

# **Lezione 6–Programmazione in Linguaggio C**

## **Corso — Programmazione**

Programmazione imperativa – Linguaggio C

**Marco Anisetti, Stefano Ferrari**

e-mail: [marco.anisetti@unimi.it](mailto:marco.anisetti@unimi.it)

web: <http://homes.di.unimi.it/anisetti/>

# Scanf... again

- Specificatore di formato:

`%[*][width][modifiers]<type>`

*[] indicano una cosa opzionale*

- **\*** Serve per ignorare quello che viene letto nel formato che segue
- **width** numero di caratteri da leggere
- **modifiers** modificatore di dimensioni: h (con d è short int) l (con d long int) L (con f long double)
- **Type** come solito d, c, e, f, g, s(stringa), u(unsigned int), o (intero ottale), x (intero esadecimale)

# Scanf... again

L'input da tastiere resta in un buffer fino a che qualcuno non lo rimuove



scanf cerca nel buffer **un pattern** (numeri decimali, numeri floating point, caratteri ecc) e si prende quello che riconosce

elimina di solito gli spazi bianchi prima

Pattern: es leggere una data (gg/mm/aaaa):

```
scanf("%2d/%2d/%4d", &giorno, &mese, &anno);
```

**Fare esperimenti**

# Esempio

```
scanf("%lf %c %lf", &a, &b, &c);
```

Come interpreta l'input?

```
scanf("test");
```

```
scanf("test",&a);
```

Funzionano?

# Grammatica dello scanf

Proviamo a pensare ad una ipotetica grammatica usata  
dallo scanf per interpretare la stringa

# Grammatica dello scanf

Proviamo a pensare ad una ipotetica grammatica usata dallo scanf per interpretare la stringa

```
<parametro> ::= <stringa> | <stringa> <specificatore> | <specificatore>
<stringa> ::= <carattere> | <stringa> <carattere>
<carattere> ::= "a" | "b" | ... | "/" | "%" | "..." |
...
<specificatore> ::= "%" <type> | "%" <modifier> <type> | "%" <width> <modifier> <type> |
%"* <type> | "%" * <modifier> <type> | "%" * <width> <modifier> <type>
<type> ::= "d" | "f" | "c" | ...
```

# Approfondimenti sul costrutto di selezione

- Operatore ternario (unico nel C)

*(espressione1 ? espressione2 : espressione3 )*

- Costituito dai due simboli ? e : che separano i tre operandi
- Si valuta espressione1 che è di tipo logico
  - se espressione1 è vera, si valuta espressione2
  - se espressione1 è falsa, si valuta espressione3
- Si assegna all'espressione composta il valore dell'espressione valutata
- Posso assegnare il valore dell'operatore ternario ad una variabile

*v=(espressione1 ? espressione2 : espressione3 )*

# Else pendente

- Ogni else si riferisce all'if più vicino non ancora accoppiato
- se le parentesi graffe non indicano altrimenti
- Questo può facilmente indurre in errore: meglio usare sempre le parentesi

```
if (y != 0)
    if (x != 0)
        r = x / y;
else
    printf("Errore!");
```

# Selezione multipla: switch (i)

- L'istruzione *switch* consente di separare blocchi alternativi che corrispondono ai singoli valori possibili per un'espressione

```
switch (espressione)
{
    case costante1 : istruzione/i
    case costante2 : istruzione/i
    ...
    default: istruzione/i
}
```

- Il blocco default corrisponde ai casi non elencati esplicitamente, cioè all'ultimo else
- Il blocco default può mancare (come l'ultimo else)

## Selezione multipla: switch (ii)

- Si valuta l'espressione
- Si cerca la costante di valore uguale all'espressione
  - Se c'è, si eseguono le istruzioni associate alla costante
  - e quelle associate a tutte le costanti successive
  - Altrimenti si eseguono le istruzioni associate a default
- **Per eseguire le istruzioni associate a una sola costante, occorre terminarle con l'istruzione break;**
- Il costrutto switch viola (in teoria) la programmazione strutturata, perché in generale non ha un solo punto di ingresso e un solo punto di uscita:
  - ogni case è un diverso punto di entrata
  - ogni istruzione break è un diverso punto di uscita

# Esercizio Switch

Supponiamo di dover scrivere un programma che riceve in ingresso il numero di tentativi di accesso da parte di un certo utente che sono stati bloccati dal sistema. Il programma deve:

nel caso sia il **terzo tentativo** i) allertare admin, ii) bloccare l'account, iii) loggare;

nel caso sia il **secondo tentativo** solo i) bloccare l'account e ii) loggare;

nel caso sia il primo tentativo solo loggare.

