



# Strutture di Controllo

Fondamenti di Informatica, AA 2022/23

Luca Cassano

[luca.cassano@polimi.it](mailto:luca.cassano@polimi.it)



# Operatori ed Espressioni Logiche

- Algebra di Boole



## Non solo operazioni aritmetiche

- **Espressione booleana:** espressione con valore **vero (1)** o **falso (0)**, determinata dagli **operatori e dal valore delle costanti o variabili** in essa contenute.



## Non solo operazioni aritmetiche

- In C abbiamo:

- **operatori relazionali**: si applicano a **variabili, costanti** o espressioni e sono `==`, `!=`, `>`, `<`, `>=`, `<=`

*Es:* `(a > 7)` , `(b % 2 == 0)` , `(x <= w)`

danno luogo ad espressioni Booleane.

- **operatori logici**: applicati a **espressioni booleane**, permettono di costruire **espressioni composte** e sono `!`, `&&`, `||`

*Es:* `(a > 7) && (b % 2 == 0)`

`! (x >= 7) || (a == 0)`



## Operazioni built-in per dati di tipo `int`

- `=` Assegnamento di un valore `int` a una variabile `int`
- `+` Somma (tra `int` ha come risultato un `int`)
- `-` Sottrazione (tra `int` ha come risultato un `int`)
- `*` Moltiplicazione (tra `int` ha come risultato un `int`)
- `/` Divisione con troncamento della parte non intera (risultato `int`)
- `%` Resto della divisione intera
- `==` Relazione di uguaglianza
- `!=` Relazione di diversità
- `<` Relazione “minore di”
- `>` Relazione “maggiore di”
- `<=` Relazione “minore o uguale a”
- `>=` Relazione “maggiore o uguale a”

Operatori  
Aritmetici

Operatori  
Relazionali



## Operazioni built-in per dati di tipo `int`

- `=` Assegnamento di un valore `int` a una variabile `int`
- `+` Somma (tra `int` ha come risultato un `int`)
- `-` Sottrazione (tra `int` ha come risultato un `int`)
- `*` Moltiplicazione (tra `int` ha come risultato un `int`)
- `/` Divisione con troncamento della parte non intera (risultato `int`)
- `%` Resto della divisione intera
- `==` Relazione di uguaglianza
- `!=` Relazione di diversità
- `<` Relazione “minore di”
- `>` Relazione “maggiore di”
- `<=` Relazione “minore o uguale a”
- `>=` Relazione “maggiore o uguale a”

Operatori  
Aritmetici

Operatori  
Relazionali



## Non solo operazioni aritmetiche

- In C abbiamo:
  - **operatori relazionali**: si applicano a **variabili, costanti** o espressioni e sono `==`, `!=`, `>`, `<`, `>=`, `<=`  
*Es:* `(a > 7)` , `(b % 2 == 0)` , `(x <= w)`  
danno luogo ad espressioni Booleane.
  - **operatori logici**: applicati a **espressioni booleane**, permettono di costruire **espressioni composte** e sono `!`, `&&`, `||`  
*Es:* `(a > 7) && (b % 2 == 0)`  
`!(x >= 7) || (a == 0)`



## Aritmetica Operatori Logici

Ordine Operatori Logici in assenza di parentesi (elementi a priorità maggiore in alto):

1. negazione (NOT) !
2. operatori di relazione <, >, <=, >=
3. uguaglianza ==, disuguaglianza !=,
4. congiunzione (AND) &&
5. disgiunzione (OR) ||

Esempi

- `x > 0 || y == 3 && !z`
- `(x > 0) || ((y == 3) && (!z))`





## Aritmetica degli Operatori Logici

- Gli operatori `&&` e `||` sono commutativi
  - $(a \ \&\& \ b) == (b \ \&\& \ a)$
  - $(a \ || \ b) == (b \ || \ a)$
- Le doppie negazioni si elidono:  $!!a == a$



## Come Funzionano gli Operatori Logici?

- Ogni espressione booleana può assumere solo due valori
- Posso quindi considerare tutti i possibili valori degli ingressi ad un'espressione booleana e calcolare i valori di output corrispondenti
- Questo corrisponde alla tabella di verità
- Incominciamo a farla per definire gli operatori `!`, `&&` e `||`



# Tabelle di Verità degli Operatori Logici

- Le tabelle di verità stabiliscono i valori di predicati composti
- Il NOT è un operatore **unario**, che prende in ingresso **una** sola espressione.
- **$!A$**  è l'opposto di  **$A$**

negazione (NOT)	
<b><math>A</math></b>	<b><math>!A</math></b>
0	1
1	0



# Tabelle di Verità degli Operatori Logici

- Le tabelle di verità stabiliscono i valori di predicati composti
- L'operatore AND è **binario**, prende in ingresso **due** espressioni.
- **A && B** è vero se e solo se sia **A** che **B** sono vere.

