Commodore 64

Borris Allan

Commodore 64

Borris Allan

Titolo originale: POCKET GUIDE COMMODORE 64
Editore originale: PITMAN PUBLISHING LTD

GRUPPO EDITORIALE JACKSON - Maggio 1985
SUPERVISIONE TECNICA: Vittorio Riva
TRADUZIONE: Studio redazionale CM
COPERTINA: Silvana Corbelli
FOTOCOMPOSIZIONE: Cencograf-Rotografica srl -
P.zza S. Marco 1 - Milano - Tel. 655.20.13 - 655.51.45
STAMPA: Grafika '78 - Pioltello (MI)

INDICE

Come usare questa guida tascabile 1
Variabili BASIC 1
Classi di valori 2
Tipi di espressioni 2
Calcolo valori 5
Informazioni sulle variabili 7
Vettori (Array) 10
Stack 13
Operatori aritmetici 15
Sintassi del BASIC 15
Valutazioni relazionali 18
Funzioni logiche 20
Parole chiave del BASIC 24

COME USARE QUESTA GUIDA TASCABILE

Per comprendere il funzionamento del COMMODORE 64 (d'ora in poi abbreviato in C64), l'utente necessita di una solida comprensione del BASIC del C64. Questa guida fornisce la conoscenza di base necessaria a tale compresione. La guida comincia illustrando il modo in cui le variabili e le funzioni sono immagazzinate nel BASIC del C64 e così facendo ci consente di indagare alcune delle procedure aritmetiche del computer. Da questa analisi emergeranno certi errori insiti nel sistema BASIC del C64.

Dopo la discussione delle variabili e dei vettori vi è un esame degli operatori aritmetici, delle valutazioni relazionali e dei connettori logici. Quando discutiamo di connessioni logiche esaminiamo la forma della aritmetica del computer conosciuta come «complemento a due».

La parte principale del testo esamina in dettaglio ciascuna parola chiave del BASIC del C64, per mostrare le limitazioni ed il potenziale del C64. Per taluni di questi comandi sono stati trovati altri «bug» (letteralmente «cimici» ma nel gergo del computer significa «errori» ndt) e la conoscenza di tali «bug» può aiutare a spiegare alcuni degli strani risultati che il C64 talvolta produce.

Le abbreviazioni MUM e GRP sono usate spesso in questa guida. MUM è il Manuale d'Uso del Microcomputer che acquistate insieme al C64, mentre GRP è la Guida di Riferimento del Programmatore per il C64, fornita separatamente dalla Commodore.

LE VARIABILI DEL BASIC C64

Nello studio delle variabili del BASIC C64 è d'aiuto chiarire la natura dei dati che il linguaggio tratta. Il termine «dati» è deliberatamente vago, e lo scopo di questa guida in parte consiste nel rendere più chiara la natura di questo termine.

Un dato o una collezione di dati sono noti come una «epressione», della quale esistono tre tipi principali.

Tipi di espressioni

(a) Le costanti sono i valori effettivi in una linea di programma, come 3, 3.4 oppure «AIUTO». Essi sono — come il nome lascia intendere — costanti, e non variano: «AIUTO» è sempre «AIUTO».

(b) Le variabili sono nomi dati a dei ricettacoli in cui depositare dei valori, dove i valori possono essere alterati. L'assegnazione X=3 deposita il valore 3 nel ricettacolo chiamato X: 'X' è un esempio di variabile, ed una nuova assegnazione X=4 cambia il valore depositato in X.

I nomi delle variabili possono essere lunghi fino a due caratteri, con il primo carattere che deve essere obbligatoriamente una lettera, ad esempio Y, YY o Y1. Se il nome di una variabile è più lungo di due caratteri sarà ignorata la parte rimanente: COSA e CORSO saranno trattati similmente, così come lo saranno COLORE1 e COLORE2. Tre nomi di variabile sono riservati (ST, TI e TI$), e nessun nome di variabile può contenere al suo interno il nome di una parola-chiave (ad esempio DANDY, che potrebbe essere considerato dal sistema come D AND Y).

(c) Quelli che denomineremo «misture» sono combinazioni di costanti e variabili (ad esempio X/5), e non devono essere confuse con le esressioni che sono solamente complesse (ad esempio X*X+Y/P↑T è complessa ma non è una mistura, poiché contiene soltanto variabili).

In termini generali, le misture e le costanti danno luogo a programmi che girano più lentamente. Impressionanti differenze di velocità possono essere ottenute utilizzando esclusivamente variabili in quante più espressioni sia possibile. È anche più facile modificare programmi con molte variabili che quelli con molte costanti, specialmente se la stessa variabile può rappresentare molte costanti dello stesso valore.

Classi di valori

Indipendentemente dalla forma della espressione vi sono quattro classi di espressioni. Per aiutare a riconoscere il tipo di espressione, generalmente, nel seguito, un intero sarà mostrato con i nomi da I0% a I9%, i numeri reali da X0 a X9, le espressioni logiche da L0% a L9% e le stringhe da S0$ a S9$.

(1) I valori interi sono numeri senza alcuna parte frazionaria. Gli interi sono memorizzati in due byte (cioé 16 bit), e le costanti intere possono essere mostrate ad esempio come 3 o come -4567E3, dove ciascun valore è memorizzato esattamente, senza alcuna approssimazione. Le variabili intere sono indicate con un suffisso «%», ad esempio X% oppure TJ%. Gli interi assumono valori esatti (cioé non frazionari) tra -32768 e +32767.
(2) I valori reali sono numeri con parti frazionarie, ad esempio, 3.0 è un numero reale, mentre 3 è un intero. 1.23456 e 1234,56 sono entrambi numeri reali e possono anche essere scritti rispettivamente come 1.23456E0 e 1.23456E3. La parte che precede la E è chiamata «mantissa», e la parte dopo la E è chiamata «esponente».
I valori reali sono memorizzati nel C64 nello stesso modo: il numero è memorizzato come una mantissa di quattro byte (32 bit, perciò una precisione di circa 9 cifre), ed un esponente di un byte (8 bit, perciò circa da E-39 ad E+38). Uno dei 32 bit della mantissa rappresenta il segno del numero (1 per i numeri relativi e 0 per i positivi).
I numeri reali assumono valori positivi nell'intervallo 2,93873588E-39 a 1.70141183E+39. I numeri negativi sono piuttosto strani.
Provate questo breve programma

```
10 T=1
20 T=T/2:PRINT T:GOTO 20
```

ed osservate la successione dei valori mentre T diviene progressivamente più piccolo. Alla fine raggiungeremo

```
5.87747176E-39
2.93873588E-39
0
```

ed è tutto a posto. Provate a lanciare

```
10 T=-1
20 T=T/2:PRINT T:GOTO 20
```

che raggiunge

```
-1.7549435E-38
-5.87747176E-39
2.93873588E-39
0
```

ed abbiamo scoperto un bug nel sistema BASIC del C64. La metà di -5.87747176E-39 non è, come ben sappiamo, 2.93873588 E-39.

Possiamo produrre altri errori:

```
T=25352959E+39
PRINT T*2
```

Genera un ?OVERFLOW ERROR, mentre

```
PRINT T+T
```

fornisce un risultato 8.50705917E+37, non un errore. T*2 è esattamente la stessa cosa di T+T: perciò abbiamo trovato un altro bug nel BASIC del C64.

Continuando con

```
T=T+T:PRINT T:PRINT T+T
```

si genera il valore di 8.50705917E+37 con la prima PRINT ed un ?OVERFLOW ERROR con la seconda PRINT.

```
PRINT 8.50705917E+37 + 8.50705917E+37
```

dà la risposta 1.70141183E+38. Moltiplicare per due (e non addizionare) dà nuovamente un ?OVERFLOW ERROR.

Alcune delle routine del BASIC C64 per l'aritmetica reale sono scorrette agli estremi, e possono esserlo probabilmente per esempi anche più comuni.

(3) I valori logici sono 1 (per «vero») e 0 (per «falso»). I valori logici sono usati per valutare il risultato di un certo insieme di condizioni: ad esempio «Se questo è vero e quello non è vero, allora fai questo...». Quando è effettuata una comparazione logica, entrambi i membri della comparazione sono in realtà convertiti in interi di 16 bit.

Valori logici possono risultare da comparazioni logiche o da operazioni, ma — siccome anch'esse hanno valori numerici — i valori logici possono essere trattati come valori interi o reali, a seconda del contesto.

(4) I valori stringa sono collezioni ordinate di caratteri, dove normalmente i caratteri sono distinti dalle variabili o dalle costanti

per l'essere inseriti tra virgolette. «X» è un carattere, ma X è una variabile; «X/2» è una stringa mentre X/2 è una operazione aritmetica; e «222222» è una stringa mentre 222222 è una costante.

Ciascun carattrere (ad esempio «X» o «/» o «2») è associato a due codici; il primo, codice ASCII, si riferisce a come ciascun carattere è immagazzinato in un byte di memoria (la maggior parte dei computer utilizza il codiceASCII), ed il secondo codice, detto codice di schermo, si riferisce al modo in cui ciascun carattere viene visualizzato sullo schermo (i valori di questo codice sono specifici del C64): i due tipi di codice non devono essere confusi tra loro.

Le variabili stringa sono distinte dal suffisso $: ad esempio A$ o THIS$ (quest'ultima equivalente a TH$).

Calcolo di valori

Il C64, a meno che non sia costretto a fare diversamente, agisce sempre come se i numeri con i quali sta lavorando fossero tutti valori reali. Questi sono immagazzinati in memoria con quattro byte per la mantissa (31 bit per il numero più uno per il segno) ed un byte per l'esponente, come si è detto.

Poiché tutti i calcoli vengono effettuati in virgola mobile, cioè con valori reali, ci devono essere procedure per convertire un valore da un tipo ad un altro. L'esame della GRP (pag. 318-131 dell'edizione inglese) mostra che alla locazione 3 e 4 vi è un puntatore (ADRAY1) alla routine che converte numeri a virgola mobile in numeri interi ed alla locazione 5 e 6 vi è un altro puntatore (ADRAY2) per la routine di conversione da interi in numeri a virgola mobile.

La necessità di convertire gli interi in valori a virgola mobile spiega perché l'uso di variabili intere rallenta i programmi.

Alla locazione 13 vi è un indicatore (VALTYP) che mostra se il tipo di dati corrente è numerico o stringa e, se numerico, la locazione 14 (INTFLO) indica se il numero è intero o a virgola mobile.

Vi sono due accumulatori a virgola mobile (1 e 2) ed i numeri a virgola mobile memorizzati in queste locazioni sono depositati in modo differente dai normali valori di questo tipo. L'accumulatore

a virgola mobile 1 è contenuto nelle locazioni da 97 a 102 ed un valore è memorizzato così:

LOCAZIONE	DESCRIZIONE
97	Esponente
98-101	Mantissa
102	Byte per il segno

e l'ordine è esattamente lo stesso per l'accumulatore 2, che si estende dalla locazione 105 alla locazione 110.
Noterete che il modo in cui i numeri sono immagazzinati in memoria differisce per l'aggiunta di un byte supplementare, il byte del segno. Vi è inoltre un byte lasciato per contenere qualunque cifra di overflow del byte 1 (locazione 104) ed un byte usato per arrotondare il valore contenuto nel numero 1 (locazione 112).
Dopo aver utilizzato l'accumulatore, il numero risultante è considerato negativo se sia il bit del segno per il numero sia il byte del segno indicano entrambi un risultato negativo. Nella codificazione delle routine matematiche del C64 (e del VIC 20) sono stati commessi degli errori nei controlli degli overflow: tali errori hanno come conseguenza alterazioni nei confronti di segno.
Questa è probabilmente l'origine di alcuni dei bug del BASIC C64 ed altri bug possono essere dovuti al modo in cui i valori a virgola mobile sono depositati e prelevati dalle variabili.
Se battete

```
PRINT 2↑ (-128)
```

la risposta è zero, ma

```
PRINT 2↑ (-127)/2
```

produrrà il risultato di 2.93873588E-39, ma ciò è esattamente lo stesso poiché 2↑(-128)=2↑(-127)/2.
Nonostante questi errori possano sembrare ingenui, essi gettano seri dubbi sulle altre routine matematiche usate nel C64: se il C64 deve essere usato per scopi seri tali bug devono essere eliminati.
Quando il C64 viene programmato in codice macchina le routine a virgola mobile non intaccheranno i risultati dei programi in codice macchina, a meno che non siano usate le routine a virgola mobile del BASIC dall'interno del codice macchina.

Informazioni sulle variabili

Immagazzinare una variabile X a virgola mobile, con il suo valore, richiede sette byte di memoria nel C64. Questo numero può essere ricavato abbastanza semplicemente: poiché X è una variabile a virgola mobile il cui valore può stare in cinque byte, sono necessari cinque byte per memorizzare quel valore, e poiché ciascuna variabile può avere fino a due caratteri nel nome, ed occorre un byte per ciascun carattere, occorrono due byte per contenere il nome.

Cinque byte più due byte danno i sette byte necessari per immagazzinare qualsiasi variabile a virgola mobile, includendo sia il suo nome che il suo valore.

Anche immagazzinare una variabile intera X% nel C64 richiede sette byte. Ovviamente, sono necessari almeno due byte per memorizzare il nome della variabile, ed in effetti il BASIC non ne richiede di più. Quando il nome di una variabile intera è immagazzinato, il carattere memorizzato viene modificato per poter distinguere tra un numero intero ed uno a virgola mobile.

L'interprete BASIC non immagazzina il suffisso %, ciò che accade è che al valore del codice ASCII di ciascun carattere del nome viene aggiunto 128 (cfr. l'appendice F del MUM riguardo al codice ASCII). Ad esempio il codice ASCII di X è 88 ed è questo il modo in cui il carattere X viene memorizzato per variabili a virgola mobile, mentre 88+128 è pari a 216, e questo è il modo in cui il carattere X viene memorizzato per variabili intere.

Nel caso di variabili intere entrambi i valori del codice ASCII vengono aumentati di 128, mentre per le stringhe soltanto il secondo valore è aumentato di 128; se non vi è il secondo carattere nel nome della variabile allora il valore memorizzato è 128. Per le variabili a virgola mobile di un solo carattere il secondo valore è pari a zero, in questo modo ciascun tipo di variabile può essere riconosciuto senza ambiguità.

