CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 1 ◇ 23 Settembre 2004

I linguaggi di programmazione

Un linguaggio di programmazione è un linguaggio che permette la realizzazione di algoritmi su un calcolatore. Il calcolatore esegue istruzioni numeriche (diverse a seconda del processore) che insieme formano il linguaggio macchina del calcolatore. Tipicamente queste istruzioni comprendono operazioni in memoria, istruzioni di flusso o di controllo (test, salti, subroutines, terminazione), definizioni di costanti, accesso a funzioni del sistema operativo, funzioni di input e output.

I numeri che compongono il linguaggio macchina possono essere inseriti direttamente in memoria. Un programma scritto in un altro linguaggio di programmazione deve invece essere convertito in linguaggio macchina; ciò può avvenire prima dell'esecuzione del programma tramite un compilatore che trasforma il programma scritto nel linguaggio sorgente in codice macchina oppure tramite un interprete che effettua una tale trasformazione durante l'esecuzione del programma e solo in parte per quei pezzi del programma che devono essere in quel momento eseguiti. Siccome il codice preparato da un compilatore è già pronto mentre le operazioni dell'interprete devono essere ripetute durante ogni esecuzione, è chiaro che i linguaggi compilati (assembler, C) sono più veloci di quelli interpretati (Perl, Lisp, PostScript, Apl, Basic).

La velocità del hardware moderno rende meno importanti queste differenze. Spesso, come nel Perl, il linguaggio utilizza moduli (scritti ad esempio in C) già compilati e quindi la traduzione riguarda solo una parte del programma, oppure è possibile, come in Java (e di nascosto anche in Perl), una compilazione in due tempi, che traduce il programma sorgente prima in codice indipendente dalla macchina (linguaggio per una macchina virtuale o bytecode) che su un calcolatore specifico viene poi eseguito da un interprete.

Nonostante che algoritmi e programmi dovrebbero essere separati il più possibile, la sintassi e soprattutto i tipi di dati previsti o definibili di uno specifico linguaggio inducono a paradigmi di programmazione diversi. Molto spesso è possibile, una volta appresi i meccanismi, imitare le costruzioni di un linguaggio anche in altri; per questa ragione è molto utile conoscere linguaggi diversi per avere un bagaglio di tecniche applicabili in molti casi indipendentemente dal linguaggio utilizzato.

Programmare in C

Un programma in C in genere viene scritto in più files, che conviene raccogliere nella stessa directory. Avrà anche bisogno di files che fanno parte dell'ambiente di programmazione o di una nostra raccolta di funzioni esterna. Tutto insieme si chiama un *progetto*. I files del progetto devono essere compilati e collegati (*linked*) per ottenere un file eseguibile (detto spesso *applicazione*). Il programma in C costituisce il codice sorgente (*source code*) di cui la parte principale è contenuta in files che portano l'estensione *.c*, mentre un'altra parte, che comprende soprattutto le dichiarazioni generali, è contenuta in files che portano l'estensione *.h* (da *header*, intestazione).

Cos'è una *dichiarazione*? Il C può essere considerato come un linguaggio macchina universale, le cui operazioni hanno effetti diretti in memoria, anche se la locazione effettiva degli indirizzi non è nota al programmatore. Il compilatore deve quindi sapere quanto spazio deve riservare alle variabili e quanti e di quale tipo sono gli argomenti e i risultati delle funzioni. Ogni file sorgente (con estensione *.c*) viene compilato separatamente in un file oggetto (con estensione *.o*), cioè un file in linguaggio macchina. Se il file utilizza variabili e funzioni definite in altri files sorgente, bisogna che il compilatore possa conoscere almeno le dichiarazioni di queste variabili e funzioni, dichiarazioni che saranno contenute nei files *.h* che vengono *inclusi* mediante `# include` nel file *.c*.

Dopo aver ottenuto i files oggetto (*.o*) corrispondenti ai singoli files sorgenti (*.c*), essi devono essere composti in un unico file eseguibile (a cui, sotto Unix, automaticamente dal compilatore viene assegnato il diritto di esecuzione) dal *linker*.

I comandi di compilazione e di composizione possono essere battuti dalla shell, ma ciò deve avvenire ogni volta che il codice sorgente è stato modificato e quindi consuma molto tempo e sarebbe difficile evitare errori. In compilatori commerciali, soprattutto su altri sistemi, queste operazioni vengono spesso eseguite attraverso un'interfaccia grafica; sotto Unix normalmente questi comandi vengono raccolti in un cosiddetto *makefile*, molto simile a uno script di shell, che viene poi eseguito mediante il comando *make*.

In questo numero

Procedure e funzioni

Una *procedura* in un programma è una parte del programma che può essere chiamata esplicitamente per eseguire determinate operazioni. In un linguaggio macchina una procedura o *subroutine* è un pezzo di codice, di cui è noto l'indirizzo iniziale e da cui si torna indietro quando nell'esecuzione del codice vengono incontrate istruzioni di uscita. Una procedura può dipendere da parametri (detti anche argomenti); una procedura che restituisce (in qualche senso intuitivo) un risultato si chiama *funzione* e corrisponde nell'utilizzo alle funzioni o applicazioni della matematica.

Una procedura o funzione in un programma si chiama *ricorsiva* se chiama se stessa. Il concetto di ricorsività è molto importante in matematica, informatica, logica e forse in molti comuni ragionamenti umani.

Sia $S(n) := 0 + 1 + \cdots + n$ la somma dei numeri naturali da 0 ad n. Allora S possiede la rappresentazione ricorsiva

$$S(n) = \begin{cases} 0 & \text{per } n = 0 \\ S(n-1) + n & \text{per } n > 0 \end{cases}$$

In C possiamo programmare

```
int S (int n)
{if (n==0) return 0; return S(n-1)+n;}
```

Sappiamo naturalmente che $S(n) = \dfrac{n(n+1)}{2}$. Nello stesso modo possiamo però rappresentare la funzione fattoriale $n!$, per la quale non esiste una formula semplice:

$$n! = \begin{cases} 0 & \text{per } n = 0 \\ (n-1)!\,n & \text{per } n > 0 \end{cases}$$

Nel programma in C la chiamiamo `Fatt`:

```
double Fatt (int n)
{if (n==0) return 1; return Fatt(n-1)*n;}
```

Siccome il fattoriale diventa presto molto grande, stavolta come tipo di risultato della funzione abbiamo usato `double`, che è il tipo del C che corrisponde ai numeri reali della matematica.

Il programma minimo

```
// alfa.c
# include <stdio.h>

int main ()
{puts("Ciao!");
exit(0);}
```

Analizziamo questo programma.

- `//...` è un commento, qui il nome del file (utile per la stampa).
- `# include ...` fa in modo che il preprocessore (che prepara il file per il compilatore) inserisca in questo punto il file indicato.

  Ci sono due modi per indicare il file:

  Con `# include "nomefile"` viene incluso il file il cui nome (assoluto o locale, secondo le convenzioni Unix) è *nomefile*.

  Con `# include <nomefile>` viene invece incluso il file *nomefile* che si trova in una delle cartelle dove il compilatore cerca i files richiesti con questa variante di `# include`, soprattutto in */usr/include*.

  Nel nostro caso viene incluso il file */usr/include/stdio.h* che contiene le dichiarazioni delle funzioni, costanti e variabili di I/O (input e output) del C, ad esempio di `puts` e `printf`.
- Ogni programma in C deve contenere esattamente una funzione principale che si chiama `main` e appare nella forma

  ```
  int main ()
  {...}
  ```

  come qui, oppure nella forma

  ```
  int main (int n, char **Arg)
  {...}
  ```

  quando si vuole usare il programma con argomenti della shell.
- `puts("il testo");` stampa la stringa *il testo* sullo schermo (`puts` è un'abbreviazione di *put string*), con un carattere di invio (`'\n'`) alla fine. Si potrebbe anche usare `printf("il testo\n");`. `printf` permette un output formattato molto più generale, come vedremo.
- Sotto Unix i processi attivi comunicano continuamente; `exit(0);` comunica agli altri processi che il programma è terminato senza errore.

Sotto Unix si distingue tra *programmi* e *processi*. Un processo è un ambiente capace di eseguire un programma; un programma è un codice eseguibile. Tipicamente sono attivi almeno alcune decine, ma anche centinaia di processi allo stesso tempo; possono essere elencati battendo `ps ax` dalla shell.

main con argomenti

Diamo un esempio di un programma principale con argomenti.

```
int main (int n, char **A)
{printf("Questo programma "
  "si chiama %s,\nil numero "
  "degli argomenti e' %d,\n"
  "gli argomenti sono %s e %s.\n",
  A[0],n-1,A[1],A[2]);
exit(0);}
```

Se il programma compilato si chiama *alfa*, con `alfa Ferrara Bologna` dalla shell otteniamo l'output

Questo programma si chiama alfa,
il numero degli argomenti è 2,
gli argomenti sono Ferrara e Bologna.

Infatti `A[1],A[2],...` sono gli argomenti, `A[0]` il nome del programma stesso.

Stringhe (parole) nel C, quando sono date in modo esplicito, vengono racchiuse tra virgolette; più stringhe tra virgolette possono essere concatenate semplicemente ponendo una vicina all'altra.

Il preprocessore

Quando un file sorgente viene compilato, prima del compilatore vero e proprio entra in azione il *preprocessore*. Questo non crea un codice in linguaggio macchina, ma prepara una versione modificata del codice sorgente secondo direttive (*preprocessor commands*) date dal programmatore che successivamente verrà elaborata dal compilatore (in linea di principio almeno, perché in alcune implementazioni le due operazioni sono combinate in un unico passaggio).

Le direttive del preprocessore per noi più importanti sono `# include` e `# define` (lo spazio dopo `#` può anche mancare).

Il significato e l'uso di `# include` sono già stati spiegati. Le direttive `# define` vengono utilizzate per definire abbreviazioni. Queste abbreviazioni possono contenere parametri variabili e simulare funzioni; in tal caso si parla di *macroistruzioni* o semplicemente di *macro*. Le più semplici abbreviazioni sono del tipo

```
# define base 40
# define nome "Giovanni Rossi"
# define CMPi M_PI // pi
# define CMPid180 0.01745329 // pi/180
# define CM180dPi 57.29577951
```

Il nome dell'abbreviazione deve essere un identificatore (deve cioè avere la forma dei nomi in C). Dopo il nome segue uno spazio (oppure una parentesi che inizia l'elenco dei parametri), e il resto della riga è l'espressione che verrà sostituita al nome dell'abbreviazione. Si noti che non appare il segno di uguaglianza. Come in altre parti del testo sorgente, una riga che termina con `\` (a cui non deve seguire un carattere di spazio vuoto) viene, prima ancora dell'intervento del preprocessore, unita alla riga seguente.

Si vede anche che la riga di un `# define` può contenere un commento che non viene incluso nella definizione della costante.

Comandi di compilazione

Questa parte è valida solo per Linux e Unix.

È buona abitudine creare per ogni programma una cartella (*directory*) apposita. In essa quindi si trova il nostro file *alfa.c* che è un normale file di testo. Lo trasformiamo in un file oggetto alfa.o con il *comando di compilazione*

```
gcc -c alfa.c
```

Controlliamo con `ls` l'esistenza del file oggetto *alfa.o*; questo file non è ancora eseguibile; creiamo un file eseguibile *alfa* (il programma) con il *comando di composizione*

```
gcc -o alfa alfa.o -lc -lm
```

L'indicazione `-lc -lm` significa che nella composizione del programma vengono utilizzate la libreria base del C (ad essa si riferisce `-lc`) e la libreria matematica (con `-lm`). Queste librerie sono fisicamente contenute nei files *libc.so* e *libm.so* della cartella */usr/lib*; le librerie *.so* sono *dinamiche*, non vengono cioè aggiunte al programma *alfa* stesso, ma caricate in memoria separatamente quando non già presenti. Esistono anche le librerie statiche *libc.a* e *libm.a* nella stessa cartella che possono essere usate per creare un programma indipendente dalle librerie dinamiche (ma molto più grande). In verità *libm.so* è un collegamento (*link*) simbolico a una versione precisa della libreria, nel nostro caso ad esempio a */lib/libm.so.6*, che a sua volta è un link simbolico a */lib/libm-2.2.93.so*.

Le librerie si possono trovare anche in altre cartelle, ad es. la *GNU Scientific Library (GSL)* e la sottolibreria riferita alla versione C del *BLAS* (*Basic Linear Algebra Subprograms*) vengono tipicamente installate nelle cartelle */usr/local/lib* con i nomi *libgsl.so* e *libgslcblas.so* risp. *libgsl.a* e *libgslcblas.a*. Se le vogliamo usare nel comando di composizione dobbiamo aggiungere `-lgsl -lgslcblas`; in altre parole, il prefisso *lib* diventa *l* e l'estensione *.so* risp. *.a* viene omessa.

Librerie statiche

Il file sorgente *alfa.c* occupa solo 104 B, e il file oggetto *alfa.o* è grande 828 B. Il programma eseguibile *alfa* occupa 11503 B.

Se avessimo invece utilizzato le librerie statiche con l'opzione `-static` nel comando di composizione,

```
gcc -o alfa alfa.o -lc -lc -static
```

(con lo stesso file *alfa.o* creato in precedenza), avremmo ottenuto un programma *alfa* di 447044 B, cioè molto più grande. Esso contiene infatti adesso al suo interno tutto il codice delle librerie del C e delle funzioni matematiche.

Useremo sempre le librerie dinamiche, omettendo cioè `static`. L'unico svantaggio è che, quando si trasporta un programma su un nuovo computer, bisogna trasferire anche le librerie dinamiche, se non ci sono già, oppure aggiornarle con, talvolta, qualche problema di compatibilità delle versioni.

I commenti

Normalmente i commenti vengono eliminati prima ancora che entri in attività il preprocessore. Il commento classico del C consiste di una parte del file sorgente compresa tra `/*` e `*/` (non contenuti in una stringa), che può estendersi su più righe. Esempio:

```
int n; /* Questo e' un commento
su due righe */ n=7;
```

I nuovi compilatori C accettano anche i commenti nello stile C++, che spesso sono più comodi e più facilmente distinguibili. In questo formato se una riga contiene (sempre al di fuori di una stringa) `//`, allora tutto il resto della riga è considerato un commento, compresa la successione `//` stessa.

Il makefile

Nel comando di composizione `-o alfa` significa che viene creato un file di nome *alfa*. Se avessimo anche altri files sorgente *beta.c* e *gamma.c* da compilare, dovremmo battere i seguenti comandi:

```
gcc -c alfa.c
gcc -c beta.c
gcc -c gammac.
gcc -o alfa alfa.o beta.o gamma.o -lc -lm
```

I primi tre sono comandi di compilazione e creano i files *alfa.o*, *beta.o* e *gamma.o*, l'ultimo è il comando di composizione.

Se nei comandi includiamo alcune opzioni, ad esempio `-fwritable-strings` o se vogliamo indicare in quali cartelle cercare i files di intestazione o le librerie, i comandi diventano ancora più complicati. Inoltre files *.o* già pronti che non necessitano di modifiche non dovrebbero esere ricompilati. Per ridurre e organizzare questo lavoro, sotto Unix si scrive un *makefile* che deve portare il nome di *Makefile* (o *makefile*):

```
# Makefile

lib = -lc -lm
obj = Oggetti

VPATH = ${obj}

make: alfa.o beta.o gamma.o
TAB gcc -o alfa ${obj}/*.o ${lib}

%.o: %.c
TAB gcc -o ${obj}/$*.o \
       -fwritable-strings -c $*.c

.PHONY: clean
clean:
TAB rm -f *.o ${obj}/*.o
```

Attenzione: `TAB` è qui un carattere tabulatore che non può essere sostituito da spazi. Possiamo invece continuare sulla stessa riga con `-fwritable-strings` togliendo il carattere `\`, che qui abbiamo usato solo per ragioni tipografiche per far stare il listato nei confini della colonna.

Affinché questo makefile funzioni, bisogna creare una cartella *Oggetti*, dove verranno depositati i files *.o*. Adesso battiamo `make` dalla shell.

Per eliminare i files *.o* non più necessari si usa `make clean`. Una volta composto, il programma può essere eseguito battendo `alfa` dalla shell.

L'editor

I files *.c* e *.h* sono normali files di testo. Per scriverli si può usare un editor che crea files in formato testo puro, sotto Linux ad esempio *emacs* o *kate* (*KDE advanced text editor, kate.kde.org*). Simile a *kate* è *kedit*.

alfa.h

Invece di `# include <stdio.h>` nella sorgente *alfa.c* stessa è preferibile, soprattuto quando si vogliono usare più files *.c*, creare un file *alfa.h* nella cartella del nostro progetto e inserire in esso le istruzioni di inclusioni generali nonché le dichiarazioni delle funzioni e variabili da condividere. In *alfa.c* (e negli altri files *.c*) scriveremo allora sempre `# include "alfa.h"`. Abbiamo quindi (nella versione più semplice, che subito allargheremo)

```
// alfa.h
# include <stdio.h>
```

e

```
// alfa.c
# include "alfa.h"

int main ()
{puts("Ciao!");
exit(0);}
```

Le intestazioni standard

Il C possiede molti files di intestazione, di cui inseriamo adesso i seguenti in *alfa.h*:

```
# include <ctype.h>
# include <dirent.h>
# include <errno.h>
# include <fcntl.h>
# include <float.h>
# include <grp.h>
# include <limits.h>
# include <locale.h>
# include <math.h>
# include <pwd.h>
# include <signal.h>
# include <stdarg.h>
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <sys/stat.h>
# include <sys/times.h>
# include <sys/types.h>
# include <time.h>
# include <unistd.h>
```

I più importanti sono *<stdio.h>* (input/output), *<stdlib.h>* (funzioni generali di utilità), *<math.h>* e *<float.h>* per la matematica, *<string.h>* (stringhe e altre funzioni matematiche), *<sys/stat.h>* e *<dirent.h>* (proprietà di files e cartelle), *<stdarg.h>* (funzioni con un numero variabile di argomenti).

Brevità di main

In genere il file che contiene la funzione `main` (noi lo chiameremo sempre *alfa.c*) dovrebbe contenere poche altre funzioni; la `main` dovrebbe essere breve e avere la mansione di coordinare le altre funzioni che sono raccolte nei vari files del progetto.

Come funziona make

La parte importante di un makefile (a cui si aggiungono regole piuttosto complicate per le abbreviazioni) sono i *blocchi elementari*, ognuno della forma

```
a: b c ...
TAB riga di comandi
TAB riga di comandi
```

in cui *a*, *b*, *c*, ... sono nomi qualsiasi (immaginare che siano nomi di files, anche se non è necessario che lo siano) e le righe di comandi contengono comandi della shell con alcune regole speciali per il trattamento delle abbreviazioni. `a` si chiama il *controllo primario* o *bersaglio* (*target*) del blocco, `b, c, ...` i *prerequisiti*. Un prerequisito si chiama *controllo secondario* se è a sua volta controllo primario di un altro blocco. I prerequisiti possono anche mancare.

Verificare il bersaglio `a` implica ricorsivamente le seguenti operazioni:

- Vengono verificati tutti i controlli secondari del blocco che inizia con `a`.
- Dopo la verifica dei controlli secondari vengono eseguiti i comandi del blocco salvo nel caso che il controllo sia già stato verificato oppure sia soddisfatta la seguente condizione:

  `a` *è il nome di un file esistente (nella stessa directory e nel momento in cui viene effettuata la verifica) e anche i prerequisiti* `b, c, ...` *sono nomi di files esistenti nessuno dei quali è più recente (tenendo conto della data dell'ultima modifica) del controllo primario.*

Può succedere che attraverso i comandi venga creato un file (come avviene ad esempio nella compilazione); l'esistenza di un file viene però controllata nel momento della verifica.

Per capire il funzionamento di `make` creiamo un file *Makefile* così composto:

```
# Prove per capire il makefile
.SILENT:

primobersaglio: a b c
TAB echo io primobersaglio

a:
TAB echo io a

b: d e
TAB echo io b

c: e f
TAB echo io c

d:
TAB echo io d

e:
TAB echo io e

f:
TAB echo io f
```

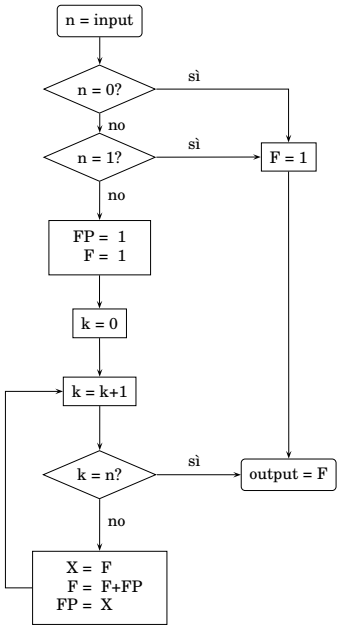
Dare dalla shell il comando `make` e vedere cosa succede. Creare poi files con alcuni dei nomi *a*, *b*, ... con `touch` oppure eliminare alcuni dei files già creati, ogni volta invocando il `make`. Variare l'esperimento modificando il makefile.

`echo x` stampa x sullo schermo.

Diagrammi di flusso

Algoritmi semplici possono essere espressi mediante diagrammi di flusso. Illustriamo questo metodo con un algoritmo per i numeri di Fibonacci che, ricordiamo, sono definiti tramite la ricorrenza

$$F_n := \begin{cases} 1 & \text{per } n = 0, 1 \\ F_{n-1} + F_{n-2} & \text{per } n \geq 2 \end{cases}$$



La tecnica dei diagrammi di flusso oggi viene usata pochissimo; è lenta e poco efficiente e non adatta alla rappresentazione di strutture di dati complesse. Può essere utile (nei primi giorni) al programmatore principiante per apprendere in un modo visivo alcuni meccanismi della programmazione procedurale.

Talvolta, soprattutto nei libri sugli algoritmi, si usano pseudolinguaggi, cioè linguaggi che non sono veri linguaggi di programmazione e nemmeno formalmente definiti in tutti i dettagli, ma le cui istruzioni e costruzioni hanno un significato facilmente intuibile. In un tale pseudolinguaggio il nostro diagramma di flusso potrebbe essere così tradotto:

```
          n=input
          if n=0 then goto uno
          if n=1 then goto uno
          FP=1
          F=1
          k=0
ciclo:    k=k+1
          if k=n then goto fine
          X=F
          F=F+FP
          FP=X
          goto ciclo
uno:      F=1
fine:     output=F
```

La rappresentazione di un algoritmo mediante un pseudolinguaggio è spesso adeguata quando si vuole studiare la complessità dell'algoritmo, perché in genere le istruzioni del pseudolinguaggio corrispondono in un modo trasparente alle operazioni effettivamente eseguite dal processore.

Si noti che nel diagramma di flusso e nel pseudolinguaggio abbiamo usato il segno di uguaglianza sia per le assegnazioni (ad esempio `k=k+1` ovviamente non può esprimere un'uguaglianza ma significa che il valore di `k` viene modificato e posto uguale a uno in più di quanto era prima) sia nei test di uguaglianza. Per distinguere i due utilizzi molti linguaggi (tra cui il C, il Perl e il Java) usano un doppio segno di uguaglianza (`==`) nei test di uguaglianza.

Nella programmazione in C i diagrammi di flusso sono del tutto superflui.

Istruzioni di controllo

Il C conosce tre tipi di istruzioni di controllo, istruzioni cioè che governano ripetizioni e ramificazioni nell'esecuzione di un programma:

Istruzioni condizionali: `if ... else`, `switch` e il costrutto `A ? x : y`.

Cicli: `for`, `while` e `do ... while`.

Salti: `goto`.

Discuteremo adesso in dettaglio questi comandi.

Blocchi

Un *blocco* in C è una parte di codice racchiusa tra parentesi graffe. In particolare è un blocco il corpo di una funzione che è sempre racchiuso tra parentesi graffe. Più istruzioni comprese in un blocco possono essere sintatticamente trattate come una singola istruzione, quindi in

```
if (x<5) {istruzione1; istruzione2;
istruzione3;}
```

tutte e tre le istruzioni vengono eseguite se la condizione dell'`if` è soddisfatta. A differenza dal Perl in C nelle istruzioni di controllo le parentesi graffe non sono necessarie per una singola istruzione. Quindi è corretta la sintassi in

```
if (x==3) istruzione1;
else {istruzione 2; istruzione 3;}
```

In seguito in genere verrà indicata solo la versione con un'istruzione singola che comunque può sempre essere sostituita da una sequenza di istruzioni comprese in un blocco.