congiunzione (AND)		
A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1



# Tabelle di Verità degli Operatori Logici

- Le tabelle di verità stabiliscono i valori di predicati composti
- L'operatore OR è **binario**, prende in ingresso **due** espressioni.
- $A \mid \mid B$  è vero se almeno una delle due è vera.
- **NB:** non è un OR esclusivo, come spesso accade nella lingua parlata

disgiunzione (OR)		
A	B	$A \mid \mid B$
0	0	0
0	1	1
1	0	1
1	1	1



## Tabelle di Verità

- Rappresenta tutti i possibili modi di valutare un'espressione booleana composta
- Una riga per ogni possibile assegnamento di valori logici alle variabili:
  - $n$  variabili logiche (espressioni booleane)  $\rightarrow 2^n$  possibili assegnamenti, quindi  $2^n$  righe.
- Una colonna per ogni espressione che compone l'espressione data (inclusa la formula stessa)



## Esempio Tabella di Verità

- $A \ \&\& \ !B \ || \ C$



## Esempio Tabella di Verità

- $A \ \&\& \ !B \ || \ C$

A	B	C
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1





## Esempio Tabella di Verità

- $A \ \&\& \ !B \ || \ C$

A	B	C	!B	$A \ \&\& \ !B$	$A \ \&\& \ !B \    \ C$
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			



## Esempio Tabella di Verità

- $A \ \&\& \ !B \ || \ C$

A	B	C	!B	$A \ \&\& \ !B$	$A \ \&\& \ !B \    \ C$
0	0	0	1		
0	0	1	1		
0	1	0	0		
0	1	1	0		
1	0	0	1		
1	0	1	1		
1	1	0	0		
1	1	1	0		



## Esempio Tabella di Verità

- $A \ \&\& \ !B \ || \ C$

A	B	C	!B	$A \ \&\& \ !B$	$A \ \&\& \ !B \    \ C$
0	0	0	1	0	
0	0	1	1	0	
0	1	0	0	0	
0	1	1	0	0	
1	0	0	1	1	
1	0	1	1	1	
1	1	0	0	0	
1	1	1	0	0	



## Esempio Tabella di Verità

- $A \ \&\& \ !B \ || \ C$

A	B	C	!B	$A \ \&\& \ !B$	$A \ \&\& \ !B \    \ C$
0	0	0	1	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	1	0	0	1



## Operatori Logici: Leggi di de Morgan

- Leggi di De Morgan: illustrano come distribuire la negazione rispetto a `||` e `&&`
  1. `!(a && b) == !a || !b`
  2. `!(a || b) == !a && !b`



## Espressioni Booleane in C

- Servono per definire condizioni che vengono impiegate in istruzioni composte:
  - Costrutti condizionali: `if`, `switch`
  - Costrutti iterativi: `while`, `do while`, `for`



## Espressioni Intere come Booleane in C

- **Espressioni intere e booleane sono intercambiabili:**  
esiste una regola di conversione automatica
  - $0 \Leftrightarrow \text{falso}$
  - **qualsiasi valore  $\neq 0 \Leftrightarrow \text{vero}$**
- Ciò utilizzato in pratica (anche se non bello dal punto di vista concettuale) per
  - **memorizzare in variabili intere risultati di condizioni** (valori di espressioni logiche, non esistono del resto variabili di un apposito tipo in C)
  - **utilizzare espressioni aritmetiche al posto di condizioni** nelle istruzioni **if** e **while**



# Linguaggio C: Costrutto Condizionale

Istruzioni composta: `if`





## Costrutto Condizionale: **if**, la sintassi

- Il costrutto **condizionale** permette di eseguire alcune istruzioni a seconda del valore di un'espressione booleana a runtime
- **if, else** keywords
- **expression** espressione booleana (vale 0 o 1)
- **corpo** sequenza di istruzioni da eseguire.
- **NB:** **corpo** è il corpo dell' **if** (o dell' **else**). Se **corpo** contiene una sola istruzione le **{ }** possono essere omesse
- **NB** indentatura irrilevante

```
if (expression) {  
    corpo_if;  
}
```

```
if (expression) {  
    corpo_if;  
}  
  
else {  
    corpo_else;  
}
```



## Costrutto Condizionale: **if**, l'esecuzione

1. Terminata **instrBefore**,  
valuto **expression**,
2. Se **expression** è vera ( $\neq 0$ ),  
allora eseguo **statement1**,  
altrimenti eseguo **statement0**.  
(se è presente **else**)
3. Terminato lo statement dell'**if** ,  
procedi con **instrAfter**, la  
prima istruzione fuori dall'**if**

**N.B.** **else** è opzionale

```
instrBefore;  
if(expression) {  
    statement1;  
}  
else {  
    statement0;  
}  
instrAfter;
```



## Costrutto Condizionale: **if**, l'esecuzione

1. Terminata **instrBefore**,  
valuto **expression**.
2. Se **expression** allora eseguo **statement1**,  
altrimenti eseguo **statement0**.  
(se è presente **else**)

