Linguaggio C: introduzione

- Il linguaggio C è un linguaggio *general purpose* sviluppato nel 1972 da Dennis Ritchie per scrivere il sistema operativo UNIX ed alcune applicazioni per un PDP-11.

- Il linguaggio C è il fondamento dei sistemi UNIX, e così anche di GNU/Linux che dispone di ANSI GNU C.

- Il progetto di standardizzazione che ha portato allo standard ANSI (American National Standard Institute) è nato nel 1983 e terminato nel 1990 (lo "standard de facto" era Kernighan & Ritchie, 1988).

- I principali vantaggi del linguaggio C sono efficenza, sinteticità e portabilità.

- Il linguaggio C richiede la presenza di un compilatore per generare un file eseguibile.

- Il contenuto di un sorgente in linguaggio C può essere suddiviso in tre parti: *commenti*, *direttive del preprocessore* e *istruzioni*.

- I commenti vanno aperti e chiusi attraverso l'uso dei simboli `/*` e `*/`.

- Un minimo di conoscenza di questo linguaggio è importante per sapersi districare tra i programmi distribuiti in forma sorgente insieme al sistema operativo.

Direttive del preprocessore

- Le direttive del preprocessore rappresentano un linguaggio che guida la compilazione del codice vero e proprio.

- L'uso più comune di queste direttive viene fatto per includere porzioni di codice sorgente esterne ad un dato file.

- Il tipico programma C richiede l'inclusione di codice esterno composto da file i cui nomi terminano con l'estensione `.h`.

- La tipica libreria che viene inclusa è quella necessaria alla gestione dei flussi di standard input, standard output e standard error. Il suo utilizzo si dichiara con la seguente direttiva

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

  N.B. Non confondere le direttive del preprocessore con i commenti negli script di shell: entrambi iniziano con `#`.

Istruzioni C

- Le istruzioni C terminano con un punto e virgola (`;`) e i raggruppamenti di queste si fanno utilizzando le parentesi graffe ({ }).

```
<istruzione>;
{<istruzione>; <istruzione>; <istruzione>; }
```

- All'inizio di ogni istruzione composta, o *blocco*, con { e } è possibile dichiarare variabili (normalmente la visibilità di tali variabili è limitata al blocco in cui sono dichiarate).

- Generalmente, un'istruzione può essere interrotta e ripresa nella riga successiva, dal momento che la sua conclusione è dichiarata esplicitamente dal punto e virgola finale.

- L'*istruzione nulla* viene rappresentata utilizzando un punto e virgola da solo.

Nomi

- I nomi scelti per identificare ciò che si utilizza all'interno di un programma devono seguire regole definite dal compilatore C a disposizione.

- Per cercare di scrivere codice portabile in altre piattaforme, conviene evitare di sfruttare caratteristiche speciali del proprio ambiente. In particolare:
  - un nome può iniziare con una lettera alfabetica e continuare con altre lettere, cifre numeriche e il simbolo di sottolineatura;
  - in teoria i nomi potrebbero inziare anche con il simbolo di sottolineatura, ma questo è sconsigliabile per evitare che possano insorgere conflitti con i nomi di sistema;
  - i nomi sono sensibili alla differenza tra lettere maiuscole e minuscole.

- La lunghezza dei nomi può essere un elemento critico; generalmente, la dimensione massima è di 32 caratteri.

- GNU C accetta più di 32 caratteri, ma ci sono versioni di C che ne possono accettare solo una quantità inferiore.

- In ogni caso, il compilatore non rifiuta i nomi troppo lunghi, semplicemente non ne distingue più la differenza oltre un certo punto.

Funzione principale

- Il codice di un programma C è scomposto in *funzioni* (il C non ha procedure) e l'esecuzione del programma corrisponde alla chiamata della funzione `main()` (che quindi deve essere presente in ogni programma).

- Le definizioni delle funzioni che compongono un programma non possono essere innestate l'una nell'altra.

- Le funzioni componenti un programma, compresa `main()`, possono essere ricorsive.

- `main()` può essere dichiarata senza argomenti oppure con due argomenti precisi:

```
int main( int argc, char *argv[] )
```

- Il comando `exit(val)` può essere utilizzato per terminare forzatamente l'esecuzione di un programma e restituire un valore all'ambiente chiamante.

- In generale, la ricorsione può produrre un codice più compatto, l'iterazione un codice più efficente.

- I parametri della funzione `main` sono chiamati `argc` e `argv` per convenzione, si potrebbero usare altri nomi.

- Il secondo parametro è un array di stringhe (precisamente, un array di puntatori a caratteri), il cui primo elemento è il nome del programma (quindi `argc` è sempre maggiore di 0) e i cui elementi successivi sono gli argomenti del programma.

Funzione principale: esempio

- Vediamo un semplice programma che emette un messaggio e poi termina la sua esecuzione.

```
/*
 *    Ciao mondo!
 */

#include <stdio.h>

/* main() viene eseguita automaticamente all'avvio. */
int main(void) {
    /* Si limita a emettere un messaggio. */
    printf("Ciao mondo!\n");
    return 0;
}
```

- Nel programma sono state inserite alcune righe di commento. In particolare, all'inizio, l'asterisco che si trova nella seconda riga non serve a nulla, se non a guidare la vista verso la conclusione del commento stesso.

- Il programma si limita a emettere la stringa `Ciao mondo!` seguita da un codice di interruzione di riga, rappresentato dal simbolo `\n`.

Compilazione

- Per compilare un programma scritto in C si utilizza generalmente il comando `cc`.

- Supponendo di avere salvato il file dell'esempio con il nome `ciao.c`, il comando per la sua compilazione è il seguente:

```
$ cc ciao.c
```

- Quello che si ottiene è il file `a.out` che dovrebbe già avere i permessi di esecuzione.

```
$ ./a.out
Ciao mondo!
```

- Se si desidera compilare il programma definendo un nome diverso per il codice eseguibile finale, si può utilizzare l'opzione `-o`.

```
$ cc -o ciao ciao.c
$ ./ciao
Ciao mondo!
```

- Il comando `cc` di solito è un collegamento simbolico al vero compilatore che si ha a disposizione (es. gcc).

Emissione di dati attraverso printf()

- L'esempio di programma presentato prima si avvale di `printf()` per emettere il messaggio attraverso lo standard output.

- Questa funzione è più sofisticata di quanto possa apparire dall'esempio, in quanto permette di formattare il risultato da emettere.

```
int printf( <stringa-di-formato> [, <espressione>]... )
```

- `printf()` emette attraverso lo standard output la stringa indicata come primo parametro, dopo averla rielaborata in base alla presenza di metavariabili riferite alle eventuali espressioni che compongono i parametri successivi.

- `printf()` restituisce il numero di caratteri emessi.

- `printf("Ciao mondo!\n");`

- Negli esempi più semplici di codice C appare immancabilmente questa funzione, per cui è necessario descrivere subito, almeno in parte, il suo funzionamento.

- L'utilizzo più semplice di `printf()` è quello che è già stato visto, cioè l'emissione di una semplice stringa senza metavariabili.

- Il codice `\n` rappresenta un carattere preciso e non è una metavariabile, piuttosto si tratta di una cosiddetta sequenza di escape.

Emissione di dati attraverso printf()

- La stringa può contenere delle metavariabili del tipo `%d`, `%c`, `%f`,... e queste fanno ordinatamente riferimento ai parametri successivi.

- Per esempio,

```
printf("Totale fatturato: %d\n", 12345 );
```

  fa in modo che la stringa incorpori il valore indicato come secondo parametro, nella posizione in cui appare `%d`.

- La metavariabile `%d` stabilisce anche che il valore in questione deve essere trasformato secondo una rappresentazione decimale intera.

- Il risultato sarà esattamente quello che ci si aspetta.

```
Totale fatturato: 12345
```

Variabili e tipi

- I tipi di dato elementari gestiti dal linguaggio C dipendono molto dall'architettura dell'elaboratore sottostante.

- In questo senso, è difficile definire la dimensione assoluta delle variabili numeriche; si possono soltanto dare delle definizioni relative.

- Solitamente, il riferimento è dato dal tipo numerico intero (`int`) la cui dimensione in bit è data dalla dimensione della parola, ovvero dalla capacità dell'unità aritmetico-logica del microprocessore.

- In pratica, con l'architettura i386 la dimensione di un intero normale è di 32 bit.

Tipi di dato primitivi

- I tipi di dato primitivi rappresentano un singolo valore numerico, nel senso che anche il tipo `char` può essere trattato come un numero.

- Tipi di dato primitivi del C

```
Tipo      Descrizione

char      Un singolo carattere (generalmente di 8 bit).
int       Intero normale.
float     Virgola mobile a singola precisione.
double    Virgola mobile a doppia precisione.
```

- Come già accennato, non si può stabilire in modo generale quali siano le dimensioni esatte in bit dei vari tipi di dato, si può solo stabilire una relazione tra loro.

```
char <= int <= float <= double
```

Tipi di dato primitivi

- I tipi di dato primitivi possono essere estesi attraverso l'uso di alcuni qualificatori: `short`, `long` e `unsigned`.

- I primi due si riferiscono alla dimensione, mentre l'ultimo si riferisce alla modalità di valutazione.

- Tipi di dato primitivi in C con qualificatori.

Tipo	Abbreviazione	Descrizione
`char`		
`unsigned char`		Tipo char usato numericamente senza segno.
`short int`	`short`	Intero piu' breve di int.
`unsigned short int`	`unsigned short`	Tipo short senza segno.
`int`		Intero normale.
`unsigned int`	`unsigned`	Tipo int senza segno.
`long int`	`long`	Intero piu' lungo di int.
`unsigned long int`	`unsigned long`	Tipo long senza segno.
`float`		
`double`		
`long double`		Tipo a virgola mobile piu' lungo di double.

- `char <= short <= int <= long`
  `float <= double <= long double`

- I tipi `long` e `float` potrebbero avere una dimensione uguale, altrimenti non è detto quale dei due sia più grande.

- Così, il problema di stabilire le relazioni tra le dimensioni si complica.

Tipi di dato primitivi

- Il programma seguente potrebbe essere utile per determinare la dimensione dei vari tipi primitivi nella propria piattaforma.

```
/* dimensione_variabili */

#include <stdio.h>

int main(void) {
    printf( "char        %d\n", (int)sizeof(char) );
    printf( "short       %d\n", (int)sizeof(short) );
    printf( "int         %d\n", (int)sizeof(int) );
    printf( "long        %d\n", (int)sizeof(long) );
    printf( "float       %d\n", (int)sizeof(float) );
    printf( "double      %d\n", (int)sizeof(double) );
    printf( "long double %d\n", (int)sizeof(long double) );
    return 0;
}
```

Possibile risultato

```
char        1
short       2
int         4
long        4
float       4
double      8
long double 12
```

- Come si può osservare, la dimensione in byte è restituita dalla funzione `sizeof()`, che nell'esempio risulta preceduta dalla notazione `(int)`.

- `sizeof()` restituisce un valore di tipo `size_t` il quale nello standard ANSI corrisponde ad un intero senza segno. Solitamente `size_t` è definito in `stdlib.h` tramite `typedef` ed è equivalente a `unsigned int`.

  ```
  typedef unsigned int size_t
  ```

- La notazione `(int)` è un *cast*, cioè una conversione di tipo esplicita.

- Il cast è solo precauzionale perché generalmente tutto funziona in modo regolare senza questa indicazione.

- I numeri rappresentano la quantità di caratteri, nel senso di valori `char`, per cui il tipo `char` dovrebbe sempre avere una dimensione unitaria.

Valori contenibili nelle variabili

- I tipi di dato primitivi sono tutti utili alla memorizzazione di valori numerici nelle variabili relative.

- A seconda che il valore in questione sia trattato con segno o senza segno, varia lo spettro di valori che possono essere contenuti.

- Nel caso di interi senza segno (`char`, `short`, `int` e `long`), la variabile può essere utilizzata per tutta la sua estensione per contenere un numero binario.

- In pratica, il massimo valore ottenibile è (2**n)-1, dove n rappresenta il numero di bit a disposizione.

- Quando invece si vuole trattare il dato come un numero con segno, il valore numerico massimo ottenibile è circa la metà.

- Nel caso di variabili a virgola mobile, non c'è più la possibilità di rappresentare esclusivamente valori senza segno, e non c'è più un limite di dimensione, ma di approssimazione.

Valori contenibili

- Le variabili `char` sono fatte, in linea di principio, per contenere il codice di rappresentazione di un carattere, secondo la codifica utilizzata nel sistema. Generalmente si utilizza la codifica ASCII.

- Generalmente si tratta di un dato di 8 bit (un byte), ma non è detto che debba sempre essere così.

- Il fatto che una variabile di tipo `char` possa essere gestita in modo numerico (in pratica si tratta di un "piccolo" intero), permette una facile conversione da lettera a codice numerico corrispondente.

- Un tipo di valore che non è stato ancora visto è quello *logico*: *Vero* è rappresentato da un qualsiasi valore numerico diverso da 0, mentre *Falso* corrisponde a 0.

Costanti letterali

- Quasi tutti i tipi di dato primitivi hanno la possibilità di essere rappresentati in forma di *costante letterale* (differentemente da una costante simbolica, una costante letterale rappresenta se stessa).

- In particolare, si distingue tra:
  - costanti *carattere*, rappresentate da un singolo carattere alfanumerico racchiuso tra apici singoli, come `'A'`, `'B'`,...;
  - costanti *intere*, rappresentate da un numero senza decimali, e a seconda delle dimensioni, può trattarsi di uno dei vari tipi di interi (escluso `char`);
  - costanti *con virgola*, rappresentate da un numero con decimali (un punto seguito da altre cifre, anche se si tratta solo di zeri), e indipendentemente dalle dimensioni si tratta sempre di un tipo `double`.

Costanti letterali

- Per esempio, 123 è generalmente una costante `int`, mentre 123.0 è una costante `double`.

- Per le costanti che rappresentano numeri con virgola si può usare anche la notazione scientifica.

  Per esempio, `7e+15` rappresenta l'equivalente di 7*(10**15), cioè un 7 con 15 zeri. Nello stesso modo, `7e-5`, rappresenta l'equivalente di 7*(10**-5), cioè 0.00007.

- È possibile rappresentare anche le stringhe in forma di costante, e questo attraverso l'uso degli apici doppi (`"`), ma la stringa non è un tipo di dato primitivo, trattandosi piuttosto di un array di caratteri.

- Per il momento è importante fare attenzione a non confondere il tipo `char` con la stringa.

  Per esempio, `'F'` è un carattere, mentre `"F"` è una stringa, e la differenza, come vedremo, è notevole.

Caratteri speciali

- Alcuni caratteri non hanno una rappresentazione grafica e non possono essere inseriti attraverso la tastiera.

- In questi casi, si possono usare tre tipi di notazione: *ottale*, *esadecimale* e *simbolica*.

- In tutti i casi si utilizza la barra obliqua inversa (\) come carattere di escape, cioè come simbolo per annunciare che ciò che segue immediatamente deve essere interpretato in modo particolare.

- La notazione ottale usa la forma \`ooo`, dove ogni lettera `o` rappresenta una cifra ottale.

- La notazione esadecimale usa la forma \`xhh`, dove ogni `h` rappresenta una cifra esadecimale.

- La notazione simbolica permette di fare riferimento facilmente a codici di uso comune, quali `<CR>`, `<HT>`,... Inoltre, permette anche di indicare caratteri che altrimenti verrebbero interpretati in maniera differente dal compilatore.

- A questo proposito, è opportuno notare che se la dimensione di un carattere fosse superiore ai fatidici 8 bit, occorrerebbero probabilmente più cifre ottali (una cifra ottale rappresenta un gruppo di 3 bit).

- Anche in questo caso vale la considerazione per cui ci vorranno più di due cifre esadecimali per rappresentare un carattere più lungo di 8 bit.

  È comunque importante tener presente che in C non esistono costanti di tipo carattere: 'F' indica in realtà l'intero che corrisponde alla sua codifica.

Caratteri speciali

- Modi di rappresentazione delle costanti carattere attraverso codici di escape.

Codice di escape	Descrizione
\ooo	Notazione ottale.
\xhh	Notazione esadecimale.
\\	Una singola barra obliqua inversa (\).
\'	Un apice singolo destro.
\"	Un apice doppio.
\0	Il codice <NUL>.
\a	Il codice <BEL> (bell).
\b	Il codice <BS> (backspace).
\f	Il codice <FF> (formfeed).
\n	Il codice <LF> (linefeed).
\r	Il codice <CR> (carriage return).
\t	Una tabulazione orizzontale (<HT>).
\v	Una tabulazione verticale (<VT>).

- Nell'esempio introduttivo, è già stato visto l'uso della notazione \n per rappresentare l'inserzione di un codice di interruzione di riga alla fine del messaggio di saluto.

```
printf("Ciao mondo!\n");
```

  Senza di questo, il cursore resterebbe a destra del messaggio alla fine dell'esecuzione di quel programma, ponendo lì l'invito.

Variabili

- Il **campo d'azione** delle variabili in C viene determinato dalla posizione in cui queste vengono dichiarate e dall'uso di particolari qualificatori.

  Quanto dichiarato all'interno di una funzione o di un blocco ha valore *locale* alla funzione o al blocco stesso, mentre quanto dichiarato al di fuori, ha valore *globale* per tutto il file.

- La **dichiarazione** di una variabile avviene specificando il tipo e il nome della variabile.

```
int numero;
```

  La variabile può anche essere *inizializzata* contestualmente, assegnandogli un valore.

```
int numero = 1000;
```

Costanti simboliche

- Una *costante* è qualcosa che non varia e generalmente si rappresenta attraverso una notazione che ne definisce il valore.

- A volte può essere più comodo definire una costante in modo *simbolico*, cioè dando un nome al valore come se fosse una variabile, per facilitarne l'utilizzo e l'identificazione all'interno del programma. Si ottiene questo con il modificatore `const` (o con la direttiva del preprocessore `#define`).

- Ovviamente, è obbligatorio inizializzare la costante contestualmente alla sua dichiarazione.

```
const float pi = 3.14159265;
```

- Le costanti simboliche di questo tipo sono delle variabili per le quali il compilatore non concede che avvengano delle modifiche.

- È il caso di osservare, tuttavia, che l'uso di costanti simboliche di questo tipo è piuttosto limitato.

- Generalmente è preferibile utilizzare delle macro definite e gestite attraverso il preprocessore (come vedremo più avanti).

Convenzioni

- Una caratteristica fondamentale del linguaggio C è quella di essere piuttosto liberale sul tipo di operazioni che si possono fare con i vari tipi di dato.