Ritornando alle variabili intere, sette byte meno due byte dà cinque byte; tuttavia, come abbiamo notato, gli interi occupano soltanto due byte (e variano tra -32768 e +32767). Per comprendere il problema inserite

```
P=1E16
P%=P
```

al che la risposta è ?ILLEGAL QUANTITY ERROR.
Infatti, per quanto si tenti, non è possibile pigiare più di due byte in una variabile intera: che ne è degli altri tre byte?
Ciò che accade agli altri tre byte è che essi sono persi, non fanno nulla: dopo i due byte del nome della variabile ed i due byte del valore intero vi sono altri tre byte che contengono zero.
Pertanto l'uso degli interi non risparmia spazio ma anzi utilizza lo stesso spazio ed il programma gira più lentamente (i valori devono essere convertiti in numeri a virgola mobile e viceversa).
Anche memorizzare una variabile stringa X$ richiede sette byte: due byte sono ancora occupati dal nome della variabile (modificati per distinguere X$ da X o X%), il che lascia cinque byte.
Il primo dei rimanenti cinque byte è utilizzato per la lunghezza della stringa (di conseguenza una stringa può essere lunga da 0 a 255 caratteri, cfr. MUM); i successivi due byte servono come puntatore alla locazione di memoria dalla quale comincia la stringa: i due byte finali sono entrambi posti a zero.
Tutte le variabili sono memorizzate alla fine del programma BASIC e le locazioni 45 e 46 funzionano da puntatore per l'inizio delle variabili BASIC. Quindi, per trovare dove iniziano le variabili, battete

```
PRINT PEEK(45)+PEEK(46)*256
```

e se sottraete da questo numero la locazione d'inizio del BASIC (normalmente 2049) scoprirete l'estensione del testo del programma BASIC.
Quando l'interprete BASIC raggiunge il nome di una variabile in una linea di programma deve trovare il valore immagazzinato in quella variabile e per trovare tale valore deve innanzitutto trovare la variabile. Per far ciò l'inteprete prende il puntatore alle locazioni 45 e 46 e poi comincia a verificare i nomi delle variabili, cominciando dalla locazione alla quale rimandano le locazioni 45-46.
Se il primo nome della variabile non corrisponde l'interprete muove al successivo, e sa esattamente dove guardare: esso cerca facendo salti di sette byte alla volta. Il BASIC C64 è un tipo di BASIC chiamato «Microsoft» BASIC e i suoi progettisti hanno deciso di accelerare la ricerca delle variabili a costo di uno spreco di memoria.

Poiché l'interprete comincia sempre dall'inizio della lista di variabili il programma può essere reso più veloce definendo all'inizio del programma le variabili più comunemente richiamate. Se X è usata prima di X$ allora X sarà prima di X$ nella lista delle variabili; se comunque X$ fosse usata molto più frequentemente di X allora (particolarmente nei programmi più complessi) ha più senso avere X$ prima di X nella lista delle variabili.

Un breve esame del modo in cui i nomi delle variabili sono immagazzinati rivela che ciascuno dei due byte assegnati al nome possono essere «alti» (cioè >128) o «bassi» (cioè >0): ciò rende possibili quattro combinazioni e così, avendo solo tre tipi distinti di variabili, una delle possibili combinazioni non è utilizzata.

La quarta combinazione è destinata ai nomi delle funzioni definite dall'utente. Invece di un suffisso (ad esempio % o $), le funzioni sono distinte da un prefisso, cioè FN: il nome ed il «valore» di una funzione insieme utilizzano sette byte.

Il nome della funzione (privato di FN) è memorizzato in due byte come al solito, con il primo byte contenente il valore ASCII + 128; il secondo byte è invece il valore ASCII effettivo: questa è la quarta combinazione. Aggiungere ulteriori tipi di dati richiede una riscrittura completa dell'interprete.

I due byte successivi puntano al luogo del programma in cui è definita la funzione; questa è la ragione per cui la funzione deve essere stata definita all'interno della parte di programma che è già stata eseguita. Se la definizione di funzione non è stata ancora raggiunta allora sarà generato un errore.

```
10 DEF FNX(Y)=Y
20 W = FNX(8)
```

funzionerà e W assumerà il valore 8, ma invertendo

```
10 W = FNX(8)
20 DEF FNX(Y)=Y
```

non funzionerà perché il nome della funzione non sarà trovato.

Anche i successivi due byte si comportano come puntatore, e precisamente alla variabile usata nella definizione come parametro (cioè in FNX(Y) Y è il parametro); il byte finale non è utilizzato.
La classificazione dei tipi di variabile, e la codifica con cui è memorizzato il nome, è mostrata di seguito:

valore ASCII

Byte 0	*Byte 1*	*Tipo di variabile*
+0	+0	Virgola mobile
+128	+128	Intero
+0	+128	Stringa
+128	+0	Definizione di funzione

Per i primi tre tipi, se ci si riferisce alla variabile prima che le sia stato assegnato un valore, essa ha un valore nullo; per le variabili a virgola mobile e le intere il valore nullo è zero e per le stringhe il valore nullo è una stringa vuota.
Per la definizione di funzione non esiste un valore nullo e quindi si genera un errore. Battere in modo diretto

```
PRINT X,X%,X$,FN(X)
```

senza alcun programma in memoria, produce 0.0, un niente (cioè la stringa vuota X$), e poi ?UNDEF'D FUNCTION ERROR, pertanto state attenti con le funzioni.
Ecco perché è una buona norma inserire prima di eseguire qualsiasi cosa: in questo modo tutte le funzioni sono inizializzate.

I vettori «array»
Un vettore (detto anche «array», dall'inglese) è una collezione ordinata di valori tutti dello stesso tipo, tutti raccolti sotto lo stesso nome. La collezione è ordinata in modo tale che i singoli termini all'interno dell'insieme vengono selezionati in base alla loro posizione numerica (Cfr. MUM e PREG).
Quando il nome di una variabile è seguito da una espressione aritmetica tra parentesi, l'interprete BASIC assume che il nome si riferisca ad un vettore: se il vettore non è stato dimensionato allora esso viene inizializzato ad un valore nullo.

Il valore nullo di ciascun termine del vettore dipende dal tipo di vettore (se a virgola mobile, intero o stringa); il numero di termini nullo od omesso viene automaticamente posto ad 11 per tutti i tipi di vettore. È come se vi fosse una dichiarazione standard di questo tipo:

```
10 DIM X(10)
20 DIM X%(10)
30 DIM X$(10)
```

oppure, in alternativa,

```
10 DIM X(10),X%(10),X$(10)
```

I vettori possono avere più di una dimensione (fino al magico 255) ed in tal caso si dicono anche matrici; il numero di termini può giungere fino all'altrettanto magico 32767. Come possa essere immagazzinato in memoria un vettore a 255 dimensioni, ciascuna con 32767 elementi, è un altro problema.
I vettori sono memorizzati con due byte per il nome. È possibile avere un vettore chiamato X e contemporaneamente una variabile ordinaria X, poiché i vettori sono immagazzinati in un'area distinta di memoria. La locazione alla quale iniziano i vettori (e dove le variabili hanno fine) può essere trovata con

```
PRINT PEEK(47)+PEEK(48)*256
```

e la fine dei vettori (più 1) è alla locazione

```
PRINT PEEK(49)+PEEK(50)*256
```

Di seguito ai due byte del nome (con +0 o +128 a seconda del tipo), vi è un puntatore di due byte all'inizio del successivo vettore. Questo puntatore è necessario perché, al contrario delle semplici variabili, ciascun vettore può essere di lunghezza differente. Dopo questi quattro byte c'è un byte che fornisce il numero delle dimensioni del vettore, con il massimo pari a 255, il significato di questi cinque byte è lo stesso per tutti i vettori.
I successivi due byte contengono il numero di elementi dell'ultima dimensione del vettore. Nel caso di

```
DIM HW(1,900)
```

questo sarà pari a 901. Se vi è più di una dimensione allora il

numero di elementi della penultima dimensione è contenuto nei due byte successivi, e così via fiché non vi sono più dimensioni.
È a questo punto che i requisiti di memorizzazione divergono per ciascun tipo di vettore.
I vettori depositano gli elementi di ciascuna dimensione cosicché i valori di tutti gli elementi in ordine della prima dimensione (cominciando dall'elemento zero) sono memorizzati in successione. Per esempio, nel caso di HW, gli elementi sono memorizzati nell'ordine HW(0,0), HW(1,0), HW(0,1), HW(1,1).....HW(1,898), HW(0,899), HW(1,899), HW(0,900), HW(1,900).
Nel caso di vettori a virgola mobile ciascun elemento occupa cinque byte, come fanno i normali numeri a virgola mobile.
Gli interi, comunque, occupano solamente due byte e non hanno i tre byte superflui che troviamo con gli interi ordinari: la memorizzazine di vettori di interi è pertanto molto più economica che la memorizzazione di collezioni di normali variabili intere.
Occorrono tre byte per ciascun elemento di un vettore di stringhe, oltre al contenuto della stringa in sè stessa. Un byte fornisce la lunghezza della stringa e due byte funzionano come puntatore all'inizio della stringa, nella parte superiore della memoria; nella normale memorizzazione di stringhe vi sono dunque due byte superflui, mentre per i vettori non vi è alcun byte sprecato.
La PREG (pag. 9 dell'edizione inglese) fornisce un metodo per calcolare l'occupazione di memoria dei vettori, ma non dà spiegazioni: prevedete cinque byte per il nome, la lunghezza ed il numero di dimensioni, due byte per la grandezza di ciascuna dimensione e successivamente analizzate il tipo di vettore.
Per ciascun elemento di vettore a virgola mobile prevedete cinque byte; per ciascun elemento di una matrice intera due byte; e per ciascun elemento di una matrice stringa tre byte per la lunghezza ed il puntatore, più un byte per ciascun carattere della stringa corrispondente a quell'elemento.
L'occupazione di memoria di un vettore di stringhe può variare col procedere del programma, mentre lo spazio per i vettori numerici è fisso.

Lo Stack

Vi sono due locazioni fisse (FPA#1 e FPA#2) dove possono essere immagazzinati valori provvisori a virgola mobile, benché il sistema necessiti di più di due locazioni come queste, il BASIC opera con due numeri alla volta, ma spesso si desidera utilizzare più di due numeri.

Il problema di immagazzinare queste informazioni, di cui non si ha immediatamente bisogno, ma che saranno necessarie prima o poi, può essere risolto immagazzinando i valori in locazioni temporanee della memoria principale. Dire che il BASIC immagazzinerà i valori nella memoria principale non è sufficiente, poiché esso deve ricordare il posto dove sono immagazzinati i valori. Mantenere traccia di dove i valori sono immagazzinati potrebbe essere in se stesso un compito oneroso, per cui il sistema necessita di qualche meccanismo automatico.

Manteniamo il controllo richiesto utilizzando la struttura detta «stack» (catasta). Sul C64 lo stack è sistemato in memoria nelle locazioni da 256 a 511, ed accoglie i suoi elementi a partire dalle locazioni da 511 in giù: lo stack è perciò costituito da 256 locazioni.

Le locazioni da 256 a 511 sono comuni a tutti i computer che usano il microprocessore MOSTEK 6502, sebbene il C64 utilizzi una versione migliorata del 6502, chiamata 6510.

Cosideriamo come il sistema tratterrà questa espressione aritmetica:

```
PRINT(1+(1+(1+(1+(1+1)))))
```

Dopo aver esaminato il comando PRINT il sistema esamina l'epsressione seguente per decidere quale deve essere il risultato. Prima incontra la «(», ed immagazzina questa informazione nello stack (spinge l'informazione sullo stack); alla fine della valutazione dell'espressione la routine si aspetta di trovare una «)» opposta, altrimenti vi sarà un ?SYNTAX ERROR. Di seguito nella linea vi è un 1 che viene caricato in FPA#1.

Quando viene incontrato un «+» viene chiamata una routine di addizione, pronta a sommare insieme due numeri. Comunque successivamente non vi è un numero ma una «(»; il valore da FPA#1 ed il puntatore alla routine di addizione sono spinti nello stack, insieme al simbolo «(».

Questo processo continua fino a quando non viene incontrata la prima «)»: a questo punto i due numeri e l'operazione sono caricati dallo stack (tirati fuori -pull-) e viene eseguito il calcolo. Successivi incontri con «)» fanno estrarre dallo stack ulteriori informazioni da combinare con le istruzioni già presenti nell'accumulatore a virgola mobile.
Lo stack è utilizzato non solo per immagazzinare informazioni intermedie nella valutazione di espressioni aritmetiche, ma anche in ogni altra circostanza in cui il sistema necessiti di immagazzinare temporaneamente informazioni, e in cui l'ordine di memorizzazione dell'informazioni sia importante.
Queste circostanze ricorrono ovviamente nella valutazione di espressioni aritmetiche, ma si verificano anche nella memorizzazione delle informazioni dei cicli FOR e delle GOSUB: per questi ultimi due esempi vedi più avanti, quando saranno discusse le parole chiave del BASIC.
L'espressione matematica

```
PRINT 1+1+1+1+1
```

non è lo stesso che

```
PRINT(1+(1+(1+(1+1))))
```

almeno per il sistema. La seconda versione impiega molto più tempo per essere elaborata, perché è ampiamente usato lo stack: la differenza nel tempo impiegato può essere notevolissima.
Eseguite questo programma:

```
10 FOR I=1 TO 2000
20 X= 1+1+1+1+1
30 NEXT I
40 PRINT "#########"
```

che impiega circa 18.5 secondi dalla pressione del tasto RETURN all'output #########. Modificate lievemente il programma

```
10 FOR I=1 TO 2000
20 X = (1+(1+(1+(1+1))))
30 NEXT I
40 PRINT "#########"
```

ed ora il tempo impiegato è di circa 21.7 secondi; utilizzare l'assegnazione più complicata di X impiega circa il 17% in più di tempo.
Secondo la GRP (pag 15 della versione americana) non vi possono essere più di dieci livelli di parentesi all'interno di una espressione aritmetica poiché non vi è più spazio disponibile nello stack; ma questo non è del tutto vero.
Battete

```
PRINT ((((((((((1+1)+1)+1)+1)+1)+1)+1)+1)+1)+1)+1)+1)
```

e la riposta è 13, sebbene vi siano 12 livelli di parentesi. Però

```
PRINT(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+(1+1)+1)+1)
+1)+1)+1)+1)+1)+1)+1)+1)+1)
```

dà un ?OUT OF MEMORY ERROR. Il sistema BASIC non ha esaurito la memoria come tale, ma piuttosto ha esaurito le locazioni di memoria dello stack. Questo esempio mostra che quando due parentesi sono adiacenti (ad esempio ''((('') la routine utilizzata per valutare le espressioni aritmetiche agisce differentemente dove esse sono separate da un valore (es. ''(1+('').
Il C64 (ed altre macchine che usano una versione simile del BASIC) è perciò piuttosto limitato nell'affrontare espressioni matematiche complesse; tali espressioni devono essere analizzate in parti più semplici, e poi combinate per produrre il risultato.