Dichiarazioni all'interno di un blocco sono *locali* per questo blocco, valgono cioè solo in esso. Esempio: Con

```
void Prova ()
{int x;
x=5;
{int x=8; printf("%d\n",x);}
printf("%d\n",x);}
```

otteniamo l'output

```
8
5
```

che mostra che la variabile `x` del blocco interno non influenza la variabile omonima esterna.

if ... else

```
if (A) istruzione;
```

corrisponde in pseudocodice a

```
esegui istruzione se la condizione A
  e' vera
```

In questa sintassi, se la condizione `A` non è vera, `istruzione` semplicemente non viene eseguita. Se in tal caso si vuole invece eseguire un'altra istruzione, si usa `if ... else`:

```
if (A) istruzione1; else istruzione2;
```

significa in pseudocodice

```
esegui istruzione1 se la condizione A
  e' vera
altrimenti esegui istruzione2
```

Creiamo ad esempio una funzione che stampa sullo schermo il massimo tra due numeri reali (a cui, come abbiamo già osservato, in C corrisponde il tipo `double`). Usiamo la funzione `printf` per l'output formattato il cui uso verrà spiegato più avanti.

```
void Stampamax (double a, double b)
{if (a<=b) printf("%.2f\n",b);
else printf("%.2f\n",a);}
```

Possiamo anche creare una funzione apposita per il massimo:

```
double Max (double a, double b)
{if (a<=b) return b; return a;}
```

Allora `Stampamax` può essere semplificata:

```
void Stampamax (double a, double b)
{printf("%.2f\n",Max(a,b));}
```

Si osservi che in `Max` non appare un `else`; infatti, se `a<=b` è vera, con il `return b` si esce dalla funzione direttamente. Il `return` può essere usato anche senza argomento; in tal caso indica che si esce dalla funzione senza che venga restituito un risultato.

La funzione `Stampamax` ha come tipo di risultato `void`; ciò significa che non fornisce un risultato.

È una buona abitudine usare per le proprie funzioni nomi che iniziano con una maiuscola (C e Unix distinguono tra minuscole e maiuscole!) - ciò non solo evita molte interferenze con i nomi di funzioni di sistema, ma rende anche l'organizzazione dei programmi esteticamente più trasparente.

Tra l'altro la lingua italiana si presta molto bene alla programmazione, non c'è quindi alcuna ragione per usare nomi inglesi. L'italiano è leggero e flessibile e programmi scritti in italiano sono molto leggibili.

`if` ed `else` possono essere annidati; in tal caso ogni `else` fa coppia con l'`if` precedente più vicino (tenendo ovviamente conto delle parentesi graffe che raccolgono espressioni complesse in un'unica espressione). Analizzare le seguenti istruzioni:

```
if (A) if (B) istruzione;
if (A) if (B) istruz1; else istruz2;
if (A) {if (B) istruz1;} else istruz2;
if (A) if (B) istruz1; else istruz2;
  else istruz3;
```

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 2 ◇ 27 Settembre 2004

Il goto

L'uso del `goto` *richiede un'etichetta, un identificatore seguito da* `:`*. L'etichetta deve trovarsi nella stessa funzione in cui è usata, può essere locale in un blocco. Il* `goto` *nella programmazione moderna viene spesso criticato, essendo un tipico costrutto della programmazione basata sui diagrammi di flusso; è però utile nel caso di più cicli annidati. Infatti il* `break` *esce solo dal ciclo più interno in cui si trova. Esempio:*

```
{...
while (A)
  {istruzione1; for (...)
    {istruzione2; while (B)
      {istruzione3; if (C) goto fine; istruzione4;}
     istruzione5;}
   istruzione6;}
fine: istruzione7;}
```

Anche in altri casi, se usato con parsimonia, il `goto` *può essere la soluzione più pratica e anche più trasparente. Raramente comunque si avrà bisogno di più di un* `goto` *nella stessa funzione.*

Riscriviamo il diagramma di flusso a pagina 4 in C; è solo un esempio, perché proprio l'uso del `goto` *per effettuare cicli è giustamente criticato - il C possiede istruzioni molto più potenti per i cicli come adesso vedremo.*

```
double Fibonaccicongoto (int n) // Da non imitare.
{if (n==0) goto uno; if (n==1) goto uno;
double fp=1; double f=1; int k=0; double x;
ciclo: k=k+1; if (k==n) goto fine;
x=f; f=f+fp; fp=x; goto ciclo;
uno: f=1;
fine: return f;}
```

Particolarmente pericolosi, oltre a poco trasparenti, e sicuramente da evitare sono salti nell'interno di un blocco, come in

```
// Esempio da evitare.
if (A) goto etichetta;
if (B) {istruzione1; etichetta: istruzione2;}
```

Qui può accadere che non solo il lettore umano, ma anche il compilatore stesso non riesca ad interpretare correttamente il flusso del programma.

Punto interrogativo e virgola

Un'altra costruzione del C può essere talvolta usata per la distinzione di casi al posto di un `if ... else` o di uno `switch`. L'espressione `A? u:v` è un *valore* che è uguale a `u`, se la condizione `A` è soddisfatta, altrimenti uguale a `v`.

L'istruzione `x = A ? u : v;` è quindi equivalente a `if (A) x=u; else x=v;`
Queste costruzioni possono essere annidate: `x = A ? u : B ? v : w;`

Spesso il punto interrogativo viene combinato con l'*operatore virgola*:
`(istruzione1,istruzione2,istruzione3,u)` è un valore, che è uguale al valore di `u` dopo esecuzione, nell'ordine indicato, di `istruzione1`, `istruzione2` e `istruzione3`. Quindi `x=(istruzione1,istruzione2,istruzione3,u)` è (di norma, cioè quando le istruzioni non hanno effetti collaterali) equivalente con

```
istruzione1; istruzione2; istruzione3; x=u;
```

Tipicamente l'operatore virgola viene combinato con il punto interrogativo:

```
x = A ? (istruzione1, istruzione2, u) : (istruzione3, v);
```

è (di norma) equivalente a

```
if (A) {istruzione1; istruzione2; x=u;}
else {istruzione3; x=v;}
```

Evidentemente punto interrogativo e virgola possono essere sostituiti con sequenze di `if ... else`, risultano però spesso utili come abbreviazioni o in situazioni con molte distinzioni di casi.

In questo numero

switch

L'istruzione

```
switch(t) {case 3: case 4: istruzione1;
case 1: istruzione2; break;
case 10: istruzione3; istruzione4; break;
case 12: istruzione5;
default: istruzione 6;}
```

può essere riformulata mediante `if` annidati e `goto` nel modo seguente:

```
if (t==3) goto caso3o4;
if (t==4) goto caso3o4;
if (t==1) goto caso1;
if (t==10) goto caso10;
if (t==12) goto caso12;
goto altricasi;
caso3o4: istruzione1;
caso1: istruzione2; goto uscita;
caso10: istruzione3; istruzione4; goto uscita;
caso12: istruzione5;
altricasi: istruzione6;
uscita: ...}
```

`t` deve essere un'*espressione* di tipo intero o compatibile con il tipo intero; i valori che seguono i `case` devono essere *costanti* di tipo compatibile con il tipo intero; queste costanti (nel nostro caso 3,4, 1,10,12) devono essere tutte distinte.

Attenzione: I `case` e il `default`, come si vede dalla trascrizione, nel C hanno il significato di etichette e non alterano il flusso di esecuzione del programma che continua liberamente attraverso queste etichette, finché non si incontra un `break`, senza il quale non si escludono a vicenda.

Il `default` può anche mancare. Gli `switch` possono essere annidati:

```
switch(t) {case 8: istr1; break;
case 0: switch(s) {case 2: istr2; break;
  case 3: istr3; break;
  default: istr4;} break;
case 20: istr5; break; default: istr6;}
```

***

Negazione con !

La negazione di una condizione logica `A` è denotata con `!A`, ad esempio in `if (!A) ...`

Più dettagli a pagina 7.

Segno di un numero reale

Definiamo una funzione per il segno di un numero reale:

$$\mathrm{sgn}(x) := \begin{cases} 1 & \text{se } x > 0 \\ 0 & \text{se } x = 0 \\ -1 & \text{se } x < 0 \end{cases}$$

```
int Segno (double x)
{if (x>0) return 1; if (x==0) return 0;
return -1;}
```

Più breve è la versione che usa punto e virgola:

```
int Segno (double x)
{return x>0 ? 1 : x==0 ? 0 : -1;}
```

Anche la funzione `Max` può così essere semplificata:

```
double Max (double a, double b)
{return a<=b ? b : a;}
```

for

Il `for` del C ha la forma

```
for (istruzione1;A;istruzione3)
istruzione2;
```

equivalente alla sequenza

```
istruzione1;
ciclo: if (!A) goto uscita;
  istruzione2;
  istruzione3;
  goto ciclo;
uscita: ...
```

`istruzione1` e `istruzione2` possono essere anche successioni di istruzioni, separate da virgole; l'ordine in cui le istruzioni in ogni successione vengono eseguite non è prevedibile. Ciascuno dei tre campi può essere vuoto. Per uscire da un `for` si usa il `break`:

```
for (istruzione1;A;istruzione3)
{istruzione2; if (B) break;
istruzione4;}
```

corrisponde alla sequenza

```
istruzione1;
ciclo: if (!A) goto uscita;
  istruzione2;
  if (B) goto uscita;
  istruzione4;
  istruzione3;
  goto ciclo;
uscita: ...
```

Naturalmente si esce dal `for` anche quando la condizione `A` non è più soddisfatta:

```
for (k=0;k<5;k++) printf("%d\n",k*k);
```

stampa i quadrati dei numeri $0..4$. Lo stesso effetto si ottiene con

```
for (k=0;;k++)
{if (k>=5) break;
printf("%d\n",k*k);
```

Gli operatori / e %

Per numeri reali a e b (cioè quando almeno uno dei due è di tipo `double`) l'espressione `a/b` è il quoziente reale di a e b; se invece sono entrambi interi, viene calcolato solo la parte intera, ad esempio `45/7` è uguale a 6.

L'operatore `%` fornisce il resto nella divisione intera: `a%m` è il resto nella divisione di a per m; ad esempio `45%7` è uguale a 3. In particolare `a%2` è 0 se e solo se a è pari, altrimenti è uguale a 1.

Nell'uso di `/` e `%` bisogna stare attenti per argomenti negativi; torneremo su questo punto.

k++ e altre abbreviazioni

Per aumentare una variabile intera `k` di uno si usa `k++` oppure, con una lieve differenza, `++k`. Isolatamente queste istruzioni hanno entrambe lo stesso effetto di `k=k+1` (tralasciamo il punto e virgola). Una differenza si ha se vengono usate all'interno di un'altra operazione. In tal caso con `k++` prima viene eseguita quella operazione, e solo dopo avviene l'aumento di `k`. Con `++k` invece `k` viene prima aumentata e l'operazione eseguita sul valore aumentato di `k`. Con

```
void Prova ()
{int n;
n=5; printf("%d\n",(n++)%2);
n=5; printf("%d\n",(++n)%2);}
```

otteniamo quindi l'output

```
1
0
```

In modo analogo si usano `k--` e `--k`.

Invece di `a=a+b;` si può anche scrivere `a+=b;`, e similmente si possono abbreviare assegnazioni per gli altri operatori binari, quindi si userà (tralasciamo anche qui il punto e virgola) `a*=b` per `a=a*b`, `a/=b` per `a=a/b` e `a-=b` per `a=a-b`. È una notazione buona e comoda, infatti `resistenza/=2` è più breve e più leggibile di `resistenza=resistenza/2`.

continue

L'istruzione `continue` interrompe il passaggio corrente di un ciclo e porta all'immediata esecuzione del passaggio successivo. Quindi ad esempio

```
for (istruzione1;A;istruzione3)
{istruzione2; if (B) continue;
istruzione4;}
```

corrisponde alla sequenza

```
istruzione1;
ciclo: if (!A) goto uscita;
  istruzione2;
  if (B) {istruzione3; goto ciclo;}
  istruzione4; istruzione3;
  goto ciclo;
uscita: ...
```

Si osservi che la `istruzione3` del `for` viene eseguita anche dopo il `continue`. `break` e `continue` possono essere utilizzati anche nel `while` e nel `do ... while`.

Il problema del 3x+1

Questo è sicuramente il problema più inutile della matematica. Sembra incredibile che, da quando è stato posto 70 anni fa, nessuno sia riuscito a risolverlo.

Partiamo con un numero naturale maggiore di 1 qualsiasi. Se è pari, lo dividiamo per 2, altrimenti lo moltiplichiamo per 3 e aggiungiamo 1. Questa operazione T, matematicamente espressa da

$$T(x) := \begin{cases} x/2 & \text{se } x \text{ è pari} \\ 3x+1 & \text{se } x \text{ è dispari} \end{cases}$$

può essere programmata in C con la seguente funzione:

```
int T (int x)
{if (x%2==0) return x/2; return 3*x+1;}
```

Con il numero così ottenuto ripetiamo l'operazione e ci fermiamo solo quando arriviamo a 1. Ad esempio partendo con 7 otteniamo

$$22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$$

Sembra che alla fine l'algoritmo si fermi sempre, cioè che prima o poi si arrivi sempre a 1. Ma nessuno riesce a dimostrarlo. Talvolta, ad esempio partendo con 27, si ottengono dei cicli piuttosto lunghi. È noto come il problema del $3x+1$ o anche il problema di Collatz, dal famoso analista numerico tedesco Lothar Collatz (1910-1990) che, da studente, lo ha inventato.

Lothar Collatz

while

L'istruzione `while(A) istruzione;` è equivalente a `for(;A;) istruzione;` e corrisponde quindi alla sequenza

```
ciclo: if (!A) goto uscita;
  istruzione;
  goto ciclo;
uscita: ...
```

La seguente funzione stampa sullo schermo $x, Tx, T^2x, \ldots$ fino a quando $Tx = 1$, dove T è l'operatore del $3x+1$:

```
void Trexpiu1 (int x)
{while(x>1) {printf("%d ",x); x=T(x);}
printf("\n");}
```

do ... while

L'istruzione `do istr; while(A);` è equivalente a `for(istr;A;) istr;` e corrisponde quindi alla sequenza

```
  istr;
ciclo: if (!A) goto uscita;
  istr;
  goto ciclo;
uscita: ...
```

Numeri esadecimali

Nei linguaggi macchina e assembler molto spesso si usano i numeri esadecimali o, più correttamente, la rappresentazione esadecimale dei numeri naturali, cioè la loro rappresentazione in base 16.

Per $2789 = 10{\cdot}16^2+14{\cdot}16+5{\cdot}1$ potremmo ad esempio scrivere

`2789 = (10,14,5)`$_{16}$.

In questo senso `10`, `14` e `5` sono le cifre della rappresentazione esadecimale di `2789`. Per poter usare lettere singole per le cifre si indicano le cifre `10`, ..., `15` mancanti nel sistema decimale nel modo seguente:

10	A
11	B
12	C
13	D
14	E
15	F

In questo modo adesso possiamo scrivere $2789 = (AE5)_{16}$. In genere si possono usare anche indifferentemente le corrispondenti lettere minuscole. Si noti che `(F)`$_{16}$ `= 15` assume nel sistema esadecimale lo stesso ruolo come il `9` nel sistema decimale. Quindi `(FF)`$_{16}$ `=` $255 = 16^2-1 = 2^8-1$. Un numero naturale n con $0 \le n \le 255$ si chiama un *byte*, un *bit* è invece uguale a 0 o a 1. Esempi:

	0	$(0)_{16}$
	14	$(E)_{16}$
	15	$(F)_{16}$
	16	$(10)_{16}$
	28	$(1C)_{16}$
2^5	32	$(20)_{16}$
2^6	64	$(40)_{16}$
	65	$(41)_{16}$
	97	$(61)_{16}$
	127	$(7F)_{16}$
2^7	128	$(80)_{16}$
	203	$(CB)_{16}$
	244	$(F4)_{16}$
	255	$(FF)_{16}$
2^8	256	$(100)_{16}$
2^{10}	1024	$(400)_{16}$
2^{12}	4096	$(1000)_{16}$
	65535	$(FFFF)_{16}$
2^{16}	65536	$(10000)_{16}$

Si vede da questa tabella che i byte sono esattamente quei numeri per i quali sono sufficienti al massimo due cifre esadecimali.

Nell'immissione di una successione di byte come un'unica stringa spesso si pone uno zero all'inizio di quei numeri (da `0` a `F`) che richiedono una cifra sola, ad esempio la stringa

`0532A2014E586A750EAA`

può essere usata per rappresentare la successione

(5,32,A2,1,4E,58,6A,75,E,AA)

di byte.

Il linguaggio macchina

Proviamo a scrivere nel linguaggio macchina del 6502 l'algoritmo per la moltiplicazione con 10 di un byte n. L'aritmetica nel 6502 avviene modulo 256, quindi, affinché il risultato sia corretto, deve valere $0 \le n \le 25$.

Utilizziamo la moltiplicazione russa, che verrà spiegata più avanti, per questo caso particolare: Formiamo $p := 2n$, poi raddoppiamo n due volte (cosicché n adesso ha 8 volte il suo valore iniziale) e aggiungiamo p.

Per n usiamo l'indirizzo esadecimale `F0`, per la variabile p l'indirizzo `F1`. Nel 6502 tutte le operazioni aritmetiche avvengono nell'accumulatore; dobbiamo quindi trasferire il contenuto di `F0` nell'accumulatore mediante l'istruzione `A5 F0`. Le istruzioni del 6502 consistono di al massimo 3 bytes, di cui il primo indica l'operazione, i rimanenti gli argomenti.

Disabilitiamo con `D8` la modalità decimale, poi con `0A` moltiplichiamo il contenuto a dell'accumulatore per due (in verità questa istruzione sposta i bits nella rappresentazione binaria di a di una posizione a sinistra e pone l'ultimo bit uguale a 0). Memorizziamo il nuovo valore in `F1` con `85 F1`, poi effettuiamo due raddoppi con `0A 0A`. Mettiamo il riporto uguale a zero con `18` (esadecimale) e addizioniamo all'accumulatore il contenuto di `F1` con `65 F1`; infine trasferiamo il valore dell'accumulatore nella locazione di memoria `F0` con `85 F0`. Il programma completo in linguaggio macchina a questo punto è

```
A5 F0 D8 0A 85 F1 0A 0A 18 65 F1 85 F0.
```

I linguaggi assembler

Il linguaggio macchina del processore 6502, utilizzato da alcuni dei più famosi personal computer di tutti i tempi (Apple II, Commodore, Atari) all'inizio degli anni '80, era elegante ed efficiente e molto istruttivo, perché era molto facile lavorare direttamente in memoria. Quei computer, piuttosto lenti se programmati in Basic, si rivelavano vivacissimi se programmati in linguaggio macchina (o in assembler). Il processore ha ancora oggi i suoi appassionati e una pagina Web (*www.6502.org/*) offre esempi di codice, descrizioni delle istruzioni e links ad altre pagine Web.

Come abbiamo visto, è possibile inserire direttamente un programma in linguaggio macchina in memoria. La difficoltà non consiste tanto nel memorizzare i codici (almeno per il 6502 che ha relativamente poche istruzioni), perché è facile ricordarseli dopo pochi giorni di pratica, ma piuttosto nella manutenzione del programma (modifiche, documentazione). Se ad esempio si volesse inserire un nuovo pezzo di programma in un certo punto, bisogna non solo spostare una parte del codice (ciò è possibile mediante appositi comandi dal terminale), ma è necessario anche aggiustare tutti gli indirizzi dei salti.

Per questa ragione sono stati inventati i linguaggi assembler. Così si chiamano linguaggi le cui istruzioni fondamentali corrispondono esattamente alle istruzioni in linguaggio macchina, ma sono espresse in una forma non numerica più facile da ricordare, e inoltre permettono l'utilizzo di nomi simbolici per gli indirizzi. Gli assembler più recenti (ad esempio per i processori Intel) sono molto complessi e forniscono moltissime funzioni.

Il programma in linguaggio macchina alla fine dell'articolo precedente viene tradotto così in assembler:

```
            ; questo è un commento
CLD         ; clear decimal
LDA   N     ; load accumulator
ASL         ; arithmetic shift left
STA   P     ; store accumulator
ASL
ASL
CLC         ; clear carry
ADC   P     ; add with carry
STA   N
```

Vero e falso

Il C non prevede un tipo booleano apposito, e usa invece 0 come *falso*, mentre ogni numero $\neq 0$ è considerato *vero*.

Quindi anche il numero 0.73 è considerato vero; il risultato di un'espressione logica che usa operatori di confronto o operatori logici è sempre uguale a 0 o 1, quindi la funzione

```
int maggioredelquadrato (int a, int b)
{return a>b*b;}
```

restituisce 1 se $a > b^2$ e 0 altrimenti.

Operatori logici

Nei confronto di numeri si usano `<`, `<=`, `>`, `>=`; per il testi di uguaglianza si deve usare `==`, per la disuguaglianza `!=`.

Attenzione: Il C accetta anche `if (a=1)`, ma in questo caso prima viene assegnato ad `a` il valore 1 e questo valore è poi usato per il test, quindi in questo `if` la condizione è sempre vera.

Una assegnazione in una condizione logica può avere senso invece, se anche il valore assegnato è variabile, quindi, usata con cautela, un'istruzione `if (a=b) ...` può avere senso e essere anche vantaggiosa per la leggibilità. Bisogna però ricordarsi che non equivale affatto a `if (a==b) ...` ma che si tratta invece di un'abbreviazione per

```
a=b; if (a) ...
```

Gli operatori logici sono `&&` (*AND*), `||` (*OR*) e, come già sappiamo, `!` (NOT). Invece di `if (x==0)` si può anche usare `if (!x)`, mentre `if (x%2==0)` è equivalente a `if (!(x%2))`; in questo secondo caso sono necessarie le parentesi interne, come è spiegato più sotto.

La ragione perché i simboli scelti sono doppi è che quando il C fu inventato le memorie erano piccole e costose ed erano ancora molto usati gli operatori logici *bit per bit* (ad esempio per *flags*) per i quali vennero previsti i simboli `&` e `|` che esistono ancora oggi ma sono usati solo raramente. Ad esempio $25 = (11001)_2$ e $13 = (01101)_2$ e se effettuiamo un AND bit per bit vediamo che `25&13` è uguale a $(01001)_2 = 9$. Similmente (esercizio) `25|13=29`. Invece naturalmente `25&&13` e `25||13` sono entrambi uguali ad 1. Controllare con `printf("%d\n",25&13);`.

Abbiamo già osservato che il punto esclamativo viene usato per la negazione logica. Se `A` è un'espressione vera (cioè diversa da 0), allora `!A` è falso, cioè 0, e viceversa. In altre parole `!A` è equivalente ad `A==0`.

Bisogna stare attenti al fatto che il punto esclamativo ha una priorità più alta degli altri operatori, quindi `!x>0` non è la negazione di `x>0`, ma è equivalente a `(!x)>0` e quindi, siccome `!x` può assumere solo i valoro 0 e 1, a `!x`, cioè a `x==0`.

Per la stessa ragione `!x>=0` non è la negazione di `x>=0`, ma è la condizione che `!x` sia ≥ 0, e ciò è sempre vero, ancora perché l'espressione logica `!x` assume solo i valori 0 e 1 e quindi non è mai negativa.

Nel C99 esiste anche la possibilità di usare il tipo `bool` con i valori `true` e `false`, includendo `<stdbool.h>`, e altri nomi per gli operatori logici, come `and` per `&&`, `bitand` per `&` ecc., includendo `<iso646.h>`.

Le dichiarazioni

In C ogni variabile o funzione, prima del suo utilizzo, deve essere dichiarata. Le dichiarazioni hanno essenzialmente il solo scopo di indicare la quantità di memoria che una variabile o le operazioni connesse con l'esecuzione di una funzione occupano, ma non vincolano il programmatore a usare quella parte della memoria in un modo determinato. Il codice

```
double x;
sprintf(&x,"Alberto");
puts(&x);
```

che pone la stringa `"Alberto"` *nella locazione prevista per la variabile* `x`*, che è di tipo* `double`*, e poi la visualizza, provocherà, su molti compilatori, un avvertimento (*`warning`*), perché l'uso dello spazio occupato da una variabile reale per contenere una stringa è insolito, ma non genera un errore. In questo caso non commettiamo nemmeno l'errore di scrivere in una parte non disponibile della memoria, perché la stringa* `"Alberto"` *occupa, con il suo carattere terminale (di cui parleremo più avanti), 8 byte, esattamente quanto viene riservato per una variabile di tipo* `double` *(come possiamo verificare con* `printf("%ld\n",sizeof(double));`*)*

Se con

```
double x;
sprintf((char*)&x,"Alberto");
puts((char*)&x);
```

si informa il compilatore delle proprie intenzioni, non protesterà più. Non protesterà nemmeno se invece di `"Alberto"` *usiamo la stringa* `"Albertino"` *che, essendo più lunga dello spazio disponibile in* `&x`*, sovrascriverà una parte della memoria non a nostra disposizione e provocherà quasi sicuramente errori nel programma o addirittura la necessità di riavviare il computer (ciò non accade però sotto Linux). Per le regole complete che governano le dichiarazioni si può consultare il libro di riferimento di Harbison / Steele.*

S. Harbison/G. Steele: *C. A reference manual. Prentice Hall, 2002.*

Nomi

Per designare variabili e funzioni o tipi si usano *nomi* (o identificatori, in inglese *identifiers*), che devono iniziare con una lettera compresa nell'insieme `{A,...,Z,a...z,_}` e possono continuare con lettere o cifre, cioè caratteri compresi nell'insieme `{A,...,Z,a...z,_,0,...,9}`. Il C (come tutto l'ambiente Unix, a cui C storicamente appartiene) distingue tra minuscole e maiuscole. Il carattere `$` è ammesso solo per scopi speciali per l'interfaccia con altri linguaggi di programmazione. Nomi leciti sono quindi `X, Xa, A3, Ap16_88r`, mentre non sono leciti i nomi `3x, alfa-beta, teo+`. Si devono anche evitare i nomi predefiniti del C o nomi predefiniti in librerie di cui si fa uso. Mentre l'uso dei nomi riservati

```
auto break case char const continue
default do double else enum extern
float for goto if int long register
return short signed sizeof static
struct switch typedef union unsigned
void volatile while inline restrict
_Bool _Complex _Imaginary
```

è del tutto proibito, ad esempio con

```
int return;
```

si provoca un messaggio d'errore di sintassi (`parse error`) del compilatore, è anche meglio non usare la dichiarazione

```
double cos;
```

perche `cos` è il nome della funzione coseno e non dovrebbe essere utilizzato come nome di una variabile. Infatti se successivamente vogliamo usare la funzione coseno come in

```
double cos;
double u=cos(3.14);
```

il compilatore protesta - appare un

```
called object is not a function
```

perché abbiamo ridefinito il nome della funzione che quindi non viene più riconosciuta. Anche nomi che iniziano con `_`, benché ammessi, possono andare in conflitto con vari identificatori predefiniti del C che iniziano con questo carattere, soprattutto se si usano nomi inglesi. Ma abbiamo già osservato a pag. 4 che si dovrebbero usare il più possibile nomi italiani.