Simulare le attività (allertare admin, bloccare account, loggare) con un printf (es printf("allertare per l'admin\n");)

# Esercizio Switch

Supponiamo di dover scrivere un programma che riceve in ingresso il numero di tentativi di accesso da parte di un certo utente che sono stati bloccati dal sistema. Il programma deve, nel caso sia il **terzo tentativo** i) allertare admin, ii) bloccare l'account, iii) loggare, nel caso sia il **secondo tentativo** solo i) bloccare l'account, ii) loggare, nel caso sia il primo tentativo solo loggare.

Simulare le attività (allertare admin, bloccare account, loggare) con un printf (es printf("allertare per l'admin\n");)

**Consiglio: non usiamo il break nello switch!**

# Ciclo for

- Un ciclo per cui ho un numero fissato di iterazioni note a priori
  - Il while si usa quando non si sa a priori quante volte si ciclerà
- E' molto usato dai programmatori C

*for (<espr-iniz>; <espr-cond>; <espr-incr>)  
  <istruzione>*

- *Equivale (logicamente ma non di fatto) a*

```
espr-iniz;  
while (espr-cond)  
{  
  istruzione  
  espr-incr;  
}
```

# Ciclo for

- Un sottoinsieme qualunque delle tre espressioni si può omettere. Se *espr-cond* è omessa, essa è assunta sempre vera.
- Quindi, `for(;;)`; equivale a un ciclo infinito come `while(1);`
- L'espressione *espr-iniz* è di normale assegnamento che inizializza una variabile indice.
- L'espressione *espr-incr* è spesso un incremento o un decremento della variabile indice.
  - Molto frequentemente un post incremento

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

**Nota:** *int i = 0* dichiarata e inizializzata nel for (*espr-iniz*) ha scope solo li non visibile dopo il for

# Ciclo For

Negli standard precedenti era obbligatorio fare

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

Cosa cambia?

# For senza istruzioni

- Se non servono istruzioni nel ciclo
- Possono comunque aver luogo nell'inizializzazione e aggiornamento (incremento)
- `for (espressione1;espressione2;espressione3) ;`
- Somma dei primi n numeri

```
int c=0;  
for (int d = 1 ; (d <= n ); c+=d++);  
return (c);
```

# Operatore virgola

- Modificare più variabili in una sola istruzione, volendo però evitare effetti collaterali multipli poco controllabili
- L'operatore virgola combina più espressioni qualsiasi in un'istruzione sola

espressioneA, espressioneB, espressioneC

- Associativo da sinistra a destra:
  - si valutano le espressioni da sinistra a destra
  - **il suo valore è il valore dell'ultima espressione**
  - ha la priorità più bassa (non occorrono parentesi)
- Si usa con espressioni che hanno effetti collaterali raggruppandole in una sola istruzione leggibile
- Es. Inizializzazioni e aggiornamenti multipli nei cicli for

## Esempi (i)

- Partiamo da questo codice

```
int i, q;  
for (i = 1; i <= n; i++)  
{ q = i * i;  
printf("%d",q);  
printf("\n"); }
```

- Se modifco così cosa succede?

```
for (i = 1, q = i * i; i <= n; i++)  
{printf("%d",q); printf("\n"); }
```

- E con questo cambio?

```
for (q = i * i, i = 1; i <= n; i++)  
{printf("%d",q); printf("\n"); }
```

## Esempi (i)

- Partiamo da questo codice

```
int i, q;  
for (i = 1; i <= n; i++)  
{ q = i * i;  
printf("%d",q);  
printf("\n"); }
```

- Se modifco così cosa succede?

```
for (i = 1, q = i * i; i <= n; i++)  
{printf("%d",q); printf("\n"); }
```