3. Terminato lo statement dell'**if**,  
procedi con **instrAfter**, la  
prima istruzione

**N.B.** **else** è opzionale

```
instrBefore;  
  
if (expression) {  
    statement1;  
}  
  
else {  
    statement0;  
}  
  
instrAfter;
```

Ramo then

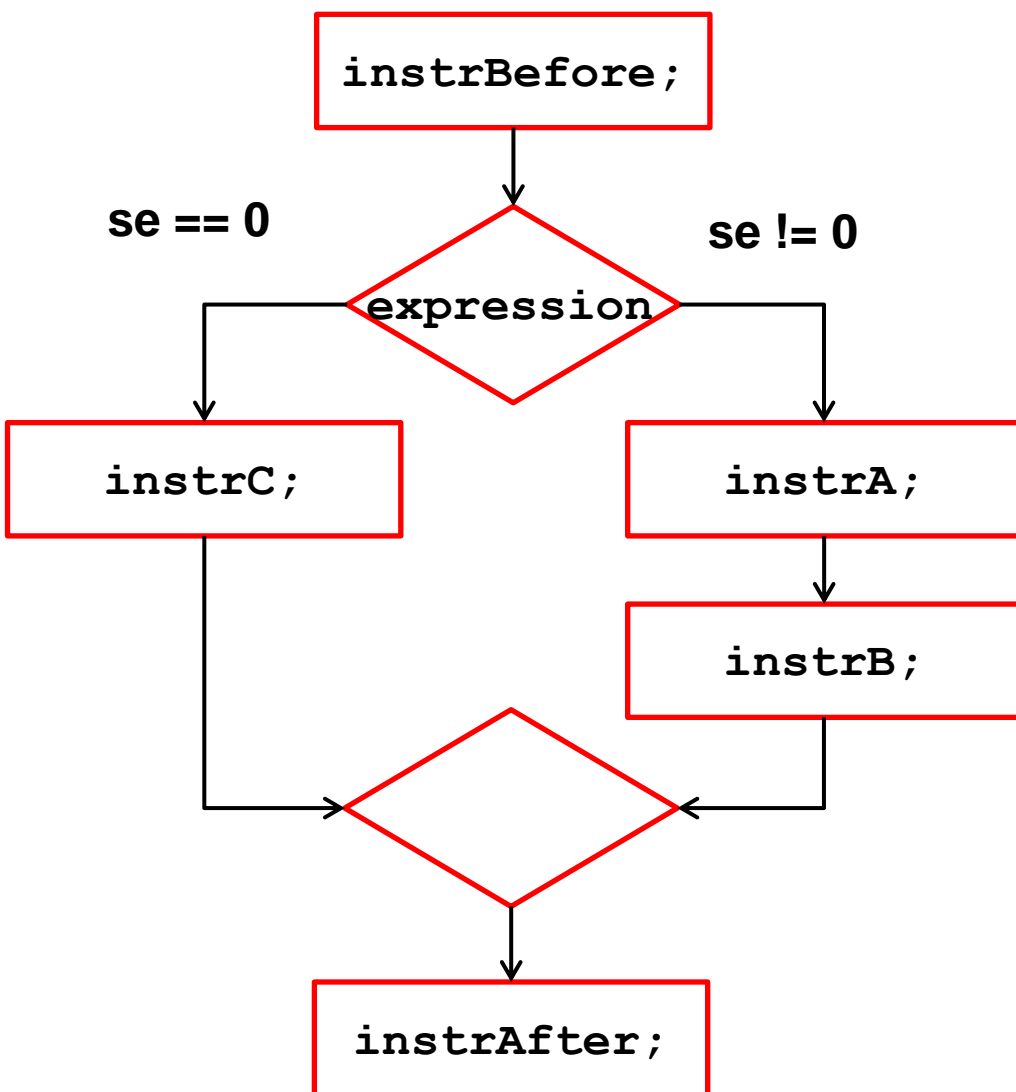
statement1;

Ramo else

statement0;