  Per esempio, il compilatore non si oppone di fronte all'assegnamento di un valore numerico a una variabile `char` o all'assegnamento di un carattere a un intero.

- Però ci possono essere situazioni in cui cose del genere accadono accidentalmente; per rendersene conto si potrebbe usare una convenzione nella definizione dei nomi delle variabili, in modo da distinguerne il tipo.

- Una *convenzione* usata nel linguaggio Java, valida e perfettamente applicabile anche in C, è la seguente.

  Si possono comporre i nomi delle variabili utilizzando un prefisso composto da una o più lettere minuscole che serve a descriverne il tipo. Nella parte restante si possono usare iniziali maiuscole per staccare visivamente i nomi composti da più parole significative.

Convenzioni

Prefisso	Tipo corrispondente
c	char
uc	unsigned char
si	short int
usi	unsigned short int
i	int
ui	unsigned int
li	long int
uli	unsigned long int
f	float
d	double
ld	long double
a	array
ac	array di char, o stringa
auc	array di unsigned char
a...	array di ...
acz	stringa terminata con \0
t	struct
u	union
p	puntatore
pc	puntatore a char
puc	puntatore a unsigned char
p...	puntatore a ...
e	enumerazione

Per esempio, `iLivello` potrebbe essere la variabile di tipo `int` che contiene il livello di qualcosa.

Nello stesso modo, `ldIndiceConsumo` potrebbe essere una variabile di tipo `long double` che rappresenta l'indice del consumo di qualcosa.

- In questa fase non abbiamo ancora usato tutti i tipi di dato che si possono gestire effettivamente.

Operatori ed espressioni

- Un operatore è qualcosa che esegue un qualche tipo di funzione su degli operandi e restituisce un valore.

- Il valore restituito è di tipo diverso a seconda degli operandi utilizzati.

  Per esempio, la somma di due interi genera un risultato intero.

- Conversioni di tipo *implicite* possono avvenire durante gli assegnamenti o nella valutazione di espressioni miste.

- Gli operatori descritti di seguito sono quelli più comuni ed importanti.

Operatori aritmetici

++<op>	Incrementa di un'unita' l'operando prima che venga restituito il suo valore.
<op>++	Incrementa di un'unita' l'operando dopo averne restituito il suo valore.
--<op>	Decrementa di un'unita' l'operando prima che venga restituito il suo valore.
<op>--	Decrementa di un'unita' l'operando dopo averne restituito il suo valore.
+<op>	Non ha alcun effetto.
-<op>	Inverte il segno dell'operando.
<op1> + <op2>	Somma i due operandi.
<op1> - <op2>	Sottrae dal primo il secondo operando.
<op1> * <op2>	Moltiplica i due operandi.
<op1> / <op2>	Divide il primo operando per il secondo.
<op1> % <op2>	Modulo: il resto della divisione tra il primo e il secondo operando.
<var> = <val>	Assegna alla variabile il valore alla destra e lo restituisce come risultato.
<op1> += <op2>	<op1> = <op1> + <op2>
<op1> -= <op2>	<op1> = <op1> - <op2>
<op1> *= <op2>	<op1> = <op1> * <op2>
<op1> /= <op2>	<op1> = <op1> / <op2>
<op1> %= <op2>	<op1> = <op1> % <op2>

Operatori aritmetici: esempi

- In C l'assegnamento di una variabile è un'espressione e come tale restituisce un valore, quello assegnato alla variabile.

- `=` ha priorità più bassa rispetto agli altri operatori ed associa da destra a sinistra.

  `i = j = k = 0` equivale a `i = (j = (k = 0))`

- Un `;` finale trasforma un'espressione in un'istruzione.

  Alcune sono molto utili

  `x = 3 + 4;` o `x++;`

  altre non lo sono

  `3 + 4;`

- Il fatto che l'assegnamento (e le sue varianti `+=`, `-=`, ...) è un'espressione permette di scrivere delle *contrazioni*.

  `j = ( i = (3+4)-7 )` `z = (x = 1) + (y = 2))`

Operatori di confronto

- Gli operatori di confronto determinano la relazione tra due operandi.

- Il risultato dell'espressione composta da due operandi posti a confronto è di tipo booleano, rappresentabile in C come !0 o non-zero (Vero), e 0 (Falso).

- È importante ricordare che qualunque valore diverso da 0, equivale a Vero in un contesto logico.

```
<op1> == <op2>    Vero se gli operandi si equivalgono.
<op1> != <op2>    Vero se gli operandi sono differenti.
<op1> < <op2>     Vero se il primo operando e' minore
                  del secondo.
<op1> > <op2>     Vero se il primo operando e' maggiore
                  del secondo.
<op1> <= <op2>    Vero se il primo operando e' minore o
                  uguale al secondo.
<op1> >= <op2>    Vero se il primo operando e' maggiore
                  o uguale al secondo.
```

Operatori logici

- Il risultato di un'espressione logica complessa è quello dell'ultima espressione elementare a essere valutata.

```
! <op>          Inverte il risultato logico dell'operando.
<op1> && <op2>   Se il risultato del primo operando e'
                 Falso non valuta il secondo.
<op1> || <op2>   Se il risultato del primo operando e'
                 Vero non valuta il secondo.
```

- *Valutazione cortocircuitata*: in un'espressione formata tramite i connettivi logici, le espressioni componenti vengono valutate sequenzialmente fino a che non viene determinato il valore dell'intera espressione.

  Per esempio, in `if ( i > 10 || ++j < 10 ) x = j`

  – se `i == 1` e `j == 8` allora ad `x` verrà assegnato `9`;

  – se `i == 11` e `j == 8` allora ad `x` verrà assegnato `8`.

  Per esempio, in `if ( i > 10 && ++j < 10 ) x = j`

  – se `i == 1` e `j == 8` allora ad `x` non verrà assegnato un nuovo valore;

  – se `i == 11` e `j == 8` allora ad `x` verrà assegnato `9`.

Operatori logici

- Un tipo particolare di operatore logico è l'operatore condizionale, che permette di valutare espressioni diverse in relazione al risultato di una condizione.

  ```
  <condizione> ? <espressione1> : <espressione2>
  ```

  In pratica, se l'espressione che rappresenta la condizione si avvera, viene eseguita la prima espressione che segue il punto interrogativo, altrimenti viene eseguita quella che segue i due punti.

Operatori binari

- In C, così come non esiste il tipo di dato booleano, non esiste nemmeno la possibilità di utilizzare variabili composte da un singolo bit.

- A questo problema si fa fronte attraverso l'utilizzo in modo binario dei tipi di dati esistenti.

```
<op1> & <op2>       AND bit per bit.
<op1> | <op2>       OR bit per bit.
<op1> ^ <op2>       XOR bit per bit (OR esclusivo).
<op1> << <op2>      Spostamento a sinistra di <op2> bit.
<op1> >> <op2>      Spostamento a destra di <op2> bit.
~<op1>              Complemento a uno.
<op1> &= <op2>      <op1> = <op1> & <op2>
<op1> |= <op2>      <op1> = <op1> | <op2>
<op1> ^= <op2>      <op1> = <op1> ^ <op2>
<op1> <<= <op2>     <op1> = <op1> << <op2>
<op1> >>= <op2>     <op1> = <op1> >> <op2>
<op1> ~= <op2>      <op1> = ~<op2>
```

Operatori binari: esempi

- In particolare, lo spostamento può avere effetti differenti a seconda che venga utilizzato su una variabile senza segno o con segno, e quest'ultimo caso può dare risultati diversi su piattaforme differenti.

- Vediamo ora alcuni esempi. Utilizzeremo due operandi di tipo `char` (a 8 bit) senza segno:
  - `a`, contenente il valore 42, pari a 00101010;
  - `b`, contenente il valore 51, pari a 00110011.

  AND: `c = a & b`

```
00101010 (42) AND
00110011 (51) =
-------------
00100010 (34)
```

  OR: `c = a | b`

```
00101010 (42) OR
00110011 (51) =
-------------
00111011 (59)
```

Operatori binari: esempi

XOR: `c = a ^ b`

```
00101010 (42) XOR
00110011 (51) =
------------
00011001 (25)
```

Spostamento a sinistra: `c = a << 1`

```
00101010 (42) <<
00000001 (1) =
------------
01010100 (84)
```

In pratica si è ottenuto un raddoppio.

Spostamento a destra: `c = a >> 1`

```
00101010 (42) >>
00000001 (1) =
------------
00010101 (21)
```

In pratica si è ottenuto un dimezzamento.

Complemento: `c = ~a`

```
00101010 (42)
11010101 (213)
```

`c` conterrà il valore 213, corrispondente all'inversione dei bit di `a`.

Priorità ed associatività degli operatori

Operatori	Associativita'
() [] . -> ++ (postfisso) -- (postfisso)	sin-dest
++ (prefisso) -- (prefisso) ! ~ sizeof + (unario) - (unario) & (indirizzo)	des-sin
* / %	sin-dest
+ -	sin-dest
<< >>	sin-dest
< <= > >=	sin-dest
== !=	sin-dest
&	sin-dest
^	sin-dest
|	sin-dest
&&	sin-dest
||	sin-dest
?:	dest-sin
= += -= *= /= ecc.	dest-sin
, (operatore virgola)	sin-dest

Conversione di tipo

- Quando si assegna un valore a una variabile, nella maggior parte dei casi, il contesto stabilisce il tipo di questo valore in modo corretto. Di fatto, è il tipo della variabile ricevente che stabilisce la conversione necessaria.

- Il problema si può porre durante la valutazione di un'espressione. Per esempio, `5/4` viene considerata la divisione di due interi, e di conseguenza l'espressione restituisce un valore intero, cioè `1` (il resto viene perso). Se si scrivesse `5.0/4.0` si tratterebbe della divisione tra due numeri in virgola mobile (per la precisione, di tipo `double`) e il risultato è un numero in virgola mobile.

- Quando si pone il problema di risolvere l'ambiguità si utilizza esplicitamente la conversione del tipo, attraverso un *cast*. In pratica, si deve indicare tra parentesi il nome del tipo di dato in cui deve essere convertita l'espressione che segue.

```
(<tipo>) <espressione>
```

Conversione di tipo

- Per chiarire l'ordine con cui gli operatori, cast compreso, vanno impiegati è bene fare uso di parentesi tonde aggiuntive.

- ```
  int x = 10;
  long y;
  ...
  y = (long)x/9;
  ```

  In questo caso, la variabile intera `x` viene convertita nel tipo `long` (a virgola mobile) prima di eseguire la divisione.

  Il cast ha precedenza sull'operazione di divisione e l'operazione avviene trasformando implicitamente il 9 intero in un 9 di tipo `long`.

  In pratica, l'operazione avviene utilizzando valori `long` e restituendo un risultato `long`.
  ```

Espressioni multiple

- Un'istruzione, cioè qualcosa che termina con un punto e virgola, può contenere diverse espressioni separate da una virgola.

- L'operatore infisso "`,`" permette di concatenare due espressioni formandone una sola. Il risultato restituito è quello della seconda espressione (il risultato della prima è ignorato).

- Per esempio, "`i = 0, j = 0`" e "`i++, j *= 2`" sono singole espressioni.

- L'esempio mostra un'istruzione contenente tre espressioni: la prima assegna a `y` il valore 10, la seconda assegna a `x` il valore 20, e la terza sovrascrive `y` assegnandole il risultato del prodotto `x*2`.

```
int x;
int y;
...
y = 10, x = 20, y = x*2;
```

  In pratica, alla fine, la variabile `y` contiene il valore 40 e la `x` contiene 20.

Strutture di controllo del flusso

- Il linguaggio C mette a disposizione praticamente tutte le strutture di controllo di flusso degli altri linguaggi di programmazione (compreso `goto`, che comunque è sempre meglio non utilizzare).

- Le strutture di controllo permettono di vincolare l'esecuzione di una parte di codice alla verifica di una condizione, oppure permettono di eseguire dei cicli, sempre sotto il controllo di una condizione.

- La parte di codice che viene sottoposta a questo controllo, può essere una singola istruzione, oppure un gruppo di istruzioni. Nel secondo caso, è necessario delimitare questo gruppo attraverso l'uso delle parentesi graffe.

- `goto` provoca un salto incondizionato ad un'istruzione etichettata del tipo

  ```
  label:  istruzione
  ```

  cosa che è sempre bene evitare se si vuole scrivere codice ben progettato secondo i dettami della *programmazione strutturata*.

If

- La struttura *condizionale* è il sistema di controllo fondamentale dell'andamento del flusso delle istruzioni.

```
if ( <condizione> ) <istruzione>
if ( <condizione> ) <istruzione> else <istruzione>
```

Se la condizione si verifica, viene eseguita l'istruzione (o il gruppo di istruzioni) seguente, altrimenti viene eseguita l'istruzione che segue la parola chiave `else`; in ogni caso, il controllo passa alle istruzioni successive alla struttura.

- ```
  int iImporto;
  ...
  if ( iImporto > 10000000 ) printf( "L'offerta e' vantaggiosa\n" );
  ```

- Nel caso di `if` innestati vale la regola che un `else` si riferisce sempre all'ultimo `if` che lo precede testualmente.
  ```

If

- ```
  int iImporto;
  int iMemorizza;
  ...
  if ( iImporto > 10000000 ) {
          iMemorizza = iImporto;
          printf( "L'offerta e' vantaggiosa\n" );
  } else {
          printf( "Lascia perdere\n" );
  }
  ```

- ```
  int iImporto;
  int iMemorizza;
  ...
  if ( iImporto > 10000000 ) {
          iMemorizza = iImporto;
          printf( "L'offerta e' vantaggiosa\n" );
  } else if ( iImporto > 5000000 ) {
          iMemorizza = iImporto;
          printf( "L'offerta e' accettabile\n" );
  } else {
          printf( "Lascia perdere\n" );
  }
  ```

Switch

- La struttura di *selezione* permette di eseguire una o più istruzioni in base al risultato di un'espressione.
- L'esempio seguente mostra la visualizzazione del nome del mese, in base al valore di un intero.

```
int iMese;
...
switch (iMese) {
    case 1: printf( "gennaio\n" ); break;
    case 2: printf( "febbraio\n" ); break;
    case 3: printf( "marzo\n" ); break;
    ...
    case 10: printf( "ottobre\n" ); break;
    case 11: printf( "novembre\n" ); break;
    case 12: printf( "dicembre\n" ); break;
}
```

- Le etichette dei `case` sono costanti intere.
- Se nel `case` selezionato all'interno di uno `switch` manca il `break`, l'esecuzione prosegue con la prima istruzione del `case` successivo. L'istruzione `break` esplicitamente interrompe l'esecuzione della struttura.

- `switch` è utile per trattare più casi differenti senza dover usare una serie di `if` innestati.

Switch

- Più casi possono essere raggruppati assieme, quando si vuole che ognuno di questi corrisponda allo stesso insieme di istruzioni.

```
int iAnno;
int iMese;
int iGiorni;
...
switch (iMese) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        iGiorni = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        iGiorni = 30;
        break;
    case 2:
        if ( ((iAnno % 4 == 0) && !(iAnno % 100 = 0)) ||
                (iAnno % 400 == 0) )
            iGiorni = 29;
        else
            iGiorni = 28;
        break;
}
```

Switch

- È anche possibile definire un caso predefinito che si verifica quando nessuno degli altri si avvera.

```
int iMese;
...
switch (iMese) {
    case 1: printf( "gennaio\n" ); break;
    case 2: printf( "febbraio\n" ); break;
    ...
    case 11: printf( "novembre\n" ); break;
    case 12: printf( "dicembre\n" ); break;
    default: printf( "mese non corretto\n" ); break;
}
```

while

- `while ( <condizione> ) <istruzione>`

  L'*iterazione* si ottiene normalmente in C attraverso l'istruzione `while`, che esegue un'istruzione, o un gruppo di queste, finché la condizione continua a restituire il valore Vero.

  La condizione viene valutata prima di eseguire il gruppo di istruzioni e poi, ogni volta che termina un ciclo, prima dell'esecuzione del successivo.

- L'esempio seguente stampa per 10 volte la lettera `x`.

```
int iContatore = 0;

while (iContatore < 10) {
    iContatore++;
    printf( "x" );
}
printf( "\n" );
```

while

- Nel blocco di istruzioni di un ciclo `while` (così come negli altri costrutti iterativi) ne possono apparire alcune particolari:
  - `break`, per uscire definitivamente dal ciclo;
  - `continue`, per interrompere l'esecuzione del gruppo di istruzioni, riprendendo immediatamente con l'iterazione successiva (a partire dalla valutazione della condizione).

- L'esempio seguente è una variante del precedente che evidenzia l'uso dell'istruzione `break`.

```
int iContatore = 0;

while (1) {
    if (iContatore >= 10) {
        break;
    }
    iContatore++;
    printf( "x" );
}
printf( "\n" );
```

- `while (1)` equivale a stabilire che il ciclo è senza fine, perché la condizione è sempre vera.

- In questo modo, solo la richiesta esplicita di interruzione dell'esecuzione della struttura (attraverso l'istruzione `break`) permette l'uscita da questa.

do-while

- L'istruzione `do` costituisce una variante del ciclo `while`, in cui l'analisi della condizione di uscita avviene dopo l'esecuzione del blocco di istruzioni che viene iterato.

```
do <blocco-di-istruzioni> while ( <condizione> );
```

- In questo caso, si esegue un gruppo di istruzioni una volta, e poi se ne ripete l'esecuzione finché la condizione restituisce il valore Vero.

For

- Per iterazioni in cui si deve incrementare o decrementare una variabile a ogni ciclo si usa preferibilmente la struttura `for`.

- La forma tipica di un'istruzione `for` è:

  ```
  for ( <var> = n; <condizione>; <var>++ ) <istruzione>
  ```

  `<var> = n` corrisponde all'assegnamento iniziale della variabile `<var>`,

  `<condizione>` corrisponde ad una condizione che deve verificarsi fino a che si vuole che sia eseguita l'istruzione (o il gruppo di istruzioni) `<istruzione>`, e

  `<var>++` corrisponde all'incremento (o decremento, se si usa `<var>--`) della variabile `<var>`.

For

La struttura `for` potrebbe essere definita anche in maniera più generale:

```
for ( <espressione1>; <espressione2>; <espressione3> ) <istruzione>
```

- `<espressione1>` viene eseguita una volta sola all'inizio della struttura;
- `<espressione2>` viene valutata all'inizio di ogni ciclo e il gruppo di istruzioni `<istruzione>` viene eseguito solo se il risultato è Vero;
- `<espressione3>` viene eseguita alla fine dell'esecuzione del gruppo di istruzioni, prima che si ricominci con l'analisi della condizione.

