Il primo programma

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

Un programma C consiste di una o più funzioni.

L'esecuzione inizia sempre da una funzione `main` che può poi invocare altre funzioni.

Sopra, `main` viene dichiarata come una funzione senza parametri (quando ve ne sono, vengono racchiusi tra parentesi).
Il suo corpo è racchiuso tra `{e}`

Una funzione viene invocata o chiamata con il suo nome, seguito da una lista di argomenti tra parentesi.

Un esempio di chiamata di funzione è `printf` sopra:
`printf` è una funzione di biblioteca, che ha per argomento una stringa e la stampa sulla standard output.

Nel seguito, ogni programma che invoca `printf` deve iniziare con la direttiva:

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

Ma, per brevità, I successivi esempi non la mostreranno.

Stringhe

Una stringa è racchiusa tra `"` e `"`.

Essa può contenere sequenze di escape per rappresentare caratteri che non si possono associare direttamente a un tasto:

- `\t` per il tab
- `\b` per il backspace
- `\"` per scrivere `"`
- `\\` per scrivere `\`
- `\n` per il ritorno a capo (volutamente, `printf` di per sé non produce il ritorno a capo)

Se `\x` non è una sequenza di escape, `\x` produce x.

Variabili e tipi

Output di un programma che converte gradi Fahrenheit in Celsius:

```
   0 -17.8
  20  -6.7
 ...
 300 148.9
```

E il programma che lo ha prodotto

```
/* stampa Fahrenheit -> Celsius per gradi F 0,20,...,300 */
main()
{
    int lower, upper, step;
    float fahr, celsius;

    lower = 0;      /* temp. F min della tabella */
    upper = 300;    /* temp. F max della tabella */
    step = 20;      /* incremento nella tabella */

    fahr = lower;   /* fahr è la temp. F da convertire */
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%4.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

I commenti sono tra `/*` e `*/`

Le variabili si dichiarano prima di usarle, all'inizio di una funzione; la dichiarazione ha la forma: type v1, v2, ...

Alcuni tipi fondamentali:

Type Name	Storage Byte	Range of Values
char	1	-128 to 127
int	2	-32,768 to 32,767 (=2^{15}-1)
long	4	-2,147,483,648 to 2,147,483,647 (=2^{31}-1)
float	4	3.4E ± 38 (7 cifre significative)
double	8	1.7E ± 308 (15 cifre significative)

Istruzioni

La computazione eseguita da un programma è fatta di istruzioni.

Nell'esempio si comincia con delle assegnazioni alle variabili (per inizializzarle)

```
    lower = 0;      /* temp. F min della tabella */
    upper = 300;    /* temp. F max della tabella */
    step = 20;      /* incremento nella tabella */
    fahr = lower;   /* fahr è la temp. F da convertire */
```

Si noti che ogni istruzione ha un `;` come terminatore.

Ogni riga dell'uscita va calcolata allo stesso modo,
per valori di `fahr` da `lower` ad `upper`, di qui l'uso del `while`:

```
while (fahr <= upper) {
    celsius = (5.0/9.0) * (fahr-32.0);
    printf("%4.0 f%6.1f\n", fahr, celsius);
    fahr = fahr + step;
}
```

Per eseguire il `while`:

1. si valuta la condizione in parentesi che segue `while`

2. se la condizione è vera, si esegue il corpo del `while` (l'istruzione che segue la condizione), dopodiché si torna a 1

3. se la condizione è falsa, si continua da ciò che segue il `while`

Nell'esempio il corpo del `while` è un'istruzione composta (`{...}`).

NB: le istruzioni tra `{}` e il corpo di `while` e altre istruzioni, vanno indentate (=rientrate) di un tab o numero fisso di spazi.

Aritmetica e `printf`

```
        celsius = (5.0/9.0) * (fahr-32.0);
```

è l'istruzione che converte `celsius` in `fahr`.

`/` è la divisione tra `int` o `float`, quindi è sbagliato scrivere `5/9`

Il punto decimale indica che una costante è un `float`

Gli `int` mostrati sotto in grassetto sono automaticamente convertiti in `float` prima di valutare l'espressione:

```
    int lower, upper, step;
    float fahr, celsius;
       ...
    fahr = lower;   /* fahr è la temp. F da convertire */
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%4.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
```

Se `printf` ha argomenti che seguono la stringa, la stampa sostituendo la 1ª, 2ª ... specifica di conversione con il 2°, 3°, ... argomento, convertito secondo la specifica.

Una forma semplificata delle specifiche di conversione è `%`fw.pt dove:

f (flag) può mancare (giustificazione a destra) o essere – (a sin.)

w (width) richiede per la stampa un campo di almeno w posizioni

p (precision) richiede p cifre decimali per un `float`

t (type) può valere `f` (`float`), `d` (`int` decimale), `x` (`int` esadecimale), `c` (`char`), `s` (stringa)

f, w, p possono mancare; `%%` stampa `%`.

Costanti simboliche e ciclo `for`

L'uso di variabili per quantità fisse come `step`, `lower` e `upper` è improprio.
Esse potrebbero essere sostituite ovunque da `20`, `0`, `300`.

Ma l'uso di questi "numeri fissi" in un programma è improprio e compromette leggibilità e modificabilità.

È meglio definire costanti simboliche che il compilatore sostituisce con i valori specificati prima dell'esecuzione.

```
#define LOWER   0
#define UPPER   300
#define STEP    20
```

Con queste, e con un ciclo `for`, la conversione si può riscrivere:

```
main()
{
    int fahr;        /* per illustrare la conversione %4d*/

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%4d %6.1f\n", fahr, (5.0/9.0)*(fahr-32.0));
}
```

Qui `printf` ha per 3° argomento un'espressione anziché una variabile: questo è sempre possibile per le funzioni C.

Il `for` descrive una computazione ciclica:

1. L'inizializzazione `fahr = LOWER` del `for` è eseguita prima di entrare nel ciclo;
2. Se la condizione `fahr <= UPPER` è vera, viene eseguito il corpo del `for` (l'istruzione singola `printf`);
3. si esegue la riinizializzazione `fahr=fahr+STEP` e si torna a 2

Esercizio: stampare i gradi in ordine inverso, da 300 a 0.

Input e output di caratteri

La biblioteca standard C fornisce le funzioni `getchar` e `putchar`.

`getchar()` restituisce il prossimo carattere dalla standard input

```
c = getchar()
```

`putchar(c)` scrive il `char` c sulla standard output.

Gli output prodotti da `putchar` e da `printf` si mescolano.

Esempio: copia da input a output, un carattere alla volta:

```
#include <stdio.h>
main()  /* copy stdin to stdout */
{
    int c;
    c = getchar();

    while (c != EOF) {      /* != vuol dire "diverso" */
        putchar(c);
        c = getchar();
    }
}
```

`getchar` restituisce `-1` quando trova la fine dell'input,
ma non in tutti i C, di qui la convenienza di introdurre la costante `EOF`.

In ogni caso, `getchar()` può non essere un `char`, quindi si dichiara `int c`.

In C, l'assegnazione var=expr è anche un'espressione e vale var dopo l'assegnazione.
E' dunque possibile (e più elegante) scrivere:

```
    while ((c = getchar()) != EOF)          /* != lega più di = */
        putchar(c);
```

Contare caratteri e righe

```
main()  /* conta caratteri da standard input */
{   long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

`long nc` indica che `n` è memorizzato in 4 byte (vale fino a 2^{31}).

`++nc` è un'abbreviazione per `nc=nc+1`.

`l` in `%ld` fa sì che la specifica di conversione tenga conto di `long`

Una versione con `double float` e `for` invece di `while`:

```
    double nc;
    for (nc = 0; (getchar() != EOF); ++nc)
        ;                   /* questa istruzione vuota è il corpo del for */
    printf("%.0f\n", nc);      /* f vale per float e per double */
```

In Pascal questa versione non si potrebbe scrivere, in quanto la condizione di uscita dal `for` non può essere un test arbitrario.

Per contare le righe si incrementa un contatore se si legge `\n`:

```
main()  /* conta righe da standard input */
{   int c, nl;

    nl = 0;
    while ((c=getchar()) != EOF)
        if (c=='\n')            /* == è il test di eguaglianza*/
            ++nl;
    printf("%d\n", nl);
}
```

Se c è un carattere o una sequenza di escape, `'c'` si dice costante carattere.

