funzioni matematiche e logiche

SELECT
Blocco SELECT..OF

SELECT
Blocco SELECT

SELECT..OF
Blocco SELECT..OF

SelectList
Espressioni lists e quoted

self
Metodi in E

SET
Costanti con SET

SetColour
Funzioni grafiche

SetList
Funzioni list

SetStdIn
Funzioni di input e output

SetStdOut
Funzioni di input e output

SetStdRast
Funzioni grafiche

SetStr
Funzioni stringa

SetTopaz
Funzioni grafiche

Shl
Funzioni matematiche e logiche

Shr
Funzioni matematiche e logiche

Sign
Funzioni matematiche e logiche

SIZEOF
Espressione SIZEOF

stdin
Variabili Built-In

stdout
Variabili Built-In

stdrast
Variabili Built-In

STEP
Loop FOR

StrAdd
Funzioni stringa

StrCmp
Funzioni stringa

StrCopy
Funzioni stringa

STRING
Stringhe normali ed E-strings

String
Funzioni stringa

StringF
Funzioni di input e output

StrLen
Funzioni stringa

STRLEN
Costanti Built-In

StrMax
Funzioni stringa

SUPER
Ereditá in E

TextF
Funzioni grafiche

THEN
Blocco IF

Throw
Ottenere una Exception

TO
Loop FOR

TO, (CASE range)
Blocco SELECT..OF

TO, (FOR loop)
Loop FOR

TrimStr
Funzioni stringa

TRUE
Logica e comparazione

TRUE
Costanti Built-In

UNTIL
Loop REPEAT..UNTIL

UpperStr
Funzioni stringa

Val
Funzioni stringa

VOID
Trasformare un'Espressione in una ↩
Dichiarazione

WaitIMessage
Funzioni di supporto intuition

WaitLeftMouse
Funzioni di supporto intuition

wbmessage
Variabili Built-In

WHILE
Loop WHILE

WriteF
Funzioni di input e output

1.187 beginner.guide/Indice Principale

```
              5.3 Indice Principale
    *****************

Questo indice dovrebbe essere usato  per trovare  informazioni precise  sui
vari concetti del linguaggio. Esiste un indice separato che tratta keywords
funzioni, variabili e costanti che fanno parte del linguaggio Amiga E.
Vedi
                sez. 5.2

                 A4 register
                                          A cosa stare attenti

                 A5 register
                                          A cosa stare attenti

                 Absolute value
                                       Funzioni matematiche e logiche

                 Absolute value (floating-point)
                      Funzioni in Virgola Mobile

                 Abstract class
                                       Ereditá in E

                 Abstract method
                                      Ereditá in E

                 Access array outside bounds
                          Utilizzare i dati di un array

                 Accessing array data
                                Utilizzare i dati di un array

                 Accuracy of floating-point numbers
                   Precisione e Range

                 Addition
                                           Matematica

                 Address
                                            Indirizzi

                 Address
                                            Indirizzi di memoria

                 Address, finding
                                     Trovare indirizzi (costruire puntatori)

                 Algebra
                                            Variabili ed Espressioni

                 Alignment
                                          Espressione SIZEOF
```

Tipi di variabili

Type, 16-bit
Tipi indiretti

Type, 32-bit
Tipo di default

Type, 8-bit
Tipi indiretti

Type, address
Indirizzi

Type, array
Tavole di dati

Type, complex
Tipi complessi

Type, default
Tipo di default

Type, direct
Tipi indiretti

Type, dynamic
Ereditá in E

Type, E-list
Lists ed E-lists

Type, indirect
Tipi indiretti

Type, list
Lists ed E-lists

Type, LONG
Tipo LONG

Type, LONG (definition)
Tipi indiretti

Type, object
Tipo OBJECT

Type, object elements
Selezione e tipi degli elementi

Type, pointer
Tipo PTR

Type, procedure parameters
Parametri di procedura

Type, recursive