LA SINTASSI BASIC PER IL C64

Operatori aritmetici

Vi son solo pochi operatori aritmetici nel BASIC del C64 di cui il più comunemente usato è l'operatore di assegnazione (o «uguale», =).
Questo è chiamato operatore perché è una direttiva per fare qualcosa: assegnare il valore dell'espressione sul lato destro alla variabile nominata sul lato sinistro. Ad esempio, X=Y prende il valore sul lato destro, Y (che potrebbe essere una espressione complessa), ed immagazzina quel valore in una variabile chiamata X, che è una variabile semplice.

Il termine sul lato sinistro non potrebbe essere nient'altro che una variabile semplice, come X, X(3,7), X%, X$(I), oppure X(1*2+1). Il tipo di variabile sul lato sinistro deve essere compatibile con il risultato dell'espressione sul lato destro. Assegnazioni possibili sono:

```
X=I%
X0=X1
I%=X
I0%=I1%
S0$=S1$
```

ed anche così vi sono restrizioni. Se X è più grande di 32767, o minore di -32668, allora l'assegnazione a I% produce un ?ILLEGAL QUANTITY ERROR.
Certe assegnazioni che dovrebbero essere ammissibili non lo sono a causa dei bug: considerate la sequenza

```
V=1.70141183E+38:PRINT V
V=(V/2)*2:PRINT V
```

che visualizza il valore corretto di V per la prima PRINT, ma che dà un ?OVERFLOW ERROR per la seconda assegnazione, senza mai raggiungere la seconda PRINT.
Il significato degli altri operatori aritmetici è piuttosto semplice da indovinare, poiché il loro significato è quello convenzionale. Questa semplicità è leggermente fuorviante, poiché T+T non è ad esempio lo stesso di T*2 o 2*T.
Sebbene questi due procedimenti siano identici in aritmetica, non lo sono nel funzionamento del C64. Il segno di addizione è un operatore ed esso opera su due numeri per produrre un risultato; l'operazione di addizionare insieme due numeri identici tuttavia non sempre produce la stessa risposta sul C64 che moltiplicare il numero per due.
La moltiplicazione è un operatore diverso, che non sostituisce delle addizioni successive. Nell'aritmetica ordinaria le nostre operazioni di moltiplicazione ed addizione sono coerenti tra di loro, ma sul C64 (e su molti altri computer) vi sono leggere differenze: per il C64 le differenze possono essere drammatiche.

Il solo operatore aritmetico che non è totalmente ovvio è l'operatore di esponenziazione, la freccia rivolta verso l'alto: ↑. Provate questa

```
PRINT 15*15
```

per la quale la risposta è 225. Ora battete

```
PRINT 15↑2
```

la cui risposta è sempre 225, e questo indica che il secondo valore, cioè quello dopo il segno ↑, è il numero di volte per cui il primo valore è moliplicato per se stesso: in altre parole, il numero dopo la ↑ indica la potenza alla quale deve essere elevato 15. Confrontate

```
PRINT 15*15*15,15↑3
```

entrambe le risposte sono 3375. Passando ad una potenza maggiore.

```
PRINT 15*15*15*15*15,15↑5
```

visualizza i risultati 759375 e 759375.001. La mancanza di uguaglianza si vede chiaramente: i numeri differiscono di .001. Quando battiamo

```
PRINT EXP(5'LOG(15))
```

viene visualizzato 759375.001.
La potenza di un numero è calcolata con l'uso di procedure simili alle funzioni EXP e LOG, e questo spiega perché la risposta sia spesso leggermente scorretta. Le dimensioni delle inesattezze possono essere giudicate per mezzo di questo breve programma:

```
10 FOR J=1 TO 10:FOR I=1 TO 5:REM J E' IL NUMERO
    ED I E' LA POTENZA
20 N=1:FOR K=1 TO I:N=N*J:NEXT K:REM N E' ORA J
   ELEVATO ALLA POTENZA DI I
30 PRINT J,I,J↑I-N:REM J↑I-N DA L'ERRORE
40 NEXT I:INPUT A$:REM ASPETTA PER MOSTRARE I RI
   SULTAT, PREMI RETURN PER CONTINUARE
50 NEXT J:REM PROSSIMO NUMERO
```

A parte i numeri (cioè J) 1,2,4 e 8, tutte potenze di 2, vi sono sempre inesattezze. Il C64 non è peggio di tanti altri microcom-

puter sotto questo aspetto, ma bisogna ricordarsi che la perfezione totale nei calcoli è impossibile, e che il C64 ha qualche altro piccolo bug supplementare.

Valutazioni relazionali

Il simbolo = appare di nuovo quando discutiamo di comparazioni relazionali; questa volta non indica una assegnazione ma indica una valutazione: X è uguale ad Y?
Per indagare la valutazione di uguaglianza provate a battere

```
I=567:PRINT I=I,I/3*3=I,I/3.7*3.7=I
```

le risposte sono -1, -1 e 0. Se il risultato di una valutazione è «vero» (es. -I=I) allora il valore numerico della valutazione è uguale a -1. È vero che I=I e che I/3*3=I (che è la ragione per cui otteniamo -1 per entrambi), ma non è vero che I/3.7*3.7=I, e quindi il valore numerico è 0.
Se noi battiamo

```
PRINT I/3.7*3.7-I
```

la risposta è -2.38418579E-07, e così noi possiamo sia verificare l'uguaglianza tanto con una sottrazione quanto con una valutazione relazionale (=). Alla linea 30 del programma precedente J↑I=N poteva essere usato per verificare quanto detto, tuttavia l'entità della differenza non sarebbe stata data dalla valutazione relazionale.
Il contrario di essere uguale è essere non uguale:

```
PRINT I/3.7*3.7<>I
```

dà il risultato -1, è vero che i due numeri non sono uguali. Questa reversibilità può salvarci da certe complicazioni nel programmare. Confrontate

```
10 INPUT N
20 IF N=100 THEN PRINT "UGUALE"
30 IF N<>100 THEN PRINT "NON UGUALE"
```

con questo programma:

```
10 INPUT N
20 IF N=100 THEN 50
30 PRINT "NON UGUALE"
40 GOTO 60
50 PRINT "UGUALE"
60 REM
```

e vedrete che l'uso di due istruzioni IF rende il primo programma più semplice da comprendere.
Un'altra valutazione relazionale è >, «maggiora di», usata ad esempio cosi:

```
PRINT I>1/3.7*3.7
```

per al quale la risposta è -1, «vero»: è vero che I è maggiore di 1/3.7*3.7. Il contrario di >non è <, ma piuttosto <=.

Se qualcosa non è più grande di qualcos'altro, potrebbe essere minore o uguale; un errore comune è quello di avere verifiche per > o per < e dimenticarsi il caso in cui vi sono valori uguali.
Poiché il risultato di una valutazione relazionale è un valore numerico, questi valori possono essere usati in espressioni aritmetiche.
Considerate due variabili X e Y: se Y è maggiore di X allora il valore di Y è immagazzinato in un'altra variabile Z, se X è maggiore di Y allore il valore di X è immagazzinato in Z. In altre parole, memorizza in Z il massimo tra X e Y. Se Y è il maggiore

```
Z=Y
```

e se X è il maggiore

```
Z=Y-(Y-X)
```

che è un modo di scrivere che Z è uguale a X: se X è maggiora di Y noi sottraiamo (Y-X) da Y. Per effettuare questa sottrazione perché non scrivere

```
Z=Y+(X>Y)*(Y-X)
```

che, quando X è maggiore di Y, determina (-1)*(Y-X)?

Tecniche come questa possono essere piuttosto utili, sebbene talvolta confondano i non iniziati: in una linea abbiamo due linee

```
IF Y<Y THEN Z=X
IF Y≥X THEN Z=Y
```

dove, notate, il contrario di «<» è «>», altrimenti, quando X=Y, non vi sarebbe alcun assegnamento di Z.
Le valutazioni relazionali funzionano anche con i caratteri, dove l'ordine in cui i caratteri sono valutati sono i valori del codice ASCII per quei caratteri (MUM app. F). Il risultato di

```
PRINT "A"< "B", "A" < ":"
```

è-1 e 0, poiché il codice ASCII per "A" è 65, mentre per "B" è 66 e per ":" è 58.
Esaminte la linea

```
PRINT "A"<"AA","AA"<"AB"
```

alla quale la risposta è -1 e -1. L'ordinamento non è solo per il codice ASCII del primo carattere di una stringa, al contrario la valutazione procede lungo la sequenza dei caratteri della stringa finché incontra la pima differenza.

Funzioni logiche

Il BASIC C64 ha tre funzioni logiche: NOT, AND e OR. Le funzioni operano su due tipi principali di input: considerate questo statement, dove X è maggiore di Z

```
IF X>Y AND Y>Z THEN MEZZO=Y
```

e considerate quando l'azione MEZZO=Y viene attivata. La condizione è vera quando (1) il valore di X è maggiore del valore di Y e, in aggiunta, (2) quando il valore di Y è maggiore del valore di Z, cioè la condizione globale è vera soltanto se entrambe le due sub-condizioni sono vere: se la condizione globale è vera allora Y è il valore di mezzo.
Questo è illustrato per mezzo di ciò che è noto come «tabella della verità»:

```
VERO AND VERO -> VERO
VERO AND FALSO -> FALSO
FALSO AND VERO -> FALSO
FALSO AND FALSO -> FALSO
```

la quale mostra che quando, e solo quando, entrambe le condizini sono vere, la condizione globale è vera. Pensando ai singoli bit nel computer:

1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0

L'altra principale funzione è «OR»: l'uso di OR è mostrato dalla

IF X>=Y OR X>=Z THEN PRINT''X NON È IL PIÙ PICCOLO''

Se è vero che X è maggiore o uguale a Y e/o se è vero che X è maggiore o uguale a Z, allora sappiamo che X non è il numero minore. Notate che il contrario di questa condizione è

IF X<Y AND X< THEN PRINT''X È IL PIÙ PICCOLO''

e cercate di scoprire il perché.
Le tavole dell verità per l'OR sono:

VERO OR FALSO → VERO
VERO OR FALSO → VERO
FALSO OR VERO → VERO
FALSO OR FALSO → FALSO
1 OR 1 = 1
1 OR 0 = 1
0 OR 1 = 1
0 OR 0 = 0

AND e OR sembrano essere il contrario l'uno dell'altro. Se in entrambe le tavole dei bit tutti gli 1 fossero cambiati in 0, e viceversa, allora una tavola sarebbe esattamente lo stesso dell'altra; questa semplice osservazione è codificata come le «regole di De Morgan».

A AND B è lo stesso che NOT(A) OR NOT(B)
A OR B è lo stesso che NOT(A) AND NOT(B)

Se venisse esaminata la tavola della verità dei numeri binari per l'OR, si vedrebbe che per porre un bit uguale a 1, si dovrebbe utilizzare la funzione OR con 1 per il bit corrispondente cioè

1 OR X = 1 (X=1 0 X=0)

Per lasciare inalterato il valore di un bit utilizziamo la funzione OR con 0:

0 OR X = X (X=1 o X=0)

Per vedere come utilizzare la funzione OR battete

```
POKE 53265,PEEK(53265) OR 32
```

ed appariranno strane cose sullo schermo. la linea che precede cambia il video in modo bit map (per il momento questo non ci serve, quindi battete STOP e RESTORE). per dare inizio al modo bit map, il numero 5 della locazione 53265 è cambiato in 1.
Le locazioni contengono dei byte, che sono otto bit, e se voi battete

```
PRINT PEEK(53265)
```

in circostanze normali, trovate che il numero immagazzinato alla locazione 53265 è 27.
Il numero decimale 32 corrisponde al numero binario 00100000. Quando facciamo una POKE nella locazione 53265, inseriamo il valore già presente (cioè PEEK(53265)), dopo aver effettuato l'OR con 32; effettuare l'OR con 32 significa porre il bite 5 ad 1 e lasciare tutti gli altri bit come sono:

bbbbbbbb OR 00100000 = bb1bbbbb

Porre ad 1 il bit 5 del byte alla locazione 53265, significa attivare il modo bit map, usato per i grafici in alta risoluzione.
Ora usate

```
POKE53265,PEEK(53265)OR32:FOR I=1 TO 1000:NEXT I:
POKE53265,PEEK(53265)AND223
```

e lo schermo commuterà per un breve istante alle stranezze di prima, poi ritornerà normale.
Guardando la tavole di verità dei bit per la AND:

1 AND X = X (X=1 = X=0)

0 AND X = 0 (X=1 o X=o)

Per porre un bit a 0, eseguite un AND con 0, o per lasciare inalterato un bit, eseguite un AND con 1; per cambiare a 0 il bit 5,

e lasciare immutati gli altri bit, fate un AND con il numero binario 11011111: questo ha valore decimale 223 (più facilmente calcolato come 255-32).
Nella precedente linea di programma si entra in modo bit map (ponendo ad 1 il bit 5 utilizzando un OR), c'è una breve pausa (il loop), e poi il bit 5 viene riportato a 0 utilizzando una AND: per comprendere qualsiasi PEEK o POKE che include una funzione logica, ricordate che OR pone a 1 e AND pone a 0.
Nel discutere i numeri a virgola mobile, è stato evidenziato che essi posseggono un bit del segno. Anche gli interi hanno un bit per il segno, ma il modo in cui il bit del segno lavora, sebbene perfettamente logico, in principio appare strano.
Un intero si estende su sedici bit, e se il bit a sinistra (bit 15) è 0 allora l'intero è positivo: se il bit 15 è un 1 allora il numero è negativo. Poiché vi sono solo 65536 differenti valori in due byte, doveva essere presa una decisione sul modo in cui immagazzinare la dimensione assoluta dei numeri.
Utilizzate queste due informazioni:

```
-1 + 1 =0
```

nella normale aritmetica, e

```
65535+1=0
```

nell'aritmetica a due byte (qualcosa di simile al contachilometri di un'automobile: se il contachilometri è a 999999, un chilometro in più è 000000): pertanto -1 è equivalente a 65535 nella aritmetica interna.
In aritmetica binaria, 65535 è 1111111111111111, e, poiché il bit 15 è 1 (tutti i bit sono posti a 1), questo si accorda con il bit nel segno posto a 1. Il numero binario 32768 è 1000000000000000 e, poiché il bit 15 è uguale a 1, il numero è negativo, equivalente a -32768 (ricordate che i limiti per gli interi erano +32768 e -32767 ?).
Questo metodo di immagazzinare i numeri negativi e positivi è chiamato «aritmetica del complemento a due», e spiega perché «vero» sia -1: se qualche numero è vero allora tutti i bit dovrebbero essere veri e tutti quei bit dovrebbero essere 1. Il numero con tutti i bit posti a 1 è 65535, che è trattato come il valore -1.