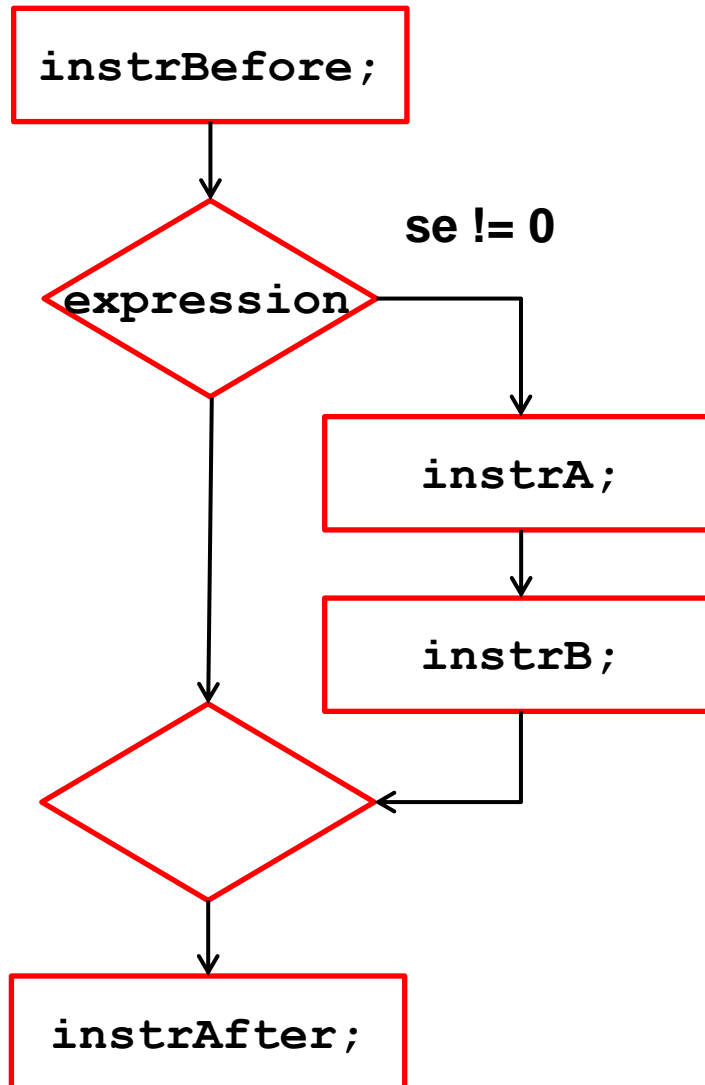
# Costrutto Condizionale: if, l'esecuzione



```
instrBefore;  
if (expression) {  
    instrA;  
    instrB;  
}  
else {  
    instrC;  
}  
instrAfter;
```



# Costrutto Condizionale: if, l'esecuzione



```
instrBefore;  
if (expression) {  
    instrA;  
    instrB;  
}  
instrAfter;
```



## Esempio

//N.B: incolonnamento codice irrilevante!

```
if (x % 7 == 0) {  
    printf("%d multiplo di 7" , x);  
}  
  
else {  
    printf("%d non multiplo di 7" , x);  
}
```



## Esempio

```
/* N.B: incolonnamento codice irrilevante! */  
if (x % 7 == 0) {  
    printf("%d multiplo di 7" , x);  
}  
else {  
    printf("%d non multiplo di 7" , x);  
}  
/* si può fare senza else? */
```



## Esempio

```
/* senza else */  
printf("%d " , x) ;  
if (x % 7 != 0) {  
    printf("non " ) ;  
}  
printf(" multiplo di 7") ;
```





## if Annidati

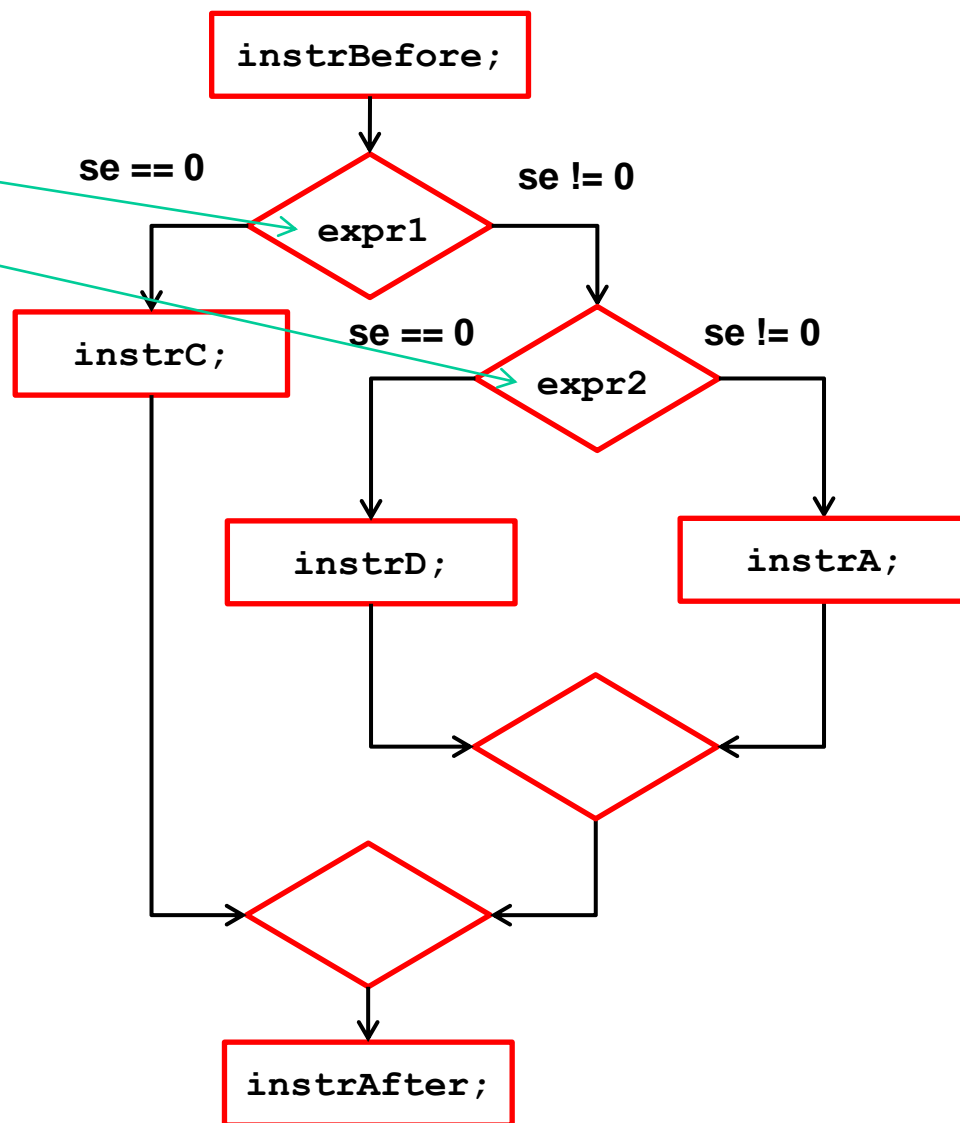
Il corpo di un **if** (cioè gli **statement**) può a sua volta contenere altri **if**: si realizzano quindi istruzioni condizionali **annidate**

```
instrBefore;  
if(expr1) {  
    if(expr2) {  
        instrA;  
    }  
    else {  
        instrD;  
    }  
}  
else {  
    instrC;  
}  
instrAfter;
```