Le espressioni in un'istruzione `for` possono anche mancare; in particolare, `<espressione2>` ha per default valore `1` (vero).

For

- L'esempio già visto, in cui veniva visualizzata per dieci volte una `x`, potrebbe tradursi nel modo seguente.

```
int iContatore;

for ( iContatore = 0; iContatore < 10; iContatore++ ) {
    printf( "x" );
}
printf( "\n" );
```

- Anche nelle istruzioni controllate da una struttura `for` si possono collocare istruzioni `break` e `continue`, con lo stesso significato visto per la struttura `while`.

- Sfruttando la possibilità di inserire più espressioni in una singola istruzione, si possono realizzare dei cicli `for` molto più complessi.

```
int iContatore;

for (iContatore = 0; iContatore < 10; printf( "x" ), iContatore++) {
    ;
}
printf( "\n" );
```

- Sfruttando la possibilità di inserire più espressioni in una singola istruzione, si possono realizzare dei cicli `for` molto più complessi, anche se però questo è sconsigliabile per evitare di scrivere codice troppo difficile da interpretare. L'esempio precedente potrebbe essere ridotto a quello che segue:

```
int iContatore;

for ( iContatore = 0; iContatore < 10; printf( "x" ),
                                              iContatore++ ) {
    ;
}
printf( "\n" );
```

- Il punto e virgola solitario rappresenta un'istruzione nulla.

Funzioni

- Il linguaggio C offre le *funzioni* come mezzo per realizzare la scomposizione del codice in subroutine.

- Prima di poter essere utilizzate attraverso una *chiamata*, le funzioni devono essere *dichiarate*, anche se non necessariamente *definite*, tramite un *prototipo*.

- Le funzioni del linguaggio C prevedono il passaggio di parametri solo *per valore*, e soltanto di tipi di dato primitivi (compresi però i puntatori che vedremo in seguito).

- Il linguaggio C offre un gran numero di funzioni predefinite, che vengono importate nel codice del programmatore attraverso l'istruzione `#include` del *preprocessore*.

  Per esempio, come si è già visto, per poter utilizzare la funzione predefinita `printf()` si deve inserire la riga

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

  nella parte iniziale del file sorgente.

- In pratica, se si vuole indicare nel codice una chiamata a una funzione che viene descritta più avanti, occorre almeno dichiararne il *prototipo*.

- Il passaggio per valore implica che i valori delle variabili passate alla funzione non possono essere modificati. Con i puntatori è possibile realizzare il passaggio per *riferimento* o *indirizzo*: anzicché la variabile viene passato il suo indirizzo.

- In pratica, con `#include` si importa la parte di codice necessaria alla dichiarazione e descrizione di queste funzioni standard.

Dichiarazione di un prototipo

```
<tipo> <nome> ([<tipo-parametro>[,...]]);
```

- Il tipo, posto all'inizio, rappresenta il tipo di valore che la funzione restituisce. Se la funzione non deve restituire alcunché, si utilizza il tipo `void`.
- Se la funzione richiede dei parametri, il tipo di questi deve essere elencato tra le parentesi tonde.
- L'istruzione con cui si dichiara il prototipo termina regolarmente con un punto e virgola.
- Lo standard ANSI C stabilisce che una funzione che non richiede parametri deve utilizzare l'identificatore `void` in modo esplicito all'interno delle parentesi.
- Per funzioni che possono ricevere un numero arbitrario di parametri vengono utilizzati tre punti (...).

```
int printf ( const char *formatstr, ... );
```

Dichiarazione di un prototipo: esempi

- `int fattoriale( int );`

  In questo caso, viene dichiarato il prototipo della funzione `fattoriale`, che richiede un parametro di tipo `int` e restituisce anche un valore di tipo `int`.

- `void elenca();`

  Si tratta della dichiarazione di una funzione che fa qualcosa senza bisogno di ricevere alcun parametro e senza restituire alcun valore (`void`).

- `void elenca( void );`

  Esattamente come nell'esempio precedente, solo che è indicato in modo esplicito il fatto che la funzione non riceve argomenti (il tipo `void` è stato messo all'interno delle parentesi), come richiede lo standard ANSI.

Definizione di una funzione

```
<tipo> <nome> ([<tipo> <parametro>[,...]]) {
        <istruzione>;...
}
```

- La definizione della funzione, costituita da *intestazione* e *corpo*, rispetto alla dichiarazione del prototipo, aggiunge l'indicazione dei nomi da usare per identificare i parametri e le istruzioni da eseguire.

- Per esempio, la funzione seguente esegue il prodotto tra i due parametri forniti e ne restituisce il risultato.

```
int prodotto( int x, int y ) {
    return x*y;
}
```

- I parametri indicati tra parentesi rappresentano dichiarazioni di variabili locali che saranno inizializzate con i valori usati nella chiamata.

- Il valore restituito da una funzione viene definito attraverso l'istruzione `return` (eventualmente seguita da un'espressione).

- La dichiarazione di funzione è il costrutto per cui i due standard ANSI e K&R differiscono maggiormente: nel caso dello standard K&R, i tipi dei parametri di una funzione non vengono specificati tra le parentesi tonde ma all'inizio del corpo della funzione (cioè dopo la {).

- Le parentesi graffe che appaiono nello schema sintattico fanno parte delle istruzioni necessarie.

- Naturalmente, le funzioni di tipo `void`, cioè quelle che non devono restituire alcun valore, non usano l'istruzione `return`.

- L'espressione che eventualmente segue `return` può essere racchiusa tra parentesi tonde. Il suo valore, se necessario, viene implicitamente convertito al tipo della funzione.

Variabili locali e globali

- Le variabili dichiarate all'interno di una funzione, oltre a quelle dichiarate implicitamente come mezzo di trasporto dei parametri, sono visibili solo al suo interno, mentre quelle dichiarate al di fuori, dette *globali*, sono accessibili a tutte le funzioni.

- Se una variabile locale ha un nome coincidente con quello di una variabile globale, allora, all'interno della funzione, quella variabile globale non sarà accessibile.

- Le regole da seguire per scrivere programmi chiari e comprensibili prevedono che si debba fare in modo di rendere le funzioni indipendenti dalle variabili globali, fornendo loro tutte le informazioni necessarie attraverso i parametri della chiamata.

- Ciò permette di non dover tenere a mente il ruolo delle variabili globali e rende il comportamento delle funzioni indipendente dal mascheramento delle variabili globali.

Struttura e campo d'azione

- Un programma scritto in linguaggio C può essere articolato in diversi file sorgenti, all'interno dei quali si può fare riferimento solo a *oggetti*, cioè variabili e funzioni, dichiarati preventivamente.

- La dichiarazione di questi oggetti non corrisponde necessariamente con la loro descrizione che può essere collocata altrove, nello stesso file o in un altro file sorgente del programma.

- Quando si vuole fare riferimento a una funzione descritta in un file sorgente differente, o in una posizione successiva dello stesso file, occorre dichiararne il prototipo in una posizione precedente.

- Se si desidera fare in modo che una funzione sia accessibile solo nel file sorgente in cui viene descritta, occorre definirla come `static`.

```
static void miafunzione(...) {
...
}
```

Variabili e classi di memorizzazione

- Quando si dichiarano delle variabili, senza specificare alcuna classe di memorizzazione (cioè quando lo si fa normalmente come negli esempi visti fino a questo punto), il loro campo d'azione è relativo alla posizione della dichiarazione:
  - le variabili dichiarate all'esterno delle funzioni sono *globali*, cioè accessibili da parte di tutte le funzioni, a partire dal punto in cui vengono dichiarate;
  - le variabili dichiarate all'interno delle funzioni sono *locali*, cioè accessibili esclusivamente dall'interno della funzione in cui si trovano.

  Le stesse regole di visibilità valgono per i blocchi.

- Si distinguono quattro tipi di *classi di memorizzazione*, a ciascuna delle quali corrisponde una parola chiave per la loro dichiarazione:
  - *automatica*, `auto`;
  - *registro*, `register`;
  - *statica*, `static`;
  - *esterna*, `extern`.

Variabili e classi di memorizzazione

- `auto`: vale in modo predefinito e non occorre indicarla quando si dichiarano le variabili (automatiche).

- Una variabile dichiarata come appartenente alla classe `register` viene posta in un registro del microprocessore.

- `static` genera due situazioni distinte a seconda della posizione in cui viene dichiarata la variabile. Se è definita al di fuori delle funzioni, risulterà accessibile solo all'interno del file sorgente in cui viene dichiarata. Se invece è interna a una funzione, la variabile mantiene il suo valore tra una chiamata e l'altra della funzione.

- Quando da un file sorgente si vuole accedere a variabili globali dichiarate in modo normale in un altro file, oppure, quando in un file si vuole poter accedere a variabili dichiarate in una posizione più avanzata dello stesso, occorre una sorta di prototipo delle variabili: la dichiarazione `extern`.

- Dichiarare una variabile come `register` può essere utile per velocizzare l'esecuzione di un programma che deve accedere frequentemente a una certa variabile, ma generalmente l'utilizzo di questa tecnica è sconsigliabile.

- In questo senso, una variabile `locale statica`, richiede generalmente un'inizializzazione all'atto della dichiarazione; tale inizializzazione avverrà una sola volta, all'avvio del programma.

- Quando da un file sorgente si vuole accedere a variabili globali dichiarate in modo normale in un altro file, oppure, quando in un file si vuole poter accedere a variabili dichiarate in una posizione più avanzata dello stesso, occorre una sorta di prototipo delle variabili: la dichiarazione `extern`.

  In questo modo si informa esplicitamente il compilatore e il linker della presenza di queste.

Classi di memorizzazione: esempi

- ```
  int accumula( int iAggiunta ) {
      static int iAccumulo = 0;
      iAccumulo += iAggiunta;
      return iAccumulo;
  }
  ```

  La funzione appena mostrata si occupa di accumulare un valore e di restituirne il livello raggiunto a ogni chiamata. Come si può osservare, la variabile statica `iAccumulo` viene inizializzata a 0, altrimenti non ci sarebbe modo di cominciare con un valore di partenza corretto.

- ```
  static int iMiaVariabile;
  ...
  int miafunzione(...) {
  ...
  }
  ...
  ```

  La variabile `iMiaVariabile` è accessibile solo alle funzioni descritte nello stesso file in cui questa si trova, impedendo l'accesso a questa da parte di funzioni di altri file attraverso la dichiarazione `extern`.

Classi di memorizzazione: esempi

- ```
  extern int iMiaVariabile;
  ...
  int miafunzione(...) {
      iMiaVariabile = ...
  }
  int iMiaVariabile = 123;
  ...
  ```

  In questo esempio, la variabile `iMiaVariabile` è dichiarata formalmente in una posizione centrale del file sorgente; per fare in modo che la funzione `miafunzione` possa accedervi, è stata necessaria la dichiarazione `extern` iniziale.

- ```
  extern int iTuaVariabile;
  ...
  int miafunzione(...) {
      iTuaVariabile = ...
  }
  ...
  ```

  Questo caso rappresenta la situazione in cui una variabile dichiarata in un altro file sorgente diventa accessibile alle funzioni del file attuale attraverso la dichiarazione `extern`. Perché ciò possa funzionare, occorre che la variabile `iTuaVariabile` sia stata dichiarata in modo normale, senza la parola chiave `static`.

I/O elementare

- Con il linguaggio C, l'I/O elementare si ottiene attraverso l'uso di due funzioni fondamentali: `printf()` e `scanf()`.

- La prima si occupa di emettere una stringa dopo averla trasformata in base a determinati codici di formattazione; la seconda si occupa di ricevere input e di trasformarlo secondo determinati codici di formattazione.

- All'inizio dell'esecuzione di un programma il sistema apre tre file standard, `stdin`, `stdout` e `stderr`. Di solito, il file `stdin` è associato alla tastiera, mentre i file `stdout` e `stderr` sono associati allo schermo.

- Per utilizzare queste due funzioni, occorre includere il file di intestazione `stdio.h`.

printf()

```
int printf( <stringa-di-formato>[, <espressione>]... )
```

- `printf()` scrive sul file di output standard `stdout` la stringa indicata come primo parametro, dopo averla rielaborata in base alla presenza di metavariabili riferite alle eventuali espressioni che compongono i parametri successivi. Restituisce il numero di caratteri emessi.

- Se viene fornito a `printf()` un unico parametro stringa, questa viene emessa così com'è, senza trasformazioni.

  Se vengono forniti anche altri parametri, il loro contenuto sostituirà le metavariabili corrispondenti inserite nel parametro stringa.

- `printf("Il capitale di %d al tasso %f ha fruttato %d", 100, 0.05, 105);` emette la frase seguente:

  ```
  Il capitale di 100 al tasso 0.05 ha fruttato 105
  ```

  Al posto della prima metavariabile `%d` è stato inserito il valore 100 dopo averlo convertito in modo da essere rappresentato da tre caratteri (`'1'`, `'0'`, `'0'`); al posto della seconda metavariabile `%f` è stato inserito il valore 0.05 dopo un'opportuna conversione in caratteri; ...

Formattazione di stringhe: metavariabili

- Quando, attraverso lo standard output, si vogliono visualizzare informazioni sullo schermo, si rende necessaria una appropriata conversione dei tipi di dato in sequenze di caratteri.

- La scelta della metavariabile corretta determina il tipo di trasformazione che il parametro corrispondente deve subire.

```
%c      Un carattere singolo.
%s      Una stringa.
%d,i    Un intero decimale con segno.
%u      Un intero decimale senza segno.
%o      Un intero ottale senza segno.
%x,X    Un intero esadecimale senza segno.
%e,E    Un numero in virgola mobile, in notazione scientifica.
%f      Un numero in virgola mobile, in notazione decimale fissa.
%g,G    Un numero nel piu' breve tra i formati %e, %E o %f.
```

- (Per rappresentare il simbolo di percentuale si usa una metavariabile fasulla composta dalla sequenza di due segni percentuali: %%.)

Simboli aggiuntivi

- Le metavariabili possono contenere informazioni aggiuntive tra il simbolo di percentuale e la lettera che definisce il tipo di trasformazione.

```
%[<simbolo>][<ampiezza>][.<precisione>][{h|l|L}]<tipo>
```

- Si può indicare il numero di caratteri complessivo (*ampiezza*), e, in presenza di valori numerici in virgola mobile, anche il numero di decimali (*precisione*). È anche possibile specificare il tipo particolare a cui appartiene il dato immesso, attraverso una lettera: `h` (`short`), `l` (`long`) e `L` (`double`). Se manca questa indicazione, si intende che si tratti di un intero normale (`int`).

- Il simbolo può essere

```
spazio   Il prefisso di un numero positivo e' uno spazio.
+        Il prefisso di un numero positivo e' il segno +.
-        Allinea a sinistra rispetto al campo.
0        Utilizza zeri, invece di spazi, per allineare a destra.
```

- Nella stringa di formattazione possono apparire anche sequenze di escape.

scanf()

```
int scanf( <stringa-di-formato>[, <puntatore>]... )
```

- `scanf()` legge l'input dal file di input standard `stdin` interpretandolo opportunamente secondo le metavariabili inserite nella stringa di formattazione (la stringa di formattazione deve contenere solo metavariabili).

- ```
  printf( "Inserisci l'importo:" );
  scanf( "%d", &iImporto );
  ```

  emette la frase seguente

  ```
  Inserisci l'importo:
  ```

  e resta in attesa dell'inserimento di un valore numerico intero, seguito da Enter. Il valore verrà inserito nella variabile `iImporto`.

- I parametri successivi alla stringa di formattazione sono dei puntatori (cioè indirizzi), per cui, avendo voluto inserire il dato nella variabile `iImporto`, questa è stata indicata preceduta dall'operatore `&` in modo da fornire alla funzione l'indirizzo corrispondente.
  ```

scanf()

- Con una stessa funzione `scanf()` è possibile inserire dati per diverse variabili (per ogni dato viene richiesta la pressione di Enter).

```
printf( "Inserisci il capitale e il tasso:" );
scanf( "%d%f", &iCapitale, &iTasso );
```

- Quando si inserisce un dato che non sia un semplice carattere alfanumerico, occorre una conversione adatta nel tipo di dato corretto.

- Le metavariabili utilizzabili sono simili a quelle già viste per `printf()`; in particolare, tra % ed il carattere di conversione, si può inserire un intero per specificare la dimensione massima della scansione.

- `scanf()` restituisce il numero di elementi che sono stati letti con successo, intendendo con questo non solo il completamento della lettura, ma anche il fatto che i dati inseriti risultano corretti in funzione delle metavariabili indicate.

`fprintf()` e `fscanf()`

- `fprintf()` e `fscanf()` sono le versioni relative ai file di `printf()` e `scanf()`.

- Rispetto a `printf()` e `scanf()`, `fprintf()` e `fscanf()` prendono un ulteriore primo parametro di tipo `FILE *` (puntatore a `FILE`) che specifica il file su cui la funzione agisce.

- `fprintf(stdout,...);` equivale a `printf(...);`
  `fscanf(stdin,...);` equivale a `scanf(...);`

- `sprintf()` e `sscanf()` sono le versioni relative alle stringhe di `printf()` e `scanf()`. La stringa su cui agiscono viene passata come primo parametro.

getchar() e putchar()

- `getchar()` e `putchar()` sono delle macro definite in `stdio.h` utilizzate, rispettivamente, per leggere caratteri dal file `stdin` associato alla tastiera e per scrivere caratteri nel file `stdout` associato allo schermo.

  Vediamo un programma che copia l'input sull'output eliminando gli spazi.

```
#include <stdio.h>

int main(void) {
   int c;
   while ( (c = getchar()) != EOF )
       if ( c != ' ' ) putchar(c);
   return 0;
}
```

- `getc()` e `putc()` sono le loro generalizzazioni in quanto possono operare su file qualsiasi il cui puntatore (di tipo `FILE *`) è passato loro come parametro.

Restituzione di un valore

- I programmi, di qualunque tipo siano, al termine della loro esecuzione, restituiscono un valore che può essere utilizzato da uno script di shell per determinare se il programma ha fatto ciò che si voleva o se è intervenuto qualche tipo di evento che lo ha impedito.

- Convenzionalmente si tratta di un valore numerico, in cui 0 rappresenta una conclusione normale, ovvero priva di eventi indesiderati, mentre qualsiasi altro valore rappresenta un'anomalia.

- Se nel sorgente C non si fa nulla per definire il valore restituito questo sarà sempre 0; per agire diversamente bisogna utilizzare la funzione `exit()`.

exit()

```
exit( <valore restituito> )
```

- La funzione `exit()` provoca la conclusione del programma, dopo aver provveduto a scaricare i flussi di dati e a chiudere i file. Per questo motivo, non restituisce un valore all'interno del programma, al contrario, fa in modo che il programma restituisca il valore indicato come argomento.

- Per poterla utilizzare occorre includere il file di intestazione `stdlib.h` che tra l'altro dichiara già due macro adatte a definire la conclusione corretta o errata del programma: `EXIT_SUCCESS` (equivale a 0) e `EXIT_FAILURE` (equivale a 1).