Un bit è «vero» se è 1, un valore è «vero» se è -1, ed entrambi sono coerenti tra loro nella aritmetica del complemento a due. Ricordate che

NOT(-64)=63
NOT(63)=-64

Il numero 63 è (come numero binario in sedici bit), uguale a 0000000000111111. Usare il NOT per ciascun singolo bit produce il numero binario 1111111111000000, che, come numero binario in complemento a due, è uguale a -64.

Parole chiave del BASIC

Nell'elenco di parole chiave che segue non vi è alcun tentativo di replicare la descrizione data dalla MUM (App. C) o nella GRP (capitolo 2). L'elenco non fornisce alcuna delle parole chiave delle funzioni logiche, poiché la loro natura ed il loro uso sono stati discussi in dettaglio (vedi sopra).

ABS(X) Questa funzione calcola il valore assoluto della espressione aritmetica in parentesi; l'espressione può essere a virgola mobile, intera, logica.
ABS funziona utilizzando l'accumulatore a virgola mobile no.1, (FPA#1), e la routine muove tutti i bit nel byte del segno di un posto verso destra, cosicché il bit del segno appare positivo.

ASC(S$) Calcola il valore ASCII del primo carattere della stringa S$. I codici ASCII usati sono quelli particolari del C64, sebbene per numeri, caratteri, simboli standard, questi siano i valori ASCII standard. La stringa può essere il risultato di ogni insieme lecito di operazioni per creare stringhe.
La funzione ASC agisce elaborando l'espressione della stringa (es. forma "AB" prima trovando ASC("A" + "B")) ed il puntatore della stringa (senza nome) viene lasciato nello stack; la funzione ASC esamina il puntatore e carica il primo carattere della stringa per la verifica.
L'applicare la funzione ASC ad una stringa vuota, cioè ASC(""), dà un ?ILLEGAL QUANTITY ERROR, un altro bug del C64 che

sembra derivare dal BASIC di altre macchine Commodore. La stringa nulla, cioè "", dovrebbe avere valore ASCII 0, come può essere visto da

```
PRINT CHR$(0);"#"
```

in cui la stringa nulla è prodotta, e seguita, nella prima colonna a sinistra, da "#" (CHR$ è la funzione opposta di ASC). Comunque, battete

```
PRINT ASC(CHR$(0))
```

e la risposta è zero.
Il bug deve derivare dall'uso dei puntatori di stringa, perché "" ha la lunghezza zero, e la routine non può ricevere una stringa (nulla) di lunghezza zero. CHR$ (vedi più avanti) restituisce sempre una stringa di lunghezza 1, anche se nulla.

ATN(X) Questa funzione calcola l'angolo, in radianti, che corrisponde al valore di una certa tangente: l'angolo è chiamato arcotangente. L'angolo risultante è sempre nell'intervallo da -PI/2 a PI/2, e se si sa che il risultato potrebbe essere fuori da questi limiti, allora devono essere fatte delle modifiche (PI è ilvalore del simbolo speciale che è prodotto utilizzando SHIFT e la freccia in su↑).
4*ATN(1) dovrebbe eguagliare il valore PI, e

```
PRINT 4*ATN(1)-PI
```

produce il risultato 9.31322575E-10.
Due volte l'arcotangente di infinito è anch'esso uguale al valore PI, e quindi

```
PRINT 2*ATN(1E38)-PI
```

dove 1E38 è la nostra approssimazione di infinito. La differenza è ancora di 9.31322575E-10, la quale mostra che è una approssimazione piuttosto accurata.
La routine della funzione ATN utilizza una speciale formula di approssimazione, che nella pratica funziona bene.

CHR$(X) Questa funzione converte una qualsiasi espressione numerica, all'interno dei limiti 0 - 255, nel corrispondente carattere ASCII; se l'espressione è a virgola mobile, viene troncata con la perdita dei decimali. Il carattere creato è una stringa di lunghezza 1 byte, questa è la ragione per cui la funzione ASC funziona con successo sul CHR$(0).

Il valore ASCII per il carattere " (doppie virgolette) è 34, e perciò battete:

```
PRINT CHR$(34);"DOPPIE VIRGOLETTE";CHR$(34)
```

per avere DOPPIE VIRGOLETTE tra virgolette, cioè "DOPPIE VIRGOLETTE".

CLOSE I% Usato per «chiudere» i file sulle periferiche, questo comando completa l'intera elaborazione di un file, e cancella i dettagli del file dalle tre speciali «tavole dei file» contenute nella RAM: per maggiori dettagli vedere la parola chiave OPEN.

CLR Questo cancella la memoria delle variabili, delle funzioni definite, delle stringhe, dei file, e cancella qualunque cosa lasciata nello stack.
Il comando opera resettando un insieme di puntatori:

Puntatore	Locazioni
1. Inizio del programma BASIC	43-44
2. Inizio delle variabili	45-46
3. Inizio degli array	47-48
4. Fine degli array (+1)	48-50
5. Inizio delle stringhe	51-52
6. Deposito temporaneo stringhe	53-54
7. Fine della memoria BASIC	55-56

Quando è attivato CLR, il puntatore 1 non è cambiato e punta sempre all'inizio della memoria BASIC; neppure il puntatore 2 è modificato, poiché questo è il punto dove termina la memoria per il programma BASIC. Il puntatore temporaneo per le stringhe (6) punta all'inizio della stringa definita in memoria per ultima, e

neppure questo è coinvolto. La fine della memoria riconosciuta dal sistema BASIC, cioè il puntatore 7, è anch'esso lasciato indenne.
Il puntatore 3 è posto uguale al puntatore 2, ed il puntatore 4 è posto a 1 in più del puntatore 2; il puntatore 5 è reso uguale al massimo della memoria BASIC, cioè il puntatore 7, ed il risultato di questi cambiamenti è che il sistema BASIC non è più informato su qualsiasi variabile o array.
Il puntatore può essere analizzato con un semplice programma

```
10 DEF FNP(I)=PEEK(I) + PEEK(I+1)*256
20 FOR T=43 TO 55 STEP 2
30 PRINT FNP(T)
40 NEXT T
```

che visualizza tutti i valori dei puntatori in ordine. L'aggiunta di una linea

```
15 FOR T=1 TO 35:A$=A$ + "!!!!!":NEXT T
```

altera la lettura dei puntatori delle stringhe più degli altri.
La parte bassa dell'immagazzinamento delle stringhe è ora molto più bassa, e la differenza tra i due puntatori delle stringhe (5 e 6) è ora 175, o la lunghezza dell'ultima A$(35*5). Lo spazio disponibile per le stringhe è ora utilizzato con tutte le differenti copie di A$, cioè 35, cominciando con «!!!!!», poi «!!!!!!!!!!» e così via.
Immettendo

```
CLR:PRINT PEEK(51)+PEEK(52)*256
```

troviamo che la locazione è nuovamente vicina alla parte superiore della memoria: il puntatore 5 è stato riinizializzato per l'utilizzazione del CLR. L'uso del PEEK alle locazioni 53 e 54, mostra che il precedente CLR non ha influenzato il puntatore transitorio delle stringhe (numero 6).

CMD I%,S$ Questo comando inizializza il device (periferica) numero I% perché riceva ogni ulteriore output, invece di mandare l'output sul video. Il device deve essere prima attivato con

un comando «OPEN I%,D%», dove D% è l'effettivo numero di device, e I% è il numero di file. Dopo il comando CMD lo schermo non visualizza più l'output, finché non è riattivato con

PRINT#I%,:CLOSE I%

che rende il device con numero di file I% «sordo». PRINT#I% permette la scrittura al file I%, ma nondimeno permetterà ad un programma di essere listato sullo schermo. CMD I%, comunque, prende qualsiasi output, incluso un LIST, e lo manda al device con numero di file I% (Cfr. PRINT# per maggiori dettagli).
Se il secondo parametro (stringa) è presente, allora i contenuti della stringa sono mandati in output al file specificato.

CONT Questo fa ripartire un programma BASIC dopo uno STOP o un END nel programma, o dopo che è stato pigiato il tasto STOP. Se è stata inserita una nuova linea di programma (con un numero di linea), il CONT non funzionerà, poiché i puntatori sono stati alterati: inoltre essi possono alterarsi se vi è un errore di sintassi, o un comando CLR.
Non tutti i comandi alterano i puntatori e quindi è possibile usare il LIST e successivamente utilizzare il CONT.

COS(X) Questa funzione calcola il coseno corrispondente all'angolo X, dove X è presupposto essere in radianti. Se un angolo è espresso in gradi, allora deve essere moltiplicato per PI/180, ad esempio.

PRINT COS(GRADI*PI/180)

L'espressione all'interno della parentesi viene valutata, il risultato immagazzinato in FPA#1, e poi chiamata la routine SIN (seno): il valore di COS(X) è pari a SIN(X+PI/180) e così la routine SIN opera sull'espressione più PI/2.

DATA Frequentemente vi sono, all'interno, di un programma, certi elementi di informazioni che non variano mai da una operazione all'altra di un programma. Ad esempio, a livello di routine in linguaggio macchina per calcolare il SIN (seno), vi sono numerosi valori fissi che sono utilizzati tutte le volte che viene chiamata la funzione SIN.

Per il SIN vi sono valori fissi che sono permanentemente immagazzinati nella ROM (Read Only Memory, memoria a sola lettura) e riprese ogni volta che viene utilizza la routine: in un programma BASIC la stessa cosa è fatta dalle istruzioni DATA.
Vi è una permanenza nell'utilizzo delle istruzioni DATA, il che significa che un errore nei dati è più facilmente rettificabile. Ad esempio l'usare le istruzioni INPUT nell'elaborazione di dati statistici lascia aperta la via agli errori di inserimento: con gli statement DATA uno può rettificare gli errori, modificando l'istruzione appropriata.
Un errore in una parte INPUT all'interno di un programma spesso significa che deve essere effettuata una nuova esecuzione.
I termini di una istruzione DATA possono essere una qualunque cosa che ci si aspetta sia valida per le INPUT: il modo in cui le DATA sono lette dall'istruzione READ è esattamente lo stesso insieme di routine delle INPUT, così che si applicano certe restrizioni. Un punto importante da ricordare, vero sia per le DATA sia per le INPUT, è che una virgola prima dell'inserimento è presa o per un numero di valore zero o per una stringa (nulla) di valore "".
Altro punto importante da ricordare è che, nel caso delle stringhe, se le stringhe contengono virgole, l'intera stringa deve essere racchiusa tra virgolette. Ad esempio

```
1000 DATA "UNA STRINGA, CON UNA
     VIRGOLA INTERNA"
```

Errori comuni nelle istruzioni DATA sono sviste nell'istruzione READ, ma lo vedrete dopo. Posizionando il cursore su un «READY.» e battendo RETURN si determinerà un ?OUT OF DATA ERROR, poiché il sistema BASIC pensa che READY. sia
READ Y.

DEF FNYZ(X) Questo è il modo in cui sono inserite le funzioni definite dall'utente. Il parametro deve essere una variabile a virgola mobile e le stringhe non sono consentite.
Inserire 10 DEF FNII(H%)=X e poi dare il RUN genera un ?SYNTAX ERROR IN 10, Se una stringa viene usata come parametro

viene prodotto un ?TYPE MISMATCH ERROR: non è consentito definire DEF FNST(A$)=ASC(A$). Soltanto una variabile reale è consentita come parametro di una funzione, e benché un parametro possa essere inutilizzato, ci deve essere sempre.

DIM X0(X1) Questa istruzione riserva spazio in memoria per un vettore di nome specificato, con X1+1 elementi. Gli elementi del vettore sono numerati da zero al numero nella parentesi dopo il nome del vettore.
L'esempio mostrato è un vettore a virgola mobile, ma sono possibili anche vettori interi o stringa.
Lo stesso vettore non può essere dimensionato due volte nello stesso programma. Per un esame più completo dell'argomento cfr. il paragrafo relativo.

END Termina un programma e scrive READY. Il programma può essere rilanciato usando CONT, se certe condizioni sono soddisfatte (cfr. CONT). Confrontate con STOP.

EXP(X) Calcola il numero di Nepero e (=2.718281828...) elevato alla potenza del valore X. Ad esempio

```
PRINT EXP(1)
```

restituisce il valore 2.71828183, e

```
PRINT EXP(2)
```

restituisce il valore 7.389051,

```
PRINT EXP(1)*EXP(1),EXP(1)↑2
```

restituisce il valore 7.3890561 per entrambe le espressioni.
Notate che

```
PRINT LOG(EXP(X))
```

dovrebbe dare il valore X. Questo programma cerca di verificare l'accuratezza dei valori delle varie X:

```
10 FOR X=0 TO 10
20 PRINT X,LOG(EXP(X))-X
30 NEXT X
```

e soltanto per X=3 sembrerebbero esservi degli errori: l'errore per X=3 è il ripetuto 9.31322575E-10
Per una verifica di tipo differente, ha valore ricordare che EXP(88) dovrebbe eguagliare EXP(87)*EXP(1). Provate a battere

```
PRINT EXP(88)
```

ed il risultato è 1.65163625E+38. L'istruzione (numericamente) equivalente.

```
PRINT EXP(87)*EXP(1)
```

genera, come ci si poteva aspettare, un ?OVERFLOW ERROR: i bug aritmetici del BASIC C64 colpiscono ancora!