Nel C99 in un nome vengono considerati i primi 63 caratteri, nelle implementazioni più vecchie i primi 31 caratteri. Nomi più lunghi che coincidono nei primi 63 o 31 caratteri vengono considerati uguali. Raramente si usano nomi così lunghi, ma soprattutto per le funzioni nomi di lunghezza media come `Cancellafinestra` (16 caratteri) possono avere un senso.

In questo numero

Il codice ASCII

Il tipo `char` viene utilizzato per rappresentare caratteri, corrisponde quasi sempre a un numero intero compreso tra 0 e 255 (un byte), a cui può essere convertiti. I caratteri „americani" (infatti ASCII significa *American Standard Code for Information Interchange*) corrispondono ai numeri da 0 a 127 e sono codificati nella tabella ASCII:

0 - 31	caratteri speciali (tasti di controllo)
32	spazio
33 - 47	! " # $ % & ' ( ) * + , - . /
48 - 57	0 1 2 3 4 5 6 7 8 9
58 - 64	: ; < = > ? @
65 - 90	A ... Z
91 - 96	[\ ] ^ _ ‘
97 - 122	a ... z
123 - 126	{ | } ~
127	tasto *bs* sopra *invio*

Per stampare un numero tra 0 e 255 come carattere si usa la sigla `%c` in `printf`. Con

```
int k;
for (k=33;k<=47;k++)
printf("%d   %c\n",k,k);
```

otteniamo quindi l'output

```
33   !
34   "
35   #
36   $
37   %
38   &
39   '
40   (
41   )
42   *
43   +
44   ,
45   -
46   .
47   /
```

Otteniamo tutti i caratteri ASCII stampabili, allineati in gruppi a dieci su ogni riga, con

```
int k;
for (k=33;k<=126;k++)
{printf("%c ",k);
if (k%10==0) printf("\n");}
printf("\n");
```

Come bisogna modificare l'algoritmo affinché già la prima riga contenga dieci caratteri?

Tipi di dati elementari

Esistono numerose variazioni dei tipi di dati elementari (descritti in dettaglio nel manuale di Harbison/Steele); in pratica però sono quasi sempre sufficienti i tipi `char`, `int`, `unsigned int`, `long`, `size_t` (e simili tipi speciali), `double`, `void`. Vettori e puntatori e tipi composti (strutture e unioni) verranni trattati più avanti. Il tipo `char` è già stato discusso a pagina 8.

Il tipo `int` rappresenta interi, sui PC tipicamente compresi tra $-2147483648 = -2^{31}$ e $2147483647 = 2^{31} - 1$, `unsigned int` interi ≥ 0, quindi tra 0 e $4294676295 = 2^{32} - 1$. Attualmente `long` è spesso uguale a `int` e `size_t` uguale a `long`. Il tipo `double` rappresenta numeri reali, naturalmente anche essi con limiti di grandezza e precisione.

Il numero di bytes occupato da un tipo o da una variabile di quel tipo può essere ottenuto con la funzione `sizeof`:

```
printf("%d %d %d %d\n",sizeof(char),
  sizeof(int),sizeof(long),
  sizeof(double));
```

con output

```
1 4 4 8
```

Includendo `<limits.h>` si possono ottenere i limiti minimi e massimi dei tipi interi:

```
printf("%d %d %u\n",INT_MAX,
  UINT_MAX,UINT_MAX);
```

con output

```
2147483647 -1 4294967295
```

Il `-1` in questo risultato deriva dal fatto che la sigla di formato `%d` in `printf` non è adatta per stampare un intero senza segno così grande. Infatti 4294967295 è congruo a -1 modulo 2^{32} e questo è il valore visualizzato se utilizziamo il formato `%d` previsto per gli interi con segno che, come detto, vanno da -2147483648 a 2147483647.

In

```
unsigned x=1000;
if (x>-1) ...
```

nell'`if` -1 viene anch'esso convertito in `unsigned int`, perché `x` è di questo tipo, per cui viene effettuato il confronto

```
if (1000 > 4294967295)
```

che è sempre falso. Quindi bisogna stare attenti in questi casi.

---

void

Il tipo `void` viene usato soprattutto in due modi; da un lato per indicare funzioni che non restituiscono un risultato, come in `void f(double)`, dall'altro lato per definire puntatori di tipo generico (indirizzi puri) che possono poi essere convertiti in puntatori di tipo determinato a seconda dei casi.

printf

Abbiamo già usato varie volte la funzione `printf` per l'output formattato. Consideriamo un altro esempio:

```
printf("%3d %-12.4f\n",m,x);
```

Il primo parametro di `printf` è sempre una stringa. Questa può contenere delle *sigle di formato* che iniziano con `%` e indicano la posizione e il formato per la visualizzazione degli argomenti aggiuntivi. Nell'esempio `%3d` *tiene il posto* per il valore della variabile `m` che verrà visualizzata come intero su uno spazio di tre caratteri, mentre `%-12.4f` indica una variabile di tipo `double` che è visualizzata su uno spazio di 12 caratteri, arrotondata a 4 cifre dopo il punto decimale, e allineata a sinistra a causa del `-` (altrimenti l'allineamente avviene a destra). Quando i valori superano lo spazio indicato dalla sigla di formato, viene visualizzato lo stesso il numero completo, rendendo però imperfetta l'intabulazione che avevamo in mente. Esempio:

```
double a=12345.67,b=20; int n=24;
printf("A%6.2fB%d\nA%6.2fB%d\n",
  a,n,b,n);
```

Otteniamo

```
A12345.67B24
A 20.00B24
```

Nonostante avessimo utilizzato la stessa sigla di formato `%6.2f` per `a` e `b`, prevedendo lo spazio di 6 caratteri per ciascuna di esse, l'allineamento in `B` non è più corretto, perché la `a` impiega più dei 6 caratteri previsti.

I formati più usati sono:

`%c`	carattere
`%d`	intero
`%ld`	intero lungo
`%u`	intero ≥ 0
`%lu`	intero lungo ≥ 0
`%f`	`double`
`%s`	stringa
`%x`	rappr. esadecimale
`%o`	rappr. ottale

Per specificare un segno di `%` nella sigla di formato di `printf` si usa `%%`.

All'interno della specificazione di formato si possono usare `-` per l'allineamento a sinistra e `0` per indicare che al posto di uno spazio venga usato `0` come carattere di riempimento negli allineamenti a destra. Quest'ultima opzione viene usata spesso per rappresentare numeri in formato esadecimale byte per byte. Vogliamo ad esempio scrivere la tripla di numeri $(58, 11, 6)$ in forma esadecimale e byte per byte (cioè riservando due caratteri per ciascuno dei tre numeri). Le rappresentazioni esadecimali sono `3a`, `b` e `6`, quindi con

```
int a=58,b=11,c=6;
printf("%2x%2x%2x\n",a,b,c);
```

otteniamo `3a b 6` come output; per ottenere il formato desiderato `3a0b06` (richiesto ad esempio dalle specifiche dei colori in HTML) dobbiamo usare invece

```
printf("%02x%02x%02x\n",a,b,c);
```

Come visto, nelle sigle di formato di `printf` può essere specificato il numero dei caratteri da usare; con `*` questo numero diventa anch'esso variabile e viene indicato negli argomenti aggiuntivi. Assumiamo ad esempio che vogliamo stampare tre stringhe variabili su righe separate; per allinearle, calcoliamo il massimo `m` delle loro lunghezze, usando la funzione `strlen` del C e la nostra funzione `Max` (pag. 4 e 6) e incorporiamo questa informazione nella sigla di formato:

```
char *A="Ferrara", *B="Roma",
  *C="Rovigo";
int m; m=Max(strlen(A),strlen(B));
m=Max(m,strlen(C));
printf("|%*s|\n|%*s|\n|%*s|\n",
  m,A,m,B,m,C);
```

Si ottiene l'output desiderato:

```
|Ferrara|
|   Roma|
| Rovigo|
```

Benché `Max` restituisca un valore di tipo `double`, l'assegnazione ad `m`, che è di tipo `int`, ha fatto in modo che al momento dell'uso all'interno di `printf` il valore usato sia di tipo `int`.

Questa tecnica di trasformare un `double` in un `int` non è sempre affidabile, quindi in casi critici sarebbe meglio creare una funzione `Maxint` apposita per gli interi.

Esiste anche una funzione di input formattato (`scanf`), un po' difficile da usare.

---

enum

La dichiarazione

```
enum {alfa=3,beta,gamma,delta=10,
  epsilon};
```

definisce delle costanti intere con i valori `alfa=3`, `beta=4`, `gamma=5`, `delta=10`, `epsilon=11`; essendo queste variabili costanti, la dichiarazione può essere scritta in un file di intestazione che verrà incluso con una direttiva `# include`. I tipi enumerativi vengono spesso usati per parametri di ramificazioni, ad esempio per poter scegliere tra più operazioni da eseguire. Esempio:

```
enum {id,quad,cubo,quarta,radice,logar};

double Operazione (double x, int k)
{switch(k) {case id: return x;
case quad: return x*x;
case cubo: return x*x*x;
case quarta: return x*x*x*x;
case radice: return sqrt(x);
case logar: return log(x);}}

void Prova ()
{double x;
x=17;
printf("%.2f\n",Operazione(x,radice));}
```

Abbiamo scelto `logar` per il nome del numero che corrisponde al logaritmo, perché il nome `log` è già occupato dalla funzione logaritmo del C.

Operazioni aritmetiche

Gli operatori `+,-,*,/` in C tengono conto del tipo delle variabili utilizzate. Il tipo `char` corrisponde ai numeri interi tra 0 e 255, perciò con

```
char u=200, v=70;
printf("\n%d\n",u+v);
```

otteniamo l'output 14, perché l'aritmetica viene effettuata modulo 256. Similmente

```
int u=2000000000, v=700000000;
printf("\n%d\n",u+v);
```

dà `-1594967296`. In particolare bisogna ricordarsi che in C (a differenza dal Perl ad esempio) l'operatore di divisione `/`, se applicato a variabili intere, dà il quoziente intero, quindi `10/4` è 2, mentre `10.0/4` è 2.5, perché il C riconosce dal punto decimale che il calcolo avviene in ambiente `double`.

L'arrotondamento avviene, a differenza dall'uso in matematica, verso il numero intero più vicino allo zero, perciò `-17/3` è -5, mentre in matematica la parte intera di $-5.66\ldots$ è -6. Per evitare questa fonte di errori è consigliabile convertire i numeri usati in valori positivi. Lo stesso vale per l'operatore `%` che calcola il resto nella divisione intera: `17%3` è 2.

Creeremo più avanti una piccola collezione di funzioni simili che operano correttamente anche per argomenti negativi.

Funzioni

Funzioni in C vengono definite nel formato

```
tris f (tipi e nomi degli argomenti)
{corpo della funzione}
```

dove tris è il tipo del risultato della funzione. Se la funzione non restituisce un risultato, si usa il tipo `void` (pag. 9). Le graffe del corpo seguono l'intestazione senza un punto e virgola interposto!

Gli argomenti della funzione e le variabili dichiarate nel corpo sono *locali* e non interferiscono con variabili omonime esterne:

```
void f (int n)
{int k;
for (k=0;k<n;k++)
{printf("%d\n",k*k);}}

void g ()
{int k=3; f(k);}
```

Nell'esecuzione il `k` di `g` in `f(k)` non interferisce con il `k` utilizzato nel corpo di `f`.

Se la funzione non ha argomenti, l'intestazione sarà semplicemente della forma

```
tris f ()
```

Altrimenti tipi e nomi degli argomenti verranno indicati come nelle dichiarazioni di variabili, cioè il tipo seguito dal nome, separando però ogni argomento da quello successivo mediante una virgola e non mediante un punto e virgola:

```
int T (int x)
{...}

double Max (double a, double b)
{...}

double Operazione (double x, int k)
{...}
```

Le parentesi vuote hanno invece un altro significato nelle dichiarazioni, come vedremo.

Una sintassi speciale hanno le funzioni con un numero variabile di argomenti, che tratteremo più avanti.

return

Se il valore di `u` deve essere restituito come risultato della funzione, si usa `return u`. Un `return` senza argomento può essere usato per uscire dalla funzione prima di raggiungere la fine del blocco che definisce la funzione. Possono apparire anche più istruzioni `return` in una funzione, come si vede dalle funzioni `S` e `Fatt` definite a pag. 1.

Il `return` deve essere distinto dall'istruzione `exit(0);` vista a pag. 2, che dovrebbe essere usata solo nella `main` e ha lo scopo di terminare correttamente il processo che eseguiva la funzione.

Dichiarazione di funzioni

Dalla *definizione* di una funzione, come descritta nell'articolo precedente, si distingue la sua *dichiarazione*.

Una funzione non può essere usata se nel punto in cui deve essere eseguita la prima volta, non è nota al compilatore. Talvolta, se la funzione `f` usa la funzione `g`, è sufficiente anteporre semplicemente la definizione della `g` a quella della `f`, come in

```
int g (double a)
{...}

void f (int x, int y)
{double u; int n;
...; n=g(u); ...}
```

Ciò non funziona però, se la `g` a sua volta usa la `f`:

```
int g (double a)
{... f(2,5); ...}

void f (int x, int y)
{double u; int n;
... n=g(u); ...}
```

In questo caso è necessario anteporre le dichiarazioni delle funzioni alle definizioni:

```
void f();
int g();
////////////////////////
int g (double a)
{... f(2,5); ...}

void f (int x, int y)
{double u; int n;
... n=g(u); ...}
```

Quando una funzione viene utilizzata in altri files, la sua dichiarazione viene inserita in un file di intestazione che verrà incluso con `# include` nei files che utilizzano la funzione.

Se in una dichiarazione non vengono indicati argomenti per una funzione, lasciando vuota la parentesi degli argomenti, come in

```
int f();
```

ciò non significa, come nella definizione della funzione, che la funzione non ha argomenti, ma che non si vogliono specificare gli argomenti e soprattutto i loro tipi. Ciò, benché talvolta criticato, è piuttosto utile quando si vogliono utilizzare queste funzioni come argomenti di altre funzioni.

Talvolta invece è necessario indicare il tipo degli argomenti (non i loro nomi) anche nelle dichiarazioni. Infatti forse l'unico veramente fastidioso difetto del C (eliminato nel C++) è che, se un argomento di una funzione nella definizione appare di tipo `double` e se manca la dichiarazione della funzione o se nella dichiarazione manca l'indicazione del tipo, interi usati per questo argomento danno tipicamente risultati errati. Ricordiamo la funzione

```
double Max (double a, double b)
{return a<=b ? b : a;}
```

definita a pagina 6. Se la dichiariamo nella forma abbreviata

```
double Max();
```

senza indicare il tipo degli argomenti, con

```
printf("%.0f\n",Max(5,8));
```

otterremo un risultato errato, probabilmente 0. Per risolvere il problema si può o rendere riconoscibili gli argomenti come `double`, usando

```
printf("%.0f\n",Max(5.0,8.0));
```

oppure, se gli argomenti sono variabili, una conversione di tipo, come in

```
int a=5, b=8;
printf("%.0f\n",Max((double)a,(double)b));
```

oppure, molto meglio e una volta per tutte, indicare il tipo degli argomenti nella dichiarazione:

```
double Max(double,double);
```

static e extern

Variabili e funzioni dello stesso nome in files diversi devono essere dichiarate di classe `static`, altrimenti il linker invierà il messaggio `multiple definition of ...`. È una buona prassi dare la classe `static` a tutte le variabili e funzioni che non vengono usate al di fuori dal file in cui sono definite.

Se una variabile dichiarata in un file deve essere invece usata anche in altri files, in questi ultimi (o nel file di inclusione, per noi sempre in *alfa.h*) deve essere dichiarata di classe `extern` in modo che già all'atto della compilazione il compilatore sappia di che tipo di dati si tratta.

Si può usare `extern` anche per le funzioni, ma non è necessario.

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 4 ◇ 11 Ottobre 2004

Vettori e puntatori

Un vettore è un indirizzo fisso insieme ad un tipo, un puntatore è un indirizzo variabile insieme ad un tipo.

Quando `a` *è un vettore o puntatore, con* `a[k]` *si denota il* `k`*-esimo elemento di quel tipo a partire da* `a`*. Questo è più precisamente l'elemento del tipo indicato il cui indirizzo si trova a una distanza di* `k` *per* `g` *byte alla destra di* `a`*, dove* `g` *è la grandezza (*`sizeof`*) di ogni singolo oggetto di quel tipo. Per esempio, abbiamo visto a pagina 9 che ogni oggetto di tipo* `double` *occupa 8 byte; quindi* `a[0]` *è l'oggetto di tipo* `double` *che si trova nell'indirizzo* `a`*,* `a[1]` *quello che si trova nell'indirizzo* `a+8`*,* `a[5]` *il numero di tipo* `double` *che si trova nell'indirizzo* `a+40`*.*

Cosa significa che un numero `double` *si trova in un certo indirizzo* `a`*? L'indirizzo è un numero anch'esso; assumiamo che sia 403005. Allora questo indirizzo corrisponde al 403005esimo byte, e gli 8 byte a partire da questo vengono interpretati come un numero di tipo* `double`*. Questo è il valore dell'elemento* `a[0]`*. Il valore di* `a+5` *si calcola similmente dal contenuto degli 8 byte negli indirizzi da 403045 a 403054.*

Quando dichiariamo una variabile scalare, ad esempio di tipo `double`*, ciò ha l'effetto che il compilatore cerca una spazio libero di 8 byte adiacenti in memoria che riserva per quella variabile. Nella dichiarazione di un vettore* `a` *dobbiamo anche riservare lo spazio per gli elementi con cui vogliamo lavorare. Questi si troveranno, come detto prima, nelle posizioni di memoria a partire da quell'indirizzo. Se vogliamo scrivere in quella parte della memoria (ad adesempio assegnando dei valori agli elementi), dobbiamo quindi anche indicare con quanti elementi vogliamo lavorare. Se chiediamo lo spazio per 7 elementi di tipo* `double`*, verranno riservati 56 byte a partire da* `a`*. La dichiarazione avviene allora in questa forma:*

```
double a[7];
```

Se `a` *viene usato in sola lettura (in questo caso ciò non ha però molto senso, se non vogliamo semplicemente esaminare il contenuto della memoria), basta anche la semplice dichiarazione dell'indirizzo con*

```
double a[];
```

Un'espressione di questa forma appare invece talvolta negli argomenti di una funzione, in cui `a` *è un vettore variabile che effettivamente, dal punto di vista di quella funzione, è utilizzato in sola lettura.*

Bisogna stare attenti che, dopo la dichiarazione `double a[7];`*, sono riservati gli spazi per i 7 elementi* `a[0]`*, ...,* `a[6]`*, ma non più per* `a[7]`*; se nel programma scriviamo in* `a[7]` *o in* `a[k]` *con un indice* `k` *che non sia compreso tra 0 e 6, quasi sicuramente provocheremo un grave errore di memoria.*

Puntatori vengono usati in due modi. Da un lato possono essere utilizzati come indirizzi variabili per muoversi all'interno della memoria, ad esempio durante l'esecuzione di un ciclo. Dall'altro lato vengono usati in modo simile ai vettori per destinare delle aree di memoria in cui conservare dei dati. In tal caso bisogna riservare quest'area di memoria con una delle istruzioni di allocazione di memoria che impareremo a pagina 13 oppure con l'assegnazione diretta di una stringa. Siccome i puntatori sono numeri, possiamo naturalmente anche lavorare con vettori di puntatori o con puntatori a puntatori; in una terminologia frequentemente usata un puntatore a puntatori si chiama un manico. Anche un vettore di stringhe è in C, come vedremo, nient'altro che un vettore di puntatori a caratteri.

Per distinguere puntatori da vettori useremo iniziali maiuscole per i puntatori e in genere minuscole per i vettori; è solo un'abitudine che proponiamo per evitare una fonte di errori, non è necessario.

In questo numero

&x e *A

Un puntatore è quindi una *variabile* che viene utilizzata per indicare un *indirizzo tipizzato* in memoria. La dichiarazione di un puntatore ha questo formato:

```
tipo *A;
```

ad esempio

```
double *A;
```

per dichiarare un puntatore a numeri del tipo `double`. Il puntatore si chiama `A`, non `*A`.

L'asterisco si spiega in questo modo: Se `A` è un puntatore, allora `*A` è il contenuto a cui punta `A`, cioè l'oggetto del tipo indicato nella dichiarazione contenuto nell'indirizzo `A`. Se `x` è invece una variabile, `&x` denota il puntatore ad `x`, che non è altro che l'indirizzo in cui `x` si trova in memoria (insieme al tipo di `x`). Aver perfettamente chiaro il significato di questi simboli è molto importante nella programmazione in C, perché vengono utilizzati continuamente. Riepilogando:

```
*A ... contenuto a cui punta A
&x ... puntatore ad x
```

Dopo

```
int *A;
```

i puntatori `&*A` e `A` coincidono quindi, così come dopo

```
int a;
```

coincidono gli oggetti `a` e `*&a`. Le istruzioni `a=7;` e `*&a=7;` hanno quindi lo stesso effetto.

I puntatori sono variabili come le altre, e quindi possono essere modificati. Ad esempio con

```
int a[]={1,3,8,7}, *X;
X=a+2;
```

si definiscono un vettore `a` di interi (si osservi il formato di inizializzazione) e un puntatore `X` a interi che successivamente punta ad `a[2]`; perciò in questo caso `*X` è uguale a 8.

Non sono invece permesse le istruzioni `a=a+2;` oppure `a=X;` perché `a` è un indirizzo fisso.

Molte operazioni sono uguali per puntatori e vettori. Ad esempio dopo

```
int *X,a[10]; X=a;
```

le istruzioni `a[4]=7;` e `X[4]=7;` sono equivalenti.

Vettori a più indici

Vettori a due indici hanno la dichiarazione

```
tipo v[n][m];
```

`v[i][j]` è l'elemento `(i,j)`-esimo del vettore; naturalmente anche qui si inizia a contare da 0. In verità un vettore a più indici è un vettore come un altro e si distingue da un vettore a un indice solo nella sintassi d'accesso. Infatti nel nostro esempio `v[0]`,..., `v[n-1]` sono a loro volta vettori che vengono collocati attigui in memoria formando un unico lungo vettore, come si vede dal seguente esempio:

```
void Prova ()
{double v[2][3]={{0,1,2},{3,4,5}},*A;
int k;
for (k=0,A=(double*)v;k<6;k++,A++)
  {printf("%.0f ",*A);}
printf("\n");}
```

con output

```
0 1 2 3 4 5.
```

L'unica cosa che abbiamo dovuto fare per eludere un'ammonimento da parte del compilatore era la conversione di tipo con

```
A=(double*)v;
```

Dopo di ciò all'inizio del ciclo `A` e `v` corrispondono allo stesso indirizzo e si distinguono solo nel tipo: `A` è un puntatore a numeri `double`, quindi un oggetto del tipo `double*`, mentre `v` è un puntatore a puntatori a numeri `double`, cioè un oggetto del tipo `double**`.

Aritmetica dei puntatori

Puntatori vengono spesso usati come variabili in cicli, per esempio

```
for (X=a;X-a<4;X++) printf("%d ",*X);
```

Se `X` è un puntatore (o vettore) di tipo `t` ed `n` è un'espressione di tipo intero (o intero lungo), allora `X+n` è il puntatore dello stesso tipo `t` e indirizzo uguale a quello di `X` aumentato di `n*d`, dove `d` è lo spazio che occupa un elemento del tipo `t`; quindi se immaginiamo la parte di memoria che inizia nell'indirizzo corrispondente a `X` occupata da elementi di tipo `t`, `X+n` punta all'elemento con indice `n` (cominciando a contare da zero). In altre parole, `*(X+n)` è lo stesso elemento come `X[n]`.

L'essenziale equivalenza tra puntatori e vettori implica che dopo

```
t a[100],*A; int k;
A=a;
```

le espressioni

```
a[k], A[k], *(A+k), *(a+k)
```

indichino lo stesso elemento. La differenza tra vettori e puntatori consiste quindi solo nel fatto che il puntatore è variabile, mentre il vettore corrisponde a una posizione fissa in memoria con un determinato spazio riservato in memoria. Una curiosità è che anche `k+A` è permesso ed equivalente a `A+k`, e che l'espressione `u[v]`, dove tra `u` e `v` uno è un puntatore, l'altro un intero, viene trasformata nell'espressione simmetrica `*(u+v)`, e quindi anche `2[A]` è corretto ed equivalente a `A[2]`!

