Operatori di incremento e decremento: ++ e --

++ e -- sono operatori unari con la stessa priorità del meno unario e associatività da destra a sinistra.

Si possono applicare solo a variabili (di tipi interi, floating o puntatori), ma non a espressioni generiche (anche se di questi tipi).

```
int c = 5;

c++;  /* c == 6 */
c--;  /* c == 5 */
--c;  /* c == 4 */
++c;  /* c == 5 */
5++;  /* Errore! */
--(7 + bubble(num,DIM)); /* Errore! */
```

Semantica degli operatori di incremento/decremento

Postfisso: Il valore dell' espressione `c++` è il valore di `c`. Mentre `c` stesso è incrementato di 1.

Prefisso: Il valore dell' espressione `++c` è il valore di `c` incrementato di 1. Inoltre `c` stesso è incrementato di 1.

Analogo discorso vale per gli operatori di decremento.

```
int c = 5;

int b = 30 / c++; /* b == 6, c == 6 */
int d = 6 + --c;  /* d = 11, c == 5 */
```

Gli operatori prefissi modificano il valore della variabile cui sono applicati prima che se ne utilizzi il valore. Gli operatori postfissi modificano il valore della variabile dopo l'utilizzo del valore (vecchio) nell'espressione.

Operatori di assegnamento

In C, l'assegnamento è un'espressione:

```
int a,b,c;

a = b = c = 0;
printf("%d\n",b = 4); /* stampa: 4 */
```

Il valore di un'espressione *variabile* = *expr* è il valore assunto da *variabile* dopo l'assegnamento, vale a dire, il valore di *expr*.

Non ci si confonda con l'operatore di uguaglianza ==

```
int b = 0;

printf("%d,", b == 4); printf("%d,", b = 4); printf("%d\n", b == 4);
/* stampa: 0,4,1 */
```

L'operatore di assegnamento ha priorità più bassa rispetto agli operatori aritmetici, e associa da destra a sinistra.

Operatori di assegnamento combinati:

`+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`

La loro semantica è:

variabile op= expr equivale a *variabile = variabile op expr*
(ma *variabile* è valutata una volta sola, questo aspetto è da ricordare quando *variabile* è una componente di un array, di una struttura, etc., esempio: `a[i++] += 2;`)

Esempio:

```
int a = 1;

a += 5; /* a == 6 */
```

Esempio di utilizzo di operatori di assegnamento combinati

```
/* power2.c */
#include <stdio.h>

#define MAXEXP  10

int main(void)
{
    int i = 0, power = 1;

    while(++i <= MAXEXP)
        printf("%6d", power *= 2);
    printf("\n");
    return 0;
}
```

`while(++i <= MAXEXP)` : il ciclo viene ripetuto fino a quando `i` non sarà maggiore di `MAXEXP`.

`power *= 2` : ad ogni iterazione, il valore di `power` viene raddoppiato, e poi stampato.

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.

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`.

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 *mascherata* in blocchi interni a quello dove è dichiarata.

Cosa stampa il frammento seguente?

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