FOR X0=X1 TO 2 STEP X3 Questa sequenza permette l'elaborazione ripetuta delle linee BASIC tra questa istruzione e la corrispondente istruzione NEXT X0.
La variabile del loop (ciclo), cioè X0 in questo esempio, deve essere una variabile a virgola mobile e non un intero. Il valore di inizio, fine ed incremento della variabile del loop (cioè X1, X2, X3 nell'ordine), possono essere solo espressioni aritmetiche.
La variabile del loop non deve essere soltanto a virgola mobile, ma deve anche essere una normale variabile: questo significa che X è ammissibile mentre la variabile complessa X(5) non è permessa. Poiché il contatore del loop è una variabile a virgola mobile, vi possono essere alcuni errori nell'arrotondamento. Ad esempio:

```
FOR I=1 TO 2 STEP .0001:NEXT I: PRINT I
```

dà il risultato 2.0000983, mentre la risposta avrebbe dovuto essere 2.0001. La ragione per cui il valore finale di I dovrebbe essere 2.0001 e non 2.0000 è resa più chiara dalla

```
FOR I=1 TO 2:NEXT I:PRINT I
```

poiché il valore finale di I è 3.
Quando l'interprete BASIC incontra un'istruzione FOR prende il primo valore del contatore di loop, ed esegue il loop per almeno una volta. Nel caso di I che varia da 1 a 2, il valore di I è incrementato di 1 quando viene raggiunto la istruzione NEXT I; questo valore incrementato è poi confrontato con il valore limite del loop, e se la variabile del loop è maggiore del limite del loop

stesso allora l'esecuzione del ciclo si arresta.
Dopo un passaggio del loop I è uguale a 2, e questo non è più grande del limite del ciclo (che è uguale a 2), perciò il loop agisce nuovamente. La volta seguente (NEXT) I=3, che è maggiore del limite del loop, quindi il loop termina con I=3 e non con I=2.
Se l'inizio del loop è maggiore del suo limite, cioè X1>X2, allora dovremmo supporre che il valore di incremento (STEP), cioè X3, debba essere negativo.
Ad ogni modo

```
FOR I=1 TO 0:PRINT I:NEXT I:PRINT I
```

fornisce i valori 1 e 2, dopo di che il loop termina. Anche se per qualche errore il valore iniziale ed il valore finale non sono corretti, il loop viene eseguito sempre una volta.
Il loop

```
FOR I=1 TO 2 STEP 0:PRINT I:NEXT I
```

non finisce mai, e viene visualizzato un flusso continuo di 1. Questo è un modo per produrre un loop infinito o un loop che agisce finché non si verifica una condizione. Qui di seguito vi è un programma che agisce finché non viene premuto il tasto F.

```
10 FOR I=1 TO 2 STEP 0
20 PRINT "#";
30 GET X$
40 IF X$ = "F" THEN I=2
50 NEXT I
```

Vi è un loop infinito nelle linee da 10 a 50, che visualizza dei # per riempire lo schermo. Alla linea 30 vi è una verifica della tastiera per controllare se è stato premuto qualche tasto, ove il risultato della ricerca è immagazzinato come carattere in X$; la linea 40 è l'equivalente di un'istruzione UNTIL (finché).
Se (IF) il carattere immagazzinato in X$ è F allora (THEN) il valore I è posto uguale a 2. Quando viene raggiunto la successiva istruzione NEXT I, la variabile I eguaglierà il suo parametro limite ed il loop terminerà.
Un problema con questa forma di istruzione di controllo è che prende una gran quantità di spazio nello stack della macchina per

immagazzinare informazioni riguardo a ciascun parametro: lo STEP è sempre incluso, anche se è sottointeso il valore 1. I loop «nidificati», cioè loop interni ad altri loop, divorano lo spazio dello stack che ben presto si esaurirà: un ciclo FOR...NEXT può utilizzare fino a 18 byte nello stack.
Se vi sono troppi loop FOR..NEXT (o troppi GOSUB - vedi oltre) potrebbe generarsi un ?OUT OF MEMORY ERROR. In effetti la memoria non si è esaurita, ma lo stack è troppo pieno.

FRE(X) Calcola il numero di byte disponibili per il BASIC: prende il puntatore della fine degli array (vedi sopra CLR) ed il valore del puntatore delle stringhe, che indirizza al termine delle stringhe.
Prima di esaminare i puntatori la FRE cancella dalla memoria qualsiasi stringa verso la quale non vi è alcun puntatore dall'interno delle variabili e degli array. Questo significa che se vi sono numerose stringhe che non sono più attive, vengono considerate e mantenute solo le stringhe attive. Le ultime stringhe attive sono poste insieme, in sequenza, dal limite superiore della memoria verso il basso.
Battete

```
10 FOR I=1 TO 2 STEP 0
20 PRINT "#";
30 N$=N$ + "!!!!!!"
40 NEXT I
```

e successivamente, dopo che il programma è stato interrotto con un ?STRING TOO LONG ERROR IN 30, battete

```
PRINT PEEK (51)+PEEK(52)*256
```

che fornirà un certo valore tra 30000 e 35000. Il battere successivamente

```
PRINT FRE(0),PEEK(51)+PEEK(52)*256
```

darà un valore negativo per FRE(0), e la nuova fine delle stringhe sarà maggiore di 40000.
La differenza del valore dell'inizio delle stringhe è un'indicazione dello spazio perso nell'effettuare copie successive di N$: FRE(0) libera questo spazio per mezzo di un processo noto come «gar-

bage collection» (raccolta della spazzatura). Il parametro di FRE non è adoperato in alcun modo.
La ragione per cui il risultato di FRE è negativo è che FRE fornisce gli spazi liberi come numero in complemento a due; per cambiare il valore negativo nel valore positivo corretto utilizziamo il fatto che, se FRE è negativo (FRE(0)<0) è uguale a -1 (vedi sopra per maggiori dettagli sull'uso delle valutazioni relazionali).
Usate

```
PRINT FRE(0)-(FRE(0)<0)*65536
```

per ricavare dalla versione in complemento a due il valore normale.

GET S$ Questo comando cerca di leggere un carattere dalla tastiera, come esempio si veda il programma FOR per emulare l'UNTIL.
Nel momento in cui i caratteri sono battuti sulla tastiera, vengono memorizzati in una parte di memoria chiamata «buffer», dalla locazione 631 alla locazione 640. Quando viene usato ciascun carattere (memorizzato con il codice ASCII corrispondente) tutti i valori avanzano di un byte ed un nuovo carattere può essere immagazzinato nel buffer.
Nella locazione 198 è immagazzinato il numero dei termini contenuti nel buffer di tastiera, cioé dei valori da 0 a 10: il numero massimo di valori che può essere immagazzinato è 10. Il numero degli elementi contenuti nel buffer di tastiera può essere modificato variando il valore della locazione 649:

```
PRINT PEEK (649)
```

normalmente fornisce il risultato 10.
La routine GET ha una speciale funzione che usa solo il primo elemento del buffer di tastiera. INPUT copia gli elementi in un buffer di input di sistema (locazioni da 512 a 600) ed aspetta finché trova un valore ASCII 13 (cioé il ritorno di carrello). Se il comando GET si aspetta un numero (es. GET X) ed è inserita una lettera, allora vi è un ?SYNTAX ERROR. Ecco perché è meglio usare GET S$, e poi usare VAL(S$) (vedi oltre). Se non è premuto nessun tasto mentre è attivo un comando GET, viene restituita o una stringa nulla o un valore numerico zero.

GET #I%,S$ Funziona in modo molto simile a GET, ma legge i caratteri singolarmente dal file specificato, invece che da tastiera. Lo schermo video è il device numero 3, e se viene battuto questo programma

```
1 OPEN 1,3
10 GET#1,S$:PRINT S$;:IF S$<>"$" THEN 10
```

ed il programma viene listato, quando il cursore è posizionato in cima allo schermo, l'esecuzione del programma modifica quanto è sul video.
Appaiono sullo schermo molte lettere e simboli ripetuti dove prima erano i caratteri. Ciò che è accaduto è che il numero di device 3 è stato assegnato al file logico numero 1 dal comando OPEN (vedi più oltre). Posizionando il cursore in cima allo schermo e poi battendo RUN lo schermo viene trattato come un file ed ogni posizione viene esaminata nell'ordine.
Poiché il contenuto di una locazione di schermo, ottenuto per mezzo del GET# è poi visualizzato dove è stato trovato, i simboli si ripetono. Vi sono parecchi simboli $ nel listato del programma, ed è praticamente certo che almeno uno viene incontrato, ma finché non viene trovato un simbolo $ il programma ripete la linea 10.

GOSUB numero Questo comando salta (come il GOTO) ad una qualsiasi linea in un programma, ma se il numero di linea non esiste vi è un ?UNDER'D STATEMENT ERROR.
Diversamente dall'istruzione GOTO, il GOSUB passa il controllo ad un'altra porzione di programma, ma si aspetta che il controllo ritorni all'istruzione seguente al GOSUB. Poiché il controllo deve essere restituito vi deve essere una istruzione RETURN.
Prima battete

```
0 PRINT "0"
1 PRINT "1"
2 RETURN
```

e poi indagate le chiamate (teoricamente senza senso)

GOSUB
GOSUB X
GOSUB GOSUB
GOSUB 0X

che sono tutte interpretate come chiamate di subroutine alla linea 0, come è indicato dalla visualizzazione di entrambi i numeri 0 e 1. Questa azione è dovuta al fatto che la routine operativa per esaminare il numero che segue il GOSUB è una stretta parente della funzione VAL.

Inserite

GOSUB 1/2
GOSUB 1GOSUB
GOSUB 1X

per ciascuna delle quali la risposta è 1, il che mostra che tutti i salti sono alla linea 1. Questo è esattamente ciò che uno si sarebbe aspettato, dato il comportamento della VAL (vedi oltre). La routine che esamina il contenuto della linea seguente il GOSUB (chiamata CHRGET, contenuta nelle locazioni da 115 a 138) si ferma al primo carattere non numerico.

Il GOSUB utilizza fino a 5 byte dello stack, che include un puntatore alla locazione in cui deve essere letto il carattere seguente il RETURN, il numero di linea corrente ed uno speciale token che indica che è stata chiamata una subroutine. Per sapere quante subroutine possono essere chiamate in una volta sola, usate il seguente programma:

```
10 INPUT N
20 F = 1
30 GOSUB 100
40 PRINT F
50 END
100 IF N=0 THEN RETURN
110 N = N-1
120 GOSUB 100
130 N = N+1
140 F = F*N
150 RETURN
```

La subroutine delle linee da 100 a 150 calcola il fattoriale di N, riportando il risultato in F: la subroutine è ricorsiva per il fatto che richiama se stessa (alla linea 120). I valori di N da 0 a 22 funzionano (ed il fattoriale di 22 è 1.12400073E+21), ma ad un valore in ingresso di N=23, vi è un ?OUT OF MEMORY ERROR IN 100.

La subroutine mostra un interessante uso di due RETURN: questo è un modo in cui le subroutine possono essere più flessibili dei GOTO.

GOTO e **GO TO** Entrambi sono salti diretti al numero di linea che segue il comando; GOTO non è la stessa cosa delle due parole separate GO TO, sebbene l'effetto sia lo stesso.

L'interprete BASIC tratta il GOTO come una sola unità, mentre considera il GO TO come due unità. GO è considerato come una parola chiave, e quando viene trattato come tale deve essere seguito da uno spazio ed il TO; il codice interno per il GOTO è differente dal codice per il GO.

La valutazione del numero di linea seguente il GOTO utilizza la routine tipo VAL, così come il GOSUB: perciò il GOTO ha disordini simili al GOSUB.

Con il GOSUB, il programma gira più in fretta se abbiamo le subroutine usate più di frequente vicino all'inizio del programma, questa è la ragione per cui molti programmi iniziano con una GOTO verso la fine del programma, saltando le subroutine usate di frequente. Poiché le subroutine sono progettate per essere usate da molte parti differenti del programma, ha senso cercare di prendere la posizione migliore.

L'ottimizzazione della ricerca dei numeri di linea utilizzando il GOTO può avere alcune interessanti conseguenze, se portata ai limiti estremi.

Regola 1: non saltate mai ad un numero di linea che è prima della linea in cui effettuate la chiamata. La routine del numero di linea verifica se il numero di linea verso il quale è effettuato il salto è prima o dopo la linea attuale: se il numero di linea è dopo allora la ricerca inizia dalla linea attuale, e non dall'inizio del programma.

Regola 2: non saltate mai ad una linea che è appena prima della linea attuale, saltate ad una linea vicino all'inizio; più la linea è vicina all'inizio, più presto sarà trovata.
Regola 3: non prendete le regole troppo seriamente.
Vi sono tre possibili forme di comando GOTO in associazione con l'istruzione IF:

IF condizione THEN GOTO numero-linea
IF condizione THEN numero-linea
IF condizione GOTO numero-linea

e IF...GOTO... è più veloce di IF...THEN...; ma IF...GO TO è inammissibile (produce un ?SYNTAX ERROR). Questo è dovuto alla differente interpretazione del GOTO e del GO TO; ed usare IF...THEN GO TO... non genera alcun errore.
È meglio mantenersi distanti dalle GO TO, poiché queta parola chiave sembra essere un ripensamento ed è incline ai bug.

IF...THEN Questa istruzione condizionale ha due principali funzioni. In primo luogo può essere usata per dirigersi verso una linea di programma oppure, può essere utilizzata per eseguire delle istruzioni sulla stessa linea dell'IF. L'accento è su «la stessa linea», cosicché l'istruzione immediata

IF X=X THEN :PRINT "#####"

produrrà #####, così come

IF X=X THEN PRINT "#####"

Nel primo caso, comunque, non vi è nessuna istruzione che segue il THEN, poiché il THEN è immediatamente seguito dal separatore di istruzioni, cioé i due punti «:». Come terzo caso battete

IF X<>X THEN :PRINT "#####"

e non apparirà nulla sullo schermo.
La routine associata con l'IF, valuta l'espressione seguente l'IF, e poi (posto che vi sia anche un THEN o un GOTO) viene verificato l'esponente del valore in FPA#1. Il valore in FPA#1 è il risultato dell'espressione dopo l'IF e, se l'esponente è zero, il risultato è zero, ed il controllo passa al numero di linea seguente, non all'istruzione successiva sulla stessa linea.

Pertanto, se la condizione è vera, saranno attivate tutte le istruzioni su quella linea di programma; se è falsa, tutto il resto delle istruzioni su quella linea di programma vengono ignorate (confrontate il caso 1 ed il caso 3). Non è necessario che vi sia alcuna istruzione di seguito al THEN.

INPUT Ha molte caratteristiche in comune con il READ (vedi più oltre), particolarmente per l'importanza delle virgole, virgolette e due punti.
Di seguito alla parola INPUT vi può essere una stringa per identificare il dato richiesto in ingresso, ad esempio

```
10 INPUT "VERIFICA";A$
```

e due linee aggiuntive aiutano ad indagare il comando INPUT:

```
20 PRINT A$
30 GOTO 10
```

Dunque provate i seguenti input:

```
J
J,
,J
J:
J;
J,
```

Per il primo inserimento viene visualizzato J, come per il secondo, sebbene con l'informazione ?EXTRA IGNORED. La routine per accettare i valori per INPUT considera la virgola come separatore, e pertanto l'utilizzatore ha inserito due valori: soltanto un valore è atteso e di conseguenza il secondo è ignorato (non è possibile immagazzinare valori per ulteriori INPUT).
Per il terzo inserimento è visualizzata una stringa nulla, e la parte eccedente, J in questo caso, viene ignorata. Siccome il primo termine che la routine incontra è una virgola, essa assume un input nullo. Il quarto inserimento mostra come anche i due punti hanno effetti simili alla virgola, visto che J è visualizzato ed il resto ignorato.