Confronto di puntatori

Puntatori possono essere confrontati tra di loro; hanno senso quindi le espressioni `X<Y` oppure `X==Y` per due puntatori `X` e `Y`; si possono usare le istruzioni `X++` e `X--` come per variabili intere.

Non raramente (come anche nel nostro esempio) si usa anche la differenza `X-Y` tra due puntatori dello stesso tipo `t`, che per `X>=Y` non è altro che il numero degli elementi di tipo `t` che si trovano tra `Y` ed `X`.

In altre parole, `X-Y==n` se e solo se `X==Y+n`.

Vettori come argomenti

Quando dichiariamo un vettore di tipo `t` nella forma

```
t v[n];
```

ciò implica soltanto che a partire da un certo indirizzo in memoria viene riservato uno spazio di complessivamente `n*sizeof(t)` byte. A differenza da altri linguaggi il vettore non contiene però l'informazione sulla proprio lunghezza (cioè su `n`).

Se ad esempio usiamo un vettore (o puntatore) come argomento di una funzione, dobbiamo indicare la lunghezza o esplicitamente come argomento addizionale oppure utilizzare un formato speciale per i dati contenuti nel vettore da cui è possibile riconoscere la fine del vettore. Se si tratta di un vettore di numeri di cui sappiamo che sono tutti ≥ 0, possiamo aggiungere un numero negativo come terminatore; per un vettore di stringhe possiamo usare la stringa vuota, se questa non viene mai usata in altro modo.

È proprio così che le funzioni per le stringhe del C riconoscono la fine di una stringa, cioè di un vettore di caratteri, dal primo carattere con codice ASCII uguale a zero.

Creiamo adesso una funzione che calcola la somma di una successione finita di numeri reali.

```
double Somma (double *A, int n)
{double s; int k;
for (s=k=0;k<n;k++) s+=A[k];
return s;}
```

In questo caso dobbiamo indicare esplicitamente nel secondo argomento il numero dei componenti del vettore che vogliamo utilizzare:

```
double a[]={1,2,3,4,8,8};
printf("%.2f\n",Somma(a,6));
```

Ciò in alcune situazioni può risultare scomodo, anche se in altre può essere accettabile o addirittura pratico. Se lo vogliamo evitare, possiamo provare a introdurre qualche convenzione sulla rappresentazione dei dati da sfruttare per riconoscere la fine di un vettore.

Assumiamo che sappiamo che i nostri numeri non sono mai negativi. Allora possiamo aggiungere un `-1` alla fine del vettore e modificare la funzione nella maniera seguente:

```
double Somma (double *A)
{double s,v; int k;
for (s=k=0;(v=A[k])>=0;k++) s+=v;
return s;}
```

Adesso la funzione può essere utilizzata così:

```
double a[]={1,2,3,4,8,8,-1};
printf("%.2f\n",Somma(a));
```

Nella condizione del `for` abbiamo dovuto mettere tra parentesi `v=A[k]`, perché altrimenti il compilatore esegue `v=(A[k]>=0)`, assegnando a `v` il valore 1 ogni volta che `A[k]>=0`.

Naturalmente anche la funzione

```
double Somma (double *A)
{double s; int k;
for (s=k=0;A[k]>=0;k++) s+=A[k];
return s;}
```

è sintatticamente corretta, richiede però che in ogni passaggio si acceda due volte al valore `A[k]`. Questa operazione, cioè l'accesso al valore `k`-esimo di un vettore, è piuttosto lenta e può, per vettori con molti elementi, comportare un sensibile rallentamento. Per questo, nella versione precedente abbiamo introdotto la variabile `v` per raccogliere il valore di `A[k]` una volta trovato.

Anche in questa forma però la funzione non è ancora programmata in modo ottimale. Infatti gli accessi ad `A[k]` avvengono indipendenti l'uno dall'altro; fisicamente ogni volta il vettore viene percorso partendo dal suo inizio fino all'indice desiderato, non si passa cioè direttamente da `A[k]` a `A[k+1]`, ma si torna prima all'inizio del vettore.

La funzione diventa più veloce, se utilizziamo un puntatore che percorre una volta sola il vettore, aggiungendo, mentre procede, i valori che trova ad `s`:

```
double Somma (double *A)
{double s,v;
for (s=0;(v=*A)>=0;A++) s+=v;
return s;}
```

Può essere sorprendente che abbiamo potuto usare il puntatore argomento `A` stesso per questo compito, soprattutto se pensiamo che nell'esecuzione di

```
double a[]={1,2,3,4,8,8,-1};
printf("%.2f\n",Somma(a));
```

dove come argomento abbiamo passato l'indirizzo fisso `a`, sembrerebbe che abbiamo modificato il valore di `a` spostandolo di `6*sizeof(double)`, cioè 80, byte verso destra. Come impareremo nel prossimo numero quando discuteremo il *passaggio di parametri*, in C gli argomenti di una funzione non vengono mai modificati, cioè quello che si muove nel `for` dell'ultima versione della funzione `Somma` non è l'argomento originale `a`, ma una copia di questo puntatore.

Se un vettore ha più indici, la dimensione che corrisponde al primo indice e solo questa può essere lasciata indeterminata:

```
tipo v[][n1][n2][n3];
```

Puntatori generici

Talvolta il programmatore avrebbe bisogno di strutture e operazioni che funzionino con elementi di tipo qualsiasi. Allora si possono usare *puntatori generici*, che formalmente vengono dichiarati come oggetti del tipo `void*`. Un esempio:

```
void Applicafunzione (void f(), void *X)
{f(X);}

void Scriviquadrato (int *X)
{int n=*X; printf("%d\n",n*n);}

void Prova ()
{int a=8;
Applicafunzione(Scriviquadrato,&a);}
```

L'output è correttamente `64`. Si osservi il modo in cui una funzione viene dichiarata come argomento di un'altra funzione. Un puntatore `void*` può essere considerato come indirizzo puro.

Conversioni di tipo

Puntatori di tipo diverso possono essere convertiti tra di loro. Se `X` è un puntatore a oggetti del tipo `t`, e se `s` è un altro tipo, allora `(s*)X` è il puntatore con lo stesso indirizzo di `X`, punta però a oggetti del tipo `s`.

Ad esempio `X+2` punta all'elemento di tipo `t` con indice 2 a partire dall'indirizzo corrispondente ad `X`, ma `(char*)X` punta al terzo *byte* (perché si comincia a contare da 0) a partire da quell'indirizzo. Qual'è invece il significato di `(char*)(X+2)`?

Conversioni di tipo fra puntatori sono frequenti soprattutto quando si utilizzano puntatori generici (indirizzi puri); abbiamo visto un esempio nell'articolo precedente. Ad esempio dopo

```
void *A; int *B;
```

con

```
B=(int*)A;
```

il puntatore `B` punta all'intero contenuto nei primi 4 byte a partire dall'indirizzo `A`.

In alcuni casi sono possibili anche conversioni di tipo fra variabili normali, ad esempio `(int)x` dopo la dichiarazione `double x;`, devono essere usate però con molta cautela o forse meglio evitate del tutto.

Verifichiamo il diverso significato di `A+k` per un puntatore, a seconda del tipo a cui punta `A`.

```
void Provapuntatori ()
{char *A="alfa"; int *B; double *C;
int k;
for (k=0;k<3;k++)
printf("%ld %ld %ld\n",A+k,B+k,C+k);}
```

Se eseguiamo questa funzione otteniamo un output della forma

```
134518240 1073897472 1073816760
134518241 1073897476 1073816768
134518242 1073897480 1073816776
```

e vediamo che `A` procede a passi di 1 byte, `B` a passi di 4 byte, `C` a passi di 8 byte, in accordo con quanto abbiamo verificato a pagina 9 quando abbiamo provato `sizeof`.

typedef

Con `typedef` si possono assegnare nomi nuovi a tipi già esistenti, ad esempio

```
typedef size_t tindirizzo;
```

Dopo di ciò invece di

```
size_t n;
```

possiamo scrivere

```
tindirizzo n;
```

`typedef` viene spesso usato per abbreviare nomi di tipi complicati, come faremo più tardi per le strutture e già da ora in avanti per semplificare i tipi di puntatori, dopo aver inserito le seguenti dichiarazioni in *alfa.h*:

```
typedef void *Void;
typedef char *Char;
typedef int *Int;
typedef uint *Uint;
// uint e ulong sono dichiarati
// in <sys/types.h> come
// unsigned int e unsigned long.
typedef ulong *Ulong;
typedef double *Double;
typedef void (*Funzione)();
typedef int (*Funzioneintera)();
typedef double (*Funzionedouble)();
```

Operazion sui byte in memoria

Le seguenti funzioni sono molto simili alle funzioni per le stringhe che vedremo più avanti e si distinguono da esse per il fatto che il carattere `0` non ha più un significato speciale; sono dichiarate in `<string.h>`.

```
void *memchr (const void *A, int x,
  size_t n);
```

`memchr(A,x,n)` restituisce un puntatore al primo `x` tra i primi `n` caratteri a partire da `A` oppure il puntatore nullo, se tra questi caratteri nessuno è uguale ad `x`.

```
void *memset (void *A, int x,
  size_t n);
```

L'istruzione `memset(A,x,n);` pone i primi `n` caratteri a partire da `A` uguali ad `x` (questo argomento dovrebbe essere del tipo `unsigned char`, anche se nella dichiarazione appare come `int`). Ad esempio `memset(A,0,50);` pone 50 byte a partire da `A` uguali a `0`. Il risultato della funzione è uguale ad `A`, ma normalmente superfluo. Lo spazio necessario deve essere stato riservato in precedenza.

```
void *memcpy (void *A, const void *B,
  size_t n);
void *memmove (void *A, const void *B,
  size_t n);
```

`memcpy(A,B,n);` e `memmove(A,B,n);` copiano entrambe `n` byte da `B` in `A`, ma mentre `memcpy` non può essere utilizzata quando le due regioni di memoria si sovrappongono, `memmove` funziona anche in questo caso. Quindi l'istruzione `memmove(A,B,strlen(B)+1);` può essere usata per copiare la stringa `B` in `A` (compreso il carattere 0 finale) anche nel caso di sovrapposizione i memoria.

Allocazione di memoria

Per riservare o liberare parti di memoria in C si usano quattro funzioni i cui prototipi sono:

```
void *malloc (size_t n);
void *calloc (size_t n, size_t dim);
void *realloc (void *A, size_t n);
void free (void *A);
```

Tutte queste funzioni richiedono il header `<stdlib.h>`.

Vengono usate nel modo seguente:

```
A=malloc(n);
```

Questa istruzione chiede al sistema di cercare in memoria uno spazio di `n` byte attigui, all'inizio del quale punterà `A`; se ciò non è possibile, `A` viene posto uguale al puntatore nullo `NULL`.

Per controllare il buon esito dell'operazione, spesso l'istruzione sarà seguita da

```
if (!A) eccezione; ...
```

La funzione `calloc` è un caso particolare di `malloc`, infatti

```
A=calloc(n,dim);
```

è equivalente a

```
A=malloc(n*dim);
```

e riserva quindi lo spazio per `n` oggetti di `dim` byte ciascuno. Se necessario, la dimensione può essere calcolata mediante `sizeof`, ad esempio

```
A=malloc(100*sizeof(vettore));
```

La funzione `realloc` viene usata per modificare lo spazio precedentemente riservato con `malloc`, `calloc` o un altro `realloc`. Le regole più importanti sono:

(1) L'istruzione `A=realloc(NULL,n);` è equivalente ad `A=malloc(n);`

(2) L'istruzione

```
A=realloc(A,0);
```

con `A` diverso da `NULL` equivale essenzialmente a

```
free(A); A=NULL;
```

(3) L'istruzione

```
A=realloc(A,n);
```

con `n` $\leq$ dello spazio già riservato per `A` libera lo spazio non più richiesto e non modifica l'indirizzo a cui punta `A`. Altrimenti viene riservato più spazio a partire da `A`, se ciò è possibile, oppure, in caso contrario, questo spazio viene cercato in un'altra parte della memoria.

Infine l'istruzione

```
free(A);
```

fa in modo che lo spazio riservato in precedenza per `A` venga liberato.

Attenzione: Se lo spazio per `A` non è riservato (ad esempio a causa di una chiamata in troppo di `free` o perché non è mai stato allocato), ciò provoca quasi sicuramente un (molto brutto) errore in memoria (`segmentation fault` sotto Linux), tranne nel caso che `A` sia il puntatore `NULL`.

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 5 ◇ 18 Ottobre 2004

Passaggio di parametri

Consideriamo la seguente funzione che dovrebbe aumentare di 1 il valore di un numero intero:

```
void Aumenta (int x) {x++;}
```

Facciamo una prova:

```
void Prova ()
{int x=5; Aumenta(x);
printf("%d\n",x);}
```

Viene stampato 5, quindi non funziona. Qual'è la ragione?

I parametri (argomenti) di una funzione in C vengono sempre passati per valore. Con ciò si intende che in una chiamata `f(x)` *alla funzione* `f` *viene passato solo il valore di* `x`*, con cui la funzione esegue le operazioni richieste, ma senza che il valore della variabile* `x` *venga modificato, anche nel caso che all'interno della funzione ci sia un'istruzione del tipo* `x=nuovovalore;`*. Infatti la variabile* `x` *che appare all'interno della funzione è un'altra variabile, che riceve come valore iniziale il valore della* `x`*.*

Per aumentare `x` *dobbiamo perciò accedere al suo indirizzo e modificare direttamente il valore contenuto in quell'indirizzo. La funzione corretta è quindi*

```
void Aumenta (Int A) {(*A)++;}
```

Ricordiamo che abbiamo introdotto `Int` *come abbreviazione di* `int*`*. Quando si applica la funzione, bisogna usare* `&x`*:*

```
void Prova ()
{int x=5; Aumenta(&x);
printf("%d\n",x);}
```

In `(*A)++` *le parentesi sono necessarie; infatti* `*A++` *aumenterebbe l'indirizzo* `A` *(secondo le regole dell'aritmetica dei puntatori) senza alcun altro effetto.*

Per la stessa ragione è corretta la funzione

```
void Stampainteri (Int A)
{for (;*A>=0;A++) printf("%d ",*A);
printf("\n");}
```

che, a partire dall'indirizzo `A` *stampa gli interi che seguono a quell'indirizzo fino a quando incontra un intero negativo.*

Quando la funzione viene utilizzata, non è il puntatore `A` *che si muove, ma una copia locale creata per la funzione. Quindi dopo l'esecuzione della funzione* `A` *punta ancora all'inizio del vettore e non al primo intero negativo incontrato. Perciò*

```
void Prova ()
{int a[]={3,5,8,0,4,-1};
Stampainteri(a);
Stampainteri(a);}
```

stampa correttamente due volte la stessa serie di numeri:

```
3 5 8 0 4
3 5 8 0 4
```

Abbiamo incontrato questa caratteristica tipica del C già nell'ultima versione della funzione `Somma` *a pagina 12.*

Numeri complessi

Definiamo un tipo `nc` per rappresentare i numeri complessi nel modo seguente:

```
typedef struct {double x,y;} nc;
```

Dopo aver inserito questa definizione in *alfa.h* possiamo creare funzioni per addizione, sottrazione e moltiplicazione di numeri complessi e per la formazione del complesso coniugato. Si noti la brevità di quest'ultima funzione che sfrutta il passaggio per valore dei parametri in C.

Il valore assoluto e il quoziente di numeri complessi richiedono qualche accorgimento per evitare risultati intermedi troppo grandi e verranno discussi nel prossimo numero.

```
nc Nccon (nc z)
{z.y=-z.y; return z;}

nc Ncadd (nc z1, nc z2)
{nc w;
w.x=z1.x+z2.x; w.y=z1.y+z2.y;
return w;}

nc Ncmolt (nc z1, nc z2)
{nc w;
w.x=z1.x*z2.x-z1.y*z2.y;
w.y=z1.x*z2.y+z2.x*z1.y;
return w;}

nc Ncsott (nc z1, nc z2)
{nc w;
w.x=z1.x-z2.x; w.y=z1.y-z2.y;
return w;}
```

In questo numero

Strutture

Il modo più conveniente per definire strutture composte in C è questo:

```
typedef struct {double x,y,z;} vettore;
```

Adesso, dopo la dichiarazione

```
vettore v;
```

le componenti di `v` sono `v.x`, `v.y` e `v.z`. Definiamo una funzione per l'addizione di due vettori:

```
vettore somma (vettore v, vettore w)
{vettore s;
s.x=v.x+w.x; s.y=v.y+w.y; s.z=v.z+w.z;
return s;}
```

Le componenti di una struttura vengono allocate una vicino all'altra in memoria. Ciò vale anche quando sono di tipo misto e permette la stessa tecnica di inizializzazione mediante parentesi graffe per le strutture come per i vettori:

```
typedef struct {double x,y;
  Char Nome; int m;} fantasia;

void Prova ()
{fantasia fan={7.8,10,"Roberto",6};
printf("%s %d %.2f %.2f\n",
  fan.Nome,fan.m,fan.x,fan.y);}
```

con output

```
Roberto 6 7.80 10.00
```

Nella componente `fan.Nome` non vengono allocati i caratteri della stringa `"Roberto"`, ma solo i quattro byte necessari per rappresentare il puntatore che corrisponde alla stringa. Il valore di questo puntatore è l'indirizzo in cui si trovano realmente i caratteri della stringa.

Se `A` è un puntatore a una struttura, per la compente `x` di `*A`, cioè per `(*A).x`, si può anche scrivere `A->x`. È un'abbreviazione usata molto spesso. Non sarebbe corretto `*A.x` che viene interpretato come `*(A.x)`.

Proviamo la correttezza di `Nccon`:

```
void Prova ()
{nc z={5,7},w; w=Nccon(z);
printf("%.1f %.1f %.1f %.1f\n",
  z.x,z.y,w.x,w.y);}
// output: 5.0 7.0 5.0 -7.0
```

Variabili di classe static

Nella dichiarazione di una variabile l'indicazione del tipo può essere preceduta dalla *classe di memoria*. Usiamo solo due di queste classi: `static` ed `extern`.

Abbiamo già discusso, a pagina 10, l'impiego di queste dichiarazioni. Per funzioni e per variabili dichiarate al di fuori di funzioni l'indicazione `static` ha semplicemente l'effetto di rendere la variabile o la funzione invisibile al linker.

Naturalmente anche variabili dichiarate internamente a una funzione non sono visibili esternamente al file (nemmeno all'esterno della funzione) e non è necessario dichiararle di classe `static`.

Per queste variabili interne a una funzione l'indicazione della classe `static` ha invece una conseguenza peculiare che talvolta è utile, ma che può implicare un comportamente della funzione misterioso, se non si conosce la regola.

Infatti mentre normalmente, se una variabile è interna a una funzione, in ogni esecuzione della funzione il sistema cerca di nuovo uno spazio in memoria per questa variabile, alle variabili di classe `static` viene assegnato uno spazio in memoria fisso, che rimane sempre lo stesso in tutte le chiamate della funzione (ciò evidentemente ha il vantaggio di impegnare la memoria molto meno); i valori di queste variabili si conservano da una esecuzione all'altra. Inoltre una eventuale inizializzazione per una variabile `static` viene eseguita solo nella prima chiamata (è soprattutto questo che puo confondere). Esempio:

```
int Sommastatic1 ()
{static int s=0,k;
for (k=0;k<5;k++) s+=k; return s;}

int Sommastatic2 ()
{static int s,k;
for (s=k=0;k<5;k++) s+=k; return s;}

void Provastatic ()
{int i;
for (i=0;i<3;i++)
printf("%d ",Sommastatic1());
printf("\n");
// output: 10 20 30
for (i=0;i<3;i++)
printf("%d ",Sommastatic2());
printf("\n");}
// output: 10 10 10
```

Si vede che `Sommastatic1` restituisce ogni volta un risultato diverso, perché l'inizializzazione `s=0` viene effettuata solo la prima volta, mentre successivamente si parte con il valore che `s` aveva alla fine dell'ultima esecuzione. Probabilmente ciò non è quello che qui il programmatore, che forse intendeva soltanto risparmiare memoria utilizzando una variabile `static`, desiderava fare. È facile rimediare, senza rinunciare a `static`, perché l'istruzione `s=0`, se data al di fuori della dichiarazione, viene eseguita normalmente ogni volta, come in `Sommastatic2`.

Esistono comunque anche situazioni in cui può essere utile che una variabile interna di una funzione conservi il suo valore da una esecuzione all'altra (un esempio è la funzione `strtok`).

Infatti un modo appropriato e molto utile dell'uso di variabili interne di classe `static` è di utilizzarle al posto di variabili globali. Se una funzione `f` ha bisogno di ricordare i valori di una variabile tra le sue successive esecuzioni, invece di creare una variabile globale a cui la funzione accede, è di gran lunga preferibile dichiarare una variabile di classe `static` all'interno di `f`.

Soprattutto vettori in funzioni che devono essere eseguite molte volte, dovrebbere essere dichiarate di classe `static`. Infatti, altrimenti la funzione in ogni esecuzione deve di nuove cercare in memoria lo spazio richiesto e ciò costa tempo.

La seguente funzione calcola la lunghezza delle stringa che corrisponde al numero reale `x` quando viene stampato con il formato indicato.

```
int Lunreale (double x, Char Formato)
{static char a[40];
sprintf(a,Formato,x);
return strlen(a);}
```

Si applica così:

```
void Prova ()
{double x=35.6891;
printf("%d\n",Lunreale(x,"%.2f"));}
```

con output `5`, perché a quel formato corrisponde la stringa `35.69`. Se in una matrice 1000×1000 la funzione viene applicata un milione di volte, potrebbe essere avvertibile la differenza.

Rappresentazione binaria

Ogni numero naturale $n > 0$ possiede una rappresentazione binaria, cioè una rappresentazione della forma

$$n = a_k 2^k + a_{k-1} 2^{k-1} + \cdots + a_2 2^2 + a_1 2 + a_0$$

con coefficienti (o cifre binarie) $a_i \in \{0, 1\}$ e $a_k = 1$ univocamente determinati. Sia $r_2(n) = (a_k, \ldots, a_0)$ la lista (o elemento di 2^*) i cui elementi sono queste cifre. Dalla rappresentazione binaria si deduce la seguente relazione ricorsiva:

$$r_2(n) = \begin{cases} (1) & \text{se } n = 1 \\ (r_2(\frac{n}{2}), 0) & \text{se } n \text{ è pari} \\ (r_2(\frac{n-1}{2}), 1) & \text{se } n \text{ è dispari} \end{cases}$$

Scriviamo una funzione che pone la stringa che contiene la rappresentazione binaria di `n` nel suo secondo argomento `A`:

```
void Rapp2 (int n, Char A)
{if (n==1) {A[0]='1'; A[1]=0;}
else if (n%2) {Rapp2((n-1)/2,A);
A=strchr(A,0); A[0]='1'; A[1]=0;}
else {Rapp2(n/2,A);
A=strchr(A,0); A[0]='0'; A[1]=0;}}
```

La proviamo con

```
void Prova ()
{char a[100]; int n;
for (n=1;n<=20;n++)
{Rapp2(n,a); printf("%s\n",a);}}
```

Un articolo a pagina 16 spiega `strchr(A,0)`. La funzione `strchr` è molto utile.

Lo schema di Horner

Sia dato un polinomio

$$f = a_0 x^n + a_1 x^{n-1} + \cdots + a_n \in A[x]$$

dove A è un qualsiasi anello commutativo.

Per $\alpha \in A$ vogliamo calcolare $f(\alpha)$.
Sia ad esempio $f = 3x^4 + 5x^3 + 6x^2 + 8x + 17$.
Poniamo

$$\begin{aligned} b_0 &= 3 \\ b_1 &= b_0\alpha + 5 &&= 3\alpha + 5 \\ b_2 &= b_1\alpha + 6 &&= 3\alpha^2 + 5\alpha + 6 \\ b_3 &= b_2\alpha + 8 &&= 3\alpha^3 + 5\alpha^2 + 6\alpha + 8 \\ b_4 &= b_3\alpha + 17 &&= 3\alpha^4 + 5\alpha^3 + 6\alpha^2 + 8\alpha + 17 \end{aligned}$$

e vediamo che $b_4 = f(\alpha)$. Lo stesso si può fare nel caso generale:

$$\begin{aligned} b_0 &= a_0 \\ b_1 &= b_0\alpha + a_1 \\ &\ldots \\ b_k &= b_{k-1}\alpha + a_k \\ &\ldots \\ b_n &= b_{n-1}\alpha + a_n \end{aligned}$$

con $b_n = f(\alpha)$, come dimostriamo adesso. Consideriamo il polinomio

$$g := b_0 x^{n-1} + b_1 x^{n-2} + \cdots + b_{n-1}.$$

Allora, usando che $\alpha b_k = b_{k+1} - a_{k+1}$ per $k = 0, \ldots, n-1$, abbiamo

$$\begin{aligned} \alpha g &= \alpha b_0 x^{n-1} + \alpha b_1 x^{n-2} + \cdots + \alpha b_{n-1} \\ &= (b_1 - a_1)x^{n-1} + (b_2 - a_2)x^{n-2} + \ldots \\ &\quad + (b_{n-1} - a_{n-1})x + b_n - a_n \\ &= (b_1 x^{n-1} + b_2 x^{n-2} + \cdots + b_{n-1}x + b_n) \\ &\quad - (a_1 x^{n-1} + a_2 x^{n-2} + \ldots a_{n-1}x + a_n) \\ &= x(g - b_0 x^{n-1}) + b_n - (f - a_0 x^n) \\ &= xg - b_0 x^n + b_n - f + a_0 x^n = xg + b_n - f \end{aligned}$$

quindi

$$\boxed{f = (x - \alpha)g + b_n}$$

e ciò implica $f(\alpha) = b_n$.

