Un esempio di utilizzo di operatori di incremento e di assegnamento

```
#include <stdio.h>

void copia(char *t, char *s)
{
    int i = 0;

    while((t[i] = s[i]) != 0)
        i++;
}

int main(void)
{
    char s[] = "pippo";
    char t[10];

    copia(t,s);
    printf(t);
    return 0;
}
```

Commenti:

`while((t[i] = s[i]) != 0)`

Si esce dal ciclo quando la *condizione* argomento di `while` diventa falsa, più precisamente, quando l'*espressione* argomento di `while` viene valutata `0`.

`(t[i] = s[i]) != 0`

Poiché l'assegnamento è un'espressione, qui se ne usa il valore: quando essa varrà 0 il ciclo terminerà. Essa varrà 0 quando varrà 0 la variabile `t[i]`, vale a dire quando varrà 0 la variabile `s[i]`.

`i++`

a ogni iterazione, il valore di `i` viene incrementato.

s:	'p'	'i'	'p'	'p'	'o'	0				
	i									
t:	?	?	?	?	?	?	?	?	?	?

`t[i] = s[i]` vale `'p'`

s:	'p'	'i'	'p'	'p'	'o'	0				
		i								
t:	'p'	?	?	?	?	?	?	?	?	?

`t[i] = s[i]` vale `'i'`

• • •

`t[i] = s[i]` vale `'o'`

s:	'p'	'i'	'p'	'p'	'o'	0				
						i				
t:	'p'	'i'	'p'	'p'	'o'	0	?	?	?	?

`t[i] = s[i]` vale 0,

cioè la condizione del ciclo è *falsa*

Esercizio:

Cosa succede se rimpiazzo `copia` con:

```
void copia(char *t, char *s)
{
    int i = 0;

    while((t[i++] = s[i++]) != 0);
}
```

e se la rimpiazzo con:

```
void copia(char *t, char *s)
{
    int i = 0;

    while(t[i] = s[i])
        ++i;
}
```

Cosa succede se invece rimpiazzo `copia` con:

```
void copia(char *t, char *s)
{
    int i;

    for(i = 0;t[i] = s[i];i++);
}
```

Si noti che in questo caso l'istruzione `for` è immediatamente seguita da `;` : il ciclo è vuoto, nessuna istruzione viene ripetuta sotto il controllo del ciclo `for`. Solo l'espressione di controllo viene valutata ad ogni iterazione.

```
t[i] = s[i]
```

L'espressione di controllo è costituita semplicemente dall'assegnamento. Poiché l'assegnamento è un'espressione, il ciclo terminerà quando questa espressione varrà `0`. Cioè quando `s[i]`, che è il valore assegnato a `t[i]` ed è anche il valore di tutta l'espressione, sarà `0`.

N.B. Abbiamo implementato `copia` per esercizio. La libreria standard contiene la funzione `strcpy`, il cui prototipo è in `string.h`.

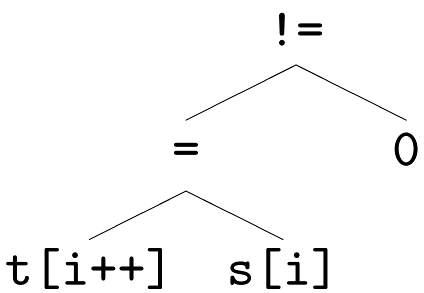
Consideriamo questa versione di `copia`:

```
void copia(char *t, char *s)
{
    int i = 0;

    while((t[i++] = s[i]) != 0);
}
```

Funziona ?

E se rimpiazziamo la condizione del ciclo `while` con `(t[i] = s[i++]) != 0` ?



Associatività e Priorità determinano
la forma dell'albero di parsing,
ma non specificano del tutto l'ordine di valutazione.
L'espressione può essere valutata in due modi diversi
a seconda di quale sottoalbero di `=` viene valutato per primo.
Lo standard non risolve questo tipo di ambiguità.

Dichiarazioni, tipi fondamentali

Dichiarazione di variabili

In C tutte le variabili devono essere dichiarate prima di essere utilizzate.

```
int main(void) {
    int x,y,z;   /* dichiarazioni */
    double alpha = 3.5, beta, gamma; /* dichiarazioni con inizializzazione */
    int i = 0; /* dichiariamo un altro intero */
    ...
}
```

Le dichiarazioni permettono al compilatore di allocare lo spazio necessario in memoria: a una variabile di tipo T sarà riservato in memoria lo spazio necessario a contenere valori di tipo T.

Informano il compilatore su come gestire le variabili dichiarate a seconda del loro tipo: ad esempio il compilatore esegue la somma di due `int` in modo diverso dalla somma di due `double`, poichè la rappresentazione interna di questi due tipi è diversa.