Il quinto input (cioé J;) è visualizzato esattamente così (cioé J;); il punto e virgola non è un separatore e in effetti è trattato come un carattere ordinario.

L'inserimento finale mostra l'uso delle doppie virgolette per rendere conscia la routine che l'insieme di caratteri seguente è solamente questo, un insieme di caratteri: se nell'insieme vi sono virgole (ecc.) devono essere prese in considerazione. In questo caso l'output è J,.

Per illustrare un bug nelle routine INPUT, battete semplicemente RETURN dopo l'ultimo termine, e di nuovo la stringa in output è J. Battendo RETURN l'input non inserisce una stringa nulla, ma la stringa contiene il suo valore precedente.

L'uso delle virgolette nelle INPUT permette l'inserimento di comandi grafici, come, ad sempio, il comando CLR, ed in questo aspetto la situazione ha molto in comune con la PRINT. Cambiate due linee nel programma,

```
10 INPUT "VERIFICA";A
20 PRINT A
30 GOTO 10
```

e poi battete

```
1
2E3
2,
,8
5:
2/3
```

Il primo non è problematico e l'output è 1; il secondo inserimento è anch'esso senza problemi e l'output è 2000. Il terzo inserimento dà ?EXTRA IGNORED, e la risposta è 2 (il quinto inserimento 5: è simile). Il risultato dell'inserimento del quarto input è ?EXTRA IGNORED ed il valore visualizzato di A è 0: la virgola è letta come inserimento nullo, cioé zero.

L'input 2/3 non è consentito e quindi viene visualizzato un messaggio ?REDO FROM START. Inserendo successivamente un valore valido è visualizzato quest'ultimo. Se comunque è inserito

il valore 2/3 (e l'errore viene segnalato) e poi si batte semplicemente RETURN si visualizza il valore 2.
Questo è ancora un altro bug. Ciò che è successo è che l'espressione 2/3 è stata valutata fino alla / dal CHRGET (in modo simile al VAL o GOSUB/GOTO), e successivamente viene evidenziato l'errore. Battendo RETURN nessun valore viene mandato ad A, ed il valore già esistente, cioé il 2 di 2/3, è trattato come il valore corretto.
Quando l'utente sta battendo molte informazioni è molto facile commettere tali errori, battendo RETURN prima del numero giusto.
Talvolta, quando è operativo un comando CMD, l'INPUT cerca di prendere i dati dalla periferica di tipo sbagliato, ad esempio la stampante, e si visualizza un ?FILE DATA ERROR. Se vi è un messaggio con l'INPUT, ad esempio il VERIFICA di prima, allora il messaggio è mandato alla periferica, il che può produrre delle complicazioni.

INPUT Questo comando prende i dati da una certa periferica, con l'esatto formato atteso per un comando INPUT ordinario, sebbene non vi sia alcun messaggio. Il file deve essere prima aperto.
Se il dato è stato mandato alla periferica con il PRINT#, il formato è esattamente come richiesto perché i due comandi sono coerenti.

INPUT# è più puntiglioso dell'INPUT ordinario per i tipi di dati ed i formati e, sebbene non vi siano avvertimenti, la parte eccedente sarà ignorata. Le routine INPUT e INPUT# sono praticamente identiche, a parte l'assegnazione del file su INPTU#.
Sia INPUT che INPUT# usano un buffer di 80 byte (cfr. GET e GET#), e questa è la ragione per cui INPUT non può essere usato in modo diretto: i comandi in modo diretto sono immagazzinati nello stesso buffer.

INT (X) Questa funzione converte l'espressione in virgola mobile tra parentesi in un valore intero che è minore o uguale al risultato

dell'espressione. Non c'è alcuna delle usuali restrizioni sulle dimensioni dell'espressione a virgola mobile (cioé all'interno dei limiti da -32768 a +32767), perché il risultato di INT è ancora un numero a virgola mobile, sebbene trasformato in un numero intero.

Notate i risultati delle seguenti espressioni

```
PRINT INT (3.1)
PRINTI INT (-3.1)
PRINTI INT (4.6+.5)
PRINT INT (-4.1+.5)
PRINT INT (1/2)
PRINT INT (-1/2)
```

che sono 3, e poi -4 (INT arrotonda sempre verso il basso); i successivi sono 5 e -4 (arrotondato al più vicino numero intero); e la coppia seguente fornisce le risposte 0 e -1 (notate che INT di -1 diviso per 2 è -1).

L'arrotondamento verso il basso è riferibile alla divisione degli interi

```
X%=-3 : Y%=2 : Z%=X%/Y% : PRINT Z%
```

per la quale la risposta è -2. La forma in complemento a due del numero -3 è 1111111111111101, e per dividere un numero binario per due muoviamo tutti i bit verso destra, inserendo 0 nella posizione più a sinistra (ad esempio 4 è 100 come numero binario e due è 010): perciò il numero diviene 0111111111111110.

Questo numero è ora positivo, e in complemento a due è 32766. Dividere il bit del segno reinserito nella posizione all'estrema sinistra è un sistema scarsamente efficace in cui la metà di un numero negativo è un numero positivo. Il numero binario formato dalla divisione per due pertanto è 1111111111111110, e come numero in complemento a due e uguale a -2.

La routine INT effettiva funziona prendendo il risultato da FPA#1, convertendolo in un intero di quattro byte, e poi convertendo l'intero a quattro byte in un numero a virgola mobile in FPA#1, mantenendo il vecchio esponente.

LEFT $ (S$,I%) Questa funzione prende S$ ed estrae da questa stringa i primi I% caratteri. Il valore I% può essere da 0 a 255, e se

è maggiore della lunghezza della stringa viene restituita l'intera stringa. Ad esempio,

```
PRINT LEFT$("1234",6),LEFT$("1234",2)
```

visualizza la stringa 1234 e 12. Se la lunghezza è zero allora viene emessa una stringa nulla.
La routine funziona prendendo i puntatori delle stringhe dallo stack, dove sono stati posti come parte di qualunque valutazione di stringa. La lunghezza della stringa è confrontata con il valore I% fornito dalla funzione, ed è preso il valore minore. La routine di selezione di stringa poi utilizza queste informazoni, che sono state riposte nello stack, per costruire la nuova stringa.

LEN (S$) Questa funzione trova la lunghezza della stringa S$, utilizzando allo scopo il byte che fornisce la lunghezza della stringa: il byte della lunghezza viene estratto dallo stack.

LET Non è necessario.

LIST Questo comando mostra il contenuto di parte o di tutto il programma, in una forma simile a quella in cui è stato inserito il programma.
Per listare un programma vi sono i seguenti metodi alternativi:

```
LIST
```

visualizzerà tutto il programma;

```
LIST200
```

visualizzerà il contenuto della linea 200;

```
LIST600-800
```

visualizzerà tutte le linee da 600 a 800 inclusa;

```
LIST-800
```

visualizzerà tutte le linee fino a 800 inclusa; e

```
LIST600-
```

visualizzerà tutte le linee da 600 fino alla fine del programma.
Se un programma è stato fermato con STOP è possibile listare e poi continuare l'esecuzione con CONT. Comunque se il coman-

do LIST è all'interno di un programma, ad esempio:

```
1 PRINT "$$$$$"
2 REM
3 REM
4 LIST
5 PRINT "#####"
```

eseguire questo programma lo lista su video ma il programma non continua alla linea 5 (la PRINT): usare il LIST all'interno di un programma ferma ogni ulteriore elaborazione. Se fate in modo di fermare (STOP) il listato mentre scorre (è più facile con i listati lunghi), il CONT trasferisce il controllo alla linea 5 ma il resto del listato è perso.

Quando viene usato CONT dopo che il programma è finito, il LIST è riattivato ed il programma è listato di nuovo. Comunque non è rieseguito l'intero programma perché la prima linea di $$$$$ non è visualizzata, solo il LIST. Il CONT dovrebbe far ripartire il programma da dove era «terminato», ma l'utilizzo di LIST confonde il risultato, ed il LIST viene ripetuto (ed interrompe il programma).
Se la linea 4 viene cancellata, e si lancia il programma, allora l'uso di CONT non fa nulla, a parte che il sistema vi dice che è pronto (READY.). Il cambiare la lina 4 in

```
4 LISY
```

determina un ?SYNTAX ERROR IN 4; e dopo tentare di battere il CONT, dà un ?CANT'T CONTINUE ERROR.
La ragione di queste peculiaretà di LIST è che il suo uso all'interno di un programma comporta l'uso estensivo dei puntatori e, se viene utilizzato il CONT, esso cerca di usare questi stessi puntatori: lo stato dei puntatori diviene non chiaro. Questo è un'altro bug, non serio, ma che non dovrebbe esserci in un BASIC ben fatto.

È possibile LISTare ad una periferica aprendo un file (OPEN) ed un device, e successivamente utilizzando un CMD (vedi sopra).

LOAD Questo comando è il contrario di SAVE, e prende un file immagazzinato di solito su disco o cassetta e carica in memoria una copia del file. Normalmente la copia sarà un programma BASIC, ma può anche essere un programma in codice macchina, o una copia del video in alta risoluzione, o qualsiasi altro insieme continuo di locazioni di memoria.
Alcuni dei più complessi SAVE utilizzano una delle routine KERNAL di sistema, anche nota come SAVE. Questa routine è attivata da SYS(65496) in congiunzione con altre routine KERNAL. Il KERNAL è un insieme di routine di sistema in linguaggio macchina che possono essere utilizzate dal programmatore, e normalmente il loro utilizzo richiede la conoscenza del linguaggio macchina.
Salvare (SAVE) il linguaggio macchina e predisporre le locazioni di memoria richiede perciò molta più conoscenza delle routine KERNAL di quanta sia possibile fornirne ora: maggiori dettagli sono forniti dalla GRP.
Con dei semplici programmi BASIC l'operazione di LOAD può essere considerata in due aree principali: la cassetta ed il disco. Dove di seguito vi è una stringa tra virgolette può essere usata una variabile stringa, ad esempio S$.
Nel caso del nastro, cioé della cassetta, tutto ciò che occorre è LOAD, ma vi sono delle varianti,

```
LOAD
LOAD"PROG"
LOAD""
```

il primo esempio caricherà il programma seguente in ordine sul nastro; il secondo caricherà il programma sul nastro che comincia con PROG (es. PROG1 o PROG2 o PROGRAMMA); ed il terzo esempio caricherà anch'esso il programma seguente sul nastro.
Le varianti per LOAD, quando si usa il disco, non sono in gran numero

```
LOAD"0:PROG",8
LOAD"1:*",8
LOAD"PROG*",8
LOAD"1:PROG??",8
LOAD"PRO?*",8
```

Notate il parametro obbligatorio ,8 per indicare che deve essere utilizzato il disco, e notate il parametro numero del drive, cioé il numero seguito dai due punti, che precede il numero del file. Il primo prende un programma chiamato PROG dal disk drive numero 0; il secondo prende il primo file dal disk drive 1. La * significa che qualsiasi (o nessun) carattere può essere sostituito in quella posizione, per formare il nome del file.

Il terzo LOAD è l'equivalente del semplice nome «PROG» per il nastro, PROG1 o PROG2 o PROGRAMMA soddisferanno la richiesta: sarà usato il primo file in ordine sul disco che inizia con PROG. Poiché non è specificato il numero del disk drive, lo si assume per default essere il drive 0.

Il quarto comando accetterà PROG1, PROG2, PROGZZ o PROBE, poiché ciascuno di essi ha cinque caratteri che iniziano con PRO: viene preso il primo. Il comando finale prenderà un file dal disk drive 0, e prenderà il primo file che comincia con PRO e che ha almeno quattro caratteri.

Il comando LOAD può essere utilizzato sia all'interno di un programma sia in modo diretto. Nel caso di un LOAD che sia in un programma non vi sono istruzioni sullo schemo, a meno che l'unità nastro non sia sul PLAY, nel qual caso vi è il messaggio PRESS PLAY ON TAPE. Per il disco non vi è alcun genere di messaggio.

Il programma non è soltanto caricato ma è anche lanciato: sotto questo aspetto è come il caricamento e l'esecuzione automatica data dai tasti SHIFT e RUN/STOP. Se il programma che carica un altro programma è più lungo del programma caricato, allora è possibile passare dei valori da un programma all'altro, eccetto le stringhe e le funzioni (per ovvie ragioni). Ad esempio,

```
10 REM
20 REM
30 REM
40 A = 10
50 B = 30
60 I% = 6
70 PRINT A,B,I%
80 REM
90 LOAD
```

visualizzerà i valori 10, 30 e 6, così velocemente che non potrete vederli prima che lo schermo diventi vuoto, per caricarne il programma seguente. Se il programma successivo sul nastro è

```
70 PRINT A,B,I%
```

allora vengono di nuovo mostrati i valori 10, 30 e 6.

LOG(X) Questa funzione calcola il logaritmo naturale (in base e), di qualsiasi espressione positiva, diversa da zero. Questa funzione è l'inverso di EXP (cfr.).
La funzione prima controlla i valori negativi o uguali a zero dell'espressione, e se il risultato è negativo o uguale a zero allora produce un ?ILLEGAL QUANTITY ERROR. Il risultato dell'espressione è lasciato in FPA#1 per cui,, se questo valore è troppo grande (es. 1E39), vi è un ?OVERFLOW ERROR.
La funzione è calcolata con l'uso di una breve serie, le cui costanti sono contenute nella ROM BASIC.

MID$(S$,I0%,I1%) Questa funzione seleziona una sottostringa dalla stringa S$: la posizione iniziale è data da 10% e la lunghezza della stringa è data da I1%.
Se la lunghezza non è indicata o è maggiore del numero di caratteri fino alla fine della stringa, allora vengono restituiti tutti i caratteri a destra: ciò la rende simile alla RIGHT$ (vedi oltre). LEFT$(S1$,13%) è equivalente a MID$(S1$,1,13%).
I valori consentiti per I0% vanno da 0 a 255, ed un valore fuori da tale intervallo dà un ?ILLEGAL QUANTITY ERROR. Tutti i valori per I1% all'interno dei limiti 0 e 255, sono ammissibili, ed al di fuori di tale intervallo producono lo stesso ?ILLEGAL QUANTITY ERROR.
Il valore sottinteso (valore di default) per il terzo parametro è 155, a meno che il parametro, cioè I1%, non sia posto ad un qualche valore: ovviamente nessuna stringa può essere più lunga di 255 caratteri. La stringa corretta è infine trovata per mezzo della manipolazione dei puntatori di stringa, che vengono immessi sullo stack: in realtà è utilizzata la routine LEFT$.
Se la stringa è S$="" allora MID$(S$,1,255) non è un errore, soltanto una stringa nulla, e se il parametro di mezzo è modificato in una qualsiasi valore allora viene restituita una stringa nulla:

questo è vero per qualunque stringa in cui il valore del parametro di mezzo è maggiore della lunghezza della stringa.
Ciò non ha senso e può essere mostrato dal programma

```
10 A$ = ""
20 PRINT MID$(A$,2,255)
30 A$ = "#"
40 PRINT MID$(A$,2,255)
```

che non produce alcun errore ma che è idiota; per mostrare quanto sia idiota provate

```
10 A$ = ""
20 PRINT MID$(A$,0,255)
30 A$ = "#"
40 PRINT MID$(A$,0,255)
```

che dà un ?ILLEGAL QUANTITY ERROR IN 20. Alla linea 20 A$ ha proprio lunghezza 0 ma, come con ASC, la codificazione delle routine di stringa sembra essere piuttosto sospetta.

