# Programmazione 2

Lezione 3 | 02 Marzo

Università degli Studi di Modena e Reggio Emilia — Unimore

## 1. Operatore di Assegnamento

### 1.1. Sintassi

**Definizione**

```
<variabile> = <espressione>;
```

```
a = 10;    // a ora contiene 10
a = -2;    // a ora contiene -2 (sovrascriviamo)
```

Finché la variabile non è `const`, possiamo assegnarla quante volte vogliamo. Il `const` fa sì che quella variabile non possa mai comparire *a sinistra* dell'operatore di assegnamento: è un controllo **lessicale** del compilatore.

### 1.2. Lvalue vs Rvalue

Una variabile può comparire sia a sinistra che a destra:

| Posizione | Significato |
|---|---|
| A sinistra (*lvalue*) | L'indirizzo della cella di memoria |
| A destra (*rvalue*) | Il valore contenuto in quella cella |

```
a = a + 1;
// Prima si valuta il lato destro (a + 1), poi si esegue l'
   assegnamento.
```

Il valore a sinistra deve essere **mappato in memoria**:

```
2 = b;  // ERRORE di compilazione: 2 non e' un lvalue
```

### 1.3. Yoda Conditions — il "barbatrucco"

Nei sistemi safety-critical (auto, aerei, medicale) si usano le cosiddette **Yoda conditions**: mettere la costante *a sinistra* del confronto.

```
// Stile normale (bug-prone):
if (a == 5) { ... }

// Yoda condition:
if (5 == a) { ... }
```

Se per errore scrivi `=` invece di `==`:

```
if (5 = a) { ... }  // <- il compilatore se ne ACCORGE: 5 non e' un
   lvalue
```

Questo trasforma un possibile bug silenzioso in un errore di compilazione. Furbo.

## 2. Inizializzazione di Variabili

**Definizione**

**Inizializzazione** = prima assegnazione che si fa a una variabile, quella che le dà un valore e quindi un significato.

```
int a = 10;   // definizione + inizializzazione (tutto in una riga)
int b;        // definizione senza inizializzazione <- PERICOLO
```

### 2.1. Una variabile non inizializzata NON è vuota

**Pericolo / UB**

Ha il contenuto *casuale* già presente nella cella di memoria occupata. Può essere qualsiasi cosa: un valore senza senso, un valore vecchio di un'altra variabile, qualcosa che causa comportamento non definito.

**MAI usare una variabile dichiarata ma non inizializzata.**

## 3. L'Assegnamento è un'Espressione in C

In C, l'operatore `=` è un'*espressione* che **denota il valore inserito nella variabile**. Quindi può comparire dentro altre espressioni:

```
3 * (a = 2);  // inserisce 2 in a, l'espressione vale 6
```

**Attenzione**

**In pratica: non farlo mai.** Non mescolare assegnamenti ed espressioni complesse nello stesso statement. Il codice diventa illeggibile e il compilatore potrebbe sorprenderti.

### 3.1. Associatività a destra

L'assegnamento è **associativo a destra**, utile per inizializzare più variabili:

```
a = b = 2;    // equivale a: b = 2; poi a = b;
              // risultato: sia a che b valgono 2
```

## 4. Operatori di Incremento e Decremento

`++` e `--` incrementano o decrementano una variabile di 1.

| Sintassi | Nome | Comportamento |
|---|---|---|
| `++k` | Pre-incremento | Incrementa *prima*, poi restituisce il nuovo valore |
| `k++` | Post-incremento | Restituisce il valore corrente, poi incrementa |
| `--k` | Pre-decremento | Decrementa *prima*, poi restituisce il nuovo valore |
| `k--` | Post-decremento | Restituisce il valore corrente, poi decrementa |