NB: `'c'` è diverso dalla stringa `"c"`.

Contare parole

```
main()     /* conta righe, parole, caratteri da standard input */
{
    int c, nl, nw, nc;
    int inword; /* 1 se l'input è all'interno di una parola, 0 altrimenti */

    inword = NO;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            inword = NO;
        else if (inword == NO) {
            inword = YES;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

La variabile `inword` sarebbe `boolean` in Pascal,
ma in C questo tipo non è disponibile.
Per chiarezza, però, si definiscono le costanti `YES` e `NO`.

L'istruzione `nl = nw = nc = 0` non è un'assegnazione multipla;
più semplicemente essa equivale a (= associa a destra):

`nc = (nl = (nw = 0))`

che ha senso perché ogni assegnazione è anche un'espressione.

L'operatore `||` è l'or logico, `&&` è l'and.
NB: la valutazione di un'espressione logica parte da sinistra
e si arresta appena il suo valore vero o falso è determinato.
P.es. sopra, se `c==' '`, `c=='\n'||c=='\t'` non è valutato.

Esercizi: scrivere dei programmi per: contare spazi e tab, sostituire 2 o più spazi con uno solo, sostituire i tab con > e – sovrapposti, stampare una parola per riga.

Array

```
main()  /* conta cifre e altri caratteri nell'input*/
{
    int i, c, nsp, naltri;
    int rip[10]; /*  rip e' un array di 10 interi
                         rip[i] == ripetizioni della cifra i */

    for (i=0; i<10; ++i)
        rip[i] = 0;
    nsp = naltri = 0;
    while ((c=getchar()) != EOF)
        if (c >= '0' && c <= '9')   /* il carattere c è una cifra */
            ++rip[c-'0'];           /* c-'0' è l'int da 0 a 9  */
        else if (c == ' ')          /* rappresentato dalla cifra c */
            ++nsp;
        else ++naltri;

    for (i=0; i<10; ++i)
        printf("rip[%d]=%d  ",i ,rip[i]);
    printf("\nspazi %d, altri %d\n", nsp, naltri);
}
```

Gli indici di un array di N elementi sono gli interi da 0 a N-1.

Nelle espressioni aritmetiche i caratteri valgono il loro codice ASCII
I codici ASCII sono conformi all'ordinamento alfabetico, per cui:
`'1'=='0'+1`, `'2'=='1'+1`,...

Questo è sfruttato sopra per stabilire se `c` è una cifra e
per calcolare come `c-'0'` il valore `int` rappresentato dal `char c`.

Si noti l'`if-else` multiplo.

Funzioni

Lo scopo delle funzioni è di dare un nome a una computazione.

Sapendo cosa fa una funzione, la si può invocare con il suo nome, astraendo da come svolge il suo compito.

Ciò rende il programma più chiaro.

```
main()   /* usa la funzione power */
{    int i;
     for (i = 0; i < 10; ++i)
          printf("%d %d %d\n", i, power(2,i), power(-3,i);
}
```

La funzione `power` può essere definita prima o dopo `main`, o anche in un file compilabile separatamente:

```
power(int x, int n)     /* eleva x alla potenza n */
{    int i, p;

     p = 1;
     for (i = 1; i <= n; ++i)
          p = p * x;
     return(p);
}
```

Le funzioni restituiscono un valore che può figurare nelle espressioni, p.es. `power` è usato due volte in `printf` di `main`.

L'esecuzione della funzione termina se si raggiunge il `}` finale o un'istruzione `return`

`return` può avere un argomento opzionale che è il valore restituito dalla funzione.

Argomenti e parametri di una funzione

```
main()
{    int i;
     for (i = 0; i < 10; ++i)
          printf("%d %d %d\n", i, power(2,i), power(-3,i);
}