- E con questo cambio?

```
for (q = i * i, i = 1; i <= n; i++)  
{printf("%d",q); printf("\n"); }
```

- In tutte e due le situazioni il quadrato si calcola solo una volta

## Esempi (ii)

- Se modifco così cosa succede?

```
for (i = 1; i <= n; q = i * i, i++)  
{printf("%d",q); printf("\n"); }
```

- E con questo cambio?

```
for (i = 1; i <= n; i++, q = i * i)  
{printf("%d",q); printf("\n"); }
```

## Esempi (ii)

- Se modifco così cosa succede?

```
for (i = 1; i <= n; q = i * i, i++)  
{printf("%d",q); printf("\n"); }
```

*Il primo quadrato è sbagliato, gli altri ritardati di un passo*

- E con questo cambio?

```
for (i = 1; i <= n; i++, q = i * i)  
{printf("%d",q); printf("\n"); }
```

*Il primo quadrato è sbagliato, gli altri giusti e sincronizzati*

- Versione corretta

```
for (i = 1, q = 1; i <= n; i++, q = i * i)  
{printf("%d",q); printf("\n"); }
```

## Esempi (iii)

- Vediamo altri effetti dell'operatore virgola

$(i = 1), j = 2, k = i + j;$

i = 1;

j = 5;

++i, i+j;

i = 1,2,3;

i = (1,2,3);

## Esempi (iii)

- Vediamo altri effetti dell'operatore virgola

$(i = 1), j = 2, k = i + j;$

L'espressione vale 3, i vale 1, j vale 2, k vale 3

i = 1;

j = 5;

++i, i+j;

L'espressione vale 7, i vale 2, j vale 5

i = 1,2,3;

L'espressione vale 3, i vale 1

i = (1,2,3);

Al termine, l'espressione vale 3, i vale 3

# Istruzione break

- In C si può uscire da un ciclo in punti diversi dalla condizione di permanenza (**ciò e contrario alla programmazione strutturata**)
- L'istruzione *break* provoca l'uscita dal blocco corrente (nei blocchi switch, while, do-while e for)
- Valuta se n è composto o primo (d sta per \divisore")

```
for (d = 2 ; d < n ; d++)  
if (n % d == 0) break;
```

```
return ( (d < n) ? 0 : 1 );
```

# Istruzione break

- In C si può uscire da un ciclo in punti diversi dalla condizione di permanenza (**ciò e contrario alla programmazione strutturata**)
- L'istruzione *break* provoca l'uscita dal blocco corrente (nei blocchi switch, while, do-while e for)
- Valuta se n è composto o primo (d sta per \"divisore")

```
for (d = 2 ; d < n ; d++)  
if (n % d == 0) break;
```

```
return ( (d < n) ? 0 : 1 );
```

Si può darne una versione strutturata complicando la condizione di permanenza?

# Istruzione break

- In C si può uscire da un ciclo in punti diversi dalla condizione di permanenza (**ciò e contrario alla programmazione strutturata**)
- L'istruzione *break* provoca l'uscita dal blocco corrente (nei blocchi switch, while, do-while e for)
- Valuta se n è composto o primo (d sta per \"divisore")

```
for (d = 2 ; d < n ; d++)  
if (n % d == 0) break;
```

```
return ( (d < n) ? 0 : 1 );
```

Si può darne una versione strutturata complicando la condizione di permanenza?

```
for (d = 2 ; (n % d != 0) && (d < n) ; d++);
```

# Istruzione continue

- L'istruzione **continue** provoca il passaggio all'iterazione successiva, ignorando il resto del corpo
- Sostituibile attraverso un if con la condizione negata
- Per la leggibilità il continue andrebbe evitato**
- Somma i quadrati dei numeri dispari compresi fra 1 e n

```
s = 0;  
for (i = 1 ; i <= n ; i++)  
{  
    if (i % 2 == 0)  
        continue;  
    s += i*i;  
}
```

```
s = 0;  
for (i = 1 ; i <= n ; i++)  
{  
    if (i % 2 != 0)  
    {  
        s += i*i;  
    }  
}
```

# Riflessioni su break e continue

L'uso di questi due istruzioni seppur ammesso rendono il flusso meno semplice da controllare

Il break viola la programmazione strutturata quando usato nei cicli, è indispensabile negli switch

