

Costrutti Iterativi e array

Laboratorio di Programmazione I

Corso di Laurea in Informatica
A.A. 2019/2020



Argomenti del Corso

Ogni lezione consta di una spiegazione assistita da slide, e seguita da esercizi in classe

- Introduzione all'ambiente Linux
- Introduzione al C
- Tipi primitivi e costrutti condizionali
- Costrutti iterativi ed array
- Funzioni, stack e visibilità variabili
- Puntatori e memoria
- Debugging
- Tipi di dati utente
- Liste concatenate e librerie
- Ricorsione

Sommario

- 1 Errori frequenti
- 2 Comandi iterativi
 - Cicli for
 - Cicli while
 - Cicli do-while
- 3 Array
 - Array
 - Array e cicli

Errori frequenti

- `int x; printf ("%d\n" x);`

Errori frequenti

- `int x; printf ("%d\n" x);`
- `int x; scanf ("%d\n", x);`

Errori frequenti

- `int x; printf ("%d\n" x);`
- `int x; scanf("%d\n", x);`
- `float x; scanf("%d", &x);`

Errori frequenti

- `int x; printf ("%d\n" x);`
- `int x; scanf("%d\n", x);`
- `float x; scanf("%d", &x);`
- `int x; scanf("%d\n", &x);`

Errori frequenti

- `int x; printf ("%d\n" x);`
- `int x; scanf("%d\n", x);`
- `float x; scanf("%d", &x);`
- `int x; scanf("%d\n", &x);`
- `printf (" Inserisci il valore :\n");`

Errori frequenti

- `int x; printf ("%d\n" x);`
- `int x; scanf("%d\n", x);`
- `float x; scanf("%d", &x);`
- `int x; scanf("%d\n", &x);`
- `printf (" Inserisci il valore :\n");`
- `int x; printf (" Il risultato e': %d\n", x);`

Errori frequenti

- `int x; printf ("%d\n" x);`
- `int x; scanf("%d\n", x);`
- `float x; scanf("%d", &x);`
- `int x; scanf("%d\n", &x);`
- `printf (" Inserisci il valore :\n");`
- `int x; printf (" Il risultato e': %d\n", x);`

Errori frequenti

- `int x; printf ("%d\n" x);`
- `int x; scanf("%d\n", x);`
- `float x; scanf("%d", &x);`
- `int x; scanf("%d\n", &x);`
- `printf (" Inserisci il valore :\n");`
- `int x; printf (" Il risultato e': %d\n", x);`

Consiglio: Quando la piattaforma vi da errore:

- Leggete l'errore.
- Controllate su che caso di test si verifica l'errore e cercate di capire perché.

Attenti ai Newline

- Il **newline** (e.s. quando premete invio) è un carattere con un suo codice ASCII

Attenti ai Newline

- Il **newline** (e.s. quando premete invio) è un carattere con un suo codice ASCII
- State attenti quando in un esercizio avete una `scanf("%c", ...)` dopo aver letto un numero da tastiera, es.
`scanf("%d", &n);`
`scanf("%c", &x);`

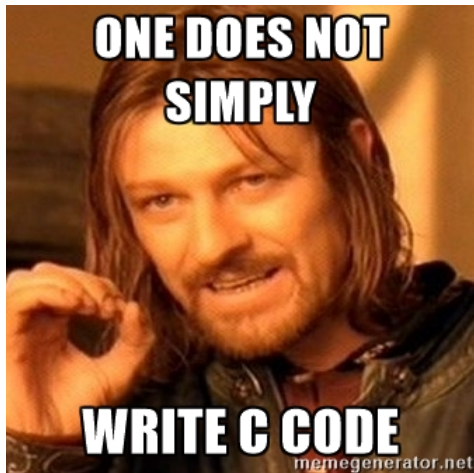
Attenti ai Newline

- Il **newline** (e.s. quando premete invio) è un carattere con un suo codice ASCII
- State attenti quando in un esercizio avete una `scanf("%c", ...)` dopo aver letto un numero da tastiera, es.
`scanf("%d", &n);`
`scanf("%c", &x);`
- La seconda `scanf` non leggerà il carattere da tastiera ma catturerà il newline che termina l'immissione del numero nella `scanf` precedente