## if Annidati

```
instrBefore;  
if (expr1) {  
    if (expr2) {  
        instrA;  
    }  
    else {  
        instrD;  
    }  
} else {  
    instrC;  
}  
instrAfter;
```





## if Annidati

Le istruzioni condizionali possono essere annidate, inserendo un ulteriore **if** sia all'interno del ramo then che del ramo else!

### *Esempio*

```
if ( cond1 ) {  
    if ( cond2 ) {  
        instr1;  
    }  
    else {  
        instr2;  
    }  
}  
else {  
    instr3;  
}
```



## if Annidati

Le istruzioni condizionali possono essere annidate, inserendo un ulteriore **if** sia all'interno del ramo then che del ramo else!

### *Esempio*

```
if ( cond1 ) {  
    instr1;  
}  
else {  
    if ( cond2 ) {  
        instr2;  
    }  
    else {  
        instr3;  
    }  
}
```



## Regole per `if` annidati

**Regola:** In caso di costrutti annidati, ed in assenza di parentesi che indichino diversamente, ogni `else` viene associato all' `if` più vicino.

```
if ( x % 5 == 0 )  
    if (x % 7 == 0 )  
        printf("x multiplo di 5 e anche di 7");  
    else  
        printf("x multiplo di 5 ma non di 7");
```



## Regole per `if` annidati

**Regola:** In caso di costrutti annidati, ed in assenza di parentesi che indichino diversamente, ogni `else` viene associato all' `if` più vicino.

```
if ( x % 5 == 0 )  
    if (x % 7 == 0 )  
        printf("x multiplo di 5 e anche di 7");  
else  
    printf("x multiplo di 5 ma non di 7");
```



## Regole per `if` annidati

**Regola:** In caso di costrutti annidati, ed in assenza di parentesi che indichino diversamente, ogni `else` viene associato all' `if` più vicino.

```
if ( x % 5 == 0 ) {  
    if (x % 7 == 0 )  
        printf("x multiplo di 5 e anche di 7");  
}  
else  
    printf("x non multiplo di 5");
```



## if Annidati

Quando il corpo di un if  
contiene più di un'istruzione  
è necessario usare parentesi.

```
instrBefore;  
if (expr1)  
    if (expr2) {  
        instrA;  
        instrB;  
    }  
    else  
        instrD;  
else  
    instrC;  
instrAfter;
```