$b_0, \ldots, b_{n-1}$ sono perciò i coefficienti del quoziente nella divisione con resto di f per $x - \alpha$, mentre b_n è il resto, uguale a $f(\alpha)$.

Questo algoritmo si chiama *schema di Horner* o *schema di Ruffini* ed è molto più veloce del calcolo separato delle potenze di α (tranne nel caso che il polinomio consista di una sola o di pochissime potenze, in cui si userà invece l'algoritmo a pagina 16). Quando serve solo il valore $f(\alpha)$, in un programma in C si può usare la stessa variabile per tutti i b_k:

```
double Horner (double alfa, Double A,
  int n)
{double b; int k;
for (b=*(A++),k=1;k<=n;k++,A++)
b=b*alfa+*A;
return b;}
```

Si osservi bene l'istruzione `b=*(A++)` in cui abbiamo prima posto `b` uguale ad `*A` e poi spostato `A` di una posizione verso destra.

Esempi per lo schema di Horner

Utilizziamo la funzione `Horner` prima per calcolare $f(3)$ dove

$$f = 3x^4 + 5x^3 + 6x^2 + 8x + 17$$

come a pagina 15.

```
void Prova ()
{double a[]={3,5,6,8,17};
printf("%.2f\n",Horner(3,a,4));}
```

L'output, `473.00`, è corretto. Si osservi che il terzo argomento, in questo caso 4, è il grado del polinomio.

Una frequente applicazione dello schema di Horner è il calcolo del valore corrispondente a una rappresentazione binaria o esadecimale.

Infatti otteniamo $(1,0,0,1,1,0,1,1,1)_2$ come

```
Horner(2,1,0,0,1,1,0,1,1,1)
```

e $(A,F,7,3,0,5,E)_{16}$ come

```
Horner(16,10,15,7,3,0,5,14).
```

Lo schema di Horner ricorsivo

Lo schema di Horner per il calcolo del valore $f(\alpha)$ di un polinomio $f = a_0x^n + a_1x^{n-1} + \cdots + a_n$ permette una elegante versione ricorsiva. Infatti, se denotiamo quel valore con $\mathrm{val}(\alpha, a_0, a_1, \ldots, a_n)$, allora abbiamo la relazione

$$\begin{aligned}&\mathrm{val}(\alpha, a_0, a_1, \ldots, a_n)\\ &\quad = \alpha \cdot \mathrm{val}(\alpha, a_0, a_1, \ldots, a_{n-1}) + a_n\end{aligned}$$

come si vede da

$$\begin{aligned}&a_0\alpha^n + a_1\alpha^{n-1} + \cdots + a_{n-1}\alpha + a_n\\ &= \alpha\cdot(a_0\alpha^{n-1} + a_1\alpha^{n-2} + \cdots + a_{n-1}) + a_n\end{aligned}$$

con la condizione iniziale $\mathrm{val}(\alpha) = 0$. Ciò può essere tradotto in un programma in C:

```
double H (double alfa, Double A, int n)
{if (n<0) return 0;
return alfa*H(alfa,A,n-1)+A[n];}
```

Questa funzione è sicuramente meno efficiente di quella precedente, non solo perché viene invocata $n+1$ volte invece di una volta sola, ma anche perché in ogni passaggio della ricorsione essa deve cercare il valore del vettore `A` per un certo indice. Questo secondo inconveniente lo potremmo eliminare programmando la funzione in modo tale che riceve i coefficienti del polinomio in ordine inverso, ciò iniziando con il termine costante. Scriviamo quindi il polinomio nella forma

$$f = c_0 + c_1x + \cdots + c_nx^n$$

cosicché

$$f(\alpha) = c_0 + \alpha\cdot(c_1 + c_2\alpha + \cdots + c_n\alpha^{n-1})$$

Il programma diventa

```
// C[0] e' il coefficiente costante.
double Hi (double alfa, Double C, int n)
{if (n<0) return 0;
return alfa*Hi(alfa,C+1,n-1)+*C;}
```

Quando si usa questa funzione, bisogna invertire l'ordine dei coefficienti:

```
void Prova ()
{double c[]={17,8,6,5,3};
printf("%.2f\n",Hi(3,c,4));}
```

Anche `Hi` è più lenta di `Horner`. È tipico per le funzioni ricorsive che in genere sono più eleganti e intuitive nella programmazione, ma più lente nell'esecuzione.

Calcolo di potenze

Per il calcolo di potenze (ad esponenti naturali) lo schema di Horner non comporta vantaggi rispetto all'algoritmo elementare mediante un ciclo, come in

```
double Potenzalenta (double x, int n)
{double p; int k;
for (p=k=1;k<=n;k++) p*=x; return p;}
```

Esiste però un algoritmo molto veloce che formuliamo in maniera ricorsiva per la funzione f definita da $f(x,n) = x^n$:

$$f(x,n) = \begin{cases} 1 & \text{se } n = 0\\ f(x^2, \frac{n}{2}) & \text{se } n \text{ è pari}\\ xf(x,n-1) & \text{se } n \text{ è dispari}\end{cases}$$

È facile tradurre questo algoritmo in un programma ricorsivo in C:

```
double Potenza (double x, int n)
{if (!n) return 1;
if (n%2) return x*Potenza(x,n-1);
return Potenza(x*x,n/2);}
```

La formula di ricorsione vale in ogni anello commutativo con unità; in particolare vale nell'aritmetica modulo m per ogni $m \in \mathbb{N}+1$. Mentre potenze con esponenti molto alti di numeri reali difficilmente sono utilizzabili nella pratica, si usano molto spesso potenze alte nell'aritmetica modulare; è facile adattare la nostra funzione a questo scopo:

```
// Calcola x alla n modulo m.
int Potenzamodulare (int x, int n, int m)
{if (!n) return 1;
if (n%2) return
  (x*Potenzamodulare(x,n-1,m))%m;
return Potenzamodulare((x*x)%m,n/2,m);}
```

All'interno della funzione i risultati intermedi (anche i quadrati) vengono subito ridotti modulo m e quindi possiamo calcolare facilmente anche potenze con esponenti nell'ordine di molte migliaia. Per provare usiamo

```
void Prova ()
{int k; for (k=0;k<4;k++)
printf("%d ",Potenzamodulare(10,k,7));
printf("\n");}
```

L'output è `1 3 2 6`, infatti

$$\begin{aligned}1 &= 0\cdot 7 + 1\\ 10 &= 1\cdot 7 + 3\\ 100 &= 14\cdot 7 + 2\\ 1000 &= 142\cdot 7 + 6\end{aligned}$$

I numeri binomiali

La formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

valida per $0 \le k \le n$ è molto importante nella teoria ma può essere utilizzata solo per numeri molto piccoli.

Infatti, se volessimo così calcolare $\binom{200}{4}$, dovremmo calcolare `200!` e `196!`, due numeri giganteschi, e formare $\frac{200!}{4!196!}$.

In pratica si usa invece la regola

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{1\cdot 2\cdots k}$$

dove sopra stanno tanti numeri quanti sotto, cioe k.

Ad esempio

$$\begin{aligned}\binom{12}{3} &= \frac{12\cdot 11\cdot 10}{1\cdot 2\cdot 3}\\ \binom{20}{5} &= \frac{20\cdot 19\cdot 18\cdot 17\cdot 16}{1\cdot 2\cdot 3\cdot 4\cdot 5}\\ \binom{200}{4} &= \frac{200\cdot 199\cdot 198\cdot 197}{1\cdot 2\cdot 3\cdot 4}\end{aligned}$$

Possiamo scrivere una funzione in C che calcola i numeri binomiali utilizzando questa regola:

```
double Bin (int n, int k)
{double num,den,i,j;
for (num=den=j=1,i=n;j<=k;i--,j++)
{num*=i; den*=j;}
return num/den;}
```

Prodotto scalare

Il prodotto scalare di due vettori `x` e `y` può essere ottenuto con somme di prodotti `x[k]*y[k]`. La ricerca del valore `x[k]` per ogni indice `k` consuma però tempo, come abbiamo già più volte osservato; un'esecuzione più veloce si ottiene con l'impiego di puntatori. Nella funzione che segue l'argomento `m` è la lunghezza dei vettori.

```
double Prodottoscalare (Double X,
  int m, Double Y)
{int k; double s;
for (s=k=0;k<m;k++,X++,Y++) s+=(*X**Y);
return s;}
```

---

strchr(A,0)

Come vedremo, una stringa in C è semplicemente un vettore di caratteri (un puntatore a caratteri, quando l'indirizzo è variabile). Molte funzioni assumono che la stringa termini con il carattere `0` (cioè il carattere il cui codice ASCII è 0, da non confondere con il carattere `'0'` il cui codice ASCII è 48). Se `A` è un puntatore a caratteri, l'indirizzo di questo carattere terminale può essere ottenuto come `strchr(A,0)`.

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 6 ◇ 25 Ottobre 2004

Funzioni con un numero variabile di argomenti

Una funzione con un numero variabile di argomenti deve avere la seguente forma, naturalmente con variazioni:

```
tipo1 f(tipo2 a, ...)
{va_list argo; ***
va_start(argo,a);
***
x=va_arg(argo,tipo);
***
va_end(argo); ***}
```

I puntini nella lista degli argomenti (dopo `tipo2`*) devono veramente essere scritti così; gli asterischi nel corpo della funzione indicano parti da completare.*
`tipo` *è il tipo della variabile che viene prelevata; la funzione* `va_arg` *può essere chiamata più volte e non è necessario che il tipo prelevato sia sempre lo stesso. Ogni chiamata di* `va_arg` *preleva la prossima variabile dalla lista degli argomenti (da sinistra a destra, cominciando con la variabile che segue la variabile con cui abbiamo inizializzato la lista mediante* `va_start`*, nel nostro caso* `a`*.*

Non dimenticare `va_end` *alla fine, altrimenti si avrà quasi certamente un errore.*

Queste istruzioni richiedono il header <stdarg.h>.

Come funziona `va_start`*? Tra gli argomenti della funzione ci deve essere almeno una variabile di tipo noto, ce ne possono essere anche più di una, e una di esse deve essere il secondo argomento di* `va_start`*; le variabili introdotte successivamente al posto dei puntini vengono collocate in memoria dopo le variabili di tipo noto.* `va_start` *può essere chiamata anche più volte.*

```
double Somma (int n, ...)
{va_list argo; int k; double s;
va_start(argo,n);
for (s=k=0;k<n;k++)
  s+=va_arg(argo,double);
va_end(argo); return s;}

void Provasomma ()
{printf("%.2f\n",
  Somma(5,3.0,2.0,1.0,4.0,8.4));}
```

Qui, come sempre nel C, quando il compilatore si aspetta un valore di tipo double, un valore intero deve essere convertito, ad esempio, come abbiamo fatto qui, scrivendolo come numero decimale con almeno una cifra dopo il punto decimale. Rileggere quanto scritto nella terza colonna a pagina 10. Solo che nelle funzioni con un numero variabile di argomenti non abbiamo la possibilità di risolvere il problema mediante una apposita dichiarazione della funzione; ogni volta che usiamo la funzione dobbiamo stare molto attenti a convertire tutti gli argomenti usati.

Funzioni con un numero variabile di argomenti sono molto importanti nel trattamento di testi (anche ad esempio di comandi impostabili dalla tastiera con numero variabile di parametri). Impareremo come crearle, se necessario, anche se spesso si possono usare le funzioni apposite `vprintf` *e* `vsprintf` *che studieremo più tardi.*

La seguente funzione, molto simile alla precedente, calcola il massimo di una lista di numeri reali; la lista può essere di una lunghezza variabile che viene indicata come primo argomento.

```
double Maxx (int n, ...)
{va_list argo; double max,x; int k;
va_start(argo,n);
max=va_arg(argo,double);
for (k=1;k<n;k++)
{x=va_arg(argo,double);
if (x>max) max=x;}
va_end(argo); return max;}

void Prova ()
{printf("%.2f\n",
  Maxx(5,0.5,1.3,0.0,1.5,11.3));}
```

È abbastanza scomodo dover ogni volta contare gli argementi; in questo caso possiamo ad esempio usare la costante `HUGE_VAL` *che, quando correttamente implementata, corrisponde a un numero* `double` *più grande di ogni altro ragionevolmente prodotto. Possiamo allora fare a meno del parametro* `n`*, aggiungendo invece alla fine della lista questo valore quasi infinito; la funzione termina l'elaborazione, quando incontra* `HUGE_VAL`*.*

```
double Maxx (double max, ...)
{va_list argo; double x;
va_start(argo,max);
for (;;) {x=va_arg(argo,double);
if (x==HUGE_VAL) break;
if (x>max) max=x;}
va_end(argo); return max;}

void Prova ()
{printf("%.2f\n",
  Maxx(0.5,1.3,0.0,6.0,HUGE_VAL));}
```

In questo numero

Alcune costanti

Inseriamo in *alfa.h* le seguenti costanti:

```
# define CME M_E
# define CMPi M_PI
# define CMPid180 0.01745329
# define CM180dPi 57.29577951
# define CMR2Pi 2.506628274631000686
# define CMR2 M_SQRT2
```

Le costanti `M_E` e `M_PI` corrispondono ad e e π e sono già presenti in C; ciò per π significa semplicemente che il file */usr/include/math.h* (o il suo equivalente sotto Windows) contiene la riga

```
# define M_PI 3.14159265358979323846
```

Usiamo il prefisso `CM` per nomi che denotano costanti matematiche. `CMPid180` corrisponde a $\frac{\pi}{180}$, `CM180dPi` a $\frac{180}{\pi}$. La definizione

```
# define CMPid180 M_PI/180
```

sarebbe lecita, ma implicherebbe che ogni volta che l'espressione è usata, viene eseguita di nuovo la divisione.

`CMR2Pi` è uguale a $\sqrt{2\pi}$ e `CMR2` indica $\sqrt{2}$, presente in C come `M_SQRT2`.

Abbiamo già osservato a pagina 2 che in `# define` non si usa il simbolo di uguaglianza.

Parte intera e parte frazionaria

Per un numero reale x denotiamo con $[x]$ la sua parte intera, cioè l'intero più vicino a sinistra di x. Si ha sempre

$$x = [x] + \alpha$$

con $0 \leq \alpha < 1$. α è univocamente determinato e si chiama la parte frazionaria di x. Naturalmente

$$\alpha = 0 \iff x \in \mathbb{Z}.$$

Per l'arrotondamento di un numero `double` a `int` (più precisamente `long`) useremo la funzione `lrint` che è più affidabile della semplice conversione di tipo mediante `(int)x`.

In C la funzione matematica $\underset{x}{\bigcirc}[x]$ è realizzata dalla funzione `floor` che, a differenza da molte altre funzioni simili del C, fornisce un risultato corretto anche per $x < 0$.

`floor` formalmente restituisce un risultato di tipo `double` che talvolta deve essere convertito mediante `lrint`, ma più essere usata correttamente in assegnamenti come `int x=floor(u);`

Divisione con resto

Lemma 18.1. *Siano $a, b \in \mathbb{R}$ con $b > 0$. Allora*

$$a = \left\lfloor \frac{a}{b} \right\rfloor b + r$$

con $0 \le r < b$.

Dimostrazione. $\frac{a}{b} = \left\lfloor \frac{a}{b} \right\rfloor + \alpha$

con $0 \le \alpha < 1$. Perciò $a = \left\lfloor \frac{a}{b} \right\rfloor b + \alpha b$ con $0 \le \alpha b < b$, perché $b > 0$. Possiamo porre $r := \alpha b$.

Osservazione 18.2. *q, b ed r siano numeri reali. Allora*

$$qb + r = (q+1)b + r - b$$

Lemma 18.3. *Siano $a, b \in \mathbb{R}$ con $b < 0$. Allora*

$$a = \left\lfloor \frac{a}{b} \right\rfloor b + s$$

con $b < s \le 0$. Se $s < 0$, allora, ponendo $r := s - b$, abbiamo

$$a = \left(\left\lfloor \frac{a}{b} \right\rfloor + 1 \right) b + r$$

con $0 < r < b$.

Dimostrazione. Abbiamo come prima

$$\frac{a}{b} = \left\lfloor \frac{a}{b} \right\rfloor + \alpha$$

con $0 \le \alpha < 1$. Perciò $a = \left\lfloor \frac{a}{b} \right\rfloor b + \alpha b$ con $b < \alpha b \le 0$, perché $b < 0$.

Possiamo porre $s := \alpha b$. L'ultima parte segue dall'osservazione 18.2.

Corollario 18.4. *Siano $a, b \in \mathbb{R}$ con $b \neq 0$. Allora*

$$a = qb + r$$

con $q \in \mathbb{Z}$ e $0 \le r < |b|$. Il quoziente q ed il resto r possono essere ottenuti nel modo seguente:

- *Calcoliamo $q = \left\lfloor \frac{a}{b} \right\rfloor$ ed $r = a - qb$.*
- *Se $b > 0$ o $r = 0$, non dobbiamo fare altro.*
- *Se $b < 0$ ed $r < 0$, allora sostituiamo q con $q + 1$ ed r con $r - b$.*

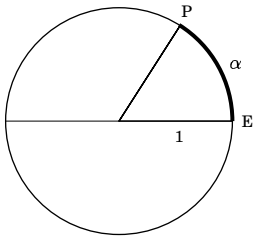
Traduciamo l'algoritmo del corollario 18.4 in tre funzioni in C, usando il prefisso `Fa` per funzioni aritmetiche. Si osservino i tipi degli argomenti.

```
// Quoziente intero di due numeri reali.
int FaDiv (double a, double b)
{int q=floor(a/b); double r=a-b*q;
if ((b<0)&&(r<0)) q++; return q;}

// Divisione e resto simultaneamente.
void FaDivResto (double a, double b,
  Int Q, Double R)
{int q=floor(a/b); double r=a-b*q;
if ((b<0)&&(r<0)) {q++; r-=b;}
*Q=q; *R=r;}

// Resto nella divisione
// di numeri reali.
double FaResto (double a, double b)
{int q=floor(a/b); double r=a-b*q;
if ((b<0)&&(r<0)) r-=b; return r;}
```

In tutti i casi `a` e `b` possono anche essere negativi, ma `b` deve essere diverso da zero.

Angoli espressi in gradi

In matematica si identifica l'angolo con la lunghezza dell'arco descritto sulla circonferenza tra i punti E e P della figura, aggiungendo però multipli del perimetro della circonferenza se l'angolo è immaginato ottenuto dopo essere girato più volte attorno al centro. Se il centro del cerchio è l'origine $(0,0)$ del piano, possiamo assumere che $E = (1,0)$. Siccome il perimetro della circonferenza di raggio 1 è 2π, si ha $360° = 2\pi$.



È chiaro che un angolo di $\alpha°$ è uguale a $\frac{\alpha}{360} 2\pi$, in altre parole

$$\alpha° = \frac{2\pi\alpha}{360} = \frac{\pi}{180}\alpha$$

in particolare

$$1° = \frac{\pi}{180}$$

quindi

$$1° \sim 0.01745329$$

e che viceversa

$$\alpha = \alpha \frac{360°}{2\pi} = \frac{180}{\pi}\alpha°$$

per ogni $\alpha \in \mathbb{R}$; in particolare

$$1 = \left(\frac{180}{\pi}\right)°$$

quindi

$$1 \sim 57.29577951°$$

Le funzioni seno e coseno sono presenti in C e si chiamano `sin` e `cos`. Come nelle funzioni matematiche corrispondenti in esse gli argomenti devono essere indicati in radianti. Se vogliamo lavorare con gradi, possiamo usare l'uguaglianza

$$\alpha° = \frac{\pi}{180}\alpha$$

vista prima, per cui definiamo, usando il prefisso `Ft` per le funzioni trigonometriche,

```
// Coseno per argomenti
// espressi in gradi.
double FtCosg (double x)
{return cos(x*CMPid180);}

// Seno per argomenti
// espressi in gradi.
double FtSing (double x)
{return sin(x*CMPid180);}
```

Una prova per il coseno:

```
void Prova ()
{int k; double x,gradi;
for (k=0;k<=24;k++)
{gradi=k*15; x=gradi*CMPid180;
printf("%.5f %.5f\n",
  cos(x),FtCosg(gradi));}}
```

Tabelle trigonometriche

Le funzioni trigonometriche sono, ancora oggi, piuttosto lente nell'esecuzione al calcolatore. Se, ad esempio in programmi di grafica, vengono usate intensamente, può essere utile creare prima una tabella per argomenti in decimi di gradi, da cui prelevare i valori.

Definiamo quindi

```
// Creazione delle tabelle
// per seno e coseno.
void Creatabtrig ()
{int k; double x; Double C,S;
for (k=0,C=tabcos,S=tabsin;k<3600;
  k++,C++,S++)
{x=k*CMPid180/10;
*C=cos(x); *S=sin(x);}}
```

L'idea è che adesso, dopo un'unica esecuzione di `Creatabrig()` all'inizio del programma, $\sin 84.7°$ è il valore `tabsin[847]`. Veniamo a questo punto però confrontati con due problemi:

In primo luogo bisogna stare molto attenti agli arrotondamenti, quando, in una tabella per decimi di gradi, moltiplichiamo gli argomenti per 10.

Inoltre argomenti non compresi tra 0 e 359.9 devono essere ridotti modulo 360, ad esempio $\sin 380.4° = \sin 20.4°$ è uguale a `tabsin[204]`, mentre naturalmente non è definito il valore `tabsin[3804]`.

Introduciamo quindi una funzione per ricavare valori da una delle tabelle di funzioni trigonometriche create con `Creatabtrig`.

```
double Trigdatabella
  (double x, Double Tabella)
{x=FaResto(x,360)*10;
return Tabella[lrint(x)];}
```

Per usarla, dobbiamo naturalmente prima aver eseguito (una volta per tutte) `Creatabtrig`. Esempio:

```
void Prova ()
{Creatabtrig(); double x;
for (x=0;x<=360;x+=11.25)
printf("%6.2f %7.3f %7.3f\n",
  x,Trigdatabella(x,tabcos),
  Trigdatabella(x,tabsin));}
```

con output

```
  0.00   1.000   0.000
 11.25   0.981   0.194
 22.50   0.924   0.383
 33.75   0.831   0.556
 45.00   0.707   0.707
 56.25   0.556   0.831
 67.50   0.383   0.924
 78.75   0.194   0.981
 90.00   0.000   1.000
101.25  -0.194   0.981
112.50  -0.383   0.924
123.75  -0.556   0.831
135.00  -0.707   0.707
...
```

Le due funzioni vengono inserite nel file *matematica.c*, all'inizio del quali si trova anche la dichiarazione

```
double tabcos[3600],tabsin[3600];
```

che deve essere ripetuta in *alfa.h* con la specificazione `extern`:

```
extern double tabcos[3600],tabsin[3600];
```

Funzioni matematiche del C

Elenchiamo le più importanti funzioni matematiche del C, alcune delle quali le abbiamo già incontrate, con i loro prototipi. Tralasciamo le funzioni per la divisione col resto che sostituiamo con le funzioni definite a pagina 18. Per i dettagli consultare il libro di Harbison/Steele (citato a pagina 8); sotto Linux anche, ad esempio, `man lgamma`.

```
\\ Massimo e minimo.
double fmax (double,double);

double fmin (double,double);

// fdim(a,b) e' uguale ad a-b se a>=b,
// altrimento e' zero.
double fdim (double,double);

// Valore assoluto di intero.
int abs (int);

// Valore assoluto di long.
long labs (long);

// Valore assoluto di double.
double fabs (double);

// Piu' vicino intero a destra.
double ceil (double);

// Piu' vicino intero a sinistra.
double floor (double);

// Arrotondamento a intero.
long lrint (double);

// Esponenziale in base e.
double exp (double);

// Esponenziale in base 2.
double exp2 (double);

// Logaritmo in base e.
double log (double);

// Logaritmo in base 10.
double log10 (double);

// Logaritmo in base 2.
double log2 (double);

// Potenza; il secondo
// argomento e' l'esponente.
double pow (double,double);

// Radice quadrata.
double sqrt (double);

// Radice cubica.
double cbrt (double);

// Valore assoluto di
// un vettore del piano;
// utile, perche' senza overflow.
double hypot (double,double);

// Funzioni trigonometriche
// e iperboliche e loro inverse.
double cos (double);

double sin (double);

double tan (double);

double cosh (double);

double sinh (double);

double tanh (double);

doube acos (double);

double asin (double);

double atan (double);

double atan2 (double,double);

double acosh (double);

double asinh (double);

double atanh (double);

// Logaritmo della funzione gamma.
double lgamma (double);

// Funzione d'errore.
double erf (double);
```

Le funzioni di Bessel sono elencate a parte.

