Capitolo 4

# Espressioni

# Operatori

- In C il concetto di espressione è fondamentale
- Le espressioni vengono costruite con le variabili, le costanti e gli operatori
- Il C offre molti operatori:
  - Operatori aritmetici
  - Operatori relazionali
  - Operatori logici
  - Operatori di assegnamento
  - Operatori di incremento e decremento

  ed diversi altri

# Operatori aritmetici

- Il C fornisce 5 ***operatori aritmetici binari*** (richiedono 2 operandi)
  - `+` addizione
  - `-` sottrazione
  - `*` moltiplicazione
  - `/` divisione
  - `%` resto

- Inoltre esistono 2 operatori aritmetici ***unari***
  - `+` addizione (unaria)
  - `-` sottrazione (unaria)

# Operatori Aritmetici Unari

- Un operatore unario richiede un solo operando

```
i = +1;
j = -i;
```

- L'operatore unario + sostanzialmente non dà nulla. Viene usato solo per enfatizzare che una costante è positiva
- L'operatore unario − cambia il segno dell'operando.

# Operatori Aritmetici Binari

- Il valore di `i % j` è il resto della divisione `i / j`
  - Il valore dell'espressione `10 % 3` è 1
  - Il valore dell'espressione `12 % 4` è 0

- Gli operatori aritmetici binari—con l'eccezione di `%`—permettono operandi sia interi che floating-point, anche misti

- Quando un operando è `int` e l'altro è `float`, il risultato è di tipo `float`.
  - Il valore dell'espressione `9 + 2.5f` è 11.5
  - Il valore dell'espressione `6.7f / 2` è 3.35.

# Gli operatori `/` and `%`

- Gli operatori `/` e `%` richiedono particolare attenzione:
  - Quando entrambi gli operandi sono `int`, l'operatore `/` "tronca" il risultato (perchè deve restituire un `int`)
  - Il valore dell'espressione `1 / 2` è 0, non 0.5.
  - L'operatore `%` richiede **due operandi interi**; altrimenti si ha un errore in fase di compilazione
  - Se si usa 0 come operando destro sia di `/` che di `%` si ottiene un comportamento indefinito
  - Quando si usano operandi negativi con `/` e `%` il risultato è ***dipendente dall'implementazione*** in C89.

# Comportamento dipendente dall'implementazione

- Lo standard C (volutamente) lascia alcuni aspetti senza specifica
- Fa parte dell'efficienza del C
  - Spesso in questi casi il compilatore fa ciò che è più veloce nella particolare implementazione (che dipende dalla macchina)
- È meglio evitare di scrivere programmi che dipendono da una particolare implementazione

# Precedenza fra gli operatori

- L'espressione `i + j * k` significa
  - "somma `i` a `j`, e moltiplica il risultato per `k`" oppure
  - "moltiplica `j` e `k`, ed al risultato somma `i`"?

- Per evitare ambiguità si possono usare le parentesi
  - scrivendo `(i + j) * k` oppure `i + (j * k)`.

- Se le parentesi non sono esplicitate, il C usa le usuali regole di ***precedenza fra gli operatori*** per determinare il valore dell'espressione

# Precedenza fra gli operatori

- Gli operatori aritmetici hanno la seguente precedenza relativa:

  Precedenza massima: `+` `-` (unari)
  `*` `/` `%`
  Precedenza minima: `+` `-` (binari)

- Esempi:

  `i + j * k` è equivalente a `i + (j * k)`
  `-i * -j` è equivalente a `(-i) * (-j)`
  `+i + j / k` è equivalente a `(+i) + (j / k)`

# Associatività degli operatori

- L'***associatività*** entra in gioco quando due o più operatori hanno la stessa precedenza
- Un operatore viene detto ***associativo a sinistra*** se raggruppa gli operandi da sinistra (a destra)
- Gli operatori aritmetici binari (`*`, `/`, `%`, `+`, e `-`) sono tutti associativi a sinistra

- Esempi

  `i - j - k` è equivalente a `(i - j) - k`

  `i * j / k` è equivalente a `(i * j) / k`

# Associatività degli operatori

- Un operatore è ***associativo a destra*** se raggruppa gli operandi da destra (a sinistra).
- Gli operatori aritmetici unari (+ e −) sono entrambi associativi a destra

  `- + i` è equivalente a `-(+i)`

# Programma: Cifra di controllo UPC

- Moltissima merce viene riconosciuta da sistemi automatici grazie ad un codice UPC (Universal Product Code)