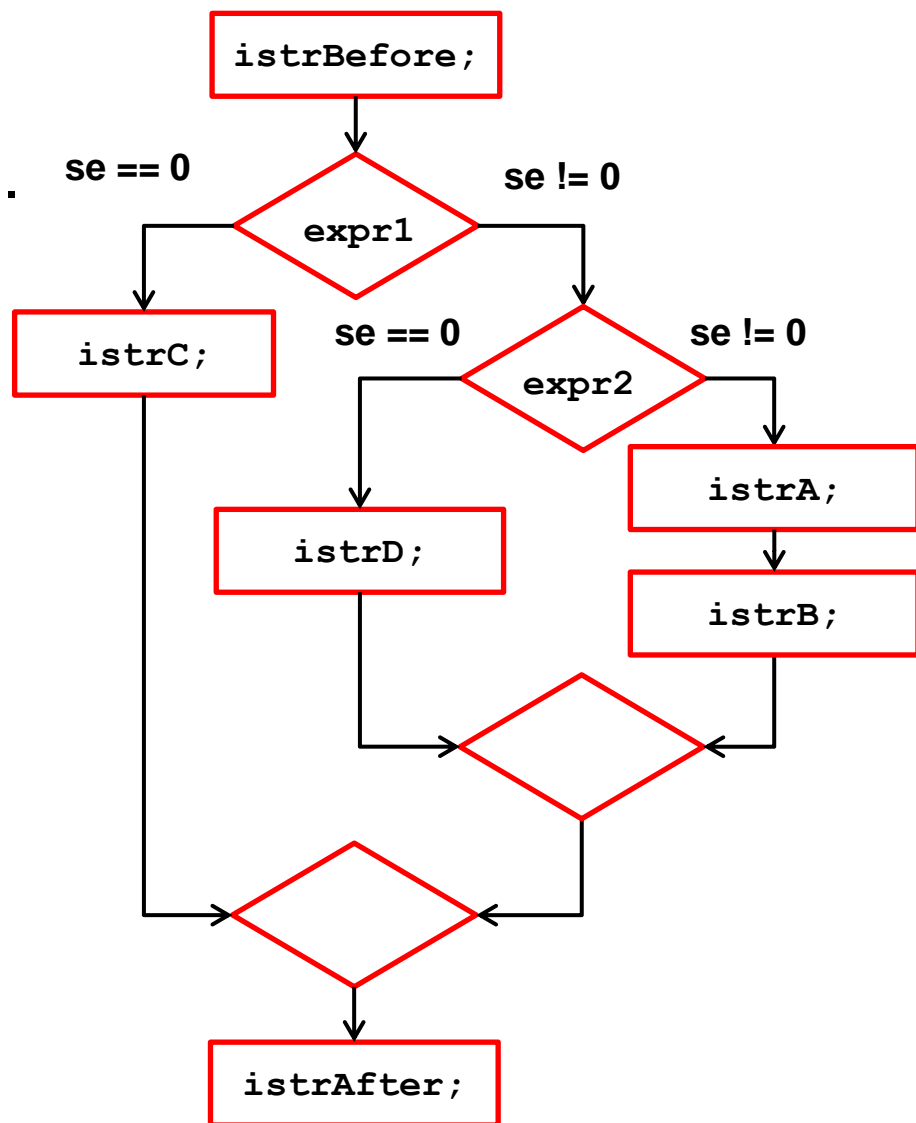




## if Annidati

Quando il corpo di un if contiene più di un'istruzione è necessario usare parentesi.

```
instrBefore;  
if (expr1)  
    if (expr2) {  
        instrA;  
        instrB;  
    }  
    else  
        instrD;  
else  
    instrC;  
instrAfter;
```





## if Annidati

L'uso delle parentesi serve per determinare quale **else** associare a quale **if**.

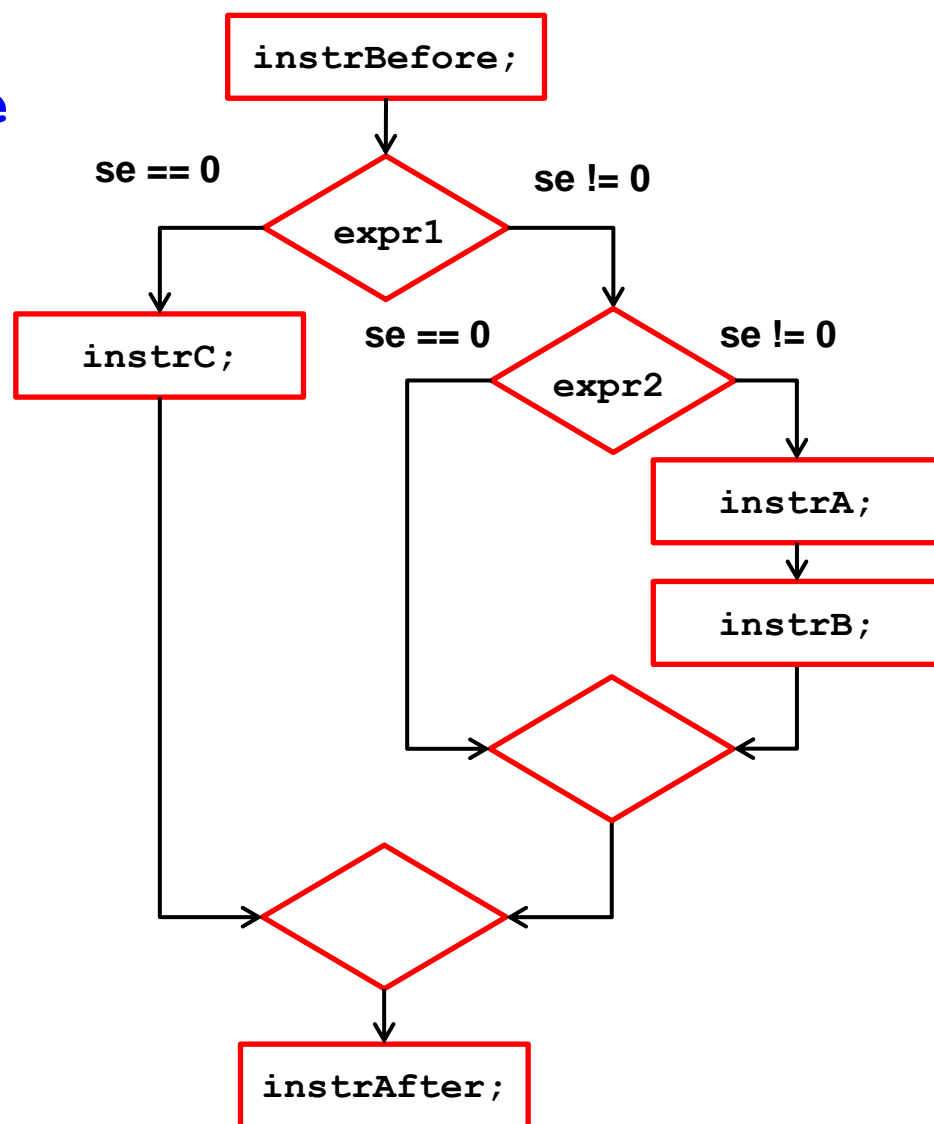
```
instrBefore;  
if (expr1)  
{ if (expr2)  
    {instrA;  
    instrB;}  
}  
else  
    instrC;  
instrAfter;
```