Il continue rende difficile avere degli invarianti di ciclo

# Esercizio

- **Distanza Euclidea:** Chiedere due coordinate e calcolarne la distanza euclidea
  - Usare i tipi double
  - double sqrt(double) è la funzione che calcola la radice quadrata dato in ingresso un double (includere math.h)

# Esercizio

- Estendere il programma della distanza Euclidea facendo in modo che l'utente possa iterare il calcolo della distanza. Una volta finito il primo calcolo viene chiesto se si vuole continuare o chiudere il programma. Se si vuole continuare si richiede di nuovo l'inserimento dei punti

# Tipi definiti da utente(1)

*typedef <tipo esistente> <tipo nuovo>;*

Si può dare un nuovo nome ai tipi di dato esistenti (predefiniti) allo scopo di chiarire il significato di un dato e facilitarne le modifiche

*typedef int anno;  
anno a; a = 2010;*

*typedef int boolean;  
boolean b; b = TRUE; // deve essere definito TRUE*

*typedef anno annobisestile;  
annobisestile ab; ab = 2000;*

## Tipi definiti da utente(2)

- La definizione di tipo se inserita in un blocco vale solo nel blocco cui appartiene (parte dichiarativa del blocco)
- Se inserita fra le direttive (fuori da tutto) vale per l'intero file (dichiarazioni globali)
- La definizione di tipo termina con ; come tutte le istruzioni
- Effetto: **Aggiunge un nuovo nome alla tabella dei tipi di dato**
- Esiste anche la tabella dei simboli a cui vengono aggiunti gli identificatori delle dichiarazioni e l'indirizzo di dove possono essere reperite

# Tipi enumerativi

- Tipo enumerativo è un tipo definito da utente con un numero finito di valori, esplicitamente enumerati nella dichiarazione
- Tipicamente, serve a definire valori simbolici
- Una variabile di tipo enumerativo si dichiara specificando
  - l'elenco dei valori possibili
  - il nome della variabile

*enum [<nomeEnum>] {valore1, valore2,...} <variabile>;*

Es. enum {PICCHE, CUORI, QUADRI, FIORI} s;

- E' sempre *<tipo> <variabile>;*" ma il tipo è composto

# Tipi enumerativi

Il C tratta le espressioni di tipo enumerativo come interi,  
Ovvero assegna agli elementi del tipo i valori 0, 1, 2,...

*enum {PICCHE, CUORI, QUADRI, FIORI} s1, s2;  
equivale a*

```
#define PICCHE 0  
#define CUORI 1  
#define QUADRI 2  
#define FIORI 3  
int s1, s2;
```

- Il precompilatore non sostituisce le costanti con i loro valori
- I valori simbolici sono locali al blocco che li dichiara
- Non si può usare lo stesso nome due volte nello stesso blocco:

```
enum { PICCHE, CUORI, QUADRI, FIORI } s1, s2;  
enum { FIORI, CIOCCOLATINI } regalo;  
• darebbe errore di duplicazione
```

# Tipi enumerativi

Si possono assegnare esplicitamente i valori alle costanti

*enum {costante1=valore1, costante2=valore2,...} variabile;*

*enum {TRUE = 1, FALSE = 0} b;*

- enumerativi e interi si convertono banalmente gli uni negli altri
- si possono applicare gli operatori interi ai tipi enumerativi
- si possono usare i valori enumerativi come indici
  - Es nei cicli e nello switch
- Ovviamente, questo comporta un rischio di overflow non segnalati

**Nota:** se il nome viene riutilizzato in una definizione in un blocco più interno, il valore associato è l'ultimo attribuito.

# Dichiarazione di tipi enumerativi

- Ripetere una dichiarazione di tipo enumerativo per ogni variabile

*enum {FALSE, TRUE} b1;*

...