- Significato del codice:
  - Cifra a sinistra : tipo di prodotto
  - Primo gruppo di 5 cifre: Produttore
  - Secondo gruppo di 5 cifre: Prodotto
  - Cifra a destra: cifra di controllo errori (nella lettura)

# Programma: Cifra di controllo UPC

- La cifra di controllo si calcola così:

1. Somma la 1ª, la 3ª, la 5ª, la 7ª, la 9ª e l'11ª cifra
2. Somma la 2ª, la 4ª, la 6ª, l' 8ª e la 10ª cifra
3. Moltiplica la prima somma per 3 e addiziona al risultato la seconda somma
4. Sottrai 1 dal totale
5. Calcola il resto della divisione per 10 del numero ottenuto al passo precedente
6. Sottrai da 9 tale resto

# Programma: Cifra di controllo UPC

- Codice UPC dell'esempio: 0 13800 15173 5

1. Prima somma: 0 + 3 + 0 + 1 + 1 + 3 = 8.
2. Seconda somma: 1 + 8 + 0 + 5 + 7 = 21.
3. Multiplicando la prima somma per 3 e sommando la seconda somma si ottiene 45.
4. Sottraendo 1 si ottiene 44.
5. Il resto della divisione di 44 per 10 è 4.
6. Sottraendo 4 da 9 si ottiene 5.
7. La cifra a destra (cifra di controllo) è infatti 5.

# Programma: Cifra di controllo UPC

- Il programma `upc.c` chiede all'utente di digitare le prime 11 cifre di un codice UPC, dopodiché visualizza la cifra di controllo:

```
Enter the first (single) digit: 0
Enter first group of five digits: 13800
Enter second group of five digits: 15173
Check digit: 5
```

- Il programma chiede all'utente di inserire separatamente la cifra a destra ed i due gruppi di 5 cifre
- Per leggere la cifra singola, useremo `scanf` con la specifica di conversione `%1d`

# upc.c

```
/* Computes a Universal Product Code check digit */

#include <stdio.h>

int main(void)
{
  int d, i1, i2, i3, i4, i5, j1, j2, j3, j4, j5,
      first_sum, second_sum, total;

  printf("Enter the first (single) digit: ");
  scanf("%1d", &d);
  printf("Enter first group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &i1, &i2, &i3, &i4, &i5);
  printf("Enter second group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &j1, &j2, &j3, &j4, &j5);
  first_sum = d + i2 + i4 + j1 + j3 + j5;
  second_sum = i1 + i3 + i5 + j2 + j4;
  total = 3 * first_sum + second_sum;

  printf("Check digit: %d\n", 9 - ((total - 1) % 10));

  return 0;
}
```

# Operatori di assegnamento

- ***Assegnamento semplice:*** si memorizza un valore in una variabile
- ***Assegnamento composto:*** si aggiorna il valore memorizzato in una variabile

# Assegnamento semplice

- L'effetto dell'assegnamento semplice $v = e$ è quello di valutare l'espressione $e$ e copiare il suo valore nella variabile $v$.
- $e$ può essere una costante, una variabile o un'espressione più complessa:

```
i = 5;                  /* i is now 5   */
j = i;                  /* j is now 5   */
k = 10 * i + j;         /* k is now 55  */
```

# Assegnamento semplice

- Se *v* ed *e* non hanno lo stesso tipo, allora il valore di *e* viene convertito al tipo di *v* prima di memorizzare il valore nella variabile

```
int i;
float f;

i = 72.99f;    /* i is now 72 */
f = 136;       /* f is now 136.0f */
```

# Assegnamento semplice

- In molti linguaggi di programmazione, l'assegnamento è un'istruzione.
- In C l'assegnamento è un operatore, proprio come l'operatore di addizione +.
- Il valore dell'assegnamento $v = e$ è il valore di $v$ dopo l'assegnamento

- Esempio:
  - Il valore di `i = 72.99f` è 72 (non 72.99).
  - Il valore di `(i = 72.99f)+1` è ?

# Effetti collaterali (Side Effects)

- Quando un operatore modifica uno dei suoi operandi si dice che ha degli "effetti collaterali" (***side effect).***
- L'assegnamento semplice ha l'effetto collaterale di modificare il valore dell'operando sinistro, la variabile che si sta assegnando
- Nella valutazione dell'espressione `i = 0` produce il risultato 0 e—come effetto collaterale—assegna 0 ad `i`.