Attenti ai Newline

- Il **newline** (e.s. quando premete invio) è un carattere con un suo codice ASCII
- State attenti quando in un esercizio avete una `scanf("%c", ...)` dopo aver letto un numero da tastiera, es.
`scanf("%d", &n);`
`scanf("%c", &x);`
- La seconda `scanf` non leggerà il carattere da tastiera ma catturerà il newline che termina l'immissione del numero nella `scanf` precedente
- Per non aver problemi inserite uno spazio prima del segnaposto del carattere in modo da catturare i caratteri whitespace (e.g. il newline, lo spazio, il tab)
`scanf(" %c", &x);`

Indentazione



Tre costrutti

- Eseguire un blocco di codice per più di una iterazione.
- Tre costrutti: **for**, **while**, e **do-while**

Regole d'oro delle buone pratiche di programmazione:

- **for**: **si sa in anticipo** il numero di iterazioni;
- **while**: **non si sa in anticipo** il numero di iterazioni;
- **do-while**: non si sa in anticipo il numero di iterazioni, ma sappiamo che deve essere eseguita **almeno una** iterazione.

Come vedremo, **for**, **while** e **do-while** hanno la stessa potenza espressiva.

Sintassi `for`

`for`: esegui un blocco di istruzioni per un numero di iterazioni pre-fissato. Sintassi:

```
for (espr1; guardia; espr2)
    blocco_istruzioni
```

Semantica:

- 1 Esegui **espr1**
- 2 Se la guardia è **falsa**, salta al passo 6
- 3 Esegui **blocco_istruzioni**
- 4 Esegui **espr2**
- 5 Vai al punto 2.
- 6 continua con l'esecuzione del programma (esci dal ciclo)

for: uso tipico

Tipicamente, **espr1** è l'inizializzazione di una variabile, **guardia** è una condizione su questa variabile, **espr2** è l'incremento di questa variabile. **blocco_istruzioni** utilizza il valore di questa variabile a seconda del risultato che vogliamo ottenere.

```
int i;  
for (i=0; i<10; i++){  
    printf("%d\n", i*3);  
}
```

Cosa fa questo programma?

for: uso tipico

Tipicamente, **espr1** è l'inizializzazione di una variabile, **guardia** è una condizione su questa variabile, **espr2** è l'incremento di questa variabile. **blocco_istruzioni** utilizza il valore di questa variabile a seconda del risultato che vogliamo ottenere.

```
int i;  
for (i=0; i<10; i++){  
    printf("%d\n", i*3);  
}
```

Cosa fa questo programma? Stampa la tabellina del 3.

for: uso tipico

```
int i;  
for (i=0; i<10; i++){  
    if (i%2 == 0)  
        printf("%d\n", i);  
}
```

Cosa fa questo programma?

for: uso tipico

```
int i;  
for (i=0; i<10; i++){  
    if (i%2 == 0)  
        printf("%d\n", i);  
}
```

Cosa fa questo programma? Stampa i numeri pari minori di 10.

for: altro esempio

In realtà, le espressioni possono essere qualunque:

```
int n;  
for (n=1; n>-50 && n<150; n*=-2){  
    printf("%d\n", n);  
}
```

Cosa fa questo programma?

for: altro esempio

In realtà, le espressioni possono essere qualunque:

```
int n;  
for (n=1; n>-50 && n<150; n*=-2){  
    printf("%d\n", n);  
}
```

Cosa fa questo programma? Partendo da $n = 1$, moltiplica n per -2 finché non esce dall'intervallo $(-50, 150)$.

for: cicli infiniti

Per uscire dal ciclo `for` la guardia deve diventare falsa. Cosa succede se la guardia del `for` è sempre verificata?

```
int i;  
for (i=0; i%2 == 0; i+=2){  
    printf("%d\n", i);  
}
```

for: cicli infiniti

Per uscire dal ciclo `for` la guardia deve diventare falsa. Cosa succede se la guardia del `for` è sempre verificata?

```
int i;  
for (i=0; i%2 == 0; i+=2){  
    printf("%d\n", i);  
}
```

Ciclo infinito: il programma non termina.

for: cicli infiniti

Per uscire dal ciclo `for` la guardia deve diventare falsa. Cosa succede se la guardia del `for` è sempre verificata?

```
int i;  
for (i=0; i%2 == 0; i+=2){  
    printf("%d\n", i);  
}
```

Ciclo infinito: il programma non termina.

```
int i;  
for (i=0; i<=0; i--){  
    printf("%d\n", i);  
}
```

Questo è uno degli errori tipici contenuti nei programmi.