Per quanto riguarda la funzione Γ, gli sviluppatori del C hanno creato un po' di confusione. Esiste infatti una funzione più vecchia che porta il nome `gamma`, restituendo però il logaritmo della funzione Γ (come `lgamma`). Si è aggiunta una funzione `tgamma` il cui nome deriva da *true gamma* (cioè vera funzione gamma) e che dovrebbe corrispondere alla Γ, la quale sembra però non implementata correttamente. Creiamo quindi una nostra funzione apposita:

```
double Gamma (double x)
{return exp(lgamma(x));}
```

La funzione d'errore `erf` è definita da

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

È legata alla distribuzione normale standardizzata dalla relazione

$$\mathrm{erf}(x) = 2\Phi(\sqrt{2}x) - 1$$

come si vede mediante una trasformazione di variabili, essendo

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}}$$

---

atan2

Le funzioni `acos`, `asin` e `atan` funzionano come uno se lo aspetta, con

$$0 \le \texttt{acos(x)} \le \pi$$

$$-\frac{\pi}{2} \le \texttt{asin(x)} \le \frac{\pi}{2}$$

$$-\frac{\pi}{2} \le \texttt{atan(x)} \le \frac{\pi}{2}$$

`atan2` calcola le coordinate polari di un punto $\neq (0,0)$ nel piano, tenendo conto del quadrante. `atan2(0,0)` non è definito; per $(x,y) \neq (0,0)$ invece `atan2(y,x)` è uguale al valore principale di $\frac{y}{x}$. Attenzione all'ordine degli argomenti.

In altre parole, usando la rappresentazione complessa, se $z = x + yi \neq 0$ con $x, y \in \mathbb{R}$ e $z = |z|e^{i\alpha}$ con $-\pi < \alpha \le \pi$, allora

$\alpha =$ `atan2(y,x)`

Funzioni di Bessel

```
// Funzioni di Bessel di prima specie.
double j0 (double);

double j1 (double);

double jn (int,double);

// Funzioni di Bessel di seconda specie.
double y0 (double);

double y1 (double);

double yn (int,double);
```

`j0` e `y0` sono le funzioni di primo ordine, `j1` e `y1` quelle di secondo ordine. L'ordine è il primo argomento in `jn` e `yn`.

Prodotto di Hadamard

Il prodotto di Hadamard P di due matrici A e B della stessa forma è definito da $P_{jk} = A_{jk}B_{jk}$; è quindi una matrice ancora della stessa forma i cui coefficienti si ottengono moltiplicando tra di loro i coefficienti corrispondenti della stessa forma.

In particolare possiamo fare il prodotto di Hadamard di due vettori della stessa lunghezza:

```
Double Hadamard (Double A, Double B,
  int m)
{int k; Double P,U;
P=calloc(m,sizeof(double));
for (k=0,U=P;k<m;k++,A++,B++,U++)
  *U=*A**B;
return P;}
```

Si noti soprattutto che in questo caso il risultato della funzione è un puntatore a `double`. Ciò significa molto concretamente che questo risultato è un indirizzo che corrisponde a una parte della memoria in cui sono riservati `m` posti per oggetti del tipo `double`, occupati dai coefficienti del prodotto di Hadamard. La lunghezza dei vettori deve essere indicato come intero nel terzo argomento. Esempio:

```
void Prova ()
{int k; double a[]={1,2,3,4,5},
  b[]={11,12,13,14,15};
Double P=Hadamard(a,b,5);
for (k=0;k<5;k++)
printf("%.1f ",P[k]);
printf("\n");}
```

con output

```
11.0 24.0 39.0 56.0 75.0
```

Il prodotto di Hadamard, detto anche prodotto di Schur, appare in moltissimi campi della matematica (analisi funzionale, calcolo combinatorio, funzioni quasiperiodiche) e in maniera naturale in statistica multivariata. Apparentemente quasi banale, è invece la bestia nera dell'algebra lineare, perché è molto difficile collegarlo con l'algebra delle matrici usuale.

R. Horn/C. Johnson: Matrix analysis. Cambridge UP 1993.

R. Horn/C. Johnson: Topics in matrix analysis. Cambridge UP 1994.

R. Horn: The Hadamard product. Proc. Symp. Appl. Math. 40 (1990), 87-169.

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 7 ◇ 1 Novembre 2004

Colori

I colori sullo schermo di un computer sono descritti da valori RGB (rosso, verde, blu) generati, nel caso dei monitor tradizionali a tubo catodico da tre cannoni elettronici (uno per ciascuno dei trei colori) oppure, nel caso degli schermi LCD da tre minuscoli transistor (di nuovo uno per ogni colore) in ogni singolo pixel dello schermo.

I valori, in verità compresi tra 0 *e* 1*, vengono spesso indicati da numeri tra* 0 *e* 255 *(ad es a* 210 *allora corrisponde il valore* $\frac{210}{255}$*) oppure da numeri tra* 0 *e* 65535 *(e quindi a 21000 corrisponde il valore* $\frac{21000}{65535}$*), utilizzando la rappresentazione esadecimale (a 2 cifre nel primo caso, a 4 cifre nel secondo) mediante stringhe a cui spesso (ad esempio nell'indicazione di colori in HTML) viene anteposto un simbolo* `#`*.*

Esempio:

	R	*G*	*B*	*esadecimale*
2 cifre	*122*	*54*	*181*	`#7a66a1`
4 cifre	*28000*	*13581*	*16255*	`#6d60350d3f7f`

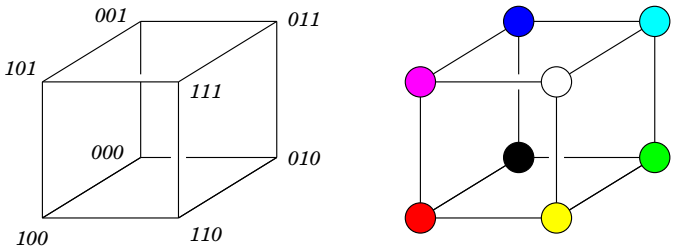
La stringa esadecimale nell'ultima riga si ottiene ad esempio con

```
printf("#%04x%04x%04x\n",28000,13581,16255);
```

Usando la rappresentazione reale (con valori in $[0, 1]$*), possiamo definire i colori principali, in cui R,G,B assumono solo i valori* 0 *ed* 1 *(e quindi 0 e 255 risp. 0 e 65535):*

R	*G*	*B*		
0	*0*	*0*	*nero*	
1	*0*	*0*	*rosso*	
0	*1*	*0*	*verde*	
0	*0*	*1*	*blu*	
1	*1*	*0*	*giallo*	
1	*0*	*1*	*magenta*	
0	*1*	*1*	*ciano*	
1	*1*	*1*	*bianco*	

Questi otto colori principali possono essere considerati come vertici di un cubo, che nel suo insieme rappresenta tutti i colori possibili.



Il sistema CMY

La rappresentazione RGB dei colori è la più adatta per lo schermo del computer. Si tratta di uno schema *additivo* che corrisponde al fatto che lo schermo emette la luce che noi vediamo. Molto diversa è la situazione per i colori su carta e quindi per la stampa. Infatti i colori che vediamo sulla carta non sono emessi dalla carta ma sono riflessi. Più precisamente la carta riceve la luce *bianca*, di cui una parte nelle regioni colorate viene assorbita e noi vediamo la luce non assorbita che viene riflessa. Per colori stampati si usa quindi un sistema *sottrattivo* (CMY o CMYK).

Come i colori sulla carta si comportano anche gli altri oggetti che noi vediamo. Le piante ad esempio nella fotosintesi usano la luce rossa e la luce blu, ma non quella verde; quest'ultima per esse è inutile e viene riflessa o può passare attraverso di esse; perciò le piante appaiono verdi, anche quando la luce si trova dietro di esse.

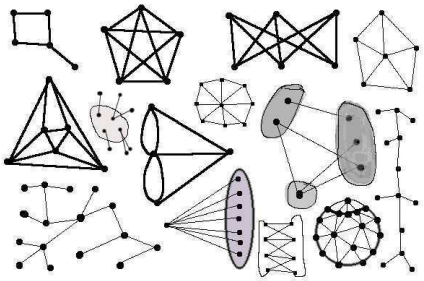
In questo numero

Gimp

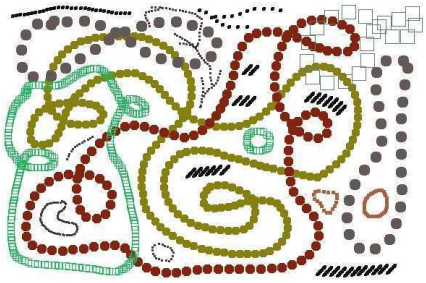
Questo programma gratuito (GPL) per la creazione ed elaborazione delle immagini (la sigla significa *GNU image manipulation program*) è confrontabile e talvolta addirittura superiore a Photoshop, un famoso programma commerciale della *Adobe*, una delle ditte più grosse del settore, nota per aver introdotto il linguaggio PostScript e per i suoi font. Per novità e informazioni si può consultare *www.gimp.org/*. Uno dei migliori libri su Gimp (*Grokking the Gimp* di Carey Bunks) può essere letto sul WWW (*gimp-savvy.com/BOOK/*). Un'introduzione (di qualche anno fa), ad esempio su come si creano disegni animati, si trova alle pagine 28-29 del corso di Sistemi 2000/01.

Gimp non fornisce strumenti diretti per disegnare forme geometriche (cerchi, ellissi, rettangoli, rette). Queste si ottengono però facilmente (dopo un po' di esercizio) mediante le funzioni di selezione.

Anche grafi si disegnano velocemente:



Si noti che Gimp non è un programma di disegno vettoriale, ma orientato ai pixel, quindi più adatto alla pittura e al ritocco che al disegno tecnico. Oltre a usi più professionali è ideale per creare immagini o icone per il Web.

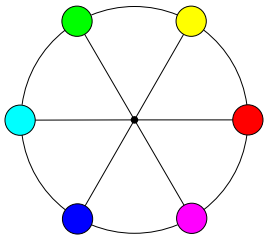


C. Bunks: Grokking the GIMP. New Riders 2000.

Il sistema HSV

È alquanto difficile giudicare un colore dai suoi valori RGB. Si usa quindi spesso un'altra rappresentazione, molto più intuitiva e corrispondente da vicino al modo in cui un pittore considera i colori. I parametri in inglese si chiamano H (*hue*, colore puro), S (*saturation*, saturazione), V (*value*, valore, luminosità):

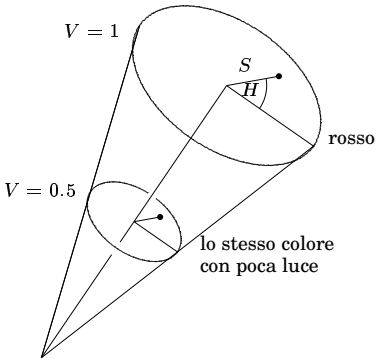
H è la posizione angolare in gradi sul cerchio dei colori:



Quindi $0 \leq H < 360$.

S indica la percentuale di colore puro, immaginando di avere aggiunto del bianco. Quindi $S = 0$ è solo bianco, $S = 0.5$ metà bianco, $S = 1$ solo colore.
V indica la luce emessa o luminosità (V=1 massima luminosità).

Nella scala HSV i colori sono perciò rappresentati come punti di un cono:



Si vede che i colori con $S = 0$ non possiedono un valore H ben definito; lo possiamo porre uguale a -1 se limitiamo H a $[0, 360)$. Questi colori hanno allora la rappresentazione $(-1, 0, V)$ e corrispondono ai *grigi* (che nella rappresentazione RGB sono i colori con R=G=B).

Anche valori positivi, ma bassi di S, ad esempio ≤ 0.3, rendono i colori molto simili al bianco o ai grigi.

Dei tre parametri V è quello più artificiale (ma anche il meno importante); infatti le scale comunemente utilizzate non rispecchiano il fatto che il nostro occhio è sensibile ai tre colori di base in modo molto diverso: 59% per il verde, 30% per il rosso, 11% per il blu. Quindi un verde ci sembra molto più luminoso di un blu, a uguali valori di saturazione e di V.

G. Wyszecki/W. Styles: Color science. Wiley 2000.

M. Fairchild: Color appearance models. Wiley 2004. *www.cis.rit.edu/people/faculty/fairchild/.*

Conversione tra RGB e HSV

Le seguenti funzioni per la conversione tra valori RGB ed HSV (non richieste all'esame) sono prese dal famoso libro sulla grafica al calcolatore di Foley e.a. L'algoritmo, talvolta criticato, consiste approssimativamente in una proiezione del cubo sul cono.

Introduciamo prima la funzini ausiliare `Max3` e `Min3` per il calcolo del massimo e del minimo di tre numeri reali.

```
double Max3 (double a, double b, double c)
{return Max(a,Max(b,c));}

double Min3 (double a, double b, double c)
{return Min(a,Min(b,c));}
```

Usiamo valori RGB tra 0 e 65535.

```
void Hsvdargb (int ri, int gi, int bi,
  Double H, Double S, Double V)
{double r=ri/65535.0, g=gi/65535.0,
  b=bi/65535.0;
double max=Max3(r,g,b), min=Min3(r,g,b);
*V=max; double d=max-min,h;
double s = max ? d/max : 0;
if (s) {if (r==max) h=(g-b)/d;
else if (g==max) h=2+(b-r)/d;
else h=4+(r-g)/d;
h*=60; if (h<0) h+=360;}
else h=-1; *H=h; *S=s;}

void Rgbdahsv (double h, double s,
  double v, Int R, Int G, Int B)
{double r,g,b;
if (!s) if (h<0) {r=g=b=v;}
else {*R=*G=*B=80000; return;}
else {double f,p,q,t; int i;
if (h==360) h=0; h/=60; i=floor(h);
f=h-i; p=v*(1-s); q=v*(1-s*f);
t=v*(1-s*(1-f));
switch(i) {case 1: r=q; g=v; b=p; break;
case 2: r=p; g=v; b=t; break;
case 3: r=p; g=q; b=v; break;
case 4: r=t; g=p; b=v; break;
case 5: r=v; g=p; b=q; break;
case 0: default: r=v; g=t; b=p;}}
*R=65535*r; *G=65535*g; *B=65535*b;}
```

Esempio:

```
void Provacolori ()
{int r,g,b; double h,s,v,h1,s1,v1;
h=235; s=0.8; v=0.9;
Rgbdahsv(h,s,v,&r,&g,&b);
Hsvdargb(r,g,b,&h1,&s1,&v1);
printf("%.0f %.2f %.2f "
  "#%04x%04x%04x\n%.0f %.2f %.2f\n",
  h,s,v,r,g,b,h1,s1,v1);}
```

con output

```
235 0.80 0.90 #2e143d70e665
235 0.80 0.90
```

Esercizio: Provare a determinare i colori di oggetti (maglioni, porte, piante, libri, mobili, astucci) in rappresentazione HSV e confrontare con la tabella dei colori su *felix.unife.it*.

J. Foley/A. Van Dam/S. Feiner/J. Hughes: Computer graphics. Addison-Wesley 1996.

La teoria dei colori è molto complicata e oltre alle rappresentazioni RGB ed HSV esistono tante altre. I colori percebili dall'occhio umano sono stati codificati dalla CIE (*Commission Internationale d'Eclairage*); il formato CIE è molto generale e non dipende dal dispositivo usato.

www.cie.co.at/cie/
en.wikipedia.org/wiki/International_Commission_on_Illumination

Il monitor CRT

Un buon monitor è sempre un buon acquisto. Incide molto sulla qualità e sulla piacevolezza del lavoro e invecchia molto meno rapidamente di altri componenti.

Non aprire mai un televisore o un monitor! Ci sono altissime tensioni anche quando il cavo della corrente è staccato.

I monitor tradizionali con tubo a raggi catodici (*cathode-ray tube, CRT*) prendono nome dall'ampio tubo di vetro (vetro di circa 2.5 cm di spessore, per questo i monitor sono così pesanti) messo (quasi) a vuoto (pericolo di implosione!), in cui elettroni emessi da tre (nel caso dei monitor a colori) cannoni elettronici (emettitori di elettroni o catodi, formati ad esempio da un filamento di tungsteno a cui viene applicata una tensione di riscaldamento portandolo a un'elevata temperatura e racchiusi in un cilindro metallico - cilindro di Wehnelt - da cui vengono emessi gli elettroni per *effetto termoionico*) e guidati da elettromagneti (bobine applicate all'esterno sul collo del tubo e formanti il *giogo di deflessione*) colpiscono la superficie anteriore fosforescente che verso l'interno è ricoperta da un sottilissimo strato riflettente di alluminio che è attraversato dagli elettroni ma riflette verso l'utente i fotoni emessi all'indietro, aumentando la luminosità verso l'esterno. Questo strato di alluminio protegge anche lo schermo da ioni negativi che si formano dalle pochissime molecole di gas presenti nel vuoto quasi assoluto del tubo. Il catodo è contenuto in un altro cilindro metallico con un foro anteriore attraverso il quale gli elettroni devono passare formando così un fascio. Tra il catodo e la bobina di deflessione si trovano gli anodi di accelerazione e di messa a fuoco, a cui vengono applicate tensioni altissime (15000-25000 V). Nei televisori a colori la maschera utilizzata per distribuire gli elettroni sui fosfori di un determinato colore assorbe circa il 75 % degli elettroni, e quindi la tensione che deve essere applicata agli anodi è molto più alta che nei televisori bianco-nero.

L'ultimo anodo è collegato a uno strato conduttore di grafite che ricopre internamente i lati del tubo, prolungando così l'azione acceleratrice dell'anodo fino in prossimità dello schermo, e raccoglie gli elettroni rimbalzati e quelli prodotti dall'emissione secondaria. Un altro strato di grafite applicato sulla parete esterna del tubo è collegato alla massa e forma con lo strato interno un condensatore il cui dielettrico è costituito dal vetro del tubo.

La tensione necessaria per guidare i raggi di elettroni è generata dagli amplificatori video (uno per ogni colore), che aumentano il segnale da circa 1 V ricevuto dal PC.

Il principio è lo stesso di quello dei televisori, ma i monitor per PC sono molto più sofisticati, hanno una risoluzione migliore e un'immagine più stabile che permette di distinguere dettagli molto più piccoli.

Esercizio: Uno schermo di 15 pollici è largo 12 pollici e alto 9. Sulla sola superficie anteriore spinge quindi un peso di 700 kg. Per questo il vetro deve essere così spesso.

I fosfori

Dove il raggio di elettroni che colpisce un fosforo consiste di molti elettroni per unità di tempo, il punto brillerà intensamente, dove il raggio è debole, si avrà meno luce. L'immagine, anche se non cambia, deve essere ricostruita in continuazione (tipicamente 70 Hz, frequenza minima per avere un'immagine senza sfarfallio su uno schermo CRT).

Il raggio di elettroni si sposta leggermente verso il basso e verso destra, tornando a sinistra alla fine di ogni riga e in alto alla fine di ogni schermata. Nel modo intrecciato (*interlaced*), usato in una variazione nei televisori, in ogni schermata viene disegnata solo la metà delle righe (una volta le righe dispari, la volta dopo quelle pari). In questo caso il numero delle righe dev'essere dispari. Virtualmente si raddoppia la frequenza, ma l'immagine ne risente (sfarfallio).

La distanza minima tra due punti creati dal monitor sullo schermo si chiama *dot pitch*. Il dot pitch corrisponde al diametro dei fori e deve essere tanto minore quanto è maggiore la dimensione dello schermo, perché con risoluzioni più elevate i pixel devono essere più precisi. Tipici valori del dot pitch (per CRT):

14 pollici	0.39 mm
15 pollici	0.28 mm
17 pollici	0.26 mm

Per ottenere i tre colori di base (rosso, verde, blu) lo strato di fosforo dello schermo consiste di tre materiali diversi che si alternano e che emettono luce ciascuno di un solo colore, con l'intensità che corrisponde al raggio di elettroni che li colpisce. I tre fosfori che corrispondono a un pixel formano una *triade* o *terzina*. Si usano tre cannoni elettronici, uno per ogni colore, che devono colpire la parte corrispondente del pixel che verrà attivato. Osservando uno schermo a colori da molto vicino si vedono abbastanza bene i fosfori rossi, verdi e blu. Si vedono anche (naturalmente meno luminosi) a televisore spento (si vede allora anche lo sfondo, cioè lo spazio tra gli fosfori, detto *matrice*). Da una certa distanza invece l'occhio sovrappone questi colori e si ha l'impressione di vedere il colore che si ottiena dalla miscela dei colori fondamentali.

In una prima tecnologia, i tre cannoni elettronici sono disposti a triangolo e una griglia (maschera) di acciaio sta vicina allo fosforo, lasciando per ogni pixel un foro attraverso il quale passano i tre raggi di elettroni che continuando il loro volo verso il vicino schermo su linee rette vanno a colpire punti leggermente distanti in cui si trovano i tre fosfori di colore diverso anch'essi disposti a triangolo.

In una tecnologia più recente invece di una griglia con fori rotondi si usano lunghi fori rettangolari e i tre cannoni elettronici sono allineati orizzontalmente uno vicino all'altro. Questa tecnologia ha vari vantaggi, tra cui la maggiore purezza dei colori, perché, essendo i fosfori sulla stessa linea verticale tutti dello stesso colore, leggere deviazioni verticali del raggio elettronico non influiscono sul colore. In entrambi i casi però la maschera respinge circa il 3/4 degli elettroni (ciò causa anche un suo aumento di temperatura che provoca piccole distorsioni).

In una tecnologia introdotta dalla Sony, i tre cannoni sono ancora allineati uno vicino all'altro, ma invece di una griglia si usa una successione di fili verticali (dello spessore di 0.18mm e distanti tra di loro di 0.25mm), tra i quali possono passare gli elettroni (si parla adesso di *slot pitch* invece di *dot pitch*). Questa è la tecnologia *Trinitron*. Questa tecnologia permette anche una notevole semplificazione dei meccanismi di deflessione e una luminosità costante su tutto lo schermo. Per impedire una vibrazione dei fili verticali, la griglia di fili viene tenuta in posizione da uno o due sottili fili orizzontali che si possono vedere sullo schermo e da cui si riconoscono gli schermi Trinitron.

La fosforescenza

La fosforescenza è un tipo di luminescenza che si manifesta anche dopo che è cessata l'eccitazione. I *fosfori* ("portatori di luce") che rivestono la superficie interna del tubo a raggi catodici non consistono di fosforo (elemento chimico), ma sono ad esempio sulfidi o silicati di zinco o cadmio con piccole aggiunte di altre sostanze che determinano la durata della luminescenza.

Gli schermi a cristalli liquidi

Cristalli liquidi sono sostanze che mantengono (in certe condizioni ad esempio di temperatura) una struttura cristallina (cioè a molecole disposte su un reticolo) anche allo stato liquido. Hanno la proprietà che quando viene applicata corrente ruotano la polarizzazione della luce.

Un pannello LCD (*liquid crystal display*) a matrice attiva (tecnologia TFT, *thin film transistor*) contiene uno strato che consiste di un reticolo di piccoli transistor, tre (rosso, verde, blu) per ogni punto, e la risoluzione migliore (tipicamente 1024x768) corrisponde esattamente al numero di questi transistor (nel nostro caso 1024x768x3 = 2359296). Scegliendo questa risoluzione naturale si avrà una rappresentazione ottimale. Ogni pixel corrisponde allora precisamente a uno di questi transistor, quindi ogni pixel dispone di un proprio circuito. Uno dei vantaggi degli schermi LCD è che l'immagine vicino al bordo è della stessa qualità come al centro. Ciò fa in modo, tra l'altro, che uno schermo LCD di 15 pollici equivale come grandezza dell'immagine a un monitor a tubo a raggi catodici di 17 pollici. L'immagine non tremola e a frequenze da 60 Hz si hanno immagini nitide e si possono seguire movimenti veloci. Si può anche provare a girare questi schermi di 90 gradi e vedere così una pagina A4 interamente (sotto Unix per molti programmi ciò non dovrebbe costituire un problema).

All'atto dell'acquisto è importante controllare se è possibile staccare lo schermo dalla base, ciò permette di piazzare lo schermo vicino alla parete, diminuendo ancora l'ingombro. Stare attenti alla qualità dei colori, sembra che ci siano notevoli differenze tra i vari modelli. Il prezzo (sempre meno) più alto dei monitor LCD viene in parte compensato dal basso consumo di energia (attorno ai 35 W, un monitor CRT a 17 pollici consuma 80-150 W).

I monitor LCD non emettono onde elettromagnetiche.

Attualmente gli schermi LCD per poter essere compatibili con gli adattatori video normalmente utilizzati in genere comunicano con il PC come se fossero dispositivi analogici, anche se il loro principio di funzionamento è intrinsicamente di natura digitale. Nel modo analogico i segnali video digitali del PC vengono trasformati in segnali analogici dal DAC (*digital analog converter*) dell'adattotore grafico come per i monitor a tubo a raggi catodici, e poi riconvertiti nel pannello LCD stesso di nuovo a segnali digitali. Questo giro artificioso non giova alla qualità dell'immagine (che però in genere è eccellente lo stesso) e, come si legge in Norton/Goodman a pag. 356, la differenza qualitativa che si ottiene mantenendo il segnale video costantemente digitale, è assolutamente sbalorditiva. Per alcuni monitor LCD esistono apposite schede video completamente digitali. Le specifiche DVI (*digital visual interface*, da non confondere con la sigla omonima per i files *device independent* prodotti dal TEX), sono recenti e ancora in elaborazione.

Oltre alla mancanza di sfarfallio gli schermi LCD sono praticamente esenti da riflessi.