# Assegnamento

- Poichè l'assegnamento è un operatore, vari assegnamenti possono essere concatenati:

- Esempio

```
i = j = k = 0;
```

- L'operatore = è associativo a destra, quindi l'assegnamento dell'esempio equivale a:

```
i = (j = (k = 0));
```

# Assegnamento

- Attenzione a risultati inattesi negli assegnamenti a catena dovuti a conversioni di tipo:

```
int i;
float f;

f = i = 33.3f;
```

- Ad `i` viene assegnato il valore 33 (perchè `i` è un intero), e quindi ad `f` viene assegnato 33.0 (non 33.3).

# Lvalues (Left values)

- L'operatore di assegnamento richiede come operando sinistro un ***lvalue*** (left value)

- Un lvalue rappresenta un oggetto (variabile) memorizzato nella memoria del computer
  - Quindi non può essere una costante o il risultato di un'espressione

- Le variabili sono lvalue
  - Espressioni quali `10` o `2 * i` non lo sono

# Lvalues

- Poichè l'operatore di assegnamento richiede un lvalue come operando a sinistra, non è permesso mettere nessun'altra cosa sulla sinistra dell'operatore di assegnamento:

```
12 = i;          /*** WRONG ***/
i + j = 0;       /*** WRONG ***/
-i = j;          /*** WRONG ***/
```

- In tali casi il compilatore produrrà un messaggio di errore del tipo: *"invalid lvalue in assignment."*

# Assegnamento composto

- Assegnamenti che usano il "vecchio" valore di una variabile per calcolare un nuovo valore sono molto comuni:

- Esempio:

```
i = i + 2;
```

- L'assegnamento composto += permette di non ripetere il nome della variabile da aggiornare:

```
i += 2;    /* same as i = i + 2; */
```

# Assegnamento composto

- Ci sono 9 altri operatori di assegnamento composto che includono i seguenti:

  `-=` `*=` `/=` `%=`

- Tutti gli operatori di assegnamento composto funzionano più o meno allo stesso modo:

  *v* += *e* somma *v* ad *e*, e memorizza il valore in *v*

  *v* -= *e* sottrae *e* da *v*, e memorizza il valore in *v*

  *v* *= *e* moltiplica *v* per *e*, e memorizza il valore in *v*

  *v* /= *e* divide *v* per *e*, e memorizza il valore in *v*

  *v* %= *e* calcola il resto di *v*/*e*, e memorizza il valore in *v*

# Assegnamento composto

- Attenti all'ordine dei due caratteri che compongono l'operatore
  - L'uguale va dopo *v* += *e* e non *v* =+ *e*

- Sebbene `i =+ j` non dà errori di compilazione essa è equivalente a `i = +j` cioè a `i = (+j)`, che memorizza il valore di `j` in `i`.

# Operatori di incremento e decremento

- Fra le operazioni più comuni troviamo l'incremento (aumentare il valore di 1) ed il decremento (diminuire il valore di 1):

```
i = i + 1;
j = j - 1;
```

- Possiamo usare gli operatori di assegnamento composto:

```
i += 1;
j -= 1;
```

# Operatori di incremento e decremento

- Il C fornisce due operatori appositi:
  `++` (***incremento***)
  `--` (***decremento***)
- L'operatore `++` addiziona 1 al suo operando
- L'operatore `--` sottrae 1 al suo operando

- Il loro uso può essere subdolo:
  - Possono essere usati in forma ***prefissa*** (`++i`, `--i`) oppure ***postfissa*** (`i++`, `i--`).
  - Hanno effetti collaterali (modificano il valore dell'operando)

# Operatori di incremento e decremento

- La valutazione dell'espressione `++i` (incremento "prefisso") produce il valore `i + 1` e—come effetto collaterale—assegna tale valore a `i` (incrementa `i`):

```
i = 1;
printf("i is %d\n", ++i);    /* prints "i is 2" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

- La valutazione dell'espressione `i++` (incremento "postfisso") produce come risultato `i`, ma assegna il valore `i+1` ad `i`:

```
i = 1;
printf("i is %d\n", i++);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

# Operatori di incremento e decremento

- `++i` significa "incrementa `i` subito," mentre `i++` significa "usa il (vecchio) valore di `i` per valutare l'espressione e poi incrementa `i`"
- Dopo, quando?
- Lo standard C non specifica esattamente quando, ma si può assumere che `i` verrà incrementata prima di eseguire la prossima istruzione del programma