Sintassi `while`

while: esegui un blocco di istruzioni per un numero di iterazioni non pre-fissato. Sintassi:

```
while (guardia)  
    blocco_istruzioni
```

Semantica:

- 1 Se la guardia è **falsa**, salta al passo 3.
- 2 Esegui **blocco_istruzioni**; vai al passo 1.
- 3 continua con l'esecuzione del programma (esci dal ciclo)

while: esempio

```
int somma = 0;
int n;
while (somma < 100){
    scanf("%d", &n);
    somma += n;
}
printf("%d\n", somma);
```

Cosa fa questo programma?

while: esempio

```
int somma = 0;
int n;
while (somma < 100){
    scanf("%d", &n);
    somma += n;
}
printf("%d\n", somma);
```

Cosa fa questo programma? Legge interi da terminale finché la somma non supera 100, quindi stampa la somma.

while: esempio

```
int somma = 0;
int n;
while (somma < 100){
    scanf("%d", &n);
    somma += n;
}
printf("%d\n", somma);
```

Cosa fa questo programma? Legge interi da terminale finché la somma non supera 100, quindi stampa la somma. A priori, è impossibile sapere quanti numeri saranno letti, quindi si usa **while**.

while: esempio

```
int n;  
int i;  
scanf("%d", &n);  
i = 2;  
while (n%i && i<n){  
    i++;  
}  
printf("%d\n", i<n);
```

Cosa fa questo programma?

while: esempio

```
int n;  
int i;  
scanf("%d", &n);  
i = 2;  
while (n%i && i<n){  
    i++;  
}  
printf("%d\n", i<n);
```

Cosa fa questo programma? Stampa 0 se n è primo, 1 altrimenti (esiste un numero $i \in [2, n - 1]$ t.c. n è divisibile per i).

while: esempio

```
int n;  
int i;  
scanf("%d", &n);  
i = 2;  
while (n%i && i<n){  
    i++;  
}  
printf("%d\n", i<n);
```

Cosa fa questo programma? Stampa 0 se n è primo, 1 altrimenti (esiste un numero $i \in [2, n - 1]$ t.c. n è divisibile per i). A priori, è impossibile sapere quanti valori di i saranno testati, quindi si usa **while**.

Esempio:

$n = 9, i = 2, 3$

$n = 7, i = 2, 3, 4, 5, 6, 7$

Cicli annidati: esempio

```
int n;  
int i;  
int j;  
for (j=0; j<10; j++){  
    scanf("%d", &n);  
    i = 2;  
    while (n%i && i<n){  
        i++;  
    }  
    printf("%d\n", i<n);  
}
```

Cosa fa questo programma?

Cicli annidati: esempio

```
int n;  
int i;  
int j;  
for (j=0; j<10; j++){  
    scanf("%d", &n);  
    i = 2;  
    while (n%i && i<n){  
        i++;  
    }  
    printf("%d\n", i<n);  
}
```

Cosa fa questo programma? Legge 10 interi, e per ognuno stampa 0 se n è primo, 1 altrimenti.

Cicli annidati: esempio

```
int n;  
int i;  
int j;  
for (j=0; j<10; j++){  
    scanf("%d", &n);  
    i = 2;  
    while (n%i && i<n){  
        i++;  
    }  
    printf("%d\n", i<n);  
}
```

Cosa fa questo programma? Legge 10 interi, e per ognuno stampa 0 se n è primo, 1 altrimenti.

Notare l'indentazione

Sintassi `do-while`

`do-while`: esegui un blocco di istruzioni per un numero di iterazioni non pre-fissato, ma almeno una volta. Sintassi:

```
do  
    blocco_istruzioni  
while (guardia)
```

Semantica:

- 1 Esegui `blocco_istruzioni`;
- 2 Se la guardia è **vera**, salta al passo 1 (continua il ciclo), se invece è **falsa**, salta al passo 3.
- 3 continua con l'esecuzione del programma (esci dal ciclo)

Questo costrutto generalmente è meno usato degli altri due.

do-while: esempio

```
int n;  
do {  
    scanf("%d", &n);  
} while (n>=0)  
printf("%d\n", n);
```

Cosa fa questo programma?

do-while: esempio

```
int n;  
do {  
    scanf("%d", &n);  
} while (n>=0)  
printf("%d\n", n);
```

Cosa fa questo programma? Legge una sequenza di interi positivi da tastiera terminati da un intero negativo che viene stampato.

do-while: esempio

```
int n;  
do {  
    scanf("%d", &n);  
} while (n>=0)  
printf("%d\n", n);
```

Cosa fa questo programma? Legge una sequenza di interi positivi da tastiera terminati da un intero negativo che viene stampato.