## if Annidati

L'uso delle parentesi serve per determinare quale **else** associare a quale **if**.

```
instrBefore;  
if(expr1)  
{ if(expr2)  
    {instrA;  
    instrB;}  
}  
else  
    instrC;  
instrAfter;
```





## if Annidati

L'uso delle parentesi serve per determinare quale **else** associare a quale **if**.

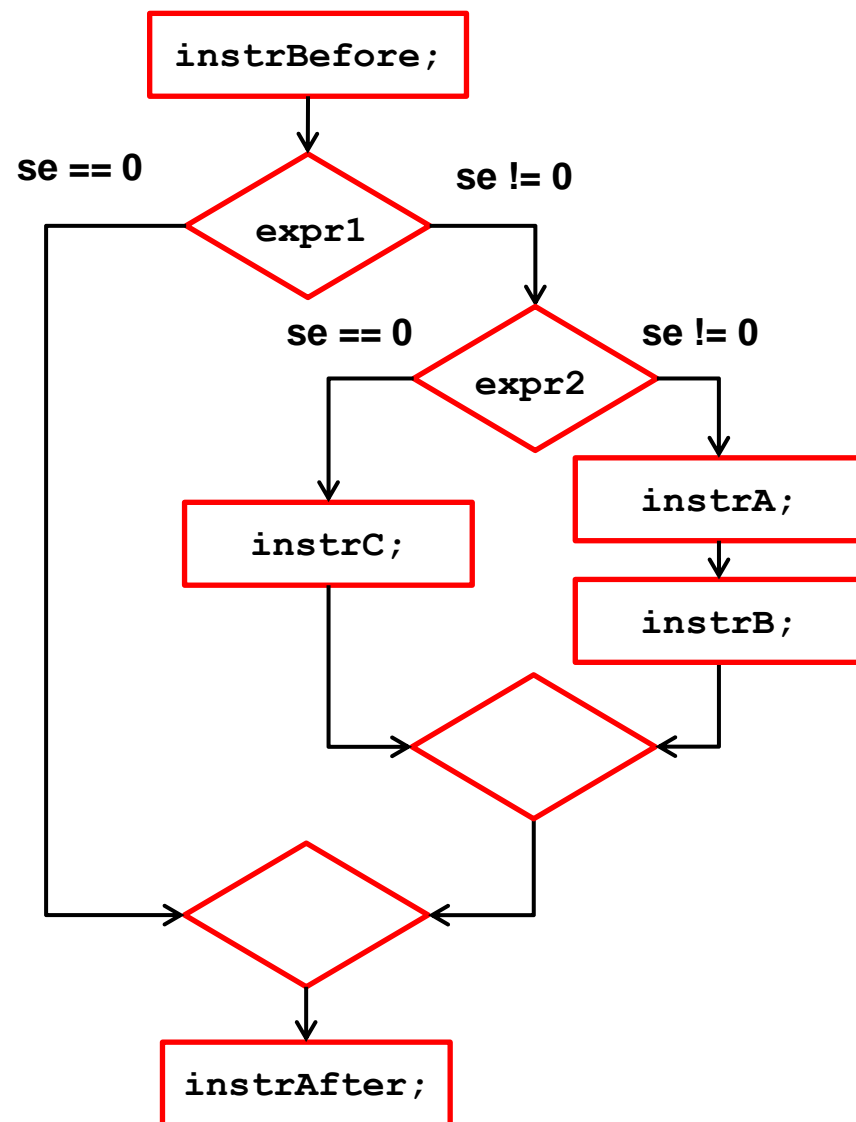
```
instrBefore;  
if (expr1)  
{ if (expr2)  
    {instrA;  
      instrB;}  
  else  
    instrC;  
}  
instrAfter;
```