Blocchi: Una porzione di codice racchiusa fra parentesi graffe $\{$ e $\}$ costituisce un **blocco**.
Le dichiarazioni, se presenti, devono precedere tutte le altre istruzioni del blocco.
Una variabile dichiarata in un blocco è visibile solo fino alla fine del blocco stesso. Può inoltre essere *occultata* in blocchi interni a quello in cui è dichiarata da un'altra variabile con lo stesso nome.

Cosa stampa il frammento seguente?

```
{
    int a = 2;
    {
        double a = 3.0;
        printf("a piu' interno -2.0: %f\n",a - 2.0);
    }
    printf("a meno interno -2.0: %f\n",a - 2.0);
}
```

Una variabile dichiarata in un blocco si dice *locale* al blocco.

Il blocco più esterno è quello dove si dichiarano le funzioni: le variabili dichiarate qui sono *globali*.

```
double b = 3.14; /* b e' una variabile globale */

int main(void)
{
        printf("b globale -2.0: %f\n",b - 2.0);
}
```

Le variabili globali sono visibili in tutto il file da dove sono state dichiarate in poi. La loro visibilità può essere modificata dichiarandole `extern` o `static` (vedremo in che senso).

Sintassi: Per la dichiarazione di variabili dei tipi fondamentali:

$$declaration ::= \quad type \quad declarator\_list\,;$$
$$declarator\_list ::= \quad declarator \quad | \quad \{\,,\, declarator\}_{opt}$$
$$declarator ::= \quad identifier \quad | \quad identifier\, =\, initializer$$

Vedremo poi come dichiarare puntatori, array, funzioni, strutture, etc.

$initializer$ è un'arbitraria espressione il cui valore appartiene al tipo $type$ della variabile $identifier$.

";" : Poichè una dichiarazione è un'istruzione, deve essere terminata da ;.
In una sola dichiarazione, però, si possono dichiarare più variabili dello stesso tipo, separandole con "," e inizializzandole o meno.

Inizializzazione e Assegnamento

L'inizializzazione non è necessaria.
Se non è presente: le variabili locali non dichiarate `static` conterranno un valore casuale, le variabili globali vengono inizializzate a 0. Alcuni compilatori inizializzano a 0 anche le variabili locali.

L'inizializzazione è concettualmente diversa dall'assegnamento.
L'assegnamento modifica il valore di una variabile già esistente.
L'inizializzazione è contestuale alla dichiarazione e fornisce il valore iniziale alla variabile appena creata.

Se dichiariamo una variabile come `const` non potremo più usare `a` come operando sinistro di un'operatore di assegnamento o di incremento/decremento:

```
const int a = 5;

a = 7;  /* Errore! */
--a;    /* Errore! */
```

Tipi Fondamentali

I tipi di dati fondamentali in C sono:

`char`	`signed char`	`unsigned char`
`signed short int`	`signed int`	`signed long int`
`unsigned short int`	`unsigned int`	`unsigned long int`
`float`	`double`	`long double`

I tipi derivati: puntatori, array, strutture, unioni, ...,
sono costruiti a partire dai tipi fondamentali.

E' possibile usare alcune abbreviazioni:

`char`	`signed char`	`unsigned char`
`short`	`int`	`long`
`unsigned short`	`unsigned`	`unsigned long`
`float`	`double`	`long double`

I tipi fondamentali possono anche essere chiamati *tipi aritmetici*, poiché oggetti di questi tipi possono comparire come operandi di operatori aritmetici.

`float`,`double`,`long double` costituiscono l'insieme dei *tipi reali*: rappresentano mediante *virgola mobile* numeri con parte frazionaria.

Gli altri tipi aritmetici sono detti *tipi interi*.

I tipi interi `signed` possono rappresentare interi negativi. Di solito la rappresentazione interna è in *complemento a 2*.

I tipi interi `unsigned` rappresentano interi non-negativi.

Il tipo `char`

Quando si pensa ai tipi interi in C, è spesso utile pensare in termini di occupazione di memoria:

Un `char` in C può spesso essere visto come un intero "piccolo", e quindi utilizzato in contesti aritmetici. Viceversa, è possibile utilizzare qualunque tipo intero per rappresentare caratteri.

Un `char` occupa 1 byte di memoria.

In genere quindi, un `char` può memorizzare 256 valori differenti.

Le costanti carattere (es.: `'x'`,`'3'`,`';'`) *sono* di tipo `int`! (in C++ sono `char`). Comunque, una costante carattere può essere assegnata a un `char`:

```
char c = 'a';
```

Si noti la differenza:

```
char c;
c = '0';
c = 0;
```

`c = '0'`: Assegna alla variabile `c` di tipo `char` il valore corrispondente al carattere `'0'` (in ASCII: 48).

`c = 0`: Assegna a `c` il valore intero 0.