A priori, è impossibile sapere quanti valori saranno letti, ma sappiamo che ne va letto almeno uno, quindi si usa **do-while**.

Tre costrutti: riepilogo

- **for**: **si sa in anticipo** il numero di iterazioni;
- **while**: **non si sa in anticipo** il numero di iterazioni;
- **do-while**: non si sa in anticipo il numero di iterazioni, ma sappiamo che deve essere eseguita **almeno una** iterazione.

In realtà **for**, **while** e **do-while** hanno la stessa potenza espressiva, cioè possono fare le stesse cose.

Equivalenza di `while` e `for`

Questi due programmi sono equivalenti:

- Programma 1:

```
for (expr_1; guardia; expr_2)
    blocco_istruzioni;
```

- Programma 2:

```
expr_1;
while (guardia){
    blocco_istruzioni;
    expr_2;
}
```

Equivalenza dei tre costrutti iterativi

- **while**, **for** e **do-while** sono equivalenti dal punto di vista funzionale. Vuol dire che possiamo intercambiarli?
- **NO!** bisogna usare le regole viste prima. Usare un **while** quando si sa a priori il numero di iterazioni è **un errore**: porta a scrivere codice poco comprensibile e poco manutenibile.
- Se gli altri non capiscono il tuo codice, vuol dire che è scritto male.
- Ricorda sempre: codice buono = codice bello (non basta che funzioni).



Array

In inglese: *oggetti messi in fila*

- È una semplice struttura dati che permette di mantenere in memoria un numero *prefissato* di elementi, tutti *dello stesso tipo*.

Array

In inglese: *oggetti messi in fila*

- È una semplice struttura dati che permette di mantenere in memoria un numero *prefissato* di elementi, tutti *dello stesso tipo*.
- Esempio: mantenere in memoria l'età di 15 persone, in modo da poterne calcolare la media.

```
int eta[15];
```

Array

In inglese: *oggetti messi in fila*

- È una semplice struttura dati che permette di mantenere in memoria un numero *prefissato* di elementi, tutti *dello stesso tipo*.
- Esempio: mantenere in memoria l'età di 15 persone, in modo da poterne calcolare la media.

```
int eta[15];
```

- Esempio: mantenere in memoria la temperatura minima degli ultimi 30 giorni, in modo da calcolarne il minimo

```
double temperatura[30];
```


Accesso agli elementi dell'array

Abbiamo l'array `int array[]`

- Accesso all'*i*-esimo elemento dell'array: `array[i]` (*i* è detto *indice*)

Accesso agli elementi dell'array

Abbiamo l'array `int array[]`

- Accesso all'*i*-esimo elemento dell'array: `array[i]` (*i* è detto *indice*)
- Esempio: scrittura di un valore

```
int array[30];  
array[17] = 5;
```

Accesso agli elementi dell'array

Abbiamo l'array `int array[]`

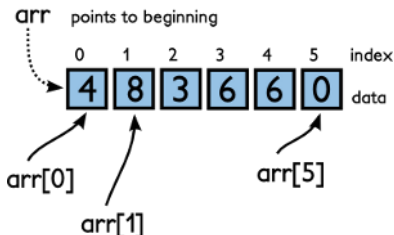
- Accesso all'*i*-esimo elemento dell'array: `array[i]` (*i* è detto *indice*)
- Esempio: scrittura di un valore

```
int array[30];  
array[17] = 5;
```

- Esempio: lettura di un valore

```
int array[30];  
int n;  
...  
n = array[17];
```

Accesso agli elementi dell'array



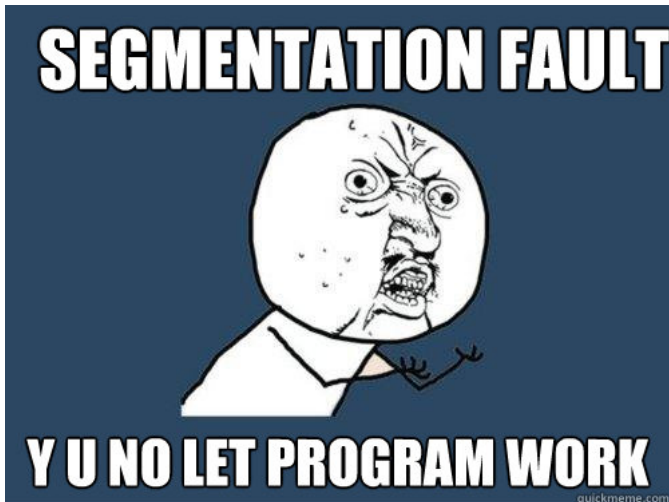
IMPORTANTE: Se la dimensione di un array è n , i suoi indici vanno da 0 a $n - 1$.