P. Norton/J. Goodman: Inside PC. Jackson 1999.

Come funzionano gli LCD

Il pannello LCD è illuminato a retro, la luce prima di attraversare lo strato dei cristalli liquidi passa per un filtro che la polarizza e dopo lo strato per un altro filtro ortogonale al primo. La luce verrebbe quindi bloccata, ma i cristalli liquidi ruotano (a seconda della corrente applicata) il piano di polarizzazione e ciò provoca il passaggio di una luce che dipende dalla corrente applicata in un determinato punto e dalla rotazione indotta dal cristallo quando non c'è corrente (se per esempio l'angolo a riposo è di 90 gradi, ciò compensa esattamente l'ortogonalità dei due filtri e quindi i punti luminosi saranno proprio quelli in cui non viene applicata corrente). Questa tecnica, ancora oggi più diffusa, viene detta *twisted nematic* (*nematic* è un aggettivo che viene usato per molecole che hanno le proprietà dei cristalli liquidi).

A questo punto si distinguono LCD a *matrice passiva* e a *matrice attiva*. Negli LCD a matrice attiva i transistor, applicati su una delle due superfici tra le quali si trova lo strato di cristalli liquidi (sull'altra si trovano elettrodi passivi), funzionano come amplificatori di segnale, cioè è sufficiente una piccola corrente che poi ne genera una più grande che viene applicata in quel punto allo strato di cristalli liquidi. La corrente più piccola può essere generata più velocemente (di circa 10 volte), circa 60 volte al secondo e quindi l'immagine di uno schermo LCD può essere aggiornata con una frequenza di 60 Hz, sufficiente per rappresentare movimenti anche veloci (il che non è possibile invece con gli schermi LCD a matrice passiva che funzionano ad esempio a 6 Hz).

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 8 ◇ 8 Novembre 2004

Stringhe

Il C non possiede un tipo per le stringhe; queste vengono invece rappresentate dai vettori di caratteri. Ciò non è uno svantaggio, ne consegue anzi una maggiore flessibilità e potenza nel trattamento delle stringhe rispetto a molti linguaggi che prevedono un tipo apposito di stringa con operatività limitata. Una stringa in C è semplicemente una successione di caratteri terminata dal carattere con codice ASCII zero. Nella rappresentazione esplicita stringhe vanno racchiuse tra virgolette, caratteri tra apici.

Dichiarazione di stringhe

Stringhe possono essere inizializzate in questo modo abbreviato:

```
char a[]="Pentecoste";
Char A="Pasqua";
```

Un vettore di stringhe viene rappresentato da un vettore di caratteri a due indici:

```
char a[10][20];
Char *AX;
```

con possibili inizializzazioni

```
char a[2][20]={"Dante","Petrarca"};
Char A[2]={"Dante","Petrarca"};
```

Per capire queste costruzioni, pensare sempre in termini di memoria! Ad esempio

```
puts("alfa"+1);
```

con output `lfa`.

In parte, le dichiarazioni `Char A;` e `char a[100];` si equivalgono. Esiste però una differenza (che vale per ogni tipo di puntatore, come abbiamo già visto in precedenza): Con `Char A;` non viene riservato lo spazio per la stringa; se si usa un vettore con indicazione della lunghezza, lo spazio riservato viene invece assegnato e la variabile del vettore contiene l'indirizzo dell'inizio di quel tratto di memoria riservato.

Una dichiarazione con inizializzazione

```
Char A="Alfa";
```

fa in modo che vengano riservati $n+1$ bytes, dove n è la lunghezza della stringa.

Mentre anche per altri tipi di vettori, ad esempio di tipo `double`, è permessa un'inizializzazione, ad esempio è corretto

```
double a[]={1,2,3,4,5};
```

a pagina 19, l'inizializzazione analoga per puntatori è possibile solo per puntatori a caratteri (cioè stringhe). Quindi mentre non possiamo scrivere

```
Double V={1,2,3,4,5}; // Errore.
```

l'istruzione

```
Char A="Alberto";
```

è permessa oppure anche un'assegnazione successiva come in

```
Char A;
A="Alberto";
```

In questo caso viene cercato prima uno spazio di 8 byte adiacenti che viene riempito con i caratteri `'A'`, `'l'`, ..., `'o'`, dopodiché `A` diventa l'indirizzo del primo byte della stringa. Lo spazio riservato per `A` è di esattamente 8 byte; è quindi possibile trasformare la stringa in `"Roberto"` mediante le istruzioni

```
A[0]='R'; A[1]='o';
```

mentre `A[20]='t';` quasi sicuramente causerà un grave errore perché si scrive su uno spazio non più riservato.

Se invece scriviamo `A="Roberto";`, vengono cercati altri 8 byte liberi in memoria al primo dei quali punterà adesso `A`. Ciò invece non è possibile dopo `char a[]="Alberto";` - in questo caso `a` è un indirizzo fisso che non può essere spostato.

A differenza dalle stringhe, singoli caratteri vanno racchiusi tra `'`, ad esempio:

```
char a,b,c;
a='x'; b='y'; c='z';
printf("%c%c%c\n",a,b,c);
```

con output `xyz`.

La seguente istruzione scrive il codice ASCII sullo schermo (cfr. pagina 8):

```
int n;
for (n=0;n<256;n++)
printf("%d %c\n",n,n);
```

La differenza tra `'A'` ed `"A"` è che `'A'` è un carattere singolo, mentre `"A"` in memoria è rappresentato da `'A'0`.

Per indicare `"` in una stringa esplicita, si usa `\"`:

```
printf
  ("Disse \"Ciao!\", sorridendo.\n");
```

con output

```
Disse "Ciao!", sorridendo.
```

In questo numero

Lo spazio di una stringa

L'istruzione

```
Char A;
A="Reno";
```

crea uno spazio riservato di 5 byte in una determinata posizione di memoria. `A+4` mostra sul carattere `0` terminale, l'indirizzo `A+5` non è più riservato; non dobbiamo perciò scrivere in `A+5` con `A[5]='x'`, così come provocherà un errore di memoria l'istruzione

```
A=A+50; *A='x';
```

se lo spazio non è stato riservato in altro modo in precedenza.

È invece corretto

```
A="Venezia";
```

perché in tal caso viene cercato uno spazio nuovo in memoria, come abbiamo già visto. Controlliamo se effettivamente lo spazio originale viene riservato:

```
void Venezia ()
{Char A,B; int k;
A="Reno"; B=A; A="Venezia";
for (k=0;k<5;k++) *B='M';
// Per vedere se viene generato
// un errore di memoria.
printf("%s %s\n",B,A);}
```

L'output è `Meno Venezia`.

La lunghezza di una stringa `A` viene calcolata con `strlen(A)`. Possiamo facilmente creare una funzione nostra allo stesso scopo:

```
int Lun (Char A)
{int n;
for (n=0;*A;A++,n++);
return n;}
```

Sappiamo che questa funzione non modifica l'indirizzo `A` perché internamente lavora con una copia. La sequenza

```
for (...;*A;A++,...)
```

è tipicamente usata per percorrere una stringa.

Confronto di stringhe

Definiamo una funzione `Tu` per l'uguaglianza di stringhe (testi) nel modo seguente:

```
int Tu (Char A, Char B)
{for (;*A;A++,B++) if (*A!=*B) return 0;
return *B==0;}
```

L'algoritmo percorre la prima stringa fino alla sua fine e confronta ogni volta il carattere nella prima stringa con il carattere nella posizione corrispondente della seconda. Quando trova la fine della prima, controlla ancora se anche la seconda termina.

Si noti anche qui che per percorrere le due stringhe usiamo le stesse variabili `A` e `B` che all'inizio denotano gli indirizzi delle stringhe. Sappiamo infatti che questo non cambia i valori originali di `A` e `B`.

`Tu(A,B)` restituisce il valore 1, se le due stringhe `A` e `B` sono uguali, altrimenti 0. Per provarla possiamo usare la funzione

```
void Provastringhe ()
{printf("%d %d %d\n",Tu("alfa","alfa"),
  Tu("alfa","alfabeto"),
  Tu("alfa","beta"));}
```

con output `1 0 0`.

In verità per il confronto di stringhe conviene usare la funzione `strcmp` del C che tratteremo fra poco:

```
int Tu (Char A, Char B)
{return strcmp(A,B)==0;}
```

Ricordiamo che un'espressione booleana in C è un numero (uguale a 0 o 1) che può essere risultato di una funzione.

Attenzione: Per l'uguaglianza di stringhe non possiamo usare `A==B`, perché questa espressione riguarda l'uguaglianza degli indirizzi in cui si trovano le due stringhe, una condizione molto più forte.

Definiamo adesso una funzione `Tui` (uguaglianza iniziale di testi) di due stringhe che restituisce 1 o 0 a seconda che la prima stringa è sottostringa della seconda o no.

```
int Tui (Char A, Char B)
{for (;*A;A++,B++) if (*A!=*B) return 0;
return 1;}
```

`Tu` (nella prima versione) e `Tui` si distinguono solo nell'ultima riga. Anche qui potremme usare le funzioni della libreria standard:

```
int Tui (Char A, Char B)
{return strncmp(A,B,strlen(A))==0;}
```

oppure, più veloce,

```
int Tui (Char A, Char B)
{return strstr(B,A)==B;}
```

Per sicurezza, all'inizio di `Tu` e `Tui` sarebbe bene inserire

```
if (!A||!B) return 0;
```

cosicché le versioni finali diventano

```
int Tu (Char A, Char B)
{if (!A||!B) return 0;
return strcmp(A,B)==0;}

int Tui (Char A, Char B)
{if (!A||!B) return 0;
return strstr(B,A)==B;}
```

Andare alla fine di una stringa

Per portare un puntator `X` alla fine di una stringa `A`, in modo che `X` punti sul carattere `0` finale, si può usare l'istruzione

```
for (X=A;*X;X++);
```

oppure la funzione di libreria `strchr` che abbiamo già incontrato alle pagine 15 (rappresentazione binaria) e 16:

```
X=strchr(A,0);
```

Input da tastiera con fgets

Per l'input di una stringa dalla tastiera in casi semplici si può usare la funzione `gets`:

```
char a[40];
gets(a);
```

Il compilatore ci avverte pero che

```
the 'gets' function is dangerous
and should not be used.
```

Infatti se l'utente immette più di 40 caratteri (per disattenzione o perché vuole danneggiare il sistema), scriverà su posizioni non riservate della memoria. Nei nostri esperimenti ciò non sarebbe un problema, ma è importante in programmi che verranno usati da utenti poco esperti o malintenzionati. Si preferisce perciò la funzione `fgets` che viene usata con questa sintassi:

```
char a[40];
fgets(a,38,stdin);
```

In questo caso nell'indirizzo `a` vengono scritti al massimo 38 caratteri; `stdin` è lo *standard input*, cioè in genere la tastiera; `fgets` può ricevere il suo input anche da altri files.

A differenza da `gets` il comando `fgets` inserisce nella stringa anche il carattere `'\n'` che termina l'input e ciò è un po' scomodo. Definiamo quindi una nostra funzione per l'immissione di dati dalla tastiera:

```
void Input (Char A, int n)
{if (n<1) n=1; fgets(A,n+1,stdin);
A=strchr(A,0)-1;
if (*A=='\n') *A=0;}
```

Naturalmente nella stringa `A` deve essere stato riservato spazio sufficiente come in

```
char a[40]; Input(a,38);
```

Creare un menu

Assumiamo che abbiamo definito funzioni `parole`, `fattoriali`, `binomiali` e `prove` che il programma ci dovrebbe permettere di scegliere dalla tastiera. Possiamo a questo scopo riscrivere la `main` nel modo seguente:

```
int main ()
{char a[50];
for (;;) {printf("\nScelta: ");
Input(a,40);
if (Tu(a,"fine")) goto fine;
if (Tu(a,"prove")) prove(); else
if (Tu(a,"bin")) binomiali(); else
if (Tu(a,"fatt")) fattoriali(); else
if (Tu(a,"parole")) parole();}
fine: exit(0);}
```

atoi, atol e atof

Nelle operazioni di input si usano spesso le funzioni `atoi`, `atol` e `atof` che convertono una stringa in un numero risp. di tipo `int`, `long` e `double`. Bisogna includere il header `<stdlib.h>`.

```
int n; double x;
n=atoi("3452"); x=atof("345.200");
```

La seguente funzione chiede ripetutamente l'input di n e stampa $n!$, terminando quando l'input è vuoto.

```
void Prova ()
{char a[40]; int n;
while (1) {printf("n: "); Input(a,35);
if (Tu(a,"")) return; n=atoi(a);
printf("%.0f\n",Fatt(n));}}
```

La funzione `Fatt` per il calcolo del fattoriale è stata definita a pagina 1.

Si noti che per (circa) $n \geq 23$ le cifre previste per il formato `%f` non sono più sufficienti; anche con il tipo `double` del C non riusciamo più a calcolare e rappresentare correttamente numeri così grandi. Si possono usare il programma *calc* oppure la libreria *gmp* oppure, se installato, *Singular*, un ottimo sistema per la geometria algebrica computazionale.

Per visualizzare i numeri binomiali (pagina 16) è sufficiente una leggera modifica del programma:

```
void Prova ()
{char a[40]; int n,k;
while (1) {printf("n: "); Input(a,35);
if (Tu(a,"")) return; n=atoi(a);
printf("k: "); Input(a,35);
if (Tu(a,"")) return; k=atoi(a);
printf("%.0f\n",Bin(n,k));}}
```

Invertire una parola

La seguente funzione chiede (ripetutamente, fino a quando non immettiamo la parola vuota) una parola, conferma la parola impostata e visualizza la parola che si ottiene dall'originale invertendone la successione dei caratteri e usando solo lettere minuscole.

```
void Invertiparola()
{char parola[70],inversa[70]; Char X,Y;
for (;;) {printf
  ("\nQuale parola vuoi invertire? ");
Input(parola,60);
if (Tu(parola,"")) break;
printf("La parola originale e' %s.\n",
  parola);
for (X=strchr(parola,0)-1,Y=inversa;
  X>=parola;X--,Y++) *Y=*X; *Y=0;
for (Y=inversa;*Y;Y++) *Y=tolower(*Y);
printf
  ("Invertita diventa %s.\n",inversa);}}
```

Si osservi che `X` nel primo `for` viene inizialmente posto a puntare sull'ultimo carattere della parola con `X=strchr(parola,0)-1;`. La penultima riga trasforma tutte le lettere della stringa invertita in minuscole, utilizzando la funzione `tolower` che richiede il header `ctype.h` come la sua gemella `toupper` che converte un carattere in maiuscola.

Per controllare se una stringa `A` è vuota, invece di `if (Tu(A,""))` si può usare anche `if (!*A)`.

Alcuni caratteri speciali

Conosciamo già `'\n'`, il carattere di nuova riga. Altri caratteri speciali sono il tabulatore `'\t'`, il backslash `\\`, la virgoletta `\"`, l'apostrofo `'\''`, il carattere ASCII 0 che può essere scritto nella forma `'\0'`. Il codice ASCII dello spazio è 32, quello del carattere *escape* 27. Come il carattere di nuova riga anche gli altri caratteri con backslash possono essere inseriti in stringhe esplicite; per l'apostrofo non è necessario il backslash quando si trova tra virgolette. Esempi:

```
int n;
Char A="\0abc\t\0de\\";
Char B="Il codice e' 'AXE36K'.\n";
for (n=0;n<9;n++) printf("%d ",A[n]);
printf("\n%s",B);
```

con output

```
0 97 98 99 9 0 100 101 92
Il codice e' 'AXE36K'.
```

sprintf ed snprintf

La funzione `sprintf` (che abbiamo incontrate alle pagine 8 e 15) è molto simile nella sintassi alla funzione `printf`, da cui si distingue per un argomento in più che precede i tipici argomenti di `printf`. Il primo argomento è un puntatore a caratteri e la chiamata della funzione fa in modo che i caratteri che con `printf` verrebbero scritti sullo schermo vengano invece scritti nell'indirizzo che corrisponde a quel puntatore.

Ci sono due usi principali di questa funzione. Da un lato può essere utilizzata per creare delle copie di stringhe, dall'altro può servire per preparare una stringa per una successiva elaborazione, ad esempio per un output grafico che non utilizza `printf`:

```
char a[200];
sprintf(a,"Il valore e' %.2f",x);
scrivinellafinestra(f,a);
```

dove immaginiamo che l'ultima istruzione effettua una visualizzazione della stringa `a` nella finestra `f`.

Quando non è sicuro che la stringa che viene posta nell'indirizzo definito dal primo argomento di `sprintf` non superi nella sua lunghezza lo spazio disponibile, si può usare la funzione `snprintf` che ha come secondo argomento il numero massimale di bytes (compreso il carattere `0` finale) che vengono trasferiti. Esempio:

```
char a[100];
snprintf(a,4,"alfabeto");
printf("%s\n",a);
```

con output `alf`.

`sprintf` e `snprintf` non devono essere utilizzate quando la stringa da trasferire si sovrappone almeno in parte con la stringa di destinazione. In questo caso il comportamento delle funzioni è indefinito e in alcune implementazioni ne può risultare anche un grave errore di memoria.

Copiare una stringa

La funzione `sprintf` può essere usata per copiare una stringa, bisogna però stare attenti a due cose: In primo luogo la parola da copiare non deve contenere caratteri che possono essere interpretati come caratteri di formattazione, quindi `\`, `%` ecc. Inoltre la copia deve essere disgiunta in memoria dall'originale, come abbiamo già osservato.

Per effettuare una copia sicura di una stringa `B` in una stringa `A` si può usare l'istruzione

```
memmove(A,B,strlen(B)+1);
```

che usa la funzione `memmove` discussa a pagina 13. Il vantaggio di `sprintf` e `snprintf` è comunque che permettono l'inserimento di parti variabili nella stringa da copiare.

vprintf

La funzione `printf` può essere utilizzata con un numero variabile di argomenti che però, quando la impieghiamo, ci deve essere noto. Come facciamo allora se la vogliamo usare all'interno di una funzione che riceve essa stessa un numero variabile di argomenti? Assumiamo ad esempio che vogliamo creare una funzione con il prototipo

```
void messaggio (int i, Char Formato, ...)
```

che a seconda del valore di `i` stampa un messaggio diverso che dipende dagli argomenti successivi. In altre parole l'idea è quella di una funzione

```
// Non puo funzionare.
void Stampa (int i, Char Formato, ...)
{if (i==1) {printf("Windows: ");
printf(Formato,...);} else
{printf("Linux: ");
printf(Formato,...);}}
```

Ma così non può funzionare perché i puntini `...` nella prima riga non possono essere usati come il nome di una variabile nel seguito.

Esiste una funzione apposita però, molto utile nella programmazione avanzata, la funzione `vprintf` (la `v` nel nome probabilmente viene da *variable*); si usa con questa sintassi:

```
void Stampa (int i, Char Formato, ...)
{va_list arg;
va_start(arg,Formato);
if (i==1) {printf("Windows: ");
vprintf(Formato,arg);} else
{printf("Linux: ");
vprintf(Formato,arg);}
va_end(arg);}
```

con la prova

```
int n=4; Char Tipo="SATO";
Stampa(1,"%s\n",Tipo);
Stampa(2,"%s %d\n",Tipo,n);
```

e l'output

```
Windows: SATO
Linux: SATO 4
```

vsprintf e vsnprintf

Ancora più utile di `vprintf` sono le funzioni `vsprintf` e `vsnprintf` che funzionano nello stesso modo come `vprintf` ma inserendo, come `sprintf`, la stringa formattata in un indirizzo invece di visualizzarla sullo schermo. Il secondo argomento di `vsnprintf` serve di nuovo per indicare il numero massimo di bytes trasferiti.

Creiamo una funzione per la concatenazione di un numero variabile di stringhe; naturalmente anche qui bisogna verificare che ci sia abbastanza spazio, che le stringhe non contengano caratteri di formattazione e che non si sovrappongano.

```
void Concatena (Char U, Char Formato, ...)
{va_list arg;
va_start(arg,Formato);
vsprintf(U,Formato,arg); va_end(arg);}
```

La proviamo con

```
void Prova ()
{char a[100];
Concatena(a,"%s-%s","alfa","beta");
puts(a);
Char A="Ferrara";
Concatena(a,"La lunghezza di %s e' %d.",
  A,strlen(A));
puts(a);}
```

ottenendo l'output

```
alfa-beta
La lunghezza di Ferrara e' 7.
```

La nostra funzione `Concatena` può essere usata anche per creare delle nuove stringhe con le quali poi possiamo eseguire altre operazioni e non solo quelle di stampa.

Similmente per poter eseguire comandi nel linguaggio grafico *Tcl/Tk* che consistono di stringhe formattate creiamo la funzione

```
void Tcl (Char Formato, ...)
{va_list arg; static char com[1000];
va_start(arg,Formato);
vsprintf(com,Formato,arg); va_end(arg);
Tcl_Eval(com);}
```

La funzione verrebbe poi utilizzata ad esempio in questo modo:

```
Tcl("%s configure %s -foreground %s",
  Finestra,Elemento,Colore);
```

Una versione sicura di Tcl

La funzione `vsprintf` presenta comunque alcuni difetti già menzionati. Una funzione sicura utilizza direttamente la tecnica delle funzioni con un numero variabile di argomenti, con due chiamate di `va_start`:

```
void Tcl (Char A, ...)
{va_list arg; Char X; int p,m;
va_start(arg,A);
for (p=0,X=A;X;)
{p+=strlen(X); X=va_arg(arg,Char);}
char b[p+1]; va_start(arg,A);
for (p=0,X=A;X;)
{m=strlen(X); memmove(b+p,X,m);
p+=m; X=va_arg(arg,Char);}
b[p]=0; Tcl_Eval(Xtvi,b); va_end(arg);}
```

CORSO DI PROGRAMMAZIONE

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 9 ◇ 15 Novembre 2004

Le funzioni per le stringhe del C

Le librerie standard del C prevedono numerose funzioni per il trattamento di stringhe che bisogna conoscere. Spesso non sarebbe difficile creare funzioni apposite simili, ma le funzioni standard sono ottimizzate e, se si conosce il loro funzionamento, affidabili. Esse richiedono il header `<string.h>`. *Le più importanti di queste funzioni sono:*

1. *funzione per calcolare la lunghezza di una stringa:* ***strlen***.
2. *funzioni per il concatenamento di stringhe:* ***strcat*** *e* ***strncat***.
2. *funzioni per la copia di stringhe:* ***strcpy*** *e* ***strncpy***.
2. *funzioni per il confronto di stringhe:* ***strcmp*** *e* ***strncmp***.
2. *funzioni per la ricerca di un singolo carattere in una stringa:* ***strchr*** *e* ***strrchr***.
3. *funzioni per la ricerca in una stringa di un carattere contenuto in un insieme di caratteri:* ***strspn***, ***strcspn*** *e* ***strpbrk***.
1. *funzione per la ricerca di una stringa in una stringa:* ***strstr***.
1. *funzione per la separazione di una stringa in sottostringhe delimitate da separatori:* ***strtok***.

Nel seguito daremo i prototipi di queste funzioni, utilizzando però le abbreviazioni per i puntatori definite a 13 e tralasciando le indicazioni `const` *dei prototipi originali.*

strlen, strcat, strncat, strcmp e strncmp

```
size_t strlen (Char A)
```

Questa funzione restituisce il numeri dei caratteri della stringa `A`, senza contare il carattere 0 finale. La stringa vuota ha lunghezza zero.

Abbiamo incontrato questa funzione già alle pagine 9, 13, 15, 23-25. Un altro esempio:

```
printf("%d %d\n",strlen(""),
  strlen("John\nBob\n"));
```

con output `0 9`.

Per il concatenamento di stringhe si possono usare le funzioni

```
Char strcat (Char A, Char B);
Char strncat (Char A, Char B,
  size_t n);
```

L'istruzione `strcat(A,B);` fa in modo che `A` diventi uguale alla concatenazione delle stringhe `A` e `B`; in altre parole `B` viene copiata nell'indirizzo occupato dal carattere 0 finale di `A`.

Bisogna stare attenti che per `A` sia riservato sufficiente spazio. In particolare è un grave errore l'istruzione `strcat("alfa","beta");`. Esempio di uso corretto:

```
char a[100]="alfa";
strcat(a,"beta"); puts(a);
```

con output `alfabeta`.

La funzione restituisce come risultato superfluo il primo argomento.

Attenzione: Le stringhe `A` e `B` non si devono sovrapporre in memoria; l'istruzione `strcat(A,A+4);` è quasi certamente un errore, se `A` consiste di più di 3 caratteri.

`strncat(A,B,n);` funziona nello stesso modo (e richiede le stesse precauzioni), ma aggiunge solo i primi `n` caratteri di `B` ad `A`, mettendo, quando necessario, un carattere 0 alla fine della nuova stringa:

```
char a[100]="alfa";
strncat(a,"012345",3); puts(a);
```

con output `alfa012`. Le istruzioni `strcat(A,B);` e `strncat(A,B,n);` sono equivalenti se $n \geq$`strlen(B)`.

Le funzioni per il confronto di stringhe `strcmp` e `strncmp` hanno i prototipi

```
int strcmp (Char A, Char B);
int strncmp (Char A, Char B, size_t n);
```

Le abbiamo già incontrate a pagina 24.

`strcmp(A,B)` confronta alfabeticamente `A` con `B` e sostituisce un intero maggiore di 0 (normalmente 1) se `A` è alfabeticamente maggiore di `B`, altrimenti 0 se le due stringhe sono uguali, e un intero minore di 0 (normalmente -1) se `A` è alfabeticamente minore di `B`.