```
#include stdlib.h
...
...
if (...) {
    exit( EXIT_SUCCESS );
} else {
    exit( EXIT_FAILURE );
}
```

Puntatori, array e stringhe

- Abbiamo finora visto i tipi di dato *primitivi*, cioè quelli a cui si fa riferimento attraverso un nome.

- Per poter utilizzare strutture di dati più complesse, come gli array o altri tipi più articolati, si devono gestire dei *puntatori* alle zone di memoria contenenti tali strutture.

- Quando si ha a che fare con i puntatori, è importante considerare che il modello di memoria che si ha di fronte è un'astrazione, nel senso che una struttura di dati appare idealmente continua, mentre nella realtà il compilatore potrebbe anche provvedere a scomporla in blocchi separati.

Puntatori

- Una variabile, di qualunque tipo sia, rappresenta un valore posto da qualche parte nella memoria del sistema. Quando si usano i tipi di dati primitivi, è il compilatore a prendersi cura di tradurre i riferimenti agli spazi di memoria rappresentati simbolicamente attraverso dei nomi.

- Attraverso l'operatore di indirizzamento `&`, è possibile ottenere il *puntatore* a (o *indirizzo* di) una variabile.

- Un puntatore è un valore appartenente ai tipi di dato primitivi e può essere a sua volta assegnato a una variabile di tipo puntatore.

- Se `p` è una variabile puntatore adatta a contenere l'indirizzo di un intero, è possibile assegnargli il puntatore alla variabile intera `i`.

```
int i = 10;
...
p = &i;
```

Dichiarazione delle variabili puntatore

- La dichiarazione di una variabile puntatore avviene in modo simile a quello delle variabili normali, con l'aggiunta di un asterisco davanti al nome.

  ```
  int *p;
  ```

  dichiara la variabile `p` come puntatore a un tipo `int`. È assolutamente necessario indicare il tipo di dato a cui si punta.

- Una volta dichiarata la variabile puntatore, questa viene utilizzata normalmente, senza asterisco, finché si intende fare riferimento al puntatore stesso.

- L'asterisco usato nella dichiarazione serve a definire il tipo di dato e non fa parte del nome della variabile; quindi, la dichiarazione `int *p`, rappresenta la dichiarazione di una variabile di tipo `int *`.

Utilizzo delle variabili puntatore

- Attraverso l'operatore di *dereferenziazione* `*`, è possibile accedere alla zona di memoria a cui la variabile punta.

  Dereferenziare significa togliere il riferimento e raggiungere i dati a cui un puntatore si riferisce.

- ```
  int i = 10;
  int *p;
  ...
  p = &i;
  ```

  Dopo aver assegnato a `p` il puntatore alla variabile `i`, è possibile accedere alla stessa area di memoria in due modi diversi: attraverso la stessa variabile `i`, oppure attraverso `*p`.

  ```
  ...
  *p = 20
  ```

  L'istruzione `*p=20` è tecnicamente equivalente a `i=20`.

  In pratica, `*&i` equivale a `i`.
  ```

Utilizzo delle variabili puntatore

- ```
  int i = 10;
  int *p;
  int *p2;
  ...
  p = &i;
  ...
  p2 = p;
  ...
  *p2 = 20
  ```

  È stata aggiunta una seconda variabile puntatore, `p2`, per mostrare che è possibile passare un puntatore anche ad altre variabili, e in tal caso non si deve usare l'asterisco. In questo caso, `*p2 = 20` è tecnicamente equivalente sia a `*p = 20` che a `i = 20`.
  ```

Passaggio di parametri per riferimento

- Il linguaggio C utilizza il passaggio dei parametri alle funzioni per *valore*.

- Per ottenere il passaggio per *riferimento* o per *indirizzo* occorre utilizzare i puntatori: invece di passare il valore della variabile da modificare, si può passare il suo puntatore, e la funzione, fatta appositamente per questo, agirà nell'area di memoria a cui punta questo puntatore.

```
...
void funzione_stupida( int *x ) {
    (*x)++;
}
...
int main(void) {
    int y = 10;
    ...
    funzione_stupida( &y );
    ...
}
```

L'esempio mostra la dichiarazione e descrizione di una funzione che non restituisce alcun valore, e riceve un parametro puntatore a un intero. Il lavoro della funzione è solo quello di incrementare il valore contenuto nell'area di memoria a cui si riferisce tale puntatore.

Passaggio di parametri per riferimento

- Se quindi si vuole usare una variabile come parametro di una funzione per restituire un valore di ritorno, alla funzione bisogna effettivamente passare come parametro l'indirizzo della variabile.

- Riepilogando, l'effetto del passaggio per riferimento è ottenuto
  - dichiarando il parametro della funzione come puntatore,
  - utilizzando il puntatore dereferenziato nel corpo della funzione,
  - passando un indirizzo come parametro quando la funzione viene chiamata.

Array

- Un *array* è una sequenza ordinata di elementi dello stesso tipo.

- Quando si dichiara un array, il programmatore ottiene il riferimento (puntatore) alla posizione iniziale di questo; gli elementi successivi verranno raggiunti tenendo conto della lunghezza di ogni elemento.

- Dal momento che quando si dichiara l'array si ottiene solo il riferimento al suo inizio, è compito del programmatore ricordare la dimensione massima dell'array, perché non c'è alcun modo per determinarla durante l'esecuzione del programma.

- La principale differenza tra un array ed un puntatore è che mentre il valore del puntatore (cioè l'indirizzo che contiene) può essere modificato, l'indirizzo iniziale dell'area di memoria riservata per l'array risulta associato al nome dell'array una volta per tutte.

- Infatti, quando un programma tenta di accedere a una posizione oltre il limite degli elementi esistenti, c'è il rischio che non si verifichi alcun errore, arrivando però a dei risultati imprevedibili.

Dichiarazione ed utilizzo degli array

- La dichiarazione di un array avviene definendo il tipo degli elementi e la loro quantità.

  ```
  int a[7];
  ```

  dichiara l'array `a` di 7 elementi di tipo `int`.

- Per accedere agli elementi dell'array si utilizza un indice, il cui valore iniziale è sempre 0; di conseguenza, l'ultimo elemento ha indice n-1, dove n corrisponde alla quantità di elementi esistenti.

  ```
  ...
  a[1] = 123;
  ```

  assegna il valore 123 al secondo elemento.

Inizializzazione di un array

- In presenza di un array di piccole dimensioni può essere sensato attribuire un valore iniziale ai suoi elementi, all'atto della dichiarazione.

- `int a[] = { 123, 453, 2, 67 };`

  L'esempio dovrebbe chiarire il modo: non occorre specificare il numero di elementi, perché questi sono esattamente quelli elencati nel raggruppamento tra le parentesi graffe.

- Alcuni compilatori consentono l'inizializzazione degli array contestualmente alla loro dichiarazione solo quando questi sono dichiarati all'esterno delle funzioni, e quindi hanno un campo di azione globale, e quando, all'interno delle funzioni, sono dichiarati `static`.

- Non è possibile assegnare a un array un altro array. L'assegnazione va fatta elemento per elemento.

Scansione di un array

- La scansione di un array avviene generalmente attraverso un'iterazione enumerativa, in pratica con un ciclo `for` che si presta particolarmente per questo scopo.

```
int a[7];
int i;
...
for (i = 0; i < 7, i++) {
    ...
    a[i] = ...;
    ...
}
```

Per scandire un array in senso opposto, si può agire in modo analogo.

```
int a[7];
int i;
...
for (i = 6; i >= 0, i--) {
    ...
    a[i] = ...;
    ...
}
```

Array e puntatori

- La variabile che contiene l'indirizzo iniziale di un array, e che rappresenta l'array stesso, è in sola lettura.

  Data la filosofia del linguaggio C, se fosse possibile assegnare un valore alla variabile che rappresenta un array, si modificherebbe il puntatore, facendo in modo che questo punti ad un array differente.

- ```
  #include <stdio.h>

  int main(void) {
      int ai[3];
      int *pi;

      pi = ai; /* pi diventa un alias dell'array ai */
      pi[0] = 10;
      pi[1] = 100;
      pi[2] = 1000;
      printf ( "%d %d %d \n",  ai[0], ai[1], ai[2] );
      return 0;
  }
  ```

  Viene creato un array, `ai`, di tre elementi di tipo `int`, e subito dopo una variabile puntatore, `pi`, al tipo `int`. Si assegna quindi alla variabile `pi` il puntatore rappresentato da `ai`; da quel momento si può fare riferimento all'array indifferentemente con il nome `ai` o `pi`.
  ```

Aritmetica dei puntatori

- Sulle variabili puntatore è possibile eseguire delle operazioni aritmetiche.

```
int a[10], *p;
...
p = a;
...
++p;
...
a[0] = *(p + 3);
...
```

- Il risultato che si ottiene è il riferimento a una zona di memoria adiacente, in funzione della dimensione del tipo di dato per il quale è stato creato il puntatore.

Aritmetica dei puntatori

- La dichiarazione di un array è in pratica la dichiarazione di un puntatore al tipo di dato degli elementi di cui questo è composto.

```
int ai[3] = { 1, 3, 5 };
int *pi;
...
pi = ai;
```

  Il puntatore `pi` punta all'inizio dell'array.

```
...
*pi = 10; /* equivale a:  ai[0] = 10 */
pi++;
*pi = 30; /* equivale a:  ai[1] = 30 */
pi++;
*pi = 50; /* equivale a:  ai[2] = 50 */
```

- *Incrementando* il puntatore si accede all'elemento successivo adiacente, in funzione della dimensione del tipo di dato.

  Decrementando il puntatore si ottiene l'effetto opposto, di accedere all'elemento precedente.

- È compito del programmatore sapere quando l'incremento o il decremento di un puntatore ha significato. Altrimenti si rischia di accedere a zone di memoria estranee all'array, con risultati imprevedibili.

Array e puntatori

- Data la dichiarazione

```
int a[10], *p;
```

  - `p = a` equivale a `p = &a[0]`
    e, in generale, per $0 \leq i < 10$
    `p = a + i` equivale a `p = &a[i]`.

  - `a[i]`, che denota l'`i`-esimo elemento dell'array `a`, equivale a `*(a + i)`, che è il dereferenziamento di `a + i` (espressione con i puntatori che restituisce l'indirizzo corrispondente ad `i` interi dopo `a`).

  - `p[i]` e `*(p + i)` sono equivalenti.

  - `&*a` e `&a[0]` sono equivalenti.

  - `a = p` è scorretto perché `a` è un puntatore costante.

  - `++p` è corretto mentre `++a` è scorretto.

Array multidimensionali

- Si possono creare array i cui elementi sono array tutti uguali.

  ```
  int a[2][3];
  ```

  dichiara un array di 2 elementi che a loro volta sono array di 3 elementi di tipo `int`.

- Si può pensare che gli elementi dell'array siano astrattamente organizzati in memoria nel modo seguente (in realtà sono memorizzati in maniera contigua):

  ```
           colonna 1  colonna2  colonna3
  riga 1   a[0][0]    a[0][1]   a[0][2]
  riga 2   a[1][0]    a[1][1]   a[1][2]
  ```

- Nello stesso modo si possono definire array con più di due dimensioni.

Array multidimensionali

- Per via delle relazioni esistenti tra puntatori ed array, esistono vari modi di accedere gli elementi di un array bidimensionale. Le seguenti espressioni sono tutte equivalenti.

```
a[i][j]           (*(a + i))[j]
*(a[i] + j)       *((*(a + i)) + j)
*(&a[0][0] + 3*i + j)   e' la mappa di memorizzazione
```

```
a[i]    *(a + i)    &a[0]+i    &a[i]    &a[i][0]
```

  L'indirizzo base dell'array è `&a[0][0]`.

- ```
  int a[2][3];
  int i;
  int j;
  ...
  for (i = 0; i < 2; i++) {
      ...
      for  (j = 0; j < 3; j++) {
          ...
          a[i][j] = ...;   /* a[i,j] e' scorretto */
          ...
      }
      ...
  }
  ```
  ```

Array come parametri di funzioni

- Quando un array viene passato come parametro ad una funzione, in realtà è il puntatore all'array che viene passato. Di conseguenza, le modifiche che verranno apportate da parte della funzione si rifletteranno nell'array di origine.

```
#include <stdio.h>

void elabora( int *pi ) {
    pi[0] = 10;
    pi[1] = 100;
    pi[2] = 1000;
}

int main(void) {
    int ai[3];
    elabora( ai );
    printf ( "%d %d %d \n",  ai[0], ai[1], ai[2] );
    return 0;
}
```

- La funzione `elabora()` utilizza un solo parametro, rappresentato da un puntatore a un tipo `int`.

- La funzione presuppone che il puntatore si riferisca all'inizio di un array di interi e così assegna alcuni valori ai primi tre elementi (anche il numero degli elementi non può essere determinato dalla funzione).

Array come parametri di funzioni

- In pratica, nell'intestazione di una funzione, `int *pi` e `int pi[]` sono equivalenti.

  L'esempio potrebbe essere modificato per presentare la gestione dell'array in modo più elegante, ponendo l'accento sul fatto che l'argomento è un array di interi.

```
void elabora( int ai[] ) {
    ai[0] = 10;
    ai[1] = 100;
    ai[2] = 1000;
}
```

- Quando si passa un array multidimensionale ad una funzione, si devono specificare tutte le dimensioni eccetto la prima, in modo che il compilatore possa allocare la memoria correttamente.

- Infatti, essendo un array di interi un puntatore a un intero, questa notazione fa sì che la lettura del sorgente diventi più facile.

```
void elabora( int ai[] ) {
    ai[0] = 10;
    ai[1] = 100;
    ai[2] = 1000;
}
```

- Volendo si può anche specificare la dimensione dell'array passato come parametro.

- Per esempio, nel caso di un array bidimensionale, l'incremento unitario del puntatore dipende dalla seconda dimensione, cioè dalla lunghezza della riga.

Stringhe

- Le *stringhe*, nel linguaggio C, non sono un tipo di dato a sé stante; si tratta solo di un array di caratteri con una particolarità: l'ultimo carattere è sempre `\0` (pari a una sequenza di bit a zero).

- In questo modo, si evita di dover accompagnare le stringhe con l'informazione della loro lunghezza, dal momento che il C non offre un metodo per determinare la dimensione degli array.

- Con questa premessa, si può intendere che il trattamento delle stringhe in C non sia una cosa tanto agevole; in particolare non si possono usare operatori di concatenamento. Per tutti i tipi di elaborazione occorre intervenire a livello di array di caratteri.

- C mette a disposizione un insieme di funzioni di manipolazione di stringhe, quali ad esempio `strcpy`, `strcat`, `strlen`, `strcmp`, `strstr` e `strchr`, i cui prototipi si trovano in `string.h`.

Array di caratteri e array stringa

- Una stringa è un array di caratteri, ma un array di caratteri non è necessariamente una stringa: per esserlo occorre che l'ultimo elemento sia il carattere \0.

```
char ac[20];

char ac[] = { 'c', 'i', 'a', 'o' };

char acz[] = { 'c', 'i', 'a', 'o', '\0' };

char acz[] = "ciao";
```

- Bisogna ricordare che la stringa, anche se espressa in forma letterale (delimitata attraverso gli apici doppi), è un array di caratteri, e come tale restituisce semplicemente il puntatore al primo di questi caratteri.

```
char *pc;
...
pc = "ciao";
...
```

- Non esistendo un tipo di dato *stringa*, si può assegnare una stringa solo a un puntatore al tipo `char`.

- Il *primo* esempio mostra la dichiarazione di un array di 20 caratteri.

- Il *secondo* esempio mostra la dichiarazione di un array di 4 caratteri. All'interno delle parentesi quadre non è stata specificata la dimensione perché questa si determina dall'inizializzazione. Anche in questo caso non si può ancora parlare di stringa, perché manca la terminazione.

- Il *terzo* esempio mostra la dichiarazione di un array di 5 caratteri corrispondente a una stringa vera e propria.

- Il *quarto* esempio è tecnicamente equivalente, solo che utilizza una rappresentazione più semplice. Pertanto, la stringa `"ciao"` è un array di cinque caratteri perché rappresenta implicitamente anche la terminazione.

Array di caratteri e array stringa

- L'esempio seguente non è valido, perché, al solito, non si può assegnare un valore alla variabile che rappresenta un array.

```
char ac[];
...
ac = "ciao"; /* non si puo' */
...
```

Stringhe come parametri di funzioni

- Quando si utilizza una stringa come parametro della chiamata di una funzione, questa riceverà il puntatore all'inizio della stringa (si ripete la stessa situazione già vista per gli array in generale).

```
#include <stdio.h>

void elabora( char *acz ) {
    printf ( acz );
}

int main(void) {
    elabora( "ciao\n" );
    return 0;
}
```

- Volendo scrivere il codice in modo più elegante si poteva dichiarare esplicitamente la variabile ricevente come array di caratteri di dimensione indefinita.

```
...

void elabora( char acz[] ) {
    printf ( acz );
}

...
```

- L'esempio mostra una funzione banale che si occupa semplicemente di emettere la stringa ricevuta come parametro, utilizzando `printf()`. La variabile utilizzata per ricevere la stringa è stata dichiarata come puntatore al tipo `char`, poi tale puntatore è stato utilizzato come parametro per la funzione `printf()`.

Parametri della funzione main()

- La funzione `main()`, se viene dichiarata con i suoi parametri tradizionali, permette di acquisire la riga di comando utilizzata per avviare il programma.

```
int main( int argc, char *argv[] ) {
   ...
}
```

- Gli argomenti della riga di comando vengono convertiti in un array di stringhe (cioè di puntatori a `char`) `argv`, in cui il primo elemento è il nome utilizzato per avviare il programma e gli elementi successivi sono gli argomenti.

- Il primo parametro, `argc`, serve a contenere la dimensione di `argv` (quindi, `argc` è maggiore di 0).