Possiamo fare lo stesso con ogni tipo intero:

```
short s;
c =  1;
s = '1';
printf("%d\n",s+c);
```

Cosa viene stampato ?

```
/* char.c */
#include <stdio.h>

int main(void)
{
    char c;
    short s;

    c = '0';
    printf("%s:  intero: %d, carattere: %c\n",
        "c = '0'",c,c);

    c = 0;
    printf("%s:  intero: %d, carattere: %c\n",
        "c = 0",c,c);

    c = 1;
    s = '1';
    printf("s+c:  %d\n",s+c);
}
```

Commenti:

```
printf("%s:  intero:  %d, carattere:  %c\n","c = '0'",c,c);
```

Il primo argomento (`%s`) della stringa formato di `printf` è una stringa: in questo caso la stringa costante `"c = '0'"`.

Il secondo argomento è un intero da visualizzare in base 10: qui passiamo `c` che è di tipo `char`.

Il terzo argomento è un intero da visualizzare come carattere: qui passiamo ancora `c`.

```
printf("s+c:  %d\n",s+c);
```

Qui passiamo come argomento `%d` il risultato della somma dello `short s` con il `char c`.

Vi sono alcuni caratteri non stampabili direttamente, ma attraverso una *sequenza di escape*:

carattere	**sequenza**	valore ASCII
allarme	\a	7
backslash	\\	92
backspace	\b	8
carriage return	\r	13
newline	\n	10
doppi apici	\"	34
form feed	\f	12
tab	\t	9
vertical tab	\v	11
apice	\'	39
carattere nullo	\0	0

Si può specificare il valore ASCII di una costante carattere in ottale (`es: '\007'`), o esadecimale (`es: '\x30'`).

Su macchine con byte di 8 bit:

`unsigned char` rappresenta l'intervallo di interi $[0, 255]$,
`signed char` rappresenta l'intervallo di interi $[-128, 127]$.
`char` equivale a uno fra `unsigned char` e `signed char`: dipende dal compilatore.

```
#include <stdio.h>

int main(void)
{
    char c = -1;
    signed char s = -1;
    unsigned char u = -1;

    printf("c = %d, s = %d, u = %d\n",c,s,u);
}
```

Provare a compilarlo con le opzioni `-funsigned-char` e `-fsigned-char` di gcc.

Il tipo `int`

L'intervallo di valori rappresentato dal tipo `int` dipende dalla macchina: solitamente un `int` viene memorizzato in una *parola* della macchina.

Assumiamo che un `int` occupi 4 byte (da 8 bit).
Allora `int` rappresenta l'intervallo di interi:

$$[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$$

Gli `int` sono rappresentati in complemento a 2.

Se durante la computazione si cerca di memorizzare in un `int` un valore esterno all'intervallo, si ha *integer overflow*. Solitamente l'esecuzione procede, ma i risultati non sono affidabili.

Le costanti intere possono essere scritte anche in ottale e esadecimale.

I tipi `short`, `long` e `unsigned`

`short` si usa quando si vuole risparmiare spazio e i valori interi in gioco sono ''piccoli''.

Spesso uno `short` occupa 2 byte. In tal caso `short` rappresenta l'intervallo $[-2^{15}, 2^{15}-1] = [-32768, 32767]$.

`long` si usa quando gli interi in gioco sono così grandi da non poter essere contenuti in un `int`: questa soluzione è efficace se effettivamente il tipo `long` occupa più byte del tipo `int`.

Lo standard ANSI assicura solo che `short` occupa al più tanti byte quanto `int`, che a sua volta occupa al più tanti byte quanto `long`.

In generale, se un tipo intero con segno T è memorizzato in k byte, allora T rappresenta l'intervallo $[-2^{8k-1}, 2^{8k-1}-1]$.

I tipi `unsigned` sono memorizzati usando la stessa quantità di byte delle corrispettive versioni con segno.

Se il tipo con segno T rappresenta l'intervallo $[-2^{8k-1}, 2^{8k-1}-1]$ allora `unsigned` T rappresenta l'intervallo $[0, 2^{8k}-1]$.

Ad esempio, se gli `int` occupano 4 byte, allora gli `unsigned` rappresentano l'intervallo $[0, 2^{32}-1] = [0, 4294967295]$.

I tipi `unsigned` sono trattati con l'aritmetica modulo 2^{8k}.

```
/* modulare.c */
#include <stdio.h>
#include <limits.h>

int main(void)
{
    int i;
    unsigned u = UINT_MAX;  /* valore massimo per tipo unsigned */

    for(i = 0;i < 10;i++)
        printf("%u + %d = %u\n",u,i,u+i);
    for(i = 0;i < 10;i++)
        printf("%u * %d = %u\n",u,i,u*i);
}
```

`limits.h`: File di intestazione che contiene macro che definiscono i valori limite per i tipi interi (es. `UINT_MAX`).