NEW Questo comando sembra eliminare il programma dalla memoria, ma in realtà cambia pochi puntatori (Cfr. CLR) cosicché essi puntano o all'inizio del BASIC o alla fine della memoria BASIC. Il programma effettivo resta inalterato e può e può essere esaminato cambiando nuovamente alcuni puntatori, un compito noioso.
La modificazione di alcuni puntatori è la base di molti sistemi o tool kit che hanno un comando OLD.

NEXT X Segnala il limite di un loop. Il controllo è ripassato al corrispondenteFOR oppure all'istruzione seguente, se il contatore è ora maggiore del limite del loop.
Il comando ha un parametro opzionale, CHE NON DOVREBBE MAI ESSERE OPZIONALE. Poiché il BASIC puo' diventare piut-

tosto confuso, non dovrebbe essere ignorata alcuna assistenza per il debugging, come, ad esempio, dire al sistema quale è un loop e quale l'altro. Tralasciare dall'istruzione NEXT il contatore di loop è estremamente disordinato e molto sciocco.
Non usare il contatore di loop dopo il NEXT può accellerare il programma, ma se il programma non funziona allora il debugging sarà più difficoltoso. Il programma

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 REM
40 NEXT I
50 REM
60 NEXT J
```

è errato, come quest'altro programma

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 REM
40 NEXT
50 REM
60 NEXT
```

Il primo programma non sarà eseguito, mentre il secondo effettivamente lo sarà: modificate quelle REM in istruzioni attive ed il secondo programma è una ricetta per un disastro.

ON I% GOTO...ON I% GOSUB Il comando ON è un comando di diramazione. Il valore della variabile che segue la parola ON indica quale delle linee che seguono il GOTO o il GOSUB deve essere usata.

L'ON può essere utilizzato in due forme, sia in modo programma che in modo diretto:

```
10 ON X GOTO 200, 300, 400, 500
20 ON Y GOSUB 2000, 3000, 4000, 5000
```

ed il valore dell'espressione che segue l'ON deve essere nei limiti 0-255.

Nell'esempio precedente se X è 0 o maggiore di 4, allora il controllo passa alla linea successiva, poiché vi sono soltanto 4 linee tra le quali scegliere. Valori negativi dell'espressione danno ?ILLEGAL QUANTITY ERROR; se non c'è un tale numero di linea nel listato, allora vi è un ?UNDEF'D STATEMENT ERROR IN 50, quando viene effettuato un salto a tale linea.
La forma alternativa GO TO non può essere usata: ecco un'altra ragione per ignorarla.

OPEN F%,D%,A%,S$ Questo comando predispone i parametri di un file, il suo numero corrispondente di device e l'indirizzo secondario, nelle tavole di memoria, ciascuna di dieci elementi.

La prima tavola si estende dalla locazione 601 alla 610, e contiene il numero di file logico come assegnato dal comando OPEN, cioè F%. Alla tavola è data la label LAT (GRP, pag 316 nella versione inglese).

La seconda tabella ha dieci elementi, ciascuna delle quali corrispondente ad un numero di file logico, questa tavola (FAT, GRP pag 316 nella versione inglese) copre le locazioni da 621 a 630, e contiene i numeri di device, cioè D%, che corrispondono ai numeri di file.

L'ultima tavola (chiamata SAT in GRP pag 316) contiene l'indirizzo «secondario» per ciascun numero di file e numero di device.

A questo punto battete

```
PRINT PEEK(601),PEEK(611),PEEK(621)
```

alla quale la risposta è 0,0,0: il valore iniziale di ciascuna tavola è zero.
Ora inserite

```
OPEN 3
```

che risponde con PRESS PLAY ON TAPE, e poi scrive FOUND XCXC, dove XCXC è il nome di un qualunque programma; se poi diamo il LIST, scopriamo che il programma non è stato caricato.

Questo metodo legge il successivo header «intestazione» di programma senza dover effettivamente leggere l'intero programma. Per continuare a leggere gli header battete CLOSE 3 e poi nuovamente OPEN 3: aprire il file, OPEN 3, senza chiuderlo, CLOSE 3, determina un ?FILE OPEN ERROR.
Battere

```
PRINT PEEK(601),PEEK(611),PEEK(621)
```

produce l'output 3,1 e 96, ed inserire

```
PRINT PEEK(184),PEEK(185),PEEK(186)
```

risulta nell'output 3,96 e 1. Queste tre locazioni (LA, FA e SA in riferimento alla GRP pag 314-31 nella versione americana) sono i valori correnti dell'ultimo file aperto.
Battete

```
OPEN4,0,0
PRINT PEEK(602),PEEK(612),PEEK(622)
PRINT PEEK(184),PEEK(185),PEEK(186)
```

ed il risultato è 4,0,96 per la prima PRINT, e 4,96,0 per la seconda.

OPEN F% è equivalente a OPEN F%,1,0,"", dove 1 è il default del secondo parametro, e corrispondente alla cassetta 1; 0 è il default del terzo parametro, e corrispondente all'ordine alla cassetta di leggere un file; e "" è il default del quarto parametro, corrispondente a nessun titolo per il file.
Per ciascun device vi sono indirizzi secondari, che ne controllano l'uso.
Ad esempio,

```
OPEN 8,1,1, "+++++"
```

produce la risposta PRESS RECORD & PLAY ON TAPE, e qualcosa viene mandato al nastro. Catalogare il nastro per mezzo di

```
CLOSE 3:OPEN 3
```

produce un SEARCHING e successivamente un FOUND +++++. Il quarto parametro fornisce il titolo ad un file, se l'indirizzo secondario indica che la periferica è aperta alla scrittura.

Device	*Numero*	*Indirizzo secondario*
Tastiera	0	Nessuno
Cassetta # 1	1	0=Legge,1=Scrive sul file più il segnalatore di fine file,2=Come 1 più il fine nastro quando il file viene chiuso
Cassetta # 2	2	Come per cassetta # 1
Schermo	3	Nessuno
Stampante	4	Varia con la stampante
Modem	5	Nessuno
Disk drive	8	0=Lettura directory,1=Scrittura directory,15=canale per gli errori

Tutti gli altri numeri, fino al massimo di 15, non sono assegnati e gli indirizzi secondari prendono i valori da 0 a 15. Perciò, leggere o scrivere dati su un disco richiede l'uso degli indirizzi secondari da 2 a 14 incluso. Poiché le unità a dischi variano, è frequente fare in modo che il parametro stringa contenga i comandi disco, riguardo la lettura e la scrittura.
L'osservazione degli indirizzi secondari, per mezzo del PEEK, mostra che essi cominciano a 96 e vanno fino a 111. Per ottenere il valore effettivo dell'indirizzo secondario battete

```
PRINT PEEK(621) AND 15
```

che vi darà la risposta esatta (così come PEEK(621)-96).

PEEK(X) Questa funzione fornisce il valore del contenuto del byte alla locazione specificata tra parentesi. Il valore è restituito come numero decimale nell'intervalle da 0 a 155; la locazione deve essere nei limiti 0-65535, altrimenti si genera un ?ILLEGAL QUANTITY ERROR.
Quando viene utilizzata, la routine prende il valore immagazzinato nelle locazioni 20 e 21, e lo trasferisce allo stack. Queste locazioni, note insieme come LINNUM (GRP pag 311 versione inglese), sono usate per il deposito temporaneo dei numeri di linea ed il trasferimento del valore è una salvaguardia.
LINNUM viene poi caricato con il valore del parametro (viene

convertito in un intero a due byte ad FPA#1). Vengono poi chiamate delle routine per trasportare il valore da quella locazione di memoria a FPA#1. I contenuti originali di LINNUM sono scaricati dallo stack e restituiti.
La necessità del caricamento del contenuto di LINNUM nello stack è mostrato quando consideriamo il gemello di PEEK, cioè POKE.

POKE X,I% Il comando POKE sostituisce il contenuto della locazione specificata al primo parametro, con il valore specificato dal secondo parametro. Battete

```
POKE 56334,0
```

e la tastiera non risponderà più ai vostri comandi. Il solo modo in cui potete rimediare è battere STOP e RESTORE simultaneamente. L'inserimento del valore 0 alla locazione 56334 ferma il timer A, ed il timer A controlla la scansione della tastiera: lo stop del timer determina l'esclusione della tastiera.
Un altro modo per ottenere lo stesso risultato è battere

```
POKE56334,PEEK(56334)AND254
```

che lascia invariati tutti i bit eccetto il bit 0, e cambia il bit 0 al valore 0. È il bit 0 che controlla il timer, e quindi la tastiera è nuovamente disattivata. Notate che lo stesso numero di linea appare due volte nella stessa istruzione. Vedete sopra, riguardo gli operatori logici, per maggiori dettagli su questo uso dei PEEK e POKE.
Se il valore che deve essere inserito, cioè il secondo parametro, è fuori dai limiti 0-255, avremo un ?ILLEGAL QUANTITY ERROR, e determiniamo lo stesso errore se la locazione è fuori dell'intervallo da o a 65535.
La routine funziona immagazzinando il valore dell'indirizzo in LINNUM, cioè le locazioni di memoria 20 e 21, valutando l'espressione che segue la virgola, ed immagazzinando il risultato dell'elaborazione all'indirizzo dato da LINNUM.

POS(X) Questa è una funzione piuttosto inutile. Quando viene chiamata, restituisce il valore della posizione corrente del cursore sullo schermo. Ad esempio

```
FOR I=1 TO 255:PRINT"#";:NEXT I:PRINT POS(0)
```

visualizzerà un flusso di # e successivamente il numero 15. Il numero 15 è calcolato come 255=3*80+15, e se la linea è modificata in

```
FOR I=1 TO 239:PRINT"#";:NEXT I:PRINT POS("")
```

la risposta è 79: notate che il valore in parentesi è senza importanza. Lo strano modo in cui questa funzione produce i suoi risultati sembra una reminiscenza delle 80 colonne del PET.

PRINT Questo è probabilmente il più complesso tra tutti i comandi del BASIC C64, e serve per visualizzare informazioni sullo schermo.

È possibile stampare stringhe:

```
A$="123":FOR I=1 TO 100:PRINT A$:NEXT I
FOR I=1 TO 100:PRINT A$,:NEXT I
FOR I=1 TO 100:PRINT A$;NEXT I
```

dove la prima linea stampa la stringa 123 su righe separate, la seconda stampa quattro esempi di 123 per linea, e la terza stampa 12312312 alla fine di ciascuna linea, e continua da dove aveva lasciato, sulla linea seguente.

Si possono stampare numeri:

```
A=123:FOR I=1 TO 100:PRINT A:NEXT I
FOR I=1 TO 100:PRINT A,:NEXT I
FOR I=1 TO 100:PRINT A;:NEXT I
```

e per i primi due esempi il risultato sembra essere lo stesso che per le stringhe, ma non proprio.
Nel secondo esempio il numero 123 comincia dopo uno spazio, e

```
PRINT -123
```

mostra il perché. La prima colonna è utilizzata per il segno del numero, segno che non appare se il numero è positivo.
Nel terzo esempio i numeri 123 non appaiono l'uno accanto all'altro, come per la stringa 123, poiché vi sono due spazi in mezzo; una stringa non ha spazi davanti o dietro: battendo

```
PRINT A;A$
```

vi è uno spazio tra il numero 123 e la stringa 123. Ciascun numero è preceduto e seguito da uno spazio.
Le linee seguenti

```
B=-10/9:FOR I=1 TO 100:PRINT B, :NEXT I
FOR I=1 TO 100:PRINT B;:NEXT I
```

mostrano che, nel primo caso, vi sono solo due numeri per linea (essendo il numero -1.11111111), perché non vi è abbastanza spazio per quattro numeri di questa lunghezza su una linea di schermo in una sola volta; ciascun numero, incluso il segno meno, è di 11 cifre, e perciò è fisicamente impossibile avere quattro numeri di undici cifre su uno schermo di 40 colonne.
Nel secondo caso i numeri -1.11111111 sono stampati ininterrottamente, con uno spazio tra loro e proseguono sulla linea seguente.
Considerate anche le seguenti istruzioni e con i relativi risultati

Istruzioni		*Risultato*
PRINT A A	*produce*	0
PRINT A$ A$		123123
PRINT 123. 9		123.9
PRINT 123. 9		123.9 0
PRINT 123. .9		123 .9
PRINT 123(9)		123 9

A A è supposto essere AA, che per default ha il valore 0; A$ A$ è supposto essere l'equivalente di A$;A$ e perciò viene visualizzato 123123 (questo è anche vero per i suffissi di array () e di intero %); 123. 9 è considerato essere 123.9; 123. 9. è supposto essere 123.9 e . (e . è sottinteso essere 0); 123. .9 è considerato essere 123 e .9; e 123(9) è considerato essere 123 9 perché anche (è un separatore.
Se dei simboli grafici seguono le virgolette, questi non agiscono immediatamente, ma solo quando la stringa viene stampata: notate che se le virgolette sono il carattere finale di una linea di PRINT, queste possono essere omesse.
Il funzionamento delle routine coinvolte dall'attivazione dell'istruzione PRINT è piuttosto complessa.