```
#include <stdio.h>

int main( int argc, char *argv[] ) {
    int i;
    printf( "Il programma si chiama %s ", argv[0] );
    printf( "ed i suoi %i argomenti sono:\n", argc-1 );
    for ( i = 1; i < argc ; i++ ) {
        printf( "argomento n. %d: %s\n", i, argv[i] );
    }
    return 0;
}
```

- Gli argomenti di `main()` sono così chiamati per convenzione, ma non è un obligo.

- `argv` avrà sempre almeno un elemento: il nome utilizzato per avviare il programma, e di conseguenza, `argc` sarà sempre maggiore o uguale a 1.

- `argv` è un array di stringhe, non un array di array di caratteri. Difatti, da un punto di vista logico è meglio pensare ad un argomento come una stringa di caratteri.

Puntatori e funzioni

- Nello standard ANSI C, la dichiarazione di una funzione è in pratica la definizione di un puntatore alla funzione stessa, un pò come accade con gli array.

- È possibile dichiarare dei puntatori a un tipo di funzione definito in base al tipo del valore restituito e ai tipi di parametri richiesti.

```
<tipo> (*<nome-puntatore>)(<tipo-parametro>[,...]);
```

- L'esempio seguente mostra la dichiarazione di un puntatore a una funzione che restituisce un valore di tipo `int` e utilizza due parametri di tipo `int`.

```
int (*f)(int, int);
```

- L'assegnamento del puntatore avviene trattando il nome della funzione nello stesso modo in cui si fa con gli array: come un puntatore.

```
int (*f)( int, int );
int prodotto( int, int ); /* prototipo di funzione */
...
f = prodotto; /* f contiene il riferimento alla funzione */
```

Puntatori e funzioni

- Una volta assegnato il puntatore, si può eseguire una chiamata di funzione semplicemente utilizzando il puntatore, per cui,

```
i = f( 2, 3 );
```

risulta equivalente a

```
i = prodotto( 2, 3 );
```

- Nel linguaggio C precedente allo standard ANSI, affinché il puntatore potesse essere utilizzato in una chiamata di funzione, occorreva dereferenziarlo.

```
i = (*f)( 2, 3 );
```

Tipi enumerativi

- Un tipo enumerativo è un insieme finito di elementi (diversi) ciascuno identificato da un nome.

  ```
  enum boolean { false, true };
  ```

- Gli identificatori degli elementi sono costanti di tipo `int` dette *enumeratori*. Il primo vale per default `0`, ed i successivi hanno valori consecutivi.

- Gli enumeratori sono in pratica costanti (per esempio, possono essere usate come etichette di un `case`) definite dal programmatore il cui uso può migliorare la chiarezza dei programmi.

- È possibile inizializzare esplicitamente gli enumeratori (eventualmente con valori ripetuti).

  ```
  enum fruit { apple=7, pear, orange=3, lemon }
  ```

  Effetto: apple=7, pear=8, orange=3, lemon=4.

- Si può accedere al valore di un enumeratore con un cast.

Tipi di dato derivati

- Abbiamo visto finora i tipi di dato primitivi e gli array di questi (incluse le stringhe).

- Nel linguaggio C, come in altri linguaggi di programmazione, è possibile definire dei tipi di dato *derivati* dai tipi primitivi, quali

  – *strutture*

  – *unioni*.

Strutture

- Si è visto che gli array sono organizzati come una serie di elementi uguali, tutti adiacenti nel modello di rappresentazione della memoria (ideale o reale che sia).
- In modo simile si possono definire strutture dati più complesse in cui gli elementi adiacenti sono di tipo differente.

  Le componenti di una struttura sono chiamate *membri*.
- In pratica, una *struttura* è una sorta di mappa di accesso ad un'area di memoria.
- Una variabile di tipo struttura, analogamente alle variabili di tipo primitivo, rappresenta tutta la zona di memoria occupata dalla struttura che contiene, e non solo il riferimento al suo inizio, come invece accade nel caso delle variabili di tipo array che in realtà sono solo dei puntatori.
- Le strutture possono essere passate come parametri (per valore) a funzioni e possono essere restituite da funzioni.

Dichiarazione di una struttura

- La dichiarazione di una struttura si articola in due fasi: la dichiarazione del tipo e la dichiarazione delle variabili che utilizzano quella struttura.

- Dal momento che il tipo di una struttura è una cosa diversa dalle variabili che l'utilizzeranno, è opportuno stabilire una *convenzione* sul modo di attribuirne il nome.

  Per esempio, si potrebbero utilizzare nomi con iniziale maiuscola.

```
struct Data { int iGiorno; int iMese; int iAnno; };
```

- L'esempio mostra la dichiarazione della struttura `Data` composta da tre interi dedicati a contenere rispettivamente: il giorno, il mese e l'anno.

```
struct Data {
    int iGiorno;
    int iMese;
    int iAnno;
};        /* il punto e virgola finale e' necessario */
```

- In questo caso, trattandosi di tre membri dello stesso tipo si sarebbe potuto utilizzare un array, ma come si vedrà in seguito, una struttura può essere conveniente anche in queste situazioni.

- È importante osservare che le parentesi graffe sono parte dell'istruzione di dichiarazione della struttura, e non rappresentano un blocco di istruzioni.

Dichiarazione di una struttura

- La dichiarazione delle variabili che utilizzano la struttura può avvenire contestualmente con la dichiarazione della struttura, oppure in un momento successivo.

- L'esempio seguente mostra la dichiarazione del tipo `Data`, seguito da un elenco di variabili che utilizzano quel tipo: `DataInizio` e `DataFine`.

```
struct Data {
    int iGiorno;
    int iMese;
    int iAnno;
} DataInizio, DataFine;
```

- Tuttavia, il modo più elegante per dichiarare delle variabili a partire da una struttura è quello seguente:

```
struct Data {
    int iGiorno;
    int iMese;
    int iAnno;
};
...
struct Data DataInizio, DataFine;
```

Accesso ad una struttura

- Quando una variabile è stata definita come organizzata secondo una certa struttura, si accede ai suoi membri attraverso l'indicazione del nome della variabile stessa, seguito dall'operatore punto (`.`) e dal nome del membro particolare.

```
DataInizio.iGiorno = 1;
DataInizio.iMese = 1;
DataInizio.iAnno = 1980;
...
DataFine.iGiorno = DataInizio.iGiorno;
DataFine.iMese = DataInizio.iMese +1;
DataFine.iAnno = DataInizio.iAnno;
```

Strutture anonime

- Una struttura può essere dichiarata in modo anonimo, definendo immediatamente tutte le variabili che fanno uso di quella struttura.

- La differenza sta nel fatto che la struttura non viene nominata nel momento della dichiarazione, e dopo la definizione dei suoi membri devono essere elencate tutte le variabili in questione.

- Non c'è la possibilità di riutilizzare questa struttura per altre variabili definite in un altro punto.

```
struct {
    int iGiorno;
    int iMese;
    int iAnno;
} DataInizio, DataFine;
```

Assegnamento e inizializzazione

- È possibile assegnare ad una variabile di tipo struttura l'intero contenuto di un'altra che appartiene alla stessa struttura.

```
DataInizio.iGiorno = 1;
DataInizio.iMese = 1;
DataInizio.iAnno = 1999;
...
DataFine = DataInizio;
DataFine.iMese++;
```

- Nel momento della dichiarazione di una struttura, è possibile anche inizializzarla utilizzando una forma simile a quella disponibile per gli array.

```
struct Data DataInizio = { 1, 1, 1999 };
```

- L'esempio mostra l'assegnamento alla variabile `DataFine` di tutta la variabile `DataInizio`. Questo è ammissibile solo perché si tratta di variabili dello stesso tipo.

- Se invece si trattasse di variabili costruite a partire da strutture differenti, anche se realizzate nello stesso modo, ciò non sarebbe ammissibile.

Campo d'azione

- Dal momento che le strutture sono tipi di dato non predefiniti, per poterne fare uso occorre che la dichiarazione relativa sia accessibile a tutte le parti del programma che hanno bisogno di accedervi.

- Il luogo più adatto è al di fuori delle funzioni, eventualmente anche in un file di intestazione realizzato appositamente.

- Le variabili che contengono una struttura vengono passate regolarmente alle funzioni, purché la dichiarazione del tipo corrispondente sia precedente ed esterno alla descrizione delle funzioni stesse.

```
...
struct Data { int iGiorno; int iMese; int iAnno; };
...
void elabora( struct Data DataAttuale ) {
    ...
}
```

Strutture e puntatori

- Così come nel caso dei tipi primitivi, si possono creare dei puntatori ad oggetti di tipo struttura.

```
struct Data *pDataFutura;
...
pDataFutura = &DataFine;
...
```

- L'operatore punto che unisce il nome della struttura a quello di un membro ha priorità rispetto all'asterisco che dereferenzia un puntatore.

```
*pDataFutura.iGiorno = 15;
```

  non è valido perché l'asterisco posto davanti viene valutato dopo l'elemento `pDataFutura.iGiorno` che non esiste.

  Per risolvere il problema si possono usare le parentesi.

```
(*pDataFutura).iGiorno = 15;      /* corretto */
```

- In alternativa, si può usare l'operatore `->`, fatto espressamente per i puntatori a una struttura.

```
pDataFutura->iGiorno = 15;      /* corretto */
```

Unioni

- Un *unione* è un tipo di dato accessibile in modi diversi, gestito come se si trattasse contemporaneamente di tipi differenti.

- La dichiarazione è simile a quella di una struttura; bisogna però tenere presente che i membri componenti condividono la stessa area di memoria.

```
union Livello {
    char cLivello;
    int iLivello;
};
```

  Il programmatore è responsabile della corretta interpretazione dei valori memorizzati.

- Anche la dichiarazione di variabili è simile a quella delle variabili di un tipo struttura (così come pure l'accesso ai membri).

```
union Livello LivelloCarburante;
...
LivelloCarburante.iLivello = 7;
```

- Si immagini, per esempio, di voler utilizzare indifferentemente una serie di lettere alfabetiche, oppure una serie di numeri, per definire un livello di qualcosa (`A` equivalente a 1, `B` equivalente a 2, ecc.).

- Le variabili generate a partire da questa unione, possono essere gestite in questi due modi, come se fossero una struttura, ma condividendo la stessa area di memoria.

- L'esempio mostra in che modo si possa dichiarare una variabile di tipo `Livello`, riferita all'omonima unione.

Unioni

- L'utilità delle unioni risiede nella possibilità di combinarle con le strutture.

- Il seguente esempio serve a chiarire le possibiltà.

```
struct Livello {
    char cTipo;
    union {
        char cLivello;    /* usato se cTipo == 'c' */
        int iLivello;     /* usato se cTipo == 'n' */
    };
};
```

  La variabile `cTipo` serve ad annotare il tipo di informazione contenuta nell'unione, se di tipo carattere o numerico.

  L'unione viene dichiarata in modo anonimo come appartenente alla struttura.

Campi

- Un elemento `int` o `unsigned` di una struttura può essere dichiarato come composto da uno specifico numero di bit.

  Tale membro è chiamato *campo* di bit ed il numero di bit componenti è la sua *ampiezza*.

- I campi possono essere utili per risparmiare memoria.

- ```
  struct Luci {
      unsigned  bA :1, bB :1, bC :1, bD :1,
                bE :1, bF :1, bG :1, bH :1;
  };
  ```

  L'esempio mostra l'abbinamento di otto nomi ai bit di un tipo `char`. Il primo, `bA`, rappresenta il bit più a destra, ovvero quello meno significativo. Se il tipo `char` occupasse una dimensione maggiore di 8 bit, la parte eccedente verrebbe semplicemente sprecata.

  In pratica viene dato un nome a ogni bit o gruppetto.
  ```

Campi

- ```
  struct Luci LuciSalotto;
  ...
  LuciSalotto.bC = 1;
  ```

  L'esempio mostra la dichiarazione della variabile `LuciSalotto` come appartenente alla struttura mostrata sopra, e poi l'assegnamento del terzo bit a uno, probabilmente per *accendere* la lampada associata.

- Volendo indicare un gruppo di bit maggiore, basta aumentare il numero indicato a fianco dei nomi dei campi.

  ```
  struct Prova {
      unsigned
          bA      :1,
          bB      :1,
          bC      :1,
          stato   :4;
  };
  ```

  Nell'esempio appena mostrato, si usano i primi tre bit in maniera singola (per qualche scopo), e altri 4 per contenere un'informazione più consistente.

  Ciò che resta (probabilmente solo un bit) viene semplicemente ignorato.
  ```

typedef

- L'istruzione `typedef` permette di definire un nuovo di tipo di dato, in modo che il suo uso sia più agevole.

```
typedef <tipo> <nuovo-tipo>
```

- Lo scopo di tutto ciò sta nell'informare il compilatore; `typedef` non ha altri effetti.

- ```
  struct Data {
      int iGiorno;
      int iMese;
      int iAnno;
  };
  typedef struct Data Data;
  Data DataInizio, DataFine;
  ```

  ```
  typedef struct Data {
       int iGiorno;
       int iMese;
       int iAnno;
  } Data;
  Data DataInizio, DataFine;
  ```

  Attraverso `typedef`, è stato definito il tipo `Data`, facilitando così la dichiarazione delle variabili `DataInizio` e `DataFine`. In pratica, `Data` si può utilizzare al posto di `struct Data`.
  ```

Oggetti dinamici

- Fino a questo punto è stato visto l'uso dei puntatori come mezzo per fare riferimento a zone di memoria già allocata.

  Es. assegnare ad un puntatore l'indirizzo di una variabile o di un array.

- È possibile gestire della memoria allocata dinamicamente durante l'esecuzione di un programma, facendovi riferimento attraverso i puntatori, utilizzando apposite funzioni per l'allocazione e deallocazione di questa memoria.

  Ciò evita di creare variabili inutili: si pensi ad esempio a strutture dati di grandezza variabile, come ad esempio le liste.

- Grazie all'uso di routine di gestione della memoria, è possibile definire ed utilizzare *strutture dati dinamiche*, quali ad esempio strutture *autoreferenzianti* definite come strutture contenenti puntatori alle strutture stesse.

Dichiarazione di un puntatore

- Nel file di intestazione `stddef.h` è definita la macro `NULL`, che, convenzionalmente, serve a rappresentare il valore di un puntatore indefinito.

- Solitamente `NULL` corrisponde a `0` o a `((void *)0)`.

- In pratica un puntatore di qualunque tipo che contenga tale valore, rappresenta il riferimento al nulla.

- A seconda del compilatore, la dichiarazione di un puntatore potrebbe coincidere con la sua inizializzazione implicita al valore `NULL`.

- Per essere sicuri che un puntatore sia inizializzato, lo si può fare in modo esplicito.

```
#include <stdio.h>
...
int main (void) {
    int *pi = NULL;
    ...
}
```

- Sul mio sistema, `NULL` corrisponde a `((void *)0)` e la dichiarazione di un puntatore implicitamente lo inizializza a `NULL`.

Verifica di un puntatore

- Dovendo gestire degli oggetti dinamici, prima di utilizzare l'area di memoria a cui dovrebbe fare riferimento un puntatore, è meglio verificare che questo punti effettivamente a qualcosa.

```
...
if ( pi != NULL ) {
    /* il puntatore e' valido e allora procede */
    ...
} else {
    /* la memoria non e' stata allocata
       e si fa qualcosa si alternativo */
    ...
}
```

Allocazione di memoria

- L'allocazione di memoria viene generalmente effettuata attraverso le funzioni `malloc()` e `calloc()`.

```
void *malloc(size_t <dimensione>)
void *calloc(size_t <quantita'>, size_t <dimensione>)
```

- Se queste funzioni riescono ad eseguire l'operazione, allora restituiscono il puntatore alla memoria allocata, altrimenti restituiscono il valore `NULL`.

- `malloc()` viene utilizzata per allocare un'area di una certa dimensione espressa generalmente in byte.

```
a = malloc (n * sizeof(int))
```

- `calloc()` permette di indicare una quantità di elementi e si presta per l'allocazione di array (la memoria allocata da `calloc()` difatti è contigua).

```
a = calloc (n, sizeof(int))
```

- Per utilizzare `malloc()` e `calloc()`, è necessario conoscere la dimensione dei tipi di dato primitivi (per evitare incompatibilità conviene usare `sizeof()`).

Allocazione di memoria

- Il valore restituito `malloc()` e `calloc()` è di tipo `void *`, cioè una specie di puntatore neutro, indipendente dal tipo di dato da utilizzare.

  Prima di assegnare a un puntatore il risultato dell'esecuzione delle funzioni di allocazione, sarebbe opportuno eseguire un cast.

```
int *pi = NULL;
...
pi = (int *)malloc(sizeof(int));

if ( pi != NULL ) {
    /* il puntatore e' valido e allora procede */
    ...
} else {
    /* la memoria non e' stata allocata
       e si fa qualcosa si alternativo */
    ...
}
```

- Lo standard ANSI C non richiede l'utilizzo di questo cast, quindi l'esempio diventa:

```
...
pi = malloc(sizeof(int));
...
```

- Come si può osservare dall'esempio, il cast viene eseguito con la notazione `(int *)` che richiede la conversione esplicita in un puntatore a `int`.

- In ANSI C il cast non sarebbe necessario poiché il valore restituito da `malloc()` è di tipo `void *` e quindi assegnabile a qualunque variabile di tipo puntatore.

- Lo standard ANSI non richiede l'utilizzo del cast esplicito, però il compilatore può dare un warning.

Deallocazione di memoria

- La memoria allocata dinamicamente deve essere liberata in modo esplicito quando non serve più (il linguaggio C non offre alcun meccanismo di *garbage collection*).

- Si utilizza la funzione `free()` che richiede semplicemente il puntatore e non restituisce alcunché.

```
void free(void *<puntatore>)
```

- È necessario evitare di deallocare più di una volta la stessa area di memoria. Ciò può provocare risultati imprevedibili.

```
int *pi = NULL;
...
pi = (int *)malloc(sizeof(int));

if ( pi != NULL ) {
    /* il puntatore e' valido e allora procede */
    ...
    free( pi ); /* libera la memoria */
    pi = NULL; /* per sicurezza azzera il puntatore */
    ...
} else {
    /* la memoria non e' stata allocata
       e si fa qualcosa si alternativo */
    ...
}
```

Strutture autoreferenzianti

- Rientrano in questa categoria strutture dati quali *liste*, *code*, *pile*, *alberi*, *grafi*, ..., molto usate in informatica.