power(int x, int n)
{    int i, p;
       ...
}
```

Una funzione può essere definita con parametri (o parametri formali), che seguono il nome della funzione racchiusi tra `(` e `)`.

I parametri sono definiti insieme al loro tipo.

Parametri e variabili dichiarate dentro una funzione sono locali, cioé distinti da entità di nome uguale definite altrove,
p.es. `i` di `power` non è `i` di `main`.

La memoria per le variabili locali viene:

- allocata alla chiamata della funzione,
- rilasciata all'uscita: ogni dato contenutovi va perso.

I parametri si comportano esattamente come variabili locali, salvo che per il modo in cui assumono un valore iniziale.

Una funzione viene invocata con il nome seguito da una lista di espressioni dette parametri attuali o argomenti, che devono corrispondere in numero e tipo agli argomenti.

All'invocazione, gli argomenti sono valutati e il loro valore è assegnato, ordinatamente, ai parametri;
quindi viene eseguito il corpo della funzione.

Questo meccanismo si dice passaggio dei parametri per valore.

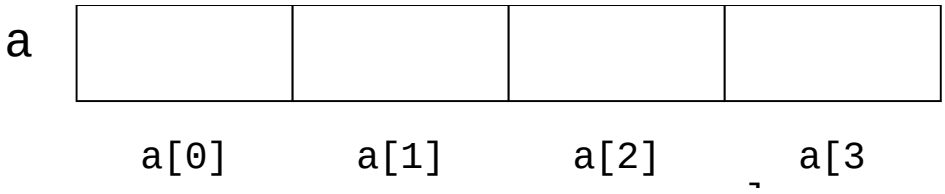
Array come puntatori e parametri

Il passaggio per valore non implica che (il codice di una) funzione non possa modificare i dati del codice chiamante.
Questo si può ottenere con i puntatori.

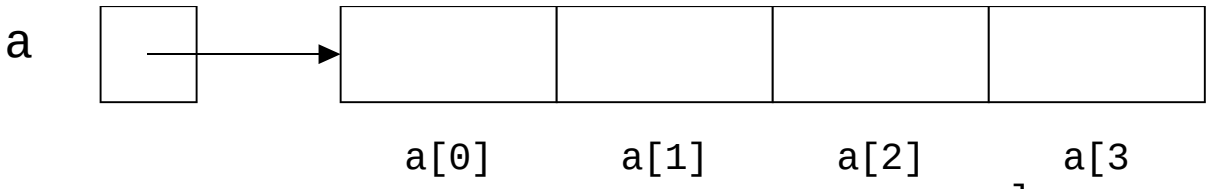
In particolare, il nome di un array è come un puntatore per accedere agli elementi dell'array:

```
int a[4];    // a è un array di 4 int
```

viene in genere pensato come:



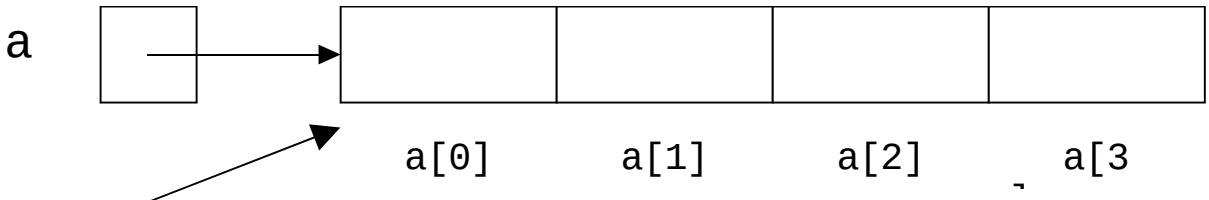
ma in C andrebbe piuttosto pensato così:



Dunque se il parametro array `x` corrisponde all'argomento array `a`,

```
f(int x[]) { x[0] = 0; }
...
f(a);
```

anche `x`, come copia di `a`, consente di accedere agli elementi di `a`:




```
f(int  x[] ) { x[0] = 0; }
```

Array di char: esempio con stringhe

Le stringhe sono in realtà array di `char` il cui ultimo elemento è `\0`

Ecco un esempio descritto top-down:

```
#define MAXLUN 1000 /* serve per fissare la dimensione degli array */
main()  /* trova la riga piu' lunga */
{
    int len, maxlen;           /* lunghezza e max lunghezza*/
    char line[MAXLUN];      /* contiene la riga letta */
    char maxline[MAXLUN];  /* max riga letta */

    maxlen = 0;
    while ((len = getline(line,MAXLUN)) > 0)
        if (len > maxlen) {       /* trovata riga > max*/
            maxlen = len;
            copy(line, maxline);
        }
    if (maxlen > 0)     /* trovata una riga (anche una vuota) */
        printf("%s",maxline);
}
```

La funzione `copy` ha per parametri due array di `char`: `s1`, `s2`.
Per parametri array, non si specifica la dimensione, che viene determinata dai parametri attuali corrispondenti (es. `line` per `s1`).