### 4.1. Esempi

```
int i, k = 5;
i = ++k;            // k diventa 6, poi i prende 6  --> i=6, k=6
```

```
int i, k = 5;
i = k++;            // i prende 5 (valore attuale), poi k diventa 6
    --> i=5, k=6
```

```
int i=4, j, k=5;
j = i + k++;        // j = 4 + 5 = 9, poi k diventa 6  --> j=9, k=6
```

### 4.2. Undefined Behavior con side effects multipli

**Pericolo / UB**

```
int j, k = 5;
j = ++k - k++;     // UNDEFINED BEHAVIOR. Non farlo MAI.
```

Modificare e leggere la stessa variabile più volte nella stessa espressione, senza un punto di sequenza definito, è UB. Il risultato dipende dal compilatore, dall'ottimizzazione, dalla luna.

**Soluzione:** spezzare in più istruzioni. Il compilatore ottimizza da solo — zero perdite in performance, massimo guadagno in leggibilità e correttezza.

**Nota**

A un certo punto la prof ha tirato fuori la tabella degli allineamenti di *Dungeons & Dragons* per mostrare le $3 \times 3$ combinazioni possibili di pre/post-incremento con vari operatori. (Legale Buono, Neutrale Buono, Caotico Malvagio... nella lezione di informatica. Sì.) Guarda le slides per la tabella completa dei personaggi del Signore degli Anelli.

## 5. Conversioni di Tipo negli Assegnamenti

### 5.1. Caso 1: la variabile è più "espressiva" → Promozione

```
float a = 5 * 2;    // risultato intero 10 promosso a float: 10.0  (ok
    )
```

### 5.2. Caso 2: la variabile è meno "espressiva" → Perdita di informazione

```
int a = 5 * 2.0;    // risultato float 10.0 troncato a int: 10  (
    attenzione!)
```

**Attenzione**

Se il troncamento è *intenzionale*, usare un cast esplicito: `int a = (int)(5 * 2.0);` Così è chiaro che la conversione è voluta e la perdita di informazione è consapevole.

### 5.3. Esempio: Celsius → Fahrenheit

```c
int main() {
    float c = 18.0;

    /* Vale l'uguaglianza: F = C * 9/5 + 32 */

    float f = 32 + c * 9 / 5;
    printf("La temperatura in Fahrenheit e': %f\n", f);
    return 0;
}
```

**Attenzione**

**Occhio all'ordine delle operazioni:** `c * 9 / 5` funziona perché `c` è `float` — la moltiplicazione produce un float e la divisione è float. Se avessi scritto `9/5` da soli (divisione intera), il risultato sarebbe `1` invece di `1.8`.

## 6. Takeaways della Lezione

- ▷ **Parentesi:** le nostre migliori amiche. Non sempre indispensabili, ma rendono esplicito il comportamento voluto e aumentano la leggibilità.
- ▷ **Confronti e logici** restituiscono `0` o `1` (tipo `int`).
- ▷ **Short-circuit:** usare `&&` e `||` per proteggere accessi e calcoli (puntatori null, divisioni per zero).
- ▷ **Signed/unsigned:** controllare promozioni e conversioni. Confronti e sottrazioni possono sorprendere.
- ▷ **Overflow signed** è **Undefined Behavior**. Overflow unsigned è wrap modulo $2^N$ (deterministico, ma non è detto sia quello che si vuole).
- ▷ **Evitare side effects multipli** sulla stessa variabile nella stessa statement. Spezzare in più istruzioni.
- ▷ **Compilare con i warning:** `-Wall -Wextra` scopre UB e bug nascosti prima che esplodano a runtime.

## 7. MISRA-C (nota a margine)

**MISRA-C** (Motor Industry Software Reliability Association) è un insieme di regole per scrivere codice C **sicuro, affidabile e portabile**. Nato per l'industria automobilistica, adottato anche in ambito aerospaziale, ferroviario e medicale.

L'idea di fondo è la stessa dei principi visti oggi: eliminare comportamenti ambigui e costrutti pericolosi *prima* ancora che il compilatore dica niente.