Ogni costante intera può essere seguita da un suffisso che ne specifichi il tipo.

Suffisso	**Tipo**	**Esempio**
`u` oppure `U`	`unsigned`	`37U`
`l` oppure `L`	`long`	`37L`
`ul` oppure `UL`	`unsigned long`	`37ul`

Se non viene specificato alcun suffisso per una costante, viene automaticamente scelto il primo fra i seguenti tipi:

`int`, `long`, `unsigned long`,

che possa rappresentare la costante.

I tipi reali `float`, `double`, `long double`

Per rappresentare costanti reali bisogna usare il punto decimale per differenziarle dalle costanti intere:

`3.14`	è una costante `double`.
`3`	è una costante `int`.
`3.0`	è una costante `double`.

E' disponibile anche la notazione scientifica:

`1.234567e5`	è	`123456.7`
`1.234567e-3`	è	`0.001234567`

I tipi reali vengono rappresentati in virgola mobile.

Il numero di byte per rappresentare un `double` non è inferiore al numero di byte usato per rappresentare un `float`.

Il numero di byte per rappresentare un `long double` non è inferiore al numero di byte usato per rappresentare un `double`.

Su molte macchine, si usano 4 byte per `float` e 8 per `double`.

Con questa rappresentazione non tutti i reali in un dato intervallo sono rappresentabili, inoltre le operazioni aritmetiche non forniscono valori esatti, ma approssimati a un certo numero di cifre decimali.

La rappresentazione in virgola mobile è caratterizzata da due parametri: *precisione* e *intervallo*.

La *precisione* descrive il numero di cifre significative rappresentabili in notazione decimale.

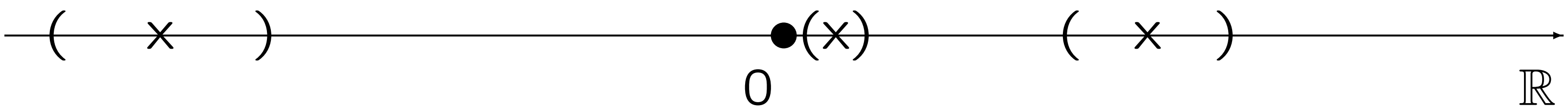
L'*intervallo* specifica il più piccolo e il più grande fra i valori reali rappresentabili.

Su macchine con `float` di 4 byte, la precisione del tipo `float` è di 6 cifre significative e un intervallo di circa $[10^{-38}, 10^{38}]$.

Su macchine con `double` di 8 byte, la precisione del tipo `double` è di 15 cifre significative e un intervallo di circa $[10^{-308}, 10^{308}]$.

La rappresentazione in virgola mobile è caratterizzata da due parametri: *precisione* e *intervallo*.

La *precisione* finita implica che non tutti i reali appartenenti all'*intervallo* sono rappresentabili: valori reali diversi possono essere rappresentati nello stesso modo (e quindi resi indistinguibili), se sufficientemente vicini rispetto al loro valore.



Nella figura gli intervalli fra tonde schematizzano insiemi di reali che in forma decimale coincidono per tutte le cifre significative. Ogni due numeri che appartengono allo stesso intervallo ( ) non sono distinguibili dalla rappresentazione in virgola mobile.

Rappresentiamo al più $2^{8*\mathtt{sizeof(double)}}$ reali (in realtà razionali) diversi.

Le costanti reali in C sono di tipo `double`.

Per modificare questa impostazione di default, si può specificare un suffisso:

Suffisso	**Tipo**	**Esempio**
`f` oppure `F`	`float`	`3.7F`
`l` oppure `L`	`long double`	`3.7L`

In C, il tipo "naturale" per le operazioni sui reali è il tipo `double`.

Anche le funzioni della libreria matematica standard, solitamente restituiscono `double` e richiedono argomenti `double`.

```
double sqrt(double); /* prototipo per funz.  radice quadrata */
```

L'operatore `sizeof`

`sizeof(`A`)` fornisce la dimensione in byte dell'oggetto A o del tipo A.

`sizeof` ha la stessa priorità degli altri operatori unari.

Devono valere le seguenti relazioni:

```
sizeof(char) == 1
sizeof(short) <= sizeof(int) && sizeof(int) <= sizeof(long)
sizeof(unsigned) == sizeof(int)
sizeof(float) <= sizeof(double) && sizeof(double) <= sizeof(long double)
```

```
#include <stdio.h>

void stampasizeof(char *s, int size)
{
    printf("%s:%d\n",s,size);
}

int main(void)
{
    stampasizeof("char",sizeof(char));
    stampasizeof("short",sizeof(short));
    stampasizeof("int",sizeof(int));
    stampasizeof("long",sizeof(long));
    stampasizeof("float",sizeof(float));
    stampasizeof("double",sizeof(double));
    stampasizeof("long double",sizeof(long double));
}
```