# Operatori di incremento e decremento

- Esempi:

```
i = 1;
printf("i is %d\n", --i);   /* prints "i is 0" */
printf("i is %d\n", i);     /* prints "i is 0" */

i = 1;
printf("i is %d\n", i--);   /* prints "i is 1" */
printf("i is %d\n", i);     /* prints "i is 0" */
```

# Valutazione di espressioni

- Operatori discussi finora

| *Precedenza* | *Nome* | *Simbolo(i)* | *Associatività* |
|---|---|---|---|
| 1 | incremento (postfisso) | `++` | sinistra |
| | decremento (postfisso) | `--` | |
| 2 | incremento (prefisso) | `++` | destra |
| | decremento (prefisso) | `--` | |
| | addizione unaria | `+` | |
| | sottrazione unaria | `-` | |
| 3 | multiplicazione | `* / %` | sinistra |
| 4 | addizione | `+ -` | sinistra |
| 5 | assegnamento | `= *= /= %= += -=` | destra |

# Valutazione di espressioni

- La tavola può essere usata per esplicitare le parentesi:
  - Iniziando dall'operatore con la precedenza massima, mettere le parentesi all'operatore ed ai suoi operandi

- Esempio:

```
a = b += c++ - d + --e / -f
```

| | Precedenza |
|---|---|
| `a = b += (c++) - d + (--e) / -f` | 1 |
| `a = b += (c++) - d + (--e) / (-f)` | 2 |
| `a = b += (c++) - d + ((--e) / (-f))` | 3 |
| `a = b += (((c++) - d) + ((--e) / (-f)))` | 4 |
| `(a = (b += (((c++) - d) + ((--e) / (-f)))))` | 5 |

# Ordine di valutazione delle sottoespressioni

- Il valore di un'espressione può dipendere dall'ordine in cui si valutano le sottoespressioni
- Il C non definisce in che ordine le sottoespressioni devono essere valutate
  - Con l'eccezione delle sottoespressioni che coinvolgono gli operatori logici e quelli condizionali
- Nell'espressione `(a + b) * (c - d)` non sappiamo se `(a + b)` sarà valutato prima o dopo di `(c - d)`.

# Ordine di valutazione delle sottoespressioni

- Molte espressioni hanno lo stesso valore indipendentemente dall'ordine in cui si valutano le sottoespressioni
- Tuttavia, ciò potrebbe non essere vero quando le sottoespressioni modificano uno degli operandi coinvolti

```
a = 5;
c = (b = a + 2) - (a = 1);
```

- L'effetto della valutazione della seconda espressione non è definito.

# Ordine di valutazione delle sottoespressioni

- Cosa deve fare il programmatore
  - Evitare di scrivere espressioni che usano il valore di una variabile e modificano la stessa variabile in altre parti dell'espressione

- Alcuni compilatori possono produrre errore (del tipo *"operation on 'a' may be undefined"*) quando trovano tali espressioni

# Ordine di valutazione delle sottoespressioni

- Esempio:

```
i = 2;
j = i * i++;
```

- È naturale assumere che a `j` venga assegnato 4. Tuttavia, `j` potrebbe anche valere 6:

1. Il secondo operando (il "vecchio" valore `i`), cioè 2, viene letto dalla memoria, e poi `i` viene incrementato.
2. Il primo operando (il nuovo valore di `i`), cioè 3, è letto dalla memoria.
3. I valori 2 e 3 vengono moltiplicati e si ottiene 6

# Espressioni ed istruzioni

- Il C ha l'inusuale regola che un'espressione può essere usata come un'istruzione

- Esempio:

```
++i;
```

`i` viene incrementato, il nuovo valore viene letto dalla memoria ma poi non viene utilizzato

# Espressioni ed istruzioni

- Poichè il valore non viene utilizzato, non ha molto senso utilizzare un'espressione come istruzione, a meno che l'espressione non abbia un effetto collaterale:

```
i = 1;          /* useful */
i--;            /* useful */
i * j - 1;      /* not useful */
```

# Espressioni ed istruzioni

- Un errore di digitazione può facilmente creare istruzioni inutili
- Ad esempio, se al posto di scrivere

```
i = j;
```

per errore scriviamo

```
i + j;
```

creiamo un'istruzione inutile

- Alcuni compilatori si accorgono di tali istruzione e generano un avvertimento del tipo *"statement with no effect."*

# … arrivederci alla prossima lezione