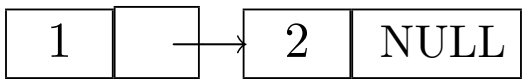
```
struct List {
     int data;
     struct List *next;   /* detto link */
}
```

- Rappresentazione intuitiva 

data next

```
struct List a, b;
a.data = 1;
b.data = 2;
a.next = b.next = NULL;
```


```
a.next = &b;    /* concatena b dopo a */
```



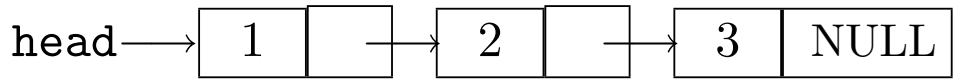
`a.next -> data` ha valore 2.

I *link* permettono di recuperare i dati degli elementi successivi.

Strutture autoreferenzianti

Tali strutture sono particolarmente utili grazie alla possibilità di usarle congiuntamente con funzioni che allocano memoria dinamicamente.

```
typedef struct List LIST;
LIST *head;
...
head = malloc(sizeof(LIST));
head -> data = 1;
head -> next = malloc(sizeof(LIST));
head -> next -> data = 2;
head -> next -> next = malloc(sizeof(LIST));
head -> next -> next -> data = 3;
head -> next -> next -> next = NULL;
...
```



File

- La gestione dei file offerta dal linguaggio C è elementare, a meno di fare uso di librerie specifiche realizzate appositamente per gestioni più sofisticate dei dati.

- A parte questa considerazione, il linguaggio C offre due tipi di accesso ai file; uno *normale* (che vediamo qui di seguito), tramite funzioni di libreria del linguaggio, ed uno *a basso livello* (che vedremo studiando le *system call*), tramite chiamate del sistema operativo.

FILE come tipo di dato

- Nel linguaggio C, i file vengono trattati come un tipo di dato derivato (cioè ottenuto dai tipi elementari esistenti) che corrisponde ad una sequenza di caratteri.

- Quando si apre e si gestisce un file si ha a che fare con un puntatore al file, o *file pointer*, che è una variabile contenente un qualche riferimento univoco al file stesso.

- Per gestire i puntatori ai file, occorre dichiarare una variabile come puntatore al tipo derivato `FILE`.

```
#include <stdio.h>
...
int main(void) {
    FILE *pf;
    ...
}
```

- `FILE` è una macro predefinita (in `stdio.h`) del preprocessore che si traduce in una struttura particolare adatta al tipo di piattaforma utilizzato.

  Il suo utilizzo richiede l'inclusione del file di intestazione `stdio.h`.

EOF

- `EOF` è un'altra macro molto importante anch'essa definita all'interno del file `stdio.h`.

- Si tratta di un valore, definito in base alle caratteristiche della piattaforma, spesso corrispondente a `-1`, utilizzato per rappresentare il raggiungimento della fine del file.

Apertura e chiusura

- L'apertura dei file viene ottenuta normalmente con la funzione `fopen()` che restituisce il puntatore al file, oppure il puntatore nullo, `NULL`, in caso di fallimento dell'operazione.
- Per poter operare su un file è prima necessario apririrlo.
- L'esempio seguente mostra l'apertura del file `mio_file` contenuto nella directory corrente, con una modalità di accesso in sola lettura.

```
#include <stdio.h>
...
int main(void) {
    FILE *pfMioFile;
    ...
    pfMioFile = fopen( "mio_file", "r" );
    ...
}
```

- È normale assegnare il puntatore ottenuto invocando `fopen()` ad una variabile adatta, che da quel momento identificherà il file finché questo resterà aperto.

Apertura e chiusura

- La chiusura del file avviene in modo analogo, attraverso la funzione `fclose()`, che restituisce 0 se l'operazione è stata conclusa con successo, oppure il valore rappresentato da `EOF`.

```
fclose( pfMioFile );
```

- La chiusura del file conclude l'attività con questo, dopo avere scritto tutti i dati eventualmente ancora rimasti in sospeso (se il file era stato aperto in scrittura).

- Normalmente, un file aperto viene definito come *flusso*, o *stream*, e nello stesso modo viene identificata la variabile puntatore che lo riferisce.

- Lo stesso file potrebbe anche essere aperto più volte con puntatori differenti, quindi è corretto distinguere tra file fisici su disco e file aperti, o flussi.

fopen()

```
FILE *fopen( char *<file>, char *<modalita'> )
```

- La funzione `fopen()` permette di aprire il file indicato attraverso la stringa fornita come primo parametro, tenendo conto che tale stringa può contenere anche informazioni sul percorso necessario per raggiungerlo, secondo la modalità stabilita dal secondo parametro, anch'esso una stringa.

- La funzione restituisce un puntatore ad un oggetto di tipo `FILE` e si tratta: o del puntatore al file aperto oppure, in caso di insuccesso, del puntatore nullo (`NULL`).

fopen()

- Modalità di apertura dei file testo:

  `r` apre il file in sola lettura, posizionandosi all'inizio del file;

  `r+` apre il file in lettura e scrittura, posizionandosi all'inizio del file;

  `w` apre il file in sola scrittura, creandolo se necessario, o troncandone a 0 il suo contenuto se questo esisteva già;

  `w+` apre il file in scrittura e lettura, creandolo se necessario, o troncandone a 0 il suo contenuto se questo esisteva già;

  `a` apre il file in scrittura in aggiunta (append), creandolo se necessario, o aggiungendovi dati a partire dalla fine, e di conseguenza posizionandosi alla fine dello stesso;

  `a+` apre il file in scrittura in aggiunta e in lettura, creandolo se necessario, o aggiungendovi dati a partire dalla fine, e di conseguenza posizionandosi alla fine dello stesso.

- Se il file è binario dopo il primo carattere della modalità bisogna aggiungere un `b` (in UNIX non c'è distinzione, in MSDOS si).

fclose()

```
int fclose( FILE *<stream> )
```

- La funzione `fclose()` chiude il file rappresentato dal puntatore indicato come parametro della funzione.

- Se il file era stato aperto in scrittura, prima di chiuderlo vengono scaricati i dati eventualmente accumulati nella memoria tampone (i buffer), utilizzando la funzione `fflush()`.

- La funzione restituisce il valore 0 se l'operazione si conclude normalmente, oppure `EOF` se qualcosa non funziona correttamente.

  In ogni caso, il file non è più accessibile attraverso il puntatore utilizzato precedentemente.

Lettura e scrittura

- L'accesso al contenuto dei file avviene a livello di byte.

- Le operazioni di lettura e scrittura dipendono da un indicatore riferito a una posizione, espressa in byte, all'interno del file stesso.

- L'indicatore viene inizialmente posizionato a seconda di come viene aperto il file e quindi viene spostato automaticamente a seconda delle operazioni di lettura e scrittura che si compiono.

Lettura e scrittura

- La lettura avviene normalmente attraverso la funzione `fread()` che legge una quantità di byte trattandoli come un array.

  Si tratta di definire la dimensione di ogni elemento, espressa in byte, e quindi la quantità di tali elementi.

  Il risultato della lettura viene inserito in un array, i cui elementi hanno la stessa dimensione.

  ```
  ...
     char ac[100];
     FILE *pf;
     int i;
     ...
     i = fread( ac, 1, 100, pf );
     ...
  ```

  In questo modo si intende leggere 100 elementi della dimensione di un solo byte, collocandoli nell'array `ac[]`, organizzato nello stesso modo.

- Il valore restituito dalla funzione rappresenta la quantità di elementi letti effettivamente.

  Se si verifica un qualsiasi tipo di errore che impedisce la lettura, la funzione si limita a restituire 0, o, in certe versioni del linguaggio C, un valore negativo.

Lettura e scrittura

- Quando il file viene aperto in lettura, l'indicatore interno viene posizionato all'inizio del file, e ogni operazione di lettura sposta in avanti il puntatore, in modo che la lettura successiva avvenga a partire dalla posizione seguente.

```
...
    char ac[100];
    FILE *pf;
    int i;
    ...
    pf = fopen( "mio_file", "r" );
    ...
    while(1) {  /* Ciclo senza fine */
        i = fread( ac, 1, 100, pf );
        if (i <= 0) {
            break;      /* Termina il ciclo */
        }
        ...
    }
    ...
```

In questo modo, come mostra l'esempio, viene letto tutto il file a colpi di 100 byte alla volta, tranne l'ultima in cui si ottiene solo quello che resta da leggere.

Lettura e scrittura

- La scrittura avviene normalmente attraverso la funzione `fwrite()` che scrive una quantità di byte trattandoli come un array, nello stesso modo già visto con la funzione `fread()`.

```
...
    char ac[100];
    FILE *pf;
    int i;
    ...
    i = fwrite( ac, 1, 100, pf );
    ...
```

  L'esempio, come nel caso di `fread()`, mostra la scrittura di 100 elementi di un solo byte, prelevati da un array.

- Il valore restituito dalla funzione è la quantità di elementi che sono stati scritti con successo.

  Se si verifica un qualsiasi tipo di errore che impedisce la scrittura, la funzione si limita a restituire 0, o, in certe versioni del linguaggio C, un valore negativo.

- Anche in scrittura è importante l'indicatore della posizione interna del file. Di solito, quando si crea un file o lo si estende, l'indicatore si trova sempre alla fine.

Lettura e scrittura

```
#include <stdio.h>
#define BLOCCO 1024
...
int main(void) {
    char ac[BLOCCO];
    FILE *pfIN;
    FILE *pfOUT;
    int i;
    ...
    pfIN = fopen( "fileIN", "r" );
    ...
    pfOUT = fopen( "fileOUT", "w" );
    ...
    while(1) {  /* Ciclo senza fine */
        i = fread( ac, 1, BLOCCO, pfIN );
        if (i <= 0) {
            break;      /* Termina il ciclo */
        }
        ...
        fwrite( ac, 1, i, pfOUT );
        ...
    }
    ...
    fclose( pfIN );
    fclose( pfOUT );
    ...
    return 0;
}
```

- L'esempio mostra lo scheletro di un programma che crea un file, copiando il contenuto di un altro (non viene effettuato alcun tipo di controllo degli errori).

fread()

```
size_t fread( void *<buffer>, size_t <dimensione>, size_t <quantita'>,
              FILE *<stream> )
```

- La funzione `fread()` permette di leggere una porzione del file identificato attraverso il puntatore riferito al flusso (`stream`), collocandola all'interno di un array.

- La lettura del file avviene a partire dalla posizione dell'indicatore interno al file, per una quantità stabilita di elementi di una data dimensione.

- La dimensione degli elementi viene espressa in byte.

- L'array ricevente, che in pratica si comporta da memoria tampone (*buffer*), deve essere in grado di accogliere la quantità di elementi letti, altrimenti i byte in più possono essere persi.

- La funzione restituisce il numero di elementi letto effettivamente.

- Il tipo di dato `size_t` serve a garantire la compatibilità con qualunque tipo intero, mentre il tipo `void` per l'array della memoria tampone, ne permette l'utilizzo di qualunque tipo.

fwrite()

```
size_t fwrite( const void *<buffer>, size_t <dimensione>,
               size_t <quantita'>, FILE *<stream> )
```

- La funzione `fwrite()` permette di scrivere il contenuto di una memoria tampone (*buffer*), contenuta in un array, in un file identificato attraverso il puntatore riferito al flusso (`stream`).

- La scrittura del file avviene a partire dalla posizione dell'indicatore interno al file, per una quantità stabilita di elementi di una data dimensione.

- La dimensione degli elementi viene espressa in byte.

- L'array in questione, che rappresenta la memoria tampone, deve contenere almeno la quantità di elementi di cui viene richiesta la scrittura.

- La funzione restituisce il numero di elementi scritti effettivamente.

Indicatore interno al file

- Lo spostamento diretto dell'indicatore interno della posizione di un file aperto è un'operazione necessaria quando il file è stato aperto sia in lettura che in scrittura, e da un tipo di operazione si vuole passare all'altro.

- Si utilizza la funzione `fseek()`, ed eventualmente anche `ftell()` per conoscere la posizione attuale.

  Posizione e spostamenti sono espressi in byte.

- `fseek()` esegue lo spostamento a partire dall'inizio del file, oppure dalla posizione attuale, oppure dalla posizione finale.

  Per questo utilizza un parametro che può avere tre valori, rispettivamente 0, 1 e 2, identificati anche da tre macro: `SEEK_SET`, `SEEK_CUR` e `SEEK_END`.

  ```
  i = fseek( pf, 10, 1);
  ```

- La funzione `fseek()` restituisce 0 se lo spostamento avviene con successo, altrimenti un valore negativo.

- L'esempio seguente mostra lo spostamento del puntatore, riferito al flusso `pf`, in avanti di 10 byte, a partire dalla posizione attuale.

```
i = fseek( pf, 10, 1);
```

- L'esempio seguente mostra lo scheletro di un programma, senza controlli sugli errori, che, dopo aver aperto un file in lettura e scrittura, lo legge a blocchi di dimensioni uguali, modifica questi blocchi e li riscrive nel file.

Indicatore interno al file: esempio

```
#include <stdio.h>

/* ------------------------------------------------------------- */
/* Dimensione del record logico                                  */
/* ------------------------------------------------------------- */
#define RECORD 10

int main(void) {
    char ac[RECORD];
    FILE *pf;
    int iQta;
    int iPosizione1;
    int iPosizione2;

    pf = fopen( "mio_file", "r+" );     /* lettura+scrittura         */

    while(1) {                          /* Ciclo senza fine          */

        /* Salva la posizione del puntatore interno al file          */
        /* prima di eseguire la lettura.                             */
        iPosizione1 = ftell( pf );
        iQta = fread( ac, 1, RECORD, pf );

        if (iQta == 0) {
            break;                      /* Termina il ciclo          */
        }

        /* Salva la posizione del puntatore interno al file          */
        /* Dopo la lettura.                                          */
        iPosizione2 = ftell( pf );

        /* Sposta il puntatore alla posizione precedente alla        */
        /* lettura.                                                  */
        fseek( pf, iPosizione1, SEEK_SET);
```

Indicatore interno al file: esempio

```
        /* Esegue qualche modifica nei dati, per esempio mette un    */
        /* punto esclamativo all'inizio.                             */
        ac[0] = '!';

        /* Riscrive il record modificato.                            */
        fwrite( ac, 1, iQta, pf );

        /* Riporta il puntatore interno al file alla posizione       */
        /* corretta per eseguire la prossima lettura.                */
        fseek( pf, iPosizione2, SEEK_SET);
    }

    fclose( pf );

    return 0;
}
```

fseek()

```
int fseek( FILE *<stream>, long <spostamento>,
           int <punto-di-partenza> )
```

- La funzione `fseek()` permette di spostare la posizione dell'indicatore interno al file a cui fa riferimento il flusso (`stream`) fornito come primo parametro.

- La posizione da raggiungere si riferisce al punto di partenza, rappresentato attraverso tre macro:
  - `SEEK_SET` rappresenta l'inizio del file;
  - `SEEK_CUR` rappresenta la posizione attuale;
  - `SEEK_END` rappresenta la fine del file.

- Il valore dello spostamento, indicato come secondo parametro, rappresenta una quantità di byte, e può essere anche negativo, a indicare un arretramento dal punto di partenza.

- Il valore restituito da `fseek()` è 0 se l'operazione viene completata con successo, altrimenti viene restituito un valore negativo (-1).

ftell()

```
long ftell( FILE *<stream> )
```

- La funzione `ftell()` permette di conoscere la posizione dell'indicatore interno al file a cui fa riferimento il flusso (`stream`) fornito come parametro.

- La posizione è assoluta, ovvero riferita all'inizio del file.

- Il valore restituito in caso di successo è positivo, a indicare appunto la posizione dell'indicatore. Se si verifica un errore viene restituito un valore negativo (-1).

File di testo

- I file di testo possono essere gestiti in modo più semplice attraverso due funzioni: `fgets()` e `fputs()`.

- Queste permettono rispettivamente di leggere e scrivere un file una riga alla volta, intendendo come riga una porzione di testo che termina con il codice di interruzione di riga.

- La funzione `fgets()` permette di leggere una riga di testo di una data dimensione massima.

  ```
  fgets( acz, 100, pf );
  ```

- Nello stesso modo funziona `fputs()`, che però richiede solo la stringa e il puntatore del file da scrivere.

- Dal momento che una stringa contiene già l'informazione della sua lunghezza perché possiede un carattere di conclusione, non è prevista l'indicazione della quantità di elementi da scrivere.

  ```
  fputs( acz, pf );
  ```

- In questo caso, viene letta una riga di testo di una dimensione massima di 99 caratteri, dal file rappresentato dal puntatore `pf`. Questa riga viene posta all'interno dell'array `acz[]`, con l'aggiunta di un carattere `\0` finale.

- Questo fatto dovrebbe spiegare il motivo per il quale il secondo parametro corrisponda a 100, mentre la dimensione massima della riga letta sia di 99 caratteri. In pratica, l'array di destinazione è sempre una stringa, terminata correttamente.

fgets()

```
char *fgets( char *<stringa>, int <dimensione>,
             FILE *<stream> )
```

- La funzione `fgets()` permette di leggere una riga di testo da un file passato come terzo parametro.

  `fgets()` richiede come secondo parametro l'indicazione della dimensione massima della riga.

  Tale dimensione sarà di un carattere in meno della dimensione dell'array di caratteri, passato come primo parametro, in cui la riga letta sarà memorizzata.

- La lettura del file avviene fino al raggiungimento della dimensione dell'array (meno uno), oppure fino al raggiungimento del codice di interruzione di riga, che viene regolarmente incluso nella riga letta, oppure anche fino alla conclusione del file.

- Se l'operazione di lettura riesce, `fgets()` restituisce l'array di caratteri di destinazione, altrimenti restituisce il puntatore nullo, `NULL`.

fputs()

```
int fputs( const char *<stringa>, FILE *<stream> )
```

- La funzione `fputs()` permette di scrivere una stringa in un file di testo.

- La stringa viene scritta senza il codice di terminazione finale, `\0`.

- Il valore restituito è un valore positivo in caso si successo, altrimenti è `EOF`.

I/O standard

- Ci sono tre flussi che risultano aperti automaticamente:
  - *standard input*, corrispondente normalmente alla tastiera;
  - *standard output*, corrispondente normalmente allo schermo del terminale;
  - *standard error*, anch'esso corrispondente normalmente allo schermo del terminale.
- A questi flussi si accede attraverso funzioni apposite (per es., `printf`, `scanf`, `getchar` e `putchar`, vedi pag. 325-332), ma si potrebbe accedere anche attraverso le normali funzioni di accesso ai file, utilizzando per identificare i flussi standard i nomi: `stdin`, `stdout` e `stderr`.

Istruzioni del preprocessore

- Il preprocessore è quella parte del compilatore che si occupa di pre-elaborare un sorgente prima della compilazione vera e propria.