Un esercizio sull'uso del preprocessore:

```
#include <stdio.h>

#define stampa(s)   stampasizeof(#s,sizeof(s))

void stampasizeof(char *s, int size)
{
    printf("%s:%d\n",s,size);
}

int main(void)
{
    stampa(char);
    stampa(short);
    stampa(int);
    stampa(long);
    stampa(float);
    stampa(double);
    stampa(long double);
}
```

Commenti:

Si sono usate alcune caratteristiche del preprocessore per rendere più “eleganti” le chiamate alla funzione che stampa il nome del tipo (una stringa costante) e la sua dimensione.

```
#define stampa(s) stampasizeof(#s,sizeof(s))
```

Definizione di macro con parametri: `s` è un parametro della macro. Quando la macro è richiamata, come in `stampa(short)`, la stringa `short` viene sostituita (in fase di preprocessamento) a ogni occorrenza di `s` nel testo della macro `stampa`.

`#s` : direttiva al preprocessore che determina la sostituzione della stringa `short` con la stringa `"short"`.

Un altro esempio di *macro con parametri*

```
#define max(A,B)    ((A) >= (B) ? (A) : (B))

printf("Il maggiore tra %d e %d e' %d \n",9,7,max(7,9));
```

Perchè tante parentesi ?
Sono necessarie per evitare effetti collaterali della macro-espansione.
Si consideri infatti la definizione (errata):

```
#define maxbis(A,B)    A >= B ? A : B

int x = 5 * maxbis(7,9); /* il valore di x sara' 7 */
```

In questo esempio la macro viene espansa come
`5 * 7 >= 9 ?  7 :  9`. Poiché la priorità di `*` è maggiore della priorità di `?:`, e dato che $35 \geq 9$, il risultato dell'intera espressione è 7.

Conversioni e Cast

Ogni espressione aritmetica ha un tipo.

Promozioni: oggetti `char` e `short` possono essere utilizzati in tutte le espressioni dove si possono usare `int` o `unsigned`.
Il tipo dell'espressione è convertito in `int` se tutti i tipi coinvolti nell'espressione possono essere convertiti in `int`.
Altrimenti, il tipo dell'espressione è convertito in `unsigned`.

```
char c = 'A';

printf("%c\n",c);
```

Nel secondo argomento di `printf`, `c` è convertito (promosso) a `int`.

Conversioni Aritmetiche

Determinano il tipo di un'espressione in cui un operatore aritmetico ha operandi di tipi differenti:

- Se uno degli operandi è `long double` anche l'altro è convertito in `long double`.

- Altrimenti, se uno è `double`, l'altro è convertito in `double`.

- Altrimenti, se uno è `float`, l'altro è convertito in `float`.

- Negli altri casi si applicano le promozioni intere e:

Negli altri casi si applicano le promozioni intere e:

- Se uno degli operandi è `unsigned long` anche l'altro è convertito in `unsigned long`.

- Se uno è `long` e l'altro è `unsigned`:
  – se `sizeof(unsigned) < sizeof(long)` l'operando `unsigned` è convertito in `long`.
  – altrimenti entrambi convertiti in `unsigned long`.

- Altrimenti, se uno è `long`, l'altro è convertito in `long`.

- Altrimenti, se uno è `unsigned`, l'altro è convertito in `unsigned`.

- Altrimenti, entrambi gli operandi sono già stati promossi a `int`.

```
char c; short s; int i; long l;
unsigned u; unsigned long ul;
float f; double d; long double ld;
```

Espressione	**Tipo**
`c - s / i`	`int`
`u * 2.0 - 1`	`double`
`c + 3`	`int`
`c + 5.0`	`double`
`d + s`	`double`
`2 * i / l`	`long`
`u * 7 - i`	`unsigned`
`f * 7 - i`	`float`
`7 * s * ul`	`unsigned long`
`ld + c`	`long double`
`u - ul`	`unsigned long`
`u - l`	`dipendente dal sistema`

Cast

Per effettuare esplicitamente delle conversioni si usa il costrutto di *cast*.

Sintassi: (*type*) *expr*

Semantica: il valore dell'espressione *expr* viene convertito al tipo *type*.

```
double x = 5.3;
int i = (int) x; /* x viene convertito in int */
```

L'operatore di cast ha la stessa priorità degli altri operatori unari.

Input/Output con `getchar()` e `putchar()`

Quando si deve gestire l'input/output a livello di singoli caratteri, è preferibile usare le *macro* `getchar()` e `putchar()` definite in `stdio.h` piuttosto che `scanf` e `printf`.