```
copy(char s1[], char s2[])
{   int i = 0;          /* dichiarazione con inizializzazione */
    while ((s2[i]=s1[i]) != '\0') /*le stringhe terminano con \0*/
        ++i;
}
```

Array di `char` - cont.

La funzione `getline` legge una riga, ne restituisce la lunghezza e la assegna al suo 1° parametro e dunque al suo 1° argomento .

Come detto prima, ciò è possibile perché il parametro è un array.

```
getline(char s[], int lim);       /* s riga letta (max lung. lim) */
{
    int c;       /* carattere letto*/
    int i;

    for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return(i);       /* a questo punto i è la lunghezza perché   */
}                    /* l'array s comincia da s[0]               */
```

Notare come le funzioni di questo programma comunichino attraverso i parametri e i `return`.

NB: anche le costanti stringa sono rappresentate con `\0` alla fine.

Ciò consente a `printf` di stabilire quando la stringa da stampare è terminata, p. es.: `"hello\n"` è rappresentata internamente con:

h	E	l	l	o	\n	\0

Esercizio: modificare il programma per stampare la lunghezza della riga, anche quando eccede MAXLUN.

Esercizio: scrivere un programma per stampare le righe più lunghe di 80 caratteri.

Esercizio: scrivere un programma per eliminare gli spazi bianchi e i tab dall'inizio di una riga.

Scope e variabili `external`

Per far comunicare le funzioni senza usare i parametri o `return`, si potrebbe pensare di usare delle variabili che siano:

- modificate dalle funzioni che producono un dato;
- lette dalle funzioni che hanno bisogno del dato come input.

Tuttavia le variabili viste finora sono inadatte a ciò:
il loro ambiente o scope è la funzione in cui sono definite;
esse si dicono locali o automatiche perché la memoria per loro è:

- riservata quando la funzione in cui compaiono è chiamata
- rilasciata all'uscita dalla funzione.

Si possono però definire variabili globali o esterne ad ogni funzione.

Queste possono essere usate da funzioni definite dopo di esse.

Una var esterna (p. es. `int i`) può anche essere usata da:

- funzioni definite prima di essa, nello stesso file (raro);
- funzioni definite in altri file

purché sia prima dichiarata con la parola chiave `extern`
(p. es. `extern int i`).

Dunque definire e dichiarare una variabile sono concetti diversi:

- la definizione avviene quando la variabile comincia a esistere, perché le viene assegnata memoria;
- la dichiarazione ha luogo dovunque venga dichiarata (resa nota alla funzione circostante) la natura della variabile, senza che sia riservata memoria.

Variabili `extern` - esempio

L'esempio precedente può essere riscritto con `line` e `maxline` variabili globali e usandole per far comunicare `main` e `copy`.

Le dichiarazioni `extern` sono ridondanti se `line`, `maxline`, `main` e `copy` sono nello stesso file.

```
#define MAXLUN 1000 /* serve per fissare la dimensione degli array */

char line[MAXLUN];          /* contiene la riga letta */
char maxline[MAXLUN];       /* max riga letta */

main()  /* trova la riga piu' lunga */
{   int len, maxlen;
    extern char line[], maxline[];

    maxlen = 0;
    while ((len = getline(line,MAXLUN) > 0)
        if (len > maxlen) {       /* trovata riga > max*/
            maxlen = len;
            copy();         /* scrivi la riga su maxline */
        }
    if (maxlen > 0)     /* trovata una riga (anche una vuota) */
        printf("%s",maxline);
}

copy()    /* copia line su maxline */
{   extern char line[], maxline[];
    int i = 0;
    while ((line[i]=maxline[i]) != '\0')
        ++i;
}
```

In generale, la comunicazione attraverso variabili non è considerata buono stile, perché in programmi complessi diventa difficile seguire come le variabili esterne vengono modificate dalle varie funzioni.

Inoltre i programmi scritti perdono generalità.
Nell'esempio, la prima `copy` si poteva usare altrove,
questa nuova è legata ai nomi `line` e `maxline`.