`strncmp(A,B,n)` equivale a `strcmp(A1,B1)` dove, posto

```
n1=min(n,strlen(A),strlen(B)),
```

con `A1` e `B1` denotiamo le stringhe che consistono dei primi `n1` caratteri di `A` e `B`.

In particolare vediamo che `A` è inizio di `B` se e solo se `strncmp(A,B,strlen(A))==0`. Si noti che in questo caso il carattere 0 finale di `A` non conta più.

In questo numero

strcpy e strncpy

```
Char strcpy (Char A, Char B);
Char strncpy (Char A, Char B, size_t n);
```

`strcpy(A,B);` copia `B` in `A`, `strncpy(A,B,n);` copia i primi `n` caratteri di `B` in `A`; se la lunghezza di `B` è minore di `n`, i caratteri mancanti vengono considerati uguali a 0; se invece `n` è $\leq$ della lunghezza di `B`, allora non viene trasferito uno 0 e la nuove fine di `A` è il primo 0 che si incontra a partire da `A`. Le funzioni restituiscono come risultato superfluo il primo argomento. Esempio:

```
void Provastrcpy ()
{char a[100]="01234";

strcpy(a,"abc"); puts(a);
// output: abc

strcpy(a,"012345"); puts(a);
// output: 012345

strncpy(a,"abcde",3); puts(a);
// output: abc345

strcpy(a,"012345"); puts(a);
// output: 012345

strncpy(a,"abc",7); puts(a);
// output: abc

strcpy(a,"012");
strcpy(a+4,"456789"); puts(a);
// output: 012

strncpy(a,"abcdefg",5); puts(a);
// output: abcde56789

strcpy(a,"abcde"); puts(a);}
//output: abcde
```

Anche in questo caso bisogna riservare spazio sufficiente per `A`; l'uso di queste funzioni costituisce un errore, se `A` e `B` si sovrappongono in memoria. Useremo quindi `memmove` in tal caso (cfr. pagine 13 e 25).

Quando le due stringhe invece non si sovrappongono, spesso si preferirà `sprintf` che permette anche di trasferire stringhe formattate; useremo `strcpy` e `strncpy` quando le stringhe possono contenere caratteri di formattazione (ad esempio `%`) che devono essere copiati così come sono e non interpretati da `sprintf`.

strchr e strrchr

Queste funzioni hanno i prototipi

```
Char strchr (Char A, int x);
Char strrchr (Char A, int x);
```

`strchr(A,x)` restituisce il puntatore `NULL` se `x` (considerato come carattere) non appare in `A` (cioè nella stringa che corrisponde ad `A`), compreso il carattere 0 finale; altrimenti restituisce un puntatore al primo `x` in `A` (quindi al carattere 0 finale, se `x` è uguale a 0, come abbiamo già visto alle pagine 14-16 e 24).

`strrchr` (la *r* sta per *reverse*) è simile, inizia la ricerca però dalla fine della stringa. Più precisamente `strrchr(A,x)` è uguale a `NULL`, se `x` non appare in `A`, altrimento è un puntatore all'ultimo `x` in `A`.

```
char a[100]; Char X,Y;
strcpy(a,"01234267");

X=strchr(a,0); Y=strrchr(a,0);
printf("%d %d\n",X-a,Y-a);
// output: 8 8

X=strchr(a,'2'); Y=strrchr(a,'2');
printf("%d %d\n",X-a,Y-a);
// output: 2 5
```

Esercizio: Analizzare questa piccola funzione:

```
int Poscar (Char A, int x)
{Char P=strchr(A,x);
if (!P) return -1;
return P-A;}
```

Usando `strchr` per determinare se un carattere appare in una stringa, possiamo definire una funzione che permette di inserire una stringa, da cui verranno eliminati tutti i caratteri che appaiono in una seconda stringa, anch'essa impostata dalla tastiera.

```
void Eliminacaratteri ()
{char a[100],b[100]; Char X,Y;
for (;;)
{printf("\nInserisci la parola: ");
Input(a,80); if (!*a) return;
printf("\nInserisci i caratteri"
  "da eliminare: ");
Input(b,80);
for (X=Y=a;*X;X++)
if (!strchr(b,*X)) *(Y++)=*X; *Y=0;
printf("\n%s\n",a);}}
```

La seguente funzione sostituisce invece ogni carattere di `L` in `A` con `x`:

```
void Sostituisci (Char A, Char L, int x)
{for (;*A;A++)
if (strchr(L,*A)) *A=x;}
```

---

strstr

Questa funzione che abbiamo usato nella funzione `Tui` a pagina 24, ha il prototipo

```
Char strstr(Char A, Char B);
```

e viene utilizzata per cercare una sottostringa in una stringa.

`strstr(A,B)` è uguale al puntatore `NULL`, se `B` non è sottostringa di `A`; altrimenti è uguale al puntatore alla prima apparizione di `B` in `A`. Esempio:

```
// Posizione di A in B.
int Pos (Char A, Char B)
{Char X=strstr(B,A);
if (!X) return -1; return X-B;}
```

---

strpbrk

```
Char strpbrk (Char A, Char L);
```

`strpbrk(A,L)` restituisce il puntatore `NULL`, se `A` non contiene caratteri di `L`; altrimenti l'istruzione restituisce un puntatore al primo carattere di `L` in `A`.

Qui, come anche in `strspn`, `strcspn` e `strtok`, il secondo argomento `L` funge da lista di caratteri da considerare.

```
strpbrk(A,"m")
```

è quindi uguale a

```
strchr(A,'m')
```

---

strspn e strcspn

Queste funzioni hanno i prototipi

```
size_t strspn (Char A, Char L);
size_t strcspn (Char A, Char L);
```

`strspn(A,L)` restituisce la lunghezza del segmento iniziale di `A` che consiste di caratteri di `L` (e quindi la lunghezza di `A` se tutti i caratteri di `A` appartengono ad `L`).

`strcspn` funziona in modo complementare a `strspn`, più precisamente `strcspn(A,L)` è la lunghezza del segmento iniziale di `A` che non consiste di carattere di `L` (e quindi la lunghezza di `A` se nessun carattere di `A` appartiene ad `L`).

Quando `A` contiene caratteri di `L`, le due istruzioni

```
X=strpbrk(A,L);
```

e

```
X=A+strcspn(A,L);
```

sono equivalenti; se invece `A` non contiene caratteri di `L`, allora `strpbrk(A,L)` è il puntatore nullo, mentre `A+strcspn(A,L)` punta al carattere 0 finale di `A`.

`strspn` e `strcspn` vengono spesso usate per separare parole in un testo.

Nell'esempio assumiamo di sapere che il testo (la stringa `A`) contiene due parole separate da caratteri `+` o `-` che possono apparire anche all'inizio o alla fine del testo. Vogliamo determinare le due parole.

```
Char Prima,Seconda,Fine;
Char Separatori="+-";
Char A="-+Romagna+++bella--";
Prima=A+strspn(A,Separatori);
Fine=Prima+strcspn(Prima,Separatori);
*Fine=0; Fine++;
Seconda=Fine+strspn(Fine,Separatori);
Fine=Seconda+
  strcspn(Seconda,Separatori);
*Fine=0;
printf("[%s]\n[%s]\n",Prima,Seconda);
```

con output

```
[Romagna]
[bella]
```

strtok

La funzione `strtok` può essere usata per suddividere una stringa in sottostringhe. È un po' difficile nell'applicazione e spesso si può usare un metodo apposito più trasparente utilizzando le altre funzioni finora incontrate. Infatti, la `strtok` stessa è implementata usando più volte `strspn` e `strpbrk`.

Il prototipo della funzione è

```
Char strtok (Char A, Char L);
```

Anche qui la stringa `L` serve come contenitore di caratteri. Tipicamente `strtok` viene chiamata più volte e ogni volta `L` può essere diversa.

La stringa che si vuole suddividere viene modificata durante le operazioni! Perciò, se la si vuole utilizzare più tardi con il valore originale, bisogna creare una copia. Infatti `strtok` pone uguale a 0 i caratteri separatori contenuti in `L`.

`strtok` utilizza un puntatore interno `P` di classe `static` (pagina 15) che viene cambiato in ogni esecuzione della funzione e utilizzato nell'esecuzione successiva. In questo modo il puntatore può avanzare dalla posizione in cui si trovava la volta precedente.

Per chi fosse interessato, rimandiamo a pagina 69 del corso di Sistemi 2001/02 per una spiegazione più dettagliata del funzionamento di `strtok`.

Tipo di un carattere

Con le seguenti funzioni si possono determinare alcune proprietà di un carattere, considerato come intero.

```
int isalpha (int x);
int isdigit (int x);
int isalnum (int x);
int iscntrl (int x);
int isprint (int x);
int isxdigit (int x);
int isspace (int x);
int islower (int x);
int isupper (int x);
```

Esse sono definite come segue:

`isalpha(x)`	... $x \in$ {'A','B',...,'Z','a','b',...,'z'} ... (non ne fa parte '_')
`isdigit(x)`	... $x \in$ {'0','1',...,'9'}
`isalnum(x)`	... `isalpha(x)` oppure `isdigit(x)`
`iscntrl(x)`	... $0 \le x \le 31$ oppure $x = 127$
`isprint(x)`	... `iscntrl(x)`==0 (normalmente)
`isxdigit(x)`	... $x \in$ {'0','1',...,'9','A',...,'F','a',...,'f'}
`isspace(x)`	... $x \in$ {' ','\t','\r','\n','\v','\f'}
`islower(x)`	... $x \in$ {'a','b',...,'z'}
`isuppper(x)`	... $x \in$ {'A','B',...,'Z'}

`'\r'` è il ritorno di carrello, `'\v'` il tabulatore verticale, `'\f'` il carattere di nuova pagina.

La seguente funzione crea in `B` la stringa che si ottiene da `A` eliminando tutti i caratteri di controllo (compreso il carattere 127, DEL) e restituisce come valore il numero dei caratteri che non sono stati trascritti.

```
int Eliminacntrl (Char A, Char B)
{int k;
for (k=0;*A;A++) if (isprint(*A))
*(B++)=*A; else k++; return k;}
```

Per trasformare minuscole in maiuscole e viceversa si utilizzano `tolower` e `toupper` che conosciamo dalla pagina 24.

Modulo di un numero complesso

Nel calcolo del valore assoluto di un numero complesso (cfr. pagina 14) dobbiamo evitare la formazione di risultati intermedi troppo grandi (che vengono approssimati male al calcolatore). Sia dato un numero complesso $z = x + iy$.

Intuitivamente $|z| = \sqrt{x^2 + y^2}$, ma il risultato intermedio $x^2 + y^2$ diventa molto più grande del risultato finale. Si usano quindi le formule

$$|z| = |x|\sqrt{1 + \left(\frac{y}{x}\right)^2}$$

(usata per $|y| \leq |x| \neq 0$)

$$|z| = |y|\sqrt{\left(\frac{x}{y}\right)^2 + 1}$$

(usata per $|x| \leq |y| \neq 0$)

che portano alla seguente funzione, in cui usiamo la funzione `fabs` (vista a pagina 19) per il calcolo del valore assoluto di un numero reale:

```
double Ncva (nc z)
{double vax,vay,t;
if (z.x==0) return fabs(z.y);
if (z.y==0) return fabs(z.x);
vax=fabs(z.x); vay=fabs(z.y);
if (vax>=vay)
{t=z.y/z.x; return vax*sqrt(1+t*t);}
t=z.x/z.y; return vay*sqrt(1+t*t);}
```

Prova:

```
nc z={3,5}, w={5,3};
printf("%.2f %.2f\n",
  Ncva(z),Ncva(w));
```

con output `5.83 5.83`. Il risultato è corretto perché $\sqrt{(34)} = 5.83\ldots$.

Quoziente di numeri complessi

Per il quoziente

$$\frac{x+iy}{x'+iy'} = \frac{(x+iy)(x'-iy')}{x'^2+y'^2} = \frac{xx'+yy'+i(yx'-xy')}{x'^2+y'^2}$$

(se $x'^2 + y'^2 \neq 0$) si procede in modo analogo. L'ultima frazione può essere scritta come

$$\frac{x + y\dfrac{y'}{x'} + i\left(y - x\dfrac{y'}{x'}\right)}{x' + y'\dfrac{y'}{x'}}$$

(usata per $|y'| \leq |x'| \neq 0$)

$$\frac{x\dfrac{x'}{y'} + y + i\left(y\dfrac{x'}{y'} - x\right)}{x'\dfrac{x'}{y'} + y'}$$

(usata per $|x'| \leq |y'| \neq 0$)

Possiamo quindi programmare la divisione in questo modo:

```
nc Ncdiv (nc z1, nc z2)
{double q,t; nc w;
if (fabs(z2.x)>=fabs(z2.y))
{q=z2.y/z2.x; t=z2.x+z2.y*q;
w.x=(z1.x+z1.y*q)/t;
w.y=(z1.y-z1.x*q)/t;
return w;}
q=z2.x/z2.y; t=z2.x*q+z2.y;
w.x=(z1.x*q+z1.y)/t;
w.y=(z1.y*q-z1.x)/t;
return w;}
```

Prova:

```
nc z={3,5}, w={1,3}, q1,q2;
q1=Ncdiv(z,w); q2=Ncdiv(w,z);
printf("%.2f %.2fi, %.2f %.2fi\n",
  q1.x,q1.y,q2.x,q2.y);
```

L'output

```
1.80 -0.40i, 0.53 0.12i
```

è corretto.

L'esponenziale complesso

Per la funzione esponenziale e le funzioni trigonometriche complesse usiamo, per

$$z = x + iy$$

le relazioni

$$\begin{aligned} e^z &= e^{x+iy} \\ &= e^x(\cos y + i \sin y) \\ e^{iz} &= e^{-y+ix} \\ &= e^{-y}(\cos x + i \sin x) \\ e^{-iz} &= e^{y-ix} \\ &= e^{y}(\cos x - i \sin x) \\ \cos z &= \frac{e^{iz} + e^{-iz}}{2} \\ &= \frac{e^y + e^{-y}}{2}\cos x - i\frac{e^y - e^{-y}}{2}\sin x \\ &= \cosh y \cos x - i \sinh y \sin x \\ \sin z &= \frac{e^{iz} - e^{-iz}}{2i} \\ &= \frac{e^y + e^{-y}}{2}\sin x + i\frac{e^y - e^{-y}}{2}\cos x \\ &= \cosh y \sin x + i \sinh y \cos x \end{aligned}$$

Definiamo quindi le seguenti funzioni in C:

```
nc Ncexp (nc z)
{double ex=exp(z.x);
nc w={ex*cos(z.y),ex*sin(z.y)};
return w;}

nc Nccos (nc z)
{nc w={cosh(z.y)*cos(z.x),
  -sinh(z.y)*sin(z.x)};
return w;}

nc Ncsin (nc z)
{nc w={cosh(z.y)*sin(z.x),
  sinh(z.y)*cos(z.x)};
return w;}
```

Si noti la tecnica di assegnazione abbreviata a una struttura (o a un vettore) mediante inizializzazione.

Funzioni per i files

Queste funzioni utilizzano le funzioni a basso livello di Unix. Per una piena comprensione dobbiamo rimandare al trattato di Stevens.

```
// Aggiunta di un testo
// al file di nome A che
// deve gia' esistere.
void Aggiungifile (Char A, Char Testo)
{int fk; fk=open(A,O_WRONLY|O_APPEND);
write(fk,Testo,strlen(Testo)); close(fk);}

// Lettura del file di nome A
// nell'indirizzo B.
void Leggifile (Char A, Char B)
{int fk; int n;
fk=open(A,O_RDONLY);
if (fk<0) {*B=0; return;}
n=Lunghezzafile(A); read(fk,B,n);
close(fk); B[n]=0;}

// Scrittura di un testo
// nel file di nome A.
void Scrivifile (Char A, Char Testo)
{int fk;
fk=open(A,O_WRONLY|O_CREAT|
  O_TRUNC|O_SYNC,S_IRUSR|
  S_IWUSR|S_IRGRP|S_IROTH);
write(fk,Testo,strlen(Testo)); close(fk);}

// Lunghezza del file di nome A.
// Il risultato e' -1, se il file non
// esiste o non e' un file regolare.
int Lunghezzafile (Char A)
{struct stat att; int e;
e=stat(A,&att); if (e<0) return -1;
if (att.st_mode && S_IFREG)
return att.st_size; return -1;}
```

Per semplicità abbiamo usato il tipo `int` invece di `size_t` per il risultato di `Lunghezzafile` e quindi anche per la variabile `n` in `Leggifile`.

Per leggere un file di lunghezza sconosciuta si determinerà prima la sua lunghezza `n` con `Lunghezzafile`, per definire successivamente un vettore di caratteri di lunghezza `n` (oppure, per sicurezza, `n+4`):

```
Char Nome="Dati";
int n=Lunghezzafile(Nome);
char a[n+4]; Leggifile(Nome,a);
puts(a);
```

Per cambiare il nome di un file si può usare la funzione `rename`, per cancellare un file `remove`, per cambiare la directory di lavoro `chdir`. I prototipi sono

```
int rename (Char Vecchio, Char Nuovo);
int remove (Char Nome);
int chdir (Char Nome);
```

Questi non sono comandi del C, ma validi sotto Unix.

Si può chiamare la shell di Unix da un programma in C con la funzione `system`:

```
system("clear");
system("rm -i lettera");
system("ls -l > alfa");
```

W. Stevens: Advanced programming in the Unix environment. Addison-Wesley 1992.

D. Curry: Unix systems programming. O'Reilly 1996.

Sistemi di Lindenmayer

Sia A un insieme. Il *monoide libero* generato da A è l'insieme

$$A^* := \bigcup_{n=0}^{\infty} A^n$$

Denotiamo con ε l'unico elemento di A^0. Poniamo $A^+ := A^* \setminus \{\varepsilon\}$.

Gli elementi di A^* si chiamano *parole* sull'alfabeto A e siccome $A^n \cap A^k = \emptyset$ per $n \neq k$, per ogni $v \in A^*$ esiste esattamente un $n \in \mathbb{N}$ tale che $v \in A^n$; questo n si chiama la *lunghezza* di v e viene denotato con $|v|$. In particolare $|\varepsilon| = 0$; ε si chiama la parola vuota.

Se usiamo la concatenazione di parole come composizione, A^* diventa un monoide, altamente noncommutativo.

T sia un monoide e $\varphi_0 : A \longrightarrow T$ un'applicazione qualsiasi. Allora esiste esiste un unico omomorfismo di monoidi $\varphi : A^* \longrightarrow T$ che su A coincide con φ_0.

Questo teorema fondamentale è stato dimostrato nel corso di Algoritmi.

Gli endomorfismi di A^* (cioè gli omomorfismi di monoidi $A^* \longrightarrow A^*$) sono noti anche come *sistemi di Lindenmayer*.

Aristid Lindenmayer (1925-1989) era un botanico olandese che utilizzò questi sistemi per descrivere (soprattutto in modo grafico) ed analizzare l'accrescimento di piante. Un endomorfismo di A^* può essere considerato come un meccanismo di *riscrittura*; l'idea è di imitare un principio generale della natura:

organismo semplice iniziale + leggi

↓

organismo complesso

Un semplicissimo, quanto antico, esempio di riscrittura è il *fiocco di neve* di Koch (1905), discusso nel corso di Algoritmi: Si parte da un elemento *iniziatore*, il triangolo equilatero, e da un elemento *generatore* (che contiene le leggi), costituita da una linea spezzata orientata che consta di quattro parti della stessa lunghezza; quindi si sostituisce ogni lato del triangolo iniziatore con una riga del generatore, ridotta in modo tale (se si vuole che lo spazio occupato dalla figura rimanga lo stesso) da avere gli estremi coincidenti con quelli del segmento da sostituire. Iterando questo procedimento si perviene ad un'immagine che assomiglia a un fiocco di neve. Per il disegno sul calcolatore naturalmente bisognerà fermarsi dopo un certo numero di iterazioni, matematicamente si può anche considerare il limite delle figure ottenute, ad esempio rispetto a una metrica (*metrica di Hausdorff*) sull'insieme dei sottoinsiemi compatti non vuoti di $\mathbb{R}^2$.

È importante che nei sistemi di Lindenmayer la riscrittura avviene *in parallelo*, cioè le regole vengono applicate simultaneamente ad ogni carattere di una data parola, a differenza da quanto accada nei linguaggi di Chomsky (usati spesso per descrivere i linguaggi di programmazione).

Elenchiamo alcune delle principali applicazioni dei sistemi di Lindenmayer.

Da un lato questi sistemi possono essere impiegati per simulare l'accrescimento di un organismo o di un intero sistema ecologico e per analizzarne i meccanismi di crescita. Si possono così individuare i parametri che determinano l'evoluzione di un organismo o di un ecosistema.

Due campi dove più intensamente si impiegano piante virtuali sono il cinema e i giochi al calcolatore, dove vengono usate in scene esterne, in effetti speciali, nella simulazione di paesaggi che possono essere esplorati interattivamente.

Sistemi di Lindenmayer possono essere usati per la memorizzazione economica di immagini. Infatti, invece di dover conservare tutto il contenuto di una parte intera dello schermo (ad esempio 600x600 pixel = 360000 bit = 45000 byte per un'immagine in bianco-nero) è sufficiente conservare la stringa che rappresenta l'iniziatore (ad esempio 50 byte) e le stringhe che contengono le leggi di crescita (ad esempio 20x30 byte = 600 byte), quindi in tutto 650 invece di 45000 byte.

La successione di Morse

Questa successione è forse il più noto esempio di un sistema di Lindenmayer. Essa compare sotto molte vesti nella *dinamica simbolica* (lo studio delle periodicità e quasiperiodicità di parole infinite, cioè di elementi di $A^{\mathbb{N}}$ o $A^{\mathbb{Z}}$). Infatti la successione di Morse è la più semplice successione quasiperiodica, ma non periodica. Essa è definita nel modo seguente:

$A = \{0, 1\}$
generatore: 0
leggi: $0 \longmapsto 01$, $1 \longmapsto 10$

Quindi la successione si sviluppa in questo modo:

0
01
0110
01101001
0110100110010110
...

Si vede che la successione può essere generata anche in altri modi, ad esempio aggiungendo alla successione ottenuta al passo precedente la successione che si ottiene da essa scambiando 1 con 0. Vediamo in particolare che la successione si allunga sempre senza mai cambiare nelle parti costruite negli stadi precedenti.

La dinamica simbolica viene classicamente e ancora oggi utilizzata nello studio di sistemi dinamici. Immaginiamo infatti un punto che si muove in uno spazio X in tempi discreti, raggiungendo le posizioni $x_0, x_1, x_2, \ldots$. Assumiamo che sia data una partizione $X = U \sqcup V$ di X e che

$x_0 \in U$ $x_1 \in U$ $x_2 \in V$ $x_3 \in U$
$x_4 \in V$ $x_5 \in U$ $x_6 \in U$ $x_7 \in V$
$x_8 \in V$ $x_9 \in V$ $x_{10} \in V$ $x_{11} \in V$
...

Allora possiamo associare a questo movimento la successione

$UUVUVUUVUUVV \ldots$

o, più brevemente,

001010010011 ...

che fornisce già alcune indicazioni sul movimento. Potremmo adesso raffinare la partizione (lavorando con più sottoinsiemi e quindi con più lettere nel nostro alfabeto) per ottenere rappresentazioni sempre più fedeli del nostro sistema dinamico. Questa tecnica è molto utilizzata in vari campi della matematica pura e della fisica statistica.

Un ramo di applicazione più recente e interessante della dinamica simbolica è l'analisi dei testi, ad esempi in informatica e bioinformatica. In questo caso parole finite vengono studiate come parti di parole infinite a cui si possono applicare i metodi della dinamica simbolica.

La funzione Linden

Non è difficile (anzi richiede soltanto due righe), ma istruttivo, creare una funzione per la realizzazione di sistemi di Lindenmayer. Nell'esempio ci limitiamo al caso $A = \{0, 1\}$. Rappresentiamo le parole di A^* come stringhe che contengono solo le lettere `'0'` e `'1'`. La funzione è semplicemente

```
void Linden (Char A, Char B, int f())
{for (;*A;A++) B+=f(*A,B); *B=0;}
```

Il terzo argomento descrive l'applicazione φ_0 del teorema fondamentale; il primo argomento corrisponde a una parola v, il secondo argomento è l'indirizzo in cui viene scritta la parola $\varphi(v)$.

La funzione `f` deve restituire, per ogni lettera v_i di v, come risultato la lunghezza della parola $\varphi_0(v_i)$; `f` dipende da due parametri, di cui il primo è il carattere da trasformare (per cui usiamo però il tipo `int`), mentre il secondo è l'indirizzo dove scrivere la parola $\varphi_0(v_i)$.

Per la successione di Morse `f` è ad esempio

```
static int Morse (int a, Char B)
{if (a=='0') sprintf(B,"01"); else
sprintf(B,"10"); return 2;}
```

Facciamo una prova con

```
void Provalinden ()
{int n=100,max=2;
char a[n+4],b[n+4];
sprintf(a,"0");
while (strlen(a)<n/max)
{printf("%s\n",a); // Meglio \n\n
Linden(a,b,Morse);
sprintf(a,b);}}
```

ottenendo

```
0
01
0110
01101001
0110100110010110
01101001100101101001011001101001
```