`scanf` e `printf` realizzano input/output *formattato* (attraverso il loro primo argomento `char *format`), e sono molto meno efficienti di `getchar()` e `putchar()` che semplicemente leggono/scrivono un carattere alla volta.

```
int c;

c = getchar();  /* legge un carattere e lo pone in c  */
putchar(c);     /* scrive il carattere contenuto in c */
```

```
/* toupper.c */
#include <stdio.h>

int main(void)
{
    int c;

    while((c = getchar()) != EOF)
        if(c >= 'a' && c <= 'z')
            putchar(c + 'A' - 'a');
        else
            putchar(c);
    return 0;
}
```

Converte l'input trasformando le lettere minuscole in maiuscole.

(N.B.: la funzione `toupper()` è già nella libreria standard: per usarla includere `ctype.h`, vedi `toupperctype.c` nel file `.zip` relativo a questa lezione)

Forniamogli un testo di prova in input: **`toupper < testo.txt`**

Commenti su `toupper.c`:

`stdio.h` : Contiene le definizioni delle macro `EOF`,`getchar`,`putchar`.

`int c` : La variabile `c`, destinata a contenere un carattere alla volta del testo in input, è dichiarata `int`, perchè ?

`while((c = getchar()) != EOF)` :
– `c = getchar()` : viene posto in `c` il valore `int` restituito da `getchar()`. Il valore dell'espressione di assegnamento è il valore restituito da `getchar()`.
– `(c = getchar()) != EOF` : il valore dell'espressione di assegnamento viene confrontato con il valore `EOF` di tipo `int`: questo valore speciale viene restituito da `getchar()` quando raggiunge la fine del file.

– `(c = getchar()) != EOF` : Si noti la parentesizzazione: poiché l'operatore relazionale `!=` ha prorità maggiore dell'operatore di assegnamento, devo parentesizzare l'espressione di assegnamento. Cosa accadrebbe se non ci fossero le parentesi ?
– **`while( ... )`** : il ciclo viene ripetuto fino a quando `getchar()` non restituisce `EOF`.

`if(c >= 'a' && c <= 'z')` : La condizione è vera sse:
il valore di `c` è maggiore o uguale al codice ASCII del carattere `'a'`
e
il valore di `c` è minore o uguale al codice ASCII del carattere `'z'`.

`&&` è l'operatore logico `AND`.

`putchar(c + 'A' - 'a');` :

`putchar` richiede un `int` come argomento.
Il carattere il cui codice ASCII è dato dal valore dell'argomento di `putchar` viene stampato sullo standard output.

Le costanti carattere, come `'A'` e `'a'`, possono comparire in contesti numerici, il loro valore è il loro codice ASCII.

I caratteri `a,b,...,z` hanno codici ASCII consecutivi: `'a' == 97`, `'b' == 98`, `....` Allo stesso modo: `'A' == 65`, `'B' == 66`, `....`
Quando `c` contiene il codice di una minuscola, l'espressione `c + 'A' -'a'` esprime il codice della maiuscola corrispondente.

Input/Output su `FILE`: `fgetc` e `fputc`

`int fgetc(FILE *fp)`
Legge un carattere dal FILE (lo *stream*) associato al puntatore `fp`.

`int fputc(int c,FILE *fp)`
Scrive il carattere `c` sul FILE puntato da `fp`.

Entrambe restituiscono il carattere letto o scritto, oppure `EOF` se la lettura o scrittura non è andata a buon fine.

Il loro prototipo è in `stdio.h`, che contiene anche due macro equivalenti a `fgetc` e `fputc`. Tali macro sono `getc` e `putc`.

Le macro sono spesso più efficienti delle funzioni, ma sono meno affidabili a causa di effetti collaterali negli argomenti (vedi pag.115).

Il programma che segue converte il file `testo.txt` in maiuscolo, ponendo il risultato nel file `maiusc.txt`.

```
/* toupperfile.c */
#include <stdio.h>

int main(void)
{
    int c;
    FILE *fr = fopen("testo.txt","r");
    FILE *fw = fopen("maiusc.txt","w");

    while((c = fgetc(fr)) != EOF)
        if(c >= 'a' && c <= 'z')
            fputc(c + 'A' - 'a',fw);
        else
            fputc(c,fw);
    return 0;
}
```

Lo standard input e lo standard output sono rispettivamente associati a puntatori a `FILE` rispettivamente denotati con `stdin` e `stdout`.

`getchar()` equivale a `fgetc(stdin)`.

`putchar(c)` equivale a `fputc(c,stdout)`.

Esiste un terzo *stream* associato per default a un processo: `stderr`, *standard error*, che per default è associato all'output su schermo.

Si esegua il programma seguente con:

`touppermsg < testo.txt > maiusc.txt`

```
/* touppermsg.c */
#include <stdio.h>

int main(void)
{
    int c;

    fprintf(stderr,"Sto computando...\n");
    while((c = fgetc(stdin)) != EOF)
        if(c >= 'a' && c <= 'z')
            fputc(c + 'A' - 'a',stdout);
        else
            fputc(c,stdout);
    fprintf(stderr,"Finito!\n");
    return 0;
}
```

Flusso del controllo, istruzioni

Operatori relazionali e di uguaglianza:

Operatori relazionali: `<`, `>`, `<=`, `>=`.