*enum {FALSE, TRUE} b2;*

- Non compatto (errore se nello stesso blocco)
- Meglio dare un nome simbolico al tipo (boolean) e separare la dichiarazione della variabile (b1, b2, . . . ) dalla dichiarazione del tipo (boolean)

## con l'istruzione **typedef**

*typedef enum {FALSE, TRUE} boolean;  
boolean b;*

## con i tag di enumerazione: usando [*<nomeEnum>*]

*enum boolean {FALSE, TRUE};  
enum boolean b;*

# Tipi enumerativi

- La scelta del tipo intero per implementare un tipo enum è lasciato al compilatore.
- Per esempio, se le costanti enumerative hanno valori negativi, GCC (compilatore) utilizza l'int, altrimenti l'unsigned int:

```
typedef enum {MENOUNO=-1, ZERO, UNO, DUE} intero;
typedef enum {ZERO_N, UNO_N, DUE_N} naturale;
intero a = MENOUNO;
naturale b = ZERO_N;
```

# Tipi enumerativi

- La scelta del tipo intero per implementare un tipo enum è lasciato al compilatore.
- Per esempio, se le costanti enumerative hanno valori negativi, GCC (compilatore) utilizza l'int, altrimenti l'unsigned int:

```
typedef enum {MENOUNO=-1, ZERO, UNO, DUE} intero;  
typedef enum {ZERO_N, UNO_N, DUE_N} naturale;  
intero a = MENOUNO;  
naturale b = ZERO_N;
```

Cosa valgono?

(a-1 >= MENOUNO)  
(b-1 >= ZERO\_N)

# Tipi enumerativi

- La scelta del tipo intero per implementare un tipo enum è lasciato al compilatore.
- Per esempio, se le costanti enumerative hanno valori negativi, GCC (compilatore) utilizza l'int, altrimenti l'unsigned int:

```
typedef enum {MENOUNO=-1, ZERO, UNO, DUE} intero;  
typedef enum {ZERO_N, UNO_N, DUE_N} naturale;  
intero a = MENOUNO;  
naturale b = ZERO_N;
```

Cosa valgono?

(a-1 >= MENOUNO) viene valutata 0.  
(b-1 >= ZERO\_N) viene valutata 1.

# Tipi enumerativi

- La scelta del tipo intero per implementare un tipo enum è lasciato al compilatore.
- Per esempio, se le costanti enumerative hanno valori negativi, GCC (compilatore) utilizza l'int, altrimenti l'unsigned int:

```
typedef enum {MENOUNO=-1, ZERO, UNO, DUE} intero;  
typedef enum {ZERO_N, UNO_N, DUE_N} naturale;  
intero a = MENOUNO;  
naturale b = ZERO_N;
```

Cosa valgono?

(a-1 >= MENOUNO) viene valutata 0.

(b-1 >= ZERO\_N) viene valutata 1.

Overflow! (unsigned int!!) ... dipende dal compilatore

# Tipi enumerativi

Consigliabile porre una costante che indichi un valore non accettabile (possibilmente come prima costante numerativa).

Esempio:

```
for (b = DUE_N; b >= ZERO_N; b--)
```

*Funziona?*

# Operatore sizeof

*sizeof(<tipo>); sizeof(<espressione>);*

- Il valore dell'espressione `sizeof(...)` è lo spazio occupato
  - da oggetti del tipo `<tipo>`
  - dall'espressione `<espressione>`
- Il valore è di tipo `size_t` (numero naturale)
  - Nota: `size_t` sarà definito con un `typedef` da qualche parte (nei file .h inclusi)
- Di solito viene usato solo per allocare memoria dinamica

## Esempi enum

```
enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
```

...

```
enum week day;  
day = Wed;  
printf("%d",day);
```

Cosa stampa?

```
enum year{Jan, Feb, Mar, Apr, May, Jun, Jul,  
          Aug, Sep, Oct, Nov, Dec};
```

```
int i;  
for (i=Jan; i<=Dec;)  
    printf("%d ", i++);
```

Cosa stampa?

## Esempi enum

```
enum state {working, failed};  
enum result {failed, passed};
```

```
int main() { return 0; }
```

Cosa succede?

## Esempi enum

```
enum state {working, failed};  
enum result {failed, passed};
```

```
int main() { return 0; }
```

Cosa succede? **enum devono essere uniche in uno scope**

```
enum State {WORKING = 0, FAILED, FREEZED};  
enum State currState = 2;
```

```
int main() {  
    (currState == WORKING)? printf("WORKING"): printf("NOT  
WORKING");  
    return 0;  
}
```

**Cosa stampa?**