PRINT#F% questa è utilizzata per mandare informazioni a periferiche quali la stampante, il disco o il nastro (Cfr. OPEN per i numeri di device).
Il file deve essere aperto in modo corretto, prima che l'informazione sia «stampata» con la PRINT#. Il primo parametro, il numero di file (compreso tra 1 e 255) è sempre seguito da una virgola: dopo la virgola vi è la lista di termini che deve essere mandata in output.
Il formato della lista seguente la PRINT# è lo stesso della PRINT. Poiché certe periferiche hanno le proprie caratteristiche particolari, non è possibile dare un formato generale, ad esempio bisogna esaminare la documentazione per la stampante.
Per le periferiche non Commodore, ad esempio le stampanti parallele, l'indirizzo secondario può essere piuttosto importante, poiché spesso viene usato per controllare il set di caratteri (cioè SET 1 o SET 2, NUM appendice E).
PRINT# attiva la periferica, stampa e poi disattiva la periferica, PRINT#21,; non stampa niente sulla periferica, ma la disattiva: soltanto se questo è seguito dal CLOSE21, i dettagli del file vengono cancellati dalle tavole dei file. Utilizzare il PRINT#21, pertanto, significa che il file 21 non sta «ascoltando», ma può essere usato ancora.
CMD attiva la periferica, stampa e lascia la periferica attiva; CMD 21 anch'esso non stampa nulla ma lascia la periferica attiva. Far seguire questo comando a CLOSE 21 cancella tutti i parametri ma il file è ancora attivo: PRINT# è il solo modo per render «sorda» ma presente la periferica.
Con le stampanti è meglio utilizzare il PRINT#, piuttosto che il CMD ed il PRINT; infatti, nella routine PRINT# sono fatte delle chiamate sia alla CMD che alla PRINT.
READ Questo comando legge informazioni dall'istruzione DATA (cfr.), e si adegua nei principi generali all'istruzione INPUT (cfr.): alcuni dei cavilli della INPUT non sono ripetuti. Ad esempio,

```
1 FOR I=1 TO 5
2 READ Y
3 PRINT Y
4 NEXT I
5 DATA ,,,,
```

produce una serie di zeri, mentre cambiare la linea 5 in

```
5 DATA 1/2,,,,
```

dà un ?SYNTAX ERROR IN 5, mentre con l'INPUT vi sarebbe stato chiesto di reinserire.

REM Questa istruzione permette di fare dei commenti nel programma: tutto ciò che segue una REM viene trattato come parte del commento: come per l'IF, il salto avviene al numero di linea seguente e non alla successiva istruzione.
Provate una linea che inizi con 10 REM seguito da SHIFT +, poi listate.

RESTORE Pone nuovamente all'inizio il puntatore che indica l'elemento in uso delle DATA (locazioni 63 e 64 per il numero di linea, e locazioni 65 e 66 per l'indirizzo dell'elemento DATA corrente, GRP =ag 312): la stessa informazione può essere letta e riletta.
RESTORE viene chiamata automaticamente da ogni altro comando che modifica i puntatori, ad esempio NEW, RUN o CLR.

RETURN Indica la fine di una subroutine, vedere GOSUB per maggiori dettagli.

RIGHT$(S$,I%) Produce gli I% caratteri più a destra della stringa S$ (confrontate con lEFT$ e MID$). Il valore I% può variare da 0 a 255, e ciò che viene restituito è inferiore di I% o della lunghezza della stringa S$.
Questa funzione è imparentata con LEFT$ e MID$, e battendo

```
A$=«1234567890»:L=LEN(A$)
FOR I=1 TO L:PRINT LEFT$(A$,I)+RIGHT$(A$,L=I):NEXT
I
```

dà una serie di copie di A$. Abbiamo preso la parte sinistra e la parte rimanente (la parte destra), e le abbiamo assommate assieme, producendo la stringa originale.

La routine funziona mediante la valutazione della stringa, prendendo i parametri dallo stack, modificando il parametro minore e poi utilizzando la routine LEFT$.

RND(X) Questa funzione genera un valore «casuale», cioè imprevedibile, tra 0 e 1. Nessuna sequenza di numeri generata da una formula fissa è tuttavia veramente casuale, perché la sequenza si ripete dopo un certo periodo.
Vi sono tre classi di valori per il parametro. Valori del parametro negativi (qualunque essi siano) producono sempre lo stesso numero casuale. Valori del parametro positivi seguono una determinata sequenza, una volta scelto il valore iniziale.
Se il valore del parametro è zero allora il numero casuale prende il valore da un orologio in tempo reale. Questo fornisce un inizio casuale a qualsiasi sequenza (l'inizio è ciamato «seme»), ma essendo basato su un orologio regolare, i valori tendono a ripetersi. Per generare una sequenza imprevedibile di valori è meglio iniziare con RND(0) e continuare con RND(1) o qualsiasi altro parametro positivo.

RUN Inizia l'esecuzione di un programma dall'inizio, o da un numero di linea specificato; i puntatori sono riinizializzati, cosicché tutte le informazioni sono effettivamente perse.
Il comando può essere usato in modo diretto o in un programma, ad esempio.

```
RUN 5000
```

```
200 INPUT "VUOI EFFETTUARE UN'ALTRA ESECUZIONE";A$
210 IF A$="SI" THEN RUN
```

Se vi è un numero di linea che segue il comando RUN viene valutato come i GOSUB e GOTO (ecc.), per cui RUN è lo stesso che RUN 0.

SAVE S$,F%,D%,A% Questo comando salva il contenuto di una porzione di memoria, solitamente un programma BASIC, su qualche dispositivo. SAVE ha gli stessi parametri di LOAD, e ne condivide molte caratteristiche (cfr. LOAD per maggiori dettagli sul significato dei parametri).

SGN(X) Questa funzione fornisce il segno dell'espressione aritmetica tra parentesi.
Il segno è calcolato prima di tutto esaminando l'esponente dell'espressione in FPA#1; se l'esponente è zero allora il segno è zero; se l'esponente è diverso da zero allora viene verificato il segno del byte e fornito il segno.
Uno dei test di bug può essere modificato così:

```
10 T = -1
20 T = T/2:PRINT SGN(T):GOTO 20
```

e produce una lunga serie di -1, un 1 finale e poi degli zero.
SGN è l'abbreviazione di SIGNUM.

SIN(X) Questa funzione dà il seno dell'angolo (in radianti) corrispondente all'espressione in parentesi.
La SIN è la funzione calcolata per determinare la funzione COSeno (vedi sopra). Oltre un certo angolo, che potreste cercare di trovare, la SIN dà sempre risposta zero.

SPC(I%) Questa funzione stampa un numero di spazi sullo schermo o sulla stampante; questa funzione è usata all'interno della lista di termini della PRINT.
Il parametro può assumere valore tra 0 e 255, ed è utilizzato per formattare l'output (cfr. TAB, più oltre).

SQR(X) Trova la radice quadrata dell'espressione aritmetica in parentesi.
Il valore della radice quadrata è determinato come caso speciale dell'operatore di potenza ↑ (vedi sopra): la radice quadrata di X è valutata come X↑.5. Questo valore non è sempre totalmente accurato per la precisione della macchina, e la migliore stima è data da

```
10 DEF FNS(X) = (SQR(X)+X/SQR(X))/2
```

ed un test è dato da

```
20 FOR I=1 TO 10
30 PRINT SQR(I) - FNS(I)
40 NEXT I
```

Per questi pochi valori vi sono due errori di 9.31322575E-10. Per avere maggiore precisione quindi utilizzate FNS.

ST È una variabile riservata (una delle tre) e fornisce una registrazione dello stato del sistema dopo ogni operazione di input-output: il nome intero di ST è STATUS.
Molti degli errori che sono più difficili da determinare non sono trattati da ST e quando uno ha raggiunto il livello in cui ci si preoccupa di tali problemi, vi sono metodi migliori.
ST è immagazzinata nella RAM come un byte in una speciale locazione, non assieme alle altre variabili (locazione 144).

STEP Parte della struttura di controllo del ciclo FOR...NEXT, cfr. FOR.

STOP Ferma un programma ed indica a quale linea il programma è stato arrestato. Il programma può esser rilanciato usando CONT.
Il funzionamento della routine associata con questo comando è praticamente identico a quello di END, con l'informazione supplementare del numero di linea.

STR$(X) Questa funzione converte l'espressione numerica all'interno della parentesi in un valore stringa. Uno dei principali utilizzi di questa funzione è nella formattazione.
La routine è abbastanza buona, ad esempio

```
PRINT STR$(1234500000000)
PRINT STR$(.000000000000001)
```

produce 1,2345E+12 e 1E-15. Notate che la routine inserisce uno spazio davanti.

SYS X Questo comando trasferisce il controllo ad una routine in codice macchina che inizia all'indirizzo che segue il comando. La routine in codice macchina viene eseguita finché non viene incontrata una istruzione di ritorno dalla subroutine (normalmente una istruzione RTS).
L'intervallo dei valori per l'espressione aritmetica deve essere entro i limiti 0-65535, e le parentesi non sono necessarie.
Il battere la

```
SYS 40960
```

«ammazza» completamente il sistema, e l'unico modo per porvi rimedio è di spegnere la macchina. La locazione 40960 è l'inizio del sistema BASIC (contenuto in ROM), e SYS 40960 salta all'inizio, (dove non dovrebbe iniziare).

TAB(I%) Questo è un comando di formattazione di stampa (confrontatelo con SPC) ma non può essere usato con una stampante.
Diversamente da SPC, che produce un certo numero di spazi a partire dalla posizione attuale, TAB muove il cursore ad una posizione assoluta. Sfortunatamente se la posizione assoluta precede quella attuale, il carattere è stampato nella prima posizione disponibile.

Ciò è illustrato da

```
PRINT TAB(3);"$#"; TAB(2);"!"
```

che lascia tre spazi e poi scrive $#!.

TAN(X) Fornisce la tangente dell'angolo in parentesi, dove l'angolo è espresso in radianti.
Il valore è calcolato come SIN(X)/COS(X), il che in effetti significa che vi sono due chiamate della routine SIN (cfr. COS).

TI e **TI$** Anche conosciute come TIME e TIME $, sono due variabili riservate che forniscono una lettura del clock interno.

TI fornisce il valore in Jiffy, cioè 1/50 di secondo (1/60 negli U.S.A.): non è possibile assegnare un valore a TI, perché si produrrebbe un ?SYNTAX ERROR.
TI$ misura il tempo in maniera differente, ed anche TI$ può avere un valore assegnato: il valore assegnato deve essere una stringa di 6 caratteri nella forma HHMMSS, dove ogni valore di ora maggiore di 23 (es. 590000) è posto uguale a 000000.
Il «Jiffy clock» è immagazzinato alle locazioni da 160 a 162 (chiamate TIME in GRP pag 314 versione inglese), e tutte e tre le locazioni sono poste uguali a zero quando la macchina è accesa. Vi sono varie routine per calcolare TI$ dal valore di TI, e varie routine di input-output faranno perdere tempo al «Jiffy clock» (tecnicamente il «jiffy clock» conta le interruzioni, che sono talvolta disattivate dalle operazioni di input-output).

USR(X) Questa è una funzione aritmetica che usa delle speciali routine in linguaggio macchina: non devono essere usate che da esperti.
Quando viene chiamata diciamo da

```
Y=USR(4)
```

vi è un salto immediato alle locazioni 785 e 786, e queste locazioni puntano all'inizio delle routine in codice macchina. Il valore in parentesi è utilizzato dalla routine verso la quale viene effettuato il salto: il parametro è posto in FPA#1 e la routine preleva poi quel valore.

VAL(S$) Abbiamo già incontrato VAL in molte situazioni (vedere GOTO, GOSUB, INPUT, tra gli altri): VAL prende un'espressione stringa e cerca di convertirla in un numero.
La stringa nulla è posta a 0, e la conversione continua fino al primo carattere non numerico (eccettuato E). VAL usa la routine CHRGET alle locazioni da 115 a 138 (GRP pag 113). La maggior parte delle altre routine per interpretare i caratteri usa CHRGET e questa è la ragione per cui vi sono molti problemi di famiglia.

VERIFY S$,F%,D% Questo comando verifica l'accuratezza della registrazione di una porzione di memoria immagazzinata.

In molti aspetti VERIFY si comporta come un comando di input-output di file (cfr.) ed in effetti utilizza routine molto simili. La differenza principale è che il programma è letto ma i byte in ingresso vengono confrontati con i contenuti della memoria: se non vi è accordo, viene alterato ST e, cosa molto più utile, è visualizzato il messaggio VERIFY ERROR.

WAIT X,I0%,I1% Questo è un comando per lo più inutile: il C64 attende finché il risultato di NOT(PEEK(X)ORNOT(I1%)ANDI0% non è uguale a zero. Se I1% non è presente allora la verifica è effettuata tra PEEK(X) e I0%.

Altri libri della collana Jackson:

Computer

Sono manuali d'uso di immediata e semplice consultazione, da tenere a portata di mano di fianco al computer.

• Sinclair Spectrum • VIC 20 • Commodore 64 • PC IBM
• Apple IIc • Sharp MZ80A

Informatica

Forniscono le conoscenze fondamentali per una cultura informatica di base.

• La programmazione

Software

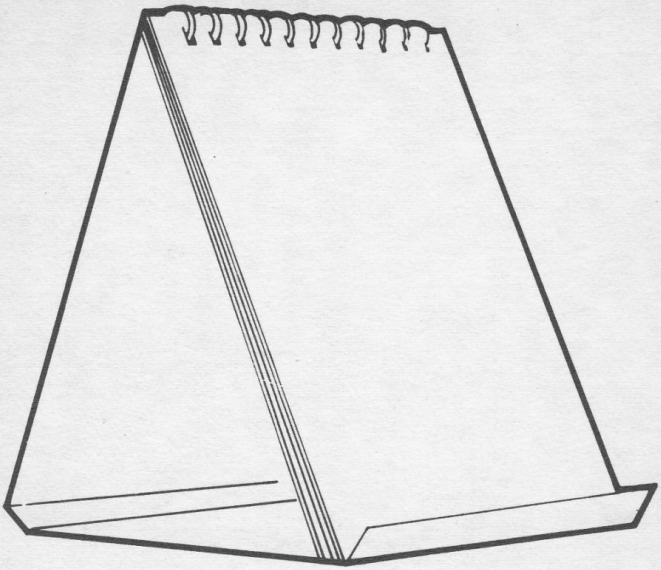
Presentano i pacchetti software e i sistemi operativi più diffusi sul mercato e suggeriscono idee e proposte di programmi per diverse applicazioni.

• WordStar • UNIX • LOGO • MS-DOS • Programmi di statistica • CP/M • PC-DOS

Linguaggi

Sono comode tabelle di riferimento delle istruzioni e i comandi dei linguaggi più famosi.

• COBOL • FORTRAN 77 • PASCAL • BASIC
• Assembler Z80 • Assembler 6502



L'illustrazione mostra come
sistemare il tascabile
per una più agevole consultazione.

I tascabili Jackson sono uno strumento prezioso per chi lavora con il computer. In poche pagine riassumono con chiarezza quanto è necessario sapere su diversi argomenti di informatica, andando incontro alle più diverse esigenze.

COD 002H ISBN 88-7056-233-6 **L. 8.500**