Operatori di uguaglianza: `==`, `!=`.

Arità: binaria.

Priorità: i relazionali hanno priorità maggiore degli operatori di uguaglianza. Entrambi i tipi hanno priorità inferiore agli operatori `+` e `-` binari, e maggiore degli operatori di assegnamento.

Associatività: da sinistra a destra.

Gli operatori relazionali e d'uguaglianza restituiscono i valori booleani *vero*,*falso*.

In C il valore booleano *falso* è rappresentato dal valore di tipo **`int`** `0`, e dal valore **`double`** `0.0`. Inoltre, ogni modo di esprimere `0` rappresenta il valore booleano *falso*. In particolare: il carattere `'\0'` e il puntatore nullo `NULL`.

Il valore booleano *vero* è rappresentato da qualunque valore diverso da `0`.

```
if(5) printf("Vero!\n");
```

Gli operatori relazionali e d'uguaglianza restituiscono `1` quando la condizione testata è vera.
Restituiscono `0` altrimenti.

Avvertenze:

Poiché la priorità degli operatori relazionali e d'uguaglianza è minore di quella degli operatori aritmetici, si può scrivere senza uso di parentesi:

```
if(7 - 1 > 1) printf("Vero!\n"); /* la condizione equivale a (7 - 1) > 1 */
```

Poiché invece tale priorità è maggiore di quella degli operatori di assegnamento, si devono usare le parentesi per eseguire l'assegnamento prima del confronto. Altrimenti:

```
int c = 4, d = 4;

if (c -= d >= 1) printf("Vero: %d\n",c); else printf("Falso: %d\n",c);
```

Non si confonda l'operatore di assegnamento = con l'operatore di uguaglianza ==.

```
int c = 0, d = 0;
if(c = d) printf ("c uguale a d?":%d,%d\n",c,d);
    else printf("c diverso da d?:%d,%d\n",c,d);
```

Operatori logici `&&`, `||`, `!`

`&&`: `AND`. Binario, associa da sinistra a destra.
`c1 && c2` restituisce `1` sse `c1 != 0` e `c2 != 0`.
Restituisce `0` altrimenti.

`||`: `OR`. Binario, associa da sinistra a destra.
`c1 || c2` restituisce `1` se almeno uno tra `c1` e `c2` è diverso da `0`.
Restituisce `0` altrimenti.

`!`: `NOT`. Unario, associa da destra a sinistra.
`!c` restituisce `1` se `c == 0`. Restituisce `0` altrimenti.

La priorità degli operatori logici è maggiore della priorità degli operatori di assegnamento. La priorità di `!` è la stessa degli altri operatori unari, la priorità di `||` è inferiore a quella di `&&` che è inferiore a quella degli operatori di uguaglianza.

Avvertenze:

Si voglia testare che il valore di `int j` cada nell'intervallo $[3, 5]$.

```
int j = 6;

if(3 <= j <= 5) printf ("SI'\n"); else printf("NO\n");      /* stampa SI' */

if(3 <= j && j <= 5) printf ("SI'\n"); else printf("NO\n"); /* stampa NO  */
```

Si noti anche:
Il valore di `!!5` è il valore di `!0`, vale a dire `1`.

Valutazione Cortocircuitata

`c1 && c2` :
Se `c1 == 0` non c'è bisogno di valutare `c2` per stabilire che
`c1 && c2 == 0`.

Analogamente:

`c1 || c2` :
Se `c1 != 0` non c'è bisogno di valutare `c2` per stabilire che
`c1 || c2 == 1`.

In C il processo evita di calcolare il valore di `c2` in questi casi.

```
int c;

while((c = getchar()) != EOF && dosomething(c)) ...
```

```c
/* divisioni.c */
#include <stdio.h>

int main(void)
{
    int v[10] = { 2, 5, 12, 7, 29, 0, 3, 4, 6, 7 };

    int i, a = 300000;

    printf("%d",a);
    for(i = 0; i < 10; i++)
        if(v[i] && (a /= v[i]) >= 1)
            printf("/%d=%d",v[i],a);
    putchar('\n');
}
```

`if(v[i] && (a /= v[i]) >= 1)` :
Grazie alla valutazione cortocircuitata non viene eseguita l'espressione `a /= v[i]` quando `v[i] == 0`.

Istruzioni

Istruzione composta, o **blocco**: Sequenza di istruzioni racchiuse fra graffe. Le dichiarazioni possono apparire solo prima di ogni altro tipo di istruzione in un blocco.