- In pratica, permette di generare un nuovo sorgente prima che questo venga compilato effettivamente.

- Per esempio, il preprocessore sostituisce le macro (costanti simboliche) con il relativo valore ed inserisce una copia del contenuto dei file di intestazione al posto della relativa riga di inclusione.

- È possibile vedere il sorgente generato dal preprocessore. Con l'opzione `-E` del compilatore

  ```
  cc -E file.c
  ```

  viene effetuata la fase di elaborazione del preprocessore ma non la compilazione e il risultato viene mostrato sullo schermo.

- Il linguaggio C non può fare a meno della presenza di un preprocessore.

- Accenneremo solo alle direttive più importanti.

- Esistono altre direttive quali `#error`, `#pragma`, ...

Istruzioni del preprocessore

- L'utilità della presenza di un preprocessore, tra le altre cose, sta nella possibilità di:
  - definire gruppi di istruzioni alternativi a seconda di circostanze determinate (per esempio, la fase di debugging);
  - definire costanti simboliche;
  - includere il contenuto di altri file.

- Le direttive del preprocessore sono state regolate anch'esse con lo standard ANSI e costituiscono un linguaggio a sé stante. In generale:
  - le direttive iniziano con il simbolo `#`, preferibilmente nella prima colonna (ma può essere preceduto da spazi);
  - le direttive non utilizzano alcun simbolo di conclusione (non si usa il punto e virgola);
  - una riga non può contenere più di una direttiva.

#include

```
#include <<file>>
#include "<file>"
```

- La direttiva `#include` permette di includere un file. Generalmente si tratta di un cosiddetto file di *intestazione*, contenente una serie di definizioni necessarie al file sorgente in cui vengono incorporate.

- Come si vede dalla sintassi, il file può essere indicato delimitandolo con le parentesi angolari, oppure con gli apici doppi.

  ```
  #include <stdio.h>
  #include "stdio.h"
  ```

- Nel primo caso si fa riferimento a un file che dovrebbe trovarsi in una posizione stabilita dalla configurazione del compilatore (nel caso del GNU C in GNU/Linux, dovrebbe trattarsi della directory `/usr/include/`).

- Nel secondo caso si fa riferimento ad una posizione precisa definita dall'utente, che richiede l'indicazione di un percorso se non si tratta della stessa posizione in cui si trova il sorgente in questione.

#include

- Quando si indica un file da includere, delimitandolo con gli apici doppi e senza indicare alcun percorso, se non si trova il file nella directory corrente, il file viene cercato nella directory predefinita, come se fosse stato indicato tra le parentesi angolari.

- Un file incorporato attraverso la direttiva `#include`, può a sua volta includerne altri, e questa possibilità va considerata per evitare di includere più volte lo stesso file.

#define

```
#define <macro> [<sequenza-di-caratteri>]
```

- La direttiva `#define` permette di definire dei nomi, in questo caso si parla di *macro* (definite anche *costanti simboliche* o *costanti manifeste*), che, quando utilizzati nel sorgente, vengono sostituiti automaticamente con la sequenza che appare dopo la definizione.

  ```
  #define SALUTO ciao come stai
  ```

- ```
  #define SALUTO "ciao come stai"
  ```

  è diverso dal caso precedente, perché ci sono in più gli apici doppi. Questa volta, la macro `SALUTO` potrebbe essere utilizzata in un'istruzione come quella seguente,

  ```
  printf( SALUTO );
  ```

  mentre non sarebbe stato possibile quando la sostituzione era stata definita senza apici.

- Per convenzione, i nomi utilizzati per le macro sono espressi solo con lettere maiuscole.
  ```

#define

- Vantaggi dell'uso di `#define`: migliore leggibibilità dei programmi, portabilità e facilità di modificare i valori delle costanti simboliche.

- Le macro sono tipicamente usate per definire le dimensioni di qualcosa, per esempio gli array, o per fissare la corrispondenza reale con valori determinati che dipendono dalla piattaforma (per esempio, `EOF`).

- `#define` può continuare sulla riga successiva usando \ subito prima del carattere di interruzione di riga.

- Normalmente le righe `#define` sono inserite all'inizio dei file, ma possono apparire in qualunque punto ed hanno effetto sulle righe che seguono.

#define

- La direttiva può essere utilizzata in modo più complesso, facendo anche riferimento ad altre macro già definite.

```
#define UNO 1
#define DUE UNO+UNO
#define TRE DUE+UNO
```

  In presenza di una situazione come questa, utilizzando la macro `TRE`, si ottiene prima la sostituzione con `DUE+UNO`, quindi con `UNO+UNO+1`, e infine con `1+1+1` (dopo, tocca al compilatore).

- È sensato anche dichiarare una macro senza assegnargli alcun valore. Ciò può servire per le direttive `#ifdef` e `#ifndef`.

#define con argomento

```
#define <macro>(<argomento>) <sequenza-di-caratteri>
```

- La direttiva `#define` può essere usata in modo simile a una funzione, per definire delle sostituzioni che includono in qualche modo un argomento.
- ```
  #define DOPPIO(a)       (a)+(a)
  ...
  ...
  i = DOPPIO(i);
  ...
  ```

  l'istruzione `i=DOPPIO(i)` si traduce in `i=(i)+(i)`.
- Si osservi il fatto che, nella definizione, la stringa di sostituzione è stata composta utilizzando le parentesi. Questo permette di evitare problemi successivamente, nelle precedenze di valutazione delle espressioni.
- Le macro con parametri vengono spesso usate per sostituire chiamate di funzione con codice in linea che risulta più efficente.
  ```

#if, #else, #elif e #endif

```
#if <espressione>
        <espressione>
#endif
```

- Le direttive `#if`, `#else`, `#elif` e `#endif`, permettono di delimitare una porzione di codice che debba essere utilizzato o ignorato in relazione a una certa espressione che può essere calcolata facendo uso soltanto delle definizioni precedenti.

```
#define DIM_MAX 1000
...
...
int main(void) {
...
#if DIM_MAX>100
    printf("Dimensione enorme.");
    ...
#else
    printf("Dimensione normale.");
    ...
#endif
...
}
```

- Un modo efficace per evitare la compilazione di parti di codice durante la fase di debugging

```
#if 0
...
#endif
```

- L'esempio mostra in che modo si possa definire questa espressione, confrontando la macro `DIM_MAX` con il valore 100. Essendo stata dichiarata per tradursi in 1000, il confronto è equivalente a $1000 > 100$ che risulta vero, pertanto il compilatore include solo le istruzioni relative.

- Una alternativa a

  ```
  #if 0
  ...
  #endif
  ```

  sarebbe quella di usare i commenti, ma non funzionerebbe in generale perché un commento, all'interno di altri commenti, causerebbe collisioni.

  ```
  /*
  ... /* questo fa parte del commento */
  questo non fa piu' parte del commento
  */
  ```

#if, #else, #elif e #endif

- L'istruzione `#elif` serve per costruire una catena di alternative *else-if*.

- Gli operatori di confronto che si possono utilizzare per le espressioni logiche sono i soliti.

- ```
  #define NAZIONE ita
  ...
  ...
  int main(void) {
  #if NAZIONE==ita
      char valuta[] = "EURO";
      ...
  #elif NAZIONE==usa
      char valuta[] = "USD";
      ...
  #endif
  ...
  }
  ```

- Le direttive condizionali possono essere annidate e possono contenere anche altri tipi di direttiva del preprocessore.
  ```

#ifdef e #ifndef

- Le direttive `#ifdef` e `#ifndef` servono per definire l'inclusione o l'esclusione di codice in base all'esistenza o meno di una macro.

- ```
  #define DEBUG
  ...
  int main(void) {
  ...
  #ifdef DEBUG
      printf( "Punto di controllo n. 1\n");
      ...
  #endif
  ...
  }
  ```

  L'esempio mostra il caso in cui sia dichiarata una macro `DEBUG` (che non si traduce in alcunché) e in base alla sua esistenza viene incluso il codice che mostra un messaggio particolare.
  ```

#ifdef e #ifndef

- ```
  #define OK
  ...
  int main(void) {
  #ifndef OK
      printf( "Punto di controllo n. 1\n");
      ...
  #endif
  ...
  }
  ```

  L'esempio mostrato è analogo al precedente, con la differenza che la direttiva `#ifndef` permette la compilazione delle istruzioni che controlla solo se la macro indicata non è stata dichiarata.

- L'uso delle direttive `#else` e `#endif` avviene nel modo già visto per la direttiva `#if`.
  ```

#undef

```
#undef <macro>
```

- La direttiva `#undef` permette di eliminare una macro a un certo punto del sorgente.

```
#define NAZIONE ita
...
/* In questa posizione, NAZIONE risulta definita */
...
#undef NAZIONE
...
/* In questa posizione, NAZIONE non e' definita */
...
```

Macro predefinite

- Il compilatore ANSI C prevede 5 macro predefinite.

- Il loro scopo è quello di annotare (a fini diagnostici) informazioni legate alla compilazione nel file eseguibile finale.

  `__FILE__` contiene una stringa che rappresenta il nome del file sorgente;

  `__LINE__` contine un intero che rappresenta il numero di riga corrente;

  `__DATE__` contiene una stringa *mese/giorno/anno* che rappresenta la data corrente;

  `__TIME__` contiene una stringa *ore:minuti:secondi* che rappresenta l'ora corrente;

  `__STDC__` contiene un intero diverso da 0 se il compilatore C che si utilizza segue lo *standard* ANSI C.

Un programma con un solo modulo

```
/* REVERSE.C */

#include <stdio.h>

/* Function Prototype */
void reverse ( char *, char * );

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

int main (void) {
  char str [100];  /* Buffer per contenere la stringa invertita */

  reverse ("cat", str); /* Inverte la stringa "cat" */
  printf ("reverse (\"cat\") = %s\n", str); /* Mostra il risultato */
  reverse ("noon", str); /* Inverte la stringa "noon" */
  printf ("reverse (\"noon\") = %s\n", str); /* Mostra il risultato */
  return 0;
}

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

void reverse ( char *before, char *after ) {
                   /* before: puntatore alla stringa originaria */
                   /* after: puntatore alla stringa invertita */
  int i, j, len;

  len = strlen (before);
  for (j=len-1, i=0; j>=0; j--, i++) /* Ciclo */
     after[i] = before[j];

  after[len] = '\0'; /* \0 termina la stringa invertita */
}
```

Programmi in un singolo modulo

- Principali *incovenienti*:
  - se si fanno anche solo delle piccole modifiche, bisogna sempre ricompilare tutto il programma;
  - le funzioni definite nel programma (per esempio, la funzione `reverse()`) non possono essere utilizzate facilmente in altri programmi.

- Il primo inconveniente porta inevitabilmente a tempi di compilazione piuttosto elevati.

- Il secondo inconveniente non può efficacemente essere risolto semplicemente ricopiando le funzioni perché
  - se ad un certo punto si trova un codice più efficente per la funzione, si dovrebbe rimpiazzare ogni copia della vecchia funzione con la nuova;
  - ciascuna copia della funzione occupa spazio disco.

Suddivisione in più moduli

- I programmi C sono articolati in più file sorgenti separati che vengono compilati in modo indipendentemente e infine collegati in un unico eseguibile.

- Nella produzione di un eseguibile, vengono ricompilati solo i file sorgente che hanno subito modifiche successivamente alla generazione del file oggetto relativo.

- Per permettere la *condivisione del codice* bisogna
  - rimuovere il codice di `reverse()` dal programma "`reverse`",
  - compilarlo separatamente e, quindi,
  - fare il *linking* (collegamento) del modulo oggetto risultante in qualsiasi programma che lo voglia utilizzare.

File di intestazione (*header*)

- Quando un programma è suddiviso in più moduli sorgente:
  - se si utilizzano delle macro del preprocessore, queste dovranno essere dichiarate in tutti i sorgenti che ne fanno uso;
  - se all'interno di un sorgente si fa riferimento ad una funzione dichiarata altrove, il sorgente deve contenere una dichiarazione del prototipo della funzione (le variabili globali vanno trattate in modo analogo).

- Per facilitarne l'inclusione in tutti i sorgenti che lo necessitano, conviene predisporre dei file di intestazione (*header*) con le dichiarazioni delle macro del preprocessore e dei prototipi delle funzioni.

Funzioni riusabili

- Per far sì che una funzione sia *riusabile* bisogna:
  - creare un modulo contenente il *codice sorgente* della funzione;
  - creare un file *header* (suffisso `.h`) contenente il *prototipo* della funzione;
  - compilare, tramite l'opzione `-c` del compilatore `cc`, il modulo con il sorgente in un modulo *oggetto*.
- Il modulo oggeto contiene il *codice macchina* e la *tabella dei simboli*.
- La tabella dei simboli permette di ricombinare (tramite il compilatore `cc` o il *loader* `ld`) il codice macchina con altri moduli oggetto per ottenere file *eseguibili*.

Un programma in tre moduli

- Il file `reverse.h`.

```
/* REVERSE.H */

void reverse ( char *, char * );  /* Dichiara ma non definisce
                                     la funzione */
```

- Il file `reverse.c`.

```
/* REVERSE.C */

#include <stdio.h>
#include "reverse.h"

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

void reverse ( char *before, char *after ) {
                    /* before: puntatore alla stringa originaria */
                    /* after: puntatore alla stringa invertita */
  int i, j, len;

  len = strlen (before);
  for (j=len-1, i=0; j>=0; j--, i++) /* Ciclo */
     after[i] = before[j];

  after[len] = '\0'; /* \0 termina la stringa invertita */
}
```

Un programma in tre moduli

Il file `main1.c`.

```
/* MAIN1.C */

#include <stdio.h>
#include "reverse.h" /* Contiene il prototipo di reverse() */

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

int main (void) {
  char str [100];  /* Buffer per contenere la stringa invertita */

  reverse ("cat", str); /* Inverte la stringa "cat" */
  printf ("reverse (\"cat\") = %s\n", str); /* Mostra il risultato */
  reverse ("noon", str); /* Inverte la stringa "noon" */
  printf ("reverse (\"noon\") = %s\n", str); /* Mostra il risultato */
  return 0;
}
```

Compilatore standard cc

- Il compilatore C di GNU/Linux è `gcc` (`cc` GNU), tuttavia, le sue caratteristiche sono tali da renderlo conforme al compilatore standard POSIX.

  Per mantenere la convenzione, è presente il collegamento `cc` che si riferisce al vero compilatore `gcc`.

- La sintassi esprime in maniera piuttosto vaga l'ordine dei vari argomenti della riga di comando, e in effetti non c'è una particolare rigidità.

  ```
  cc [ <opzioni> | <file> ]...
  ```

- Estensioni tipiche dei nomi dei file

  .c Sorgente C.

  .o File oggetto.

  .a Libreria.

Compilatore standard cc: alcune opzioni

-c

Genera come risultato i file di oggetto, senza avviare il link dell'eseguibile finale.

-g

Aggiunge delle informazioni diagnostiche utili per il debugging attraverso strumenti appositi come `gdb`.

-o *file*

Definisce il nome del file che deve essere generato a seguito della compilazione (indipendentemente dal fatto che si tratti di un file eseguibile o di un file oggetto o altro ancora).

-l*libreria*

Compila utilizzando la libreria indicata, tenendo presente che, per questo, verrà cercato un file che inizia per `lib`, continua con il nome indicato, e termina con `.a` oppure `.so`.

-v

Genera informazioni dettagliate sulla compilazione.

Compilatore standard cc: librerie

- La libreria standard del C (`libc.a`) è consultata automaticamente.

- Nelle versioni non troppo recenti del compilatore, è necessario menzionare esplicitamente nel comando di compilazione qualsiasi altra libreria necessaria per produrre l'eseguibile. Per esempio,

  `-lm:` collega la libreria contenente le funzioni matematiche (`sin()`, `cos()`, `tan()`, ... che operano su `double`);

  `-lX11:` collega la libreria contenente le funzioni grafiche di X.

- Se un programma utilizza delle funzioni di libreria, per esempio appartenenti alla libreria matematica, per generare l'eseguibile non basta utilizzare la direttiva `#include <math.h>` (comunque necessaria al compilatore per avere informazioni sui prototipi), ma bisogna anche collegare (con `-lm`) i file oggetto della libreria che effettivamente contengono le routine matematiche.

Compilazione separata e linking

- L'opzione `-c` di `cc` permette di compilare ogni file sorgente separatamente e di creare un modulo oggetto, con un suffisso `.o`, per ogni sorgente.

```
/home/rossi$ cc -c reverse.c
/home/rossi$ cc -c main1.c
/home/rossi$ ls -l
-rw-r--r--  1 rossi users  517 Apr 27 11:07 main1.c
-rw-r--r--  1 rossi users 1056 Apr 27 11:16 main1.o
-rw-r--r--  1 rossi users  503 Apr 27 11:14 reverse.c
-rw-r--r--  1 rossi users   99 Apr 27 11:13 reverse.h
-rw-r--r--  1 rossi users  884 Apr 27 11:14 reverse.o
/home/rossi$
```

  Alternativamente, si può usare un unico comando.

```
/home/rossi$ cc -c reverse.c main1.c
/home/rossi$
```

- Per fare il *linking* dei file oggetto (risolvendo i riferimenti incrociati) e creare un unico file eseguibile chiamato `main1` si può usare

```
/home/rossi$ cc reverse.o main1.o -o main1
/home/rossi$ ls -l main1
-rwxr-xr-x  1 rossi users 4325 Apr 27 11:17 main1
/home/rossi$ main1
reverse ("cat") = tac
reverse ("noon") = noon
/home/rossi$
```

- Se si volesse fare una modifica su uno solo dei file sorgenti, basterebbe rigenerare il file oggetto relativo e riunire il tutto con il comando `cc -o`.

- Per fare un pò di esperimenti può essere utile il comando `touch`. `touch` *nomefile* crea il file (con contenuto nullo) se non esiste già, altrimenti aggiorna la data di ultima modifica a quella corrente.

Il loader: ld

`ld -n` {`-L`*path*}* {*objMod*}* {*library*}* {`-lx`}* [`-o` *outputFile*]

- `ld` collega insieme i file oggetto *objMod* e i moduli di libreria per produrre un unico file eseguibile.
- Il nome dell'eseguibile per default è `a.out`, ma un nome diverso può essere specificato con l'opzione `-o`.
- L'opzione `-n` serve per creare un eseguibile stand-alone (anzicché un modulo con dynamic-linking).
- Quando viene specificata un'opzione `-lx`, `ld` cerca nelle directory standard `/lib`, `/usr/lib` e `/usr/local/lib` un file di libreria chiamato `libx.a`. Se si vuole inserire una directory `path` in questo insieme di directory allora bisogna usare l'opzione `-L`*path*.
- Se si decide di fare il linking di un programma invocando direttamente `ld`, è importante specificare come primo modulo il modulo oggetto a runtime del C, `crt0.o`, e come modulo di libreria la libreria standard del C, `libc.a`.

- Quando si usa `cc` per fare il "linking" di diversi moduli oggetto, implicitamente viene invocato il *loader* (meglio noto come *linker*), `ld`, che fa effettivamente il lavoro.

- `ld` può anche essere invocato direttamente.

- *Dynamic-linking* significa per esempio che l'eseguibile non è self-contained ma a run-time ha bisogno di caricare delle librerie *condivise*, dette anche *dinamiche*. Il vantaggio di non includere staticamente le librerie usate dal codice sta in una maggiore compattezza, lo svantaggio principale è che bisogna prestare attenzione alla versione appropriata della libreria che il codice riferisce.

- Il comando `ld` varia moltissimo da versione a versione del sistema UNIX, anche più di altri comandi ed utility UNIX. È quindi importante consultare la documentazione relativa alla versione del sistema che si sta usando.

Riusare la funzione reverse()

Vediamo ora un programma che fa anch'esso uso della funzione `reverse()`.

- Il file `palindrome.h`.