Accessi e scritture fuori dall'array

Che succede se scriviamo o leggiamo un indice fuori dall'array?

```
int array[30];  
array[30] = 5;  
array[40] = 5;  
a = array[-1];
```

Accessi e scritture fuori dall'array



Inizializzazione

Due modi per inizializzare un array, analoghi all'inizializzazione delle variabili:

- Inizializzazione *esplicita*: al momento della dichiarazione conosciamo già il contenuto

```
int eta [] = {23, 24, 17, 27, 25, 24, 24}
```

Inizializzazione

Due modi per inizializzare un array, analoghi all'inizializzazione delle variabili:

- Inizializzazione *esplicita*: al momento della dichiarazione conosciamo già il contenuto

```
int eta [] = {23, 24, 17, 27, 25, 24, 24}
```

- Inizializzazione *implicita*: al momento della dichiarazione non conosciamo il contenuto

```
int eta [15];
```


Inizializzazione

Due modi per inizializzare un array, analoghi all'inizializzazione delle variabili:

- Inizializzazione *esplicita*: al momento della dichiarazione conosciamo già il contenuto

```
int eta [] = {23, 24, 17, 27, 25, 24, 24}
```

- Inizializzazione *implicita*: al momento della dichiarazione non conosciamo il contenuto

```
int eta [15];
```

- Che succede se leggiamo un valore da un array non inizializzato? Il risultato della lettura **non** è un valore affidabile.

Riepilogo array

Evitare errori tipici:

- **Non** accedere ad indici fuori dal range $[0, n - 1]$ (segfault)

Riepilogo array

Evitare errori tipici:

- **Non** accedere ad indici fuori dal range $[0, n - 1]$ (segfault)
- **Non** leggere valori non inizializzati (valori non affidabili)

Array e cicli (for)

Array e cicli si sposano naturalmente. Esempio con `for`:

```
int eta[] = {23, 24, 17, 27, 25, 24, 24}
int N = 7;
int i;
int somma = 0;
double media = 0;
for (i = 0; i < N; i++)
    somma += eta[i]
media = (double) somma / N;
```

Cosa fa questo programma?

Array e cicli (for)

Array e cicli si sposano naturalmente. Esempio con `for`:

```
int eta[] = {23, 24, 17, 27, 25, 24, 24}
int N = 7;
int i;
int somma = 0;
double media = 0;
for (i = 0; i < N; i++)
    somma += eta[i]
media = (double) somma / N;
```

Cosa fa questo programma? Calcola la media degli elementi contenuti in `eta`.

Array e cicli (while)

Array e cicli si sposano naturalmente. Esempio con **while**:

```
int N = 7;  
double temperatura_min[] = {2, 5, 5, -1, 3, 0, -2}  
int i = 0;  
while (i < N && temperatura_min[i] > 0)  
    i++
```

Cosa fa questo programma?

Array e cicli (while)

Array e cicli si sposano naturalmente. Esempio con **while**:

```
int N = 7;  
double temperatura_min[] = {2, 5, 5, -1, 3, 0, -2}  
int i = 0;  
while (i < N && temperatura_min[i] > 0)  
    i++
```

Cosa fa questo programma? Se $i < N$ allora i memorizza l'indice del primo giorno di gelo, se $i = N$ allora non ci sono stati giorni di gelo.

Esempio lettura da terminale

```
int arr[7]
int i;
int n;
for (i=0; i<7; i++){
    scanf("%d", &n);
    arr[i] = n;
}
for (i=6; i>=0; i--){
    printf("%d", arr[i]);
}
```

Cosa fa questo programma?

Esempio lettura da terminale

```
int arr[7]
int i;
int n;
for (i=0; i<7; i++){
    scanf("%d", &n);
    arr[i] = n;
}
for (i=6; i>=0; i--){
    printf("%d", arr[i]);
}
```

Cosa fa questo programma? Legge 7 interi da terminale e li stampa nell'ordine inverso rispetto a come sono stati letti.

Array e cicli: regole generali

Regola generale:

- Ogni volta che pensate
“Per ogni elemento dell’array, ...”
→ bisogna usare un ciclo **for**
- Ogni volta che pensate
“Per almeno un elemento dell’array, ...”
→ bisogna usare un ciclo **while**