$compound\_statement ::= \{ \quad \{declaration\}_{0+} \quad \{statement\}_{0+} \quad \}$

Se in un costrutto C, come `while`,`for`,`if`, vogliamo porre più istruzioni, dobbiamo raccoglierle in un'istruzione composta.

Istruzione vuota: E' costituita dal solo simbolo `;`.

Utile quando a livello sintattico è necessaria un'istruzione, ma a livello semantico non vi è nulla da fare:

```
for(i = 0;t[i] = s[i];i++)
    ;
```

Istruzioni di selezione: `if`, `if-else`

`if` (*expr*)
    statement

L'istruzione *statement* viene eseguita solo se il valore di *expr* è diverso da `0`.

`if` (*expr*)
    statement1
`else`
    statement2

Se *expr* `!= 0` allora viene eseguita l'istruzione *statement1*, altrimenti viene eseguita l'istruzione *statement2*.

Avvertenze:

Si ricordi: in C ogni *espressione* costituisce una *condizione*: se tale espressione vale `0`, allora come condizione risulta *falsa*, altrimenti risulta *vera*.

Dangling Else:

```
if(a == 1) if(b == 2) printf("%d",b); else printf("%d",a);
```

equivale a

```
if(a == 1) {
    if(b == 2)
        printf("%d",b);
    else
        printf("%d",a);
}
```

il ramo `else` viene associato all'`if` più interno (il più vicino).

Per modificare la risoluzione standard del *Dangling Else* si utilizzano opportunamente le istruzioni composte:

```
if(a == 1)
{
    if(b == 2)
        printf("%d",b);
}
else
    printf("%d",a);
```

Serie di selezioni:

if (*expr*)
statement1
else if (*expr*)
statement2
...
else if (*expr*)
*statement*k

Istruzioni per realizzare iterazioni: `while`

```
while  (expr)
       statement
```

Si valuta *expr*: se il valore è diverso da `0` si esegue `statement`, dopodichè si ritorna all'inizio del ciclo: si valuta *expr* di nuovo.

Quando *expr* `== 0` il flusso abbandona il ciclo `while` e prosegue con l'istruzione successiva al ciclo stesso.

A volte non c'è bisogno di *statement*, poichè tutto il lavoro è svolto nella valutazione della condizione di controllo *expr*. In questi casi si usa l'istruzione vuota.

```
while((c = getchar()) != EOF && putchar(toupper(c)))
        ;
```

Esercizio sui costrutti iterativi:

Calcolo del fattoriale

Sia n un intero positivo:

$$0! = 1, \qquad\qquad n! = n((n-1)!) = n(n-1)(n-2)\cdots 2.$$

La funzione fattoriale cresce molto velocemente:

$$13! = 6227020800$$

AVVERTENZA:

-se scegliamo di rappresentare i valori del fattoriale in un `long` su implementazioni in cui `sizeof(long) == 4`, potremo calcolare correttamente il fattoriale di interi piuttosto piccoli (da 0! a 12!).

-se scegliamo di rappresentare i valori in un `double`, avremo solo valori approssimati del fattoriale.

-dovremmo implementare una sorta di tipo intero "esteso" per rappresentare i grandi valori interi della funzione fattoriale: vedremo in un esercizio come si può affrontare questo problema.

-per adesso scegliamo un'implementazione con `double`.

```c
#include <stdio.h>

int main(void)
{
    int n, i = 1;
    double factorial = 1.0;

    while(1) {

        printf("Immetti n:\n");
        if(!scanf("%d",&n)) {
            fprintf(stderr,"Errore in lettura!\n");
            return -1;
        }

        while(++i <= n)
            factorial *= i;

        printf("%d! = %.0f\n",n,factorial);
        i = 1; factorial = 1.0;
    }
}
```

Commenti su `factorial.c`

`while(1)` : L'espressione `1` è costantemente diversa da `0`: il ciclo viene ripetuto per sempre.
Questo è uno dei due modi standard del C per realizzare cicli infiniti.
Per uscire da un siffatto ciclo, o si interrompe il processo dall'esterno, oppure si usano delle istruzioni di uscita (da cicli o da funzioni) all'interno del ciclo.

`if(!scanf("%d",&n)) {`: Si usa il valore restituito da `scanf`, cioè il numero di argomenti della stringa formato letti con successo, per controllare che la lettura dell'intero `n` vada a buon fine.

```
fprintf(stderr,"Errore in lettura!\n");
return -1;
```

In caso `scanf` non abbia concluso con successo la lettura dell'intero, si stampa un messaggio d'errore su `stderr`, e poi si esce dalla funzione (e a fortiori) dal ciclo, con l'istruzione `return -1`.
Poiché la funzione in questione è `main` il processo termina e il valore `-1` è passato al sistema operativo.

```
while(++i <= n)
    factorial *= i;
```

Ciclo `while` che costruisce il valore del fattoriale.