```
/* PALINDROME.H */

int palindrome ( char * ); /* Dichiara ma non definisce la funzione */
```

- Il file `palindrome.c`.

```
/* PALINDROME.C */

#include <stdio.h>
#include "reverse.h"
#include "palindrome.h"

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

int palindrome ( char *str ) {
  char reversedStr [100];

  reverse (str, reversedStr);   /* Inverte la stringa originaria */
  return (strcmp (str, reversedStr) == 0);
                                /* Confronta le due stringhe */
}
```

main.c

- Il file `main2.c`.

```
/* MAIN2.C */

#include <stdio.h>
#include "palindrome.h" /* Contiene il prototipo di palindrome() */

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

int main (void) {
  printf ("palidrome (\"cat\") = %d\n", palindrome ("cat"));
  printf ("palidrome (\"noon\") = %d\n", palindrome ("noon"));
  return 0;
}
```

- Compilazione e linking dei moduli.

```
/home/rossi$ cc -c palindrome.c
/home/rossi$ cc -c main2.c
/home/rossi$ cc reverse.o palindrome.o main2.o -o main2
/home/rossi$ ls -l reverse.o palindrome.o main2.o main2
-rwxr-xr-x  1 rossi users  4596 Apr 27 11:26 main2
-rw-r--r--  1 rossi users  1052 Apr 27 11:25 main2.o
-rw-r--r--  1 rossi users   892 Apr 27 11:25 palindrome.o
-rw-r--r--  1 rossi users   884 Apr 27 11:14 reverse.o
/home/rossi$ main2
palindrome ("cat") = 0
palindrome ("noon") = 1
/home/rossi$
```

Gestione dei programmi

- Chi ci assicura che i moduli oggetto e gli eseguibili sono aggiornati?

  `make`: sistema per tener traccia delle dipendenze dei file in UNIX.

- Chi organizza e memorizza i moduli oggetto?

  `ar`: sistema di archiviazione dei file in UNIX.

- Chi tiene traccia delle versioni dei file sorgenti e degli header?

  `sccs`: sistema per la gestione del codice sorgente in UNIX.

Make

- La ricompilazione di un programma suddiviso in più moduli può essere un'operazione alquanto laboriosa: si potrebbe predisporre uno script di shell per eseguire sequenzialmente tutte le operazioni necessarie ma la tradizione impone di usare Make.

  `make [-f` *makefile*`]`

- `make` è una utility che aggiorna un file sulla base di *regole di dipendenza* contenute in un file con un formato speciale detto *makefile*.

- Per default, `make` si aspetta che le regole di dipendenza si trovino nel file `makefile` o, preferibilmente, `Makefile`. Quindi, se si vuole specificare un nome diverso per il file contenente le regole, si deve usare l'opzione `-f` seguita dall'appropriato *makefile* (il cui nome è consigliabile abbia suffisso `.make`).

- Il *makefile* contiene la lista di tutte le interdipendenze che esistono tra i file che sono usati per creare l'eseguibile. Al suo interno, `#` denota l'inizio di un commento.

Make

- Make può anche essere usato da solo, senza makefile, per compilare un singolo sorgente, e in questo caso, tenta di determinare l'operazione più adatta da compiere in base all'estensione del sorgente stesso.

  Per esempio, se esiste il file `prova.c` nella directory corrente, il comando

  ```
  $ make prova
  ```

  fa sì che `make` avvii in pratica il comando seguente:

  ```
  $ cc -o prova prova.c
  ```

  Se invece esistesse un makefile, lo stesso comando, `make prova`, avrebbe un significato diverso, corrispondendo alla ricerca di un *obiettivo* con il nome `prova` all'interno del makefile stesso.

- Un makefile contiene la definizione di *macro*, simili alle variabili di ambiente di uno script di shell, e di *obiettivi* che rappresentano le varie operazioni da compiere.

Regole di dipendenza

targetList:dependencyList
commandList

- *targetList* è una lista di file obiettivo, *dependencyList* è una lista di file da cui i file obiettivo dipendono, e *commandList* è una lista di zero o più comandi, separati da caratteri di interruzione di riga, che ricostruiscono i file obiettivo a partire dai file da cui essi dipendono.

- Ogni linea in *commandList* deve cominciare con un carattere di tabulazione orizzontale.

- Il comando indicato in una regola, può proseguire su più righe successive, basta concludere la riga, prima del codice di interruzione di riga, con una barra obliqua inversa \.

- Il comando di una regola può iniziare con un prefisso particolare (inserito dopo il *tab*):
  - `-` fa in modo che gli errori vengano ignorati;
  - `+` fa in modo che il comando venga eseguito sempre;
  - `@` fa in modo che il testo del comando non venga mostrato.

Regole di dipendenza

- In pratica, non si può eseguire il comando se prima non esistono i file indicati nelle dipendenze.

- Un sorgente dipende da ogni file che è direttamente o indirettamente incluso in esso con `#include`.

```
/home/rossi$ more main1.make
main1: main1.o reverse.o
        cc main1.o reverse.o -o main1
main1.o: main1.c reverse.h
        cc -c main1.c
reverse.o: reverse.c reverse.h
        cc -c reverse.c
/home/rossi$
```

Regole di dipendenza

- L'ordine con cui le regole appaiono è importante.
- `make` crea un *albero delle dipendenze* cominciando ad esaminare la prima regola.

  Ogni file nella *targetList* della prima regola è una radice di un albero delle dipendenze ed ogni file nella *dependencyList* è aggiunto come una foglia di ciascuna radice.

  Quindi viene visitata ciascuna regola associata con ciascun file nella *dependencyList* e vengono eseguite le stesse azioni.
- L'albero viene visitato dalle foglie alla radice per vedere se la data dell'ultima modifica di un nodo figlio è più recente di quella del suo immediato genitore.

  Ogni qualvolta ciò accade, vengono eseguiti i comandi nella *commandList* della regola associata al nodo genitore (cioè quella regola in cui il nodo genitore è in *targetList*).
- Se un file non esiste, la regola associata al file viene eseguita indipendentemente dai tempi di modifica dei sui nodi figli.

Regole di dipendenza

- Questo è ciò che accade nel primo esempio.

```
/home/rossi$ make -f main1.make
cc -c main1.c
cc -c reverse.c
cc main1.o reverse.o -o main1
/home/rossi$
```

- Questo è ciò che accade nel secondo esempio.

```
/home/rossi$ more main2.make
main2: main1.o reverse.o palindrome.o
      cc main1.o reverse.o palindrome.o -o main2
main2.o: main2.c palindrome.h
      cc -c main2.c
reverse.o: reverse.c reverse.h
      cc -c reverse.c
palindrome.o: palindrome.c palindrome.h reverse.h
      cc -c palindrome.c
/home/rossi$ make -f main2.make
cc -c main2.c
cc -c palindrome.c
cc main2.o reverse.o palidrome.o -o main2
/home/rossi$
```

`reverse.c` non è stata ricompilata perché il relativo modulo oggetto era stato appena generato dal `make` precedente.

Regole predefinite

- `make` contiene alcune regole predefinite. Ad esempio

```
.c.o:
	/bin/cc -c -o $<
```

  che gli dice come creare un modulo oggetto da un file sorgente.

- Il secondo esempio si può così semplificare.

```
main2: main2.o reverse.o palindrome.o
	cc main2.o reverse.o palindrome.o -o main2
main2.o: main2.c palindrome.h
reverse.o: reverse.c reverse.h
palindrome.o: palindrome.c palindrome.h reverse.h
```

- `make` sa che il nome di un modulo oggetto e quello del corrispondente file sorgente sono solitamente collegati.

- Il secondo esempio si può ulteriormente semplificare.

```
main2: main2.o reverse.o palindrome.o
	cc main2.o reverse.o palindrome.o -o main2
main2.o: palindrome.h
reverse.o: reverse.h
palindrome.o: palindrome.h reverse.h
```

Macro

- La definizione di una macro avviene indicando l'assegnamento di una stringa a un nome che da quel momento la rappresenterà.

  ```
  <nome> = <stringa>
  ```

  Per esempio,

  ```
  prefix=/usr/local
  ```

  definisce la macro `prefix` che da quel punto in poi equivale a `/usr/local`.

- La sostituzione di una macro si indica attraverso due modi possibili:

  ```
  $(<nome>)
  ${<nome>}
  ```

  come nell'esempio seguente, dove la macro `exec_prefix` viene generata a partire dal contenuto di `prefix`.

  ```
  prefix=/usr/local
  exec_prefix=$(prefix)
  ```

Makefile tipico

- Il makefile tipico permette di automatizzare tutte le fasi legate alla ricompilazione di un programma e alla sua installazione.
- Si distinguono alcuni obiettivi comuni
  - `all`: utile per definire l'azione da compiere quando non si indica alcun obiettivo;
  - `clean`: per eliminare i file oggetto e i binari già compilati;
  - `install`: per installare il programma eseguibile dopo la compilazione.
- Le fasi tipiche di un'installazione di un programma distribuito in forma sorgente sono appunto:

  ```
  $ make
  ```

  che richiama automaticamente l'obiettivo `all` del makefile, coincidente con i comandi necessari per la compilazione del programma, e

  ```
  $ make install
  ```

  che provvede a installare gli eseguibili compilati nella loro destinazione prevista.

Makefile tipico: esempio

Supponendo di avere realizzato un programma, denominato `mio_prog.c`, il cui eseguibile debba essere installato nella directory `/usr/local/bin/`, si potrebbe utilizzare il makefile

```
prefix=/usr/local
bindir=${prefix}/bin

all:
        cc -o mio_prog mio_prog.c

clean:
        rm -f core *.o mio_prog

install:
        cp mio_prog $(bindir)
```

Come si può osservare, sono state definire le macro `prefix` e `bindir` in modo da facilitare la modifica della destinazione del programma installato, senza intervenire sui comandi.

L'obiettivo `clean` elimina un eventuale file `core`, generato da un errore irreversibile durante l'esecuzione del programma, probabilmente mentre lo si prova tra una compilazione e l'altra, quindi elimina gli eventuali file oggetto e infine l'eseguibile generato dalla compilazione.

ar

`ar` *key archiveName* { *fileName* }*

- `ar` può essere utilizzata per organizzare e raggruppare i moduli oggetto in una *libreria*.

- `ar` è una utility che permette di creare e gestire archivi.

- *archiveName* (dovrebbe terminare con il suffisso `.a`) è il nome del file di archivio che si desidera accedere.

- *key* può essere una delle lettere seguenti:
  - `d`: cancella { *fileName* }* da *archiveName*;
  - `q`: concatena { *fileName* }* alla fine di *archiveName*, anche se già presente;
  - `r`: aggiunge { *fileName* }* ad *archiveName* se non è già presente, altrimenti ne rimpiazza la versione corrente;
  - `t`: mostra sullo standard output l'indice di *archiveName*;
  - `x`: copia { *fileName* }* da *archiveName* nella directory corrente;
  - `v`: produce informazioni dettagliate.

- `cc` e `ld` possono accedere ai file in un archivio semplicemente fornendo loro il nome del file di archivio come argomento.

- `make` contiene delle regole predefinite per la gestione automatica degli archivi.

- Rivediamo ulteriormente il secondo esempio.

```
/home/rossi$ more main2.make
main2: main2.o string.a(reverse.o) string.a(palindrome.o)
      cc main2.o string.a -o main2
main2.o: palindrome.h
string.a(reverse.o): reverse.h
string.a(palindrome.o): palindrome.h reverse.h
/home/rossi$ make -f main2.make
cc    -c main2.c -o main2.o
cc    -c reverse.c -o reverse.o
ar rv string.a reverse.o
a - reverse.o
cc    -c palindrome.c -o palindrome.o
ar rv string.a palindrome.o
a - palindrome.o
cc main2.o string.a -o main2
rm reverse.o palindrome.o
/home/rossi$
```

- Si noti che le regole predefinite rimuovono automaticamente i moduli oggetto nella directory una volta che sono stati inseriti nell'archivio.

- In alcuni sistemi più vecchi è importante l'ordine con cui i file sono immagazzinati in un archivio.

- Per esempio, se un modulo oggetto `A` contiene una funzione che chiama una funzione in un modulo oggetto `B`, allora `B` deve precedere `A` nella sequenza dei link.

  Se `A` e `B` si trovano nella stessa libreria, allora `B` deve apparire prima di `A`.

- Per risolvere il problema si usa

  `ranlib` { *archive* }$^+$

- `ranlib` aggiunge un indice ad un archivio tramite l'inserimento nell'archivio di una nuova entry chiamata `__.SYMDEF`.

SCCS

- Il *source code control system* (o sccs) è un insieme di utility che UNIX mette a disposizione per la gestione di progetti di grandi dimensioni (per memorizzare, accedere e proteggere tutte le versioni dei file contenenti codice sorgente).

- `admin` permette di creare e gestire file in formato sccs. Un tale file è memorizzato in una maniera particolare e non può essere modificato o compilato in modo standard. Il file contiene informazioni riguardanti la data di creazione e l'utente che l'ha creato. Modifiche successive conterrano informazioni sui cambiamenti effettuati rispetto alla versione precedente.

- `get` permette di prelevare l'ultima versione (o versioni precedenti) di un file in formato sccs e di creare un file testo in formato standard che si può modificare e compilare. `get` non permette a nessun altro di lavorare sul file finché lo stesso non viene restituito.

SCCS

- `delta` permette di restituire al sistema la nuova versione di un file in formato sccs. Il comando ottimizza l'occupazione della memoria disco salvando soltanto le differenze tra la vecchia e la nuova versione.

- `sact` permette di vedere l'attività corrente che viene fatta su un particolare file sccs.

- Altre utility che fanno parte del sistema sono `help`, `prs`, `comb`, `what` e `unget`.

prof

`prof -ln [` *executableFile* `[` *profileFile* `] ]`

- `prof` (profiler standard di UNIX), a partire dalle informazioni contenute in *profileFile*, genera una tabella con il tempo utilizzato ed il numero delle ripetizioni di ogni funzione che appare nel file *executableFile*.
- Se *profileFile* è omesso, si usa il file `mon.out`.
- Se *executableFile* è omesso, si usa il file `a.out`.
- L'eseguibile deve essere stato compilato con l'opzione `-p` di `cc`, che istruisce il compilatore a generare un codice speciale che, quando il programma è eseguito, scrive il file `mon.out`.
- Una volta terminata l'esecuzione del programma, l'utility `prof` può essere usata per esaminare il file `mon.out` e mostrare l'informazione ivi contenuta.
- Per default, l'informazione sulla performance è mostrata in ordine discendente. L'opzione `-l` ordina l'informazione per nome, e l'opzione `-n` ordina l'informazione per tempo cumulativo.

lint

`lint` { *fileName* }*

`lint` scandisce i file sorgenti specificati, esamina le potenziali interazioni e mostra qualsiasi potenziale errore trovato.

dbx

`dbx` *executableFilename*

- `dbx` è il debugger standard di UNIX.

- Il file specificato come argomento è caricato nel debugger ed è mostrato un *invito* che indica all'utente che il debugger è in attesa di un comando da eseguire.

- `dbx` permette il debugging simbolico (a livello sorgente) di un programma e supporta varie facility, tra cui:
  - esecuzione passo-passo,
  - inserimento di punti di interruzione,
  - modifica del file da dentro il debugger,
  - accesso e modifica delle variabili,
  - ricerca di funzioni,
  - tracciamento dell'esecuzione del programma.

- Per preparare un programma per il debugging bisogna compilarlo con l'opzione `-g` di `cc`, che istruisce il compilatore ad inserire informazioni per il debugging nel file generato.

Comandi di dbx

- `help` fornisce informazioni sui comandi.

- `run` esegue un programma attraverso il debugger.

  `rerun` fa ripartire l'esecuzione di un programma (la cui esecuzione era stata arrestata).

- `trace` fornisce una traccia linea-per-linea dell'esecuzione del programma.

  `trace` *variabile* `in` *funzione* fornisce la traccia dei valori assunti da una specifica variabile in una funzione.

  `trace` *funzione* fornisce la traccia delle chiamate di una funzione.

- `edit` invoca l'editor indicato nella variabile di ambiente `EDITOR`. Il programma può quindi essere modificato (prima di riprendere il debugging va ricompilato).

- `stop in` *funzione* permette di sospendere l'esecuzione di un programma quando viene incontrata l'invocazione di una determinata funzione.

Comandi di dbx

- `step` esegue un singolo passo del programma.

- `print` *variabile* permette di esaminare il valore di una variabile.

- `whatis` mostra la dichiarazione di una variabile.

  `which` indica dove una variabile è dichiarata.

- `where` mostra un traccia completa dello stack.

  `whereis` indica dove si trova all'interno del programma una particolare funzione o variabile.

- `list` mostra 10 linee di una funzione.

- `/` e `?` permettono di cercare in avanti ed indietro nel testo.

- `quit` esce dal debugger.

strip

- Sia il profiler che il debugger richiedono che un programma sia compilato usando opzioni che aggiungono codice speciale al file eseguibile.

- Per rimuovere il codice extra contenuto da un file eseguibile dopo le fasi di profiling e debugging, si può usare

  `strip` { *fileName* }*

- `strip` riumove dai file argomento tutte le informazioni relative alle tabelle dei simboli, alla rilocazione, al debugging ed al profiling.