## if Annidati

L'uso delle parentesi serve per determinare quale **else** associare a quale **if**.

```
instrBefore;  
if(expr1)  
{ if(expr2)  
  {instrA;  
   instrB;}  
  else  
    instrC;  
}  
instrAfter;
```





## if - else if - else

Nel caso le condizioni da verificare siano più di una è possibile specificare catene di **if**

```
if (expr1)
    instr1;
else if (expr2)
    instr2;
else if (expr3)
    instr3;
else
    instr4;
```



```
if (expr1)
    instr1;
else {
    if (expr2)
        instr2;
    else {
        if (expr3)
            instr3;
        else
            instr4;
    }
}
```



## Esempio

Scrivere un programma che, inserito un intero positivo, determina se corrisponde ad un anno bisestile

- Un anno è bisestile se
  - è multiplo di 4 ma non di 100
  - oppure se è multiplo di 400



## Soluzione1: if annidati

```
#include<stdio.h>
int main(int argc, char* argv[]) {
    int n; // anno
    int bis = 0;
    printf("inserire anno: ");
    scanf("%d", &n);
    if(n % 4 == 0) {
        bis = 1;
        if(n % 100 == 0)
            bis = 0;
        if(n % 400 == 0)
            bis = 1;
    }
    printf("\n%d ", n);
    if(bis == 0)
        printf("NON ");
    printf("e' bisestile!");
    return 0;
}
```





## Soluzione1: if annidati

```
#include<stdio.h>
int main(int argc, char* argv[]) {
    int n; // anno
    int bis = 0;
    printf("inserire anno: ");
    scanf("%d", &n);
    if(n % 4 == 0) {
        bis = 1;
        if(n % 100 == 0)
            bis = 0;
        if(n % 400 == 0)
            bis = 1;
    }
    printf("\n%d ", n);
    if(bis == 0)
        printf("NON ");
    printf("e' bisestile!");
    return 0;
}
```

### Osservazioni:

1. bis è una variabile che vale 1 quando una data condizione si verifica, in questo caso l'anno è bisestile
2. Le parentesi inutili sono state omesse (non è possibile togliere quella del primo if perché il corpo contiene più istruzioni)



## Soluzione2: condizioni composte e predicati

```
#include<stdio.h>
int main(int argc, char* argv[]) {
    int n; // anno
    int bis = 0;
    int d4, d100, d400;
    printf("inserire anno: ");
    scanf("%d", &n);
    d4 = (n % 4 == 0);
    d100 = (n % 100 == 0);
    d400 = (n % 400 == 0);

    if(d4 && (!d100 || d400))
        bis = 1;

    printf("\n%d ", n);
    if(bis == 0)
        printf("NON ");
    printf("e' bisestile!");
    return 0;
}
```

### Osservazioni:

1. Le variabili d4,d100 e d400 contengono il risultato di un'operazione logica (0/1)



## Esempio

Scrivere un programma che determina il massimo tra tre numeri inseriti da tastiera



## Soluzione 1: If Annidati

```
#include<stdio.h>
int main(int argc, char* argv[]) {
{
    int a,b,c;
    printf("\ninsertire a: ");
    scanf("%d", &a);
    printf("\ninsertire b: ");
    scanf("%d", &b);
    printf("\ninsertire c: ");
    scanf("%d", &c);
    if(a > b)
        if(a > c) // b non può essere il max
            printf("\nmax = %d", a);
        else
            printf("\nmax = %d", c);
    else
        if(b > c)
            printf("\nmax = %d", b);
        else
            printf("\nmax = %d", c);
    return 0;
}
```



## Soluzione 2: Condizioni Composte

```
#include<stdio.h>
int main(int argc, char* argv[]) {
    int a,b,c;
    printf("\ninserire a: ");
    scanf("%d", &a);
    printf("\ninserire b: ");
    scanf("%d", &b);
    printf("\ninserire c: ");
    scanf("%d", &c);

    if(a >= b && a >= c)
        printf("\nmax = %d", a);
    if(b >= c && b >= a)
        printf("\nmax = %d", b);
    if(c >= a && c >= b)
        printf("\nmax = %d", c)
    return 0;
}
```

E' necessario mettere  $\geq$  altrimenti non gestisce correttamente il caso in cui almeno due numeri sono uguali



## Soluzione 3: if in sequenza

```
#include<stdio.h>
int main(int argc, char* argv[]) {
    int a,b,c,max;
    printf("\ninserire a: ");
    scanf("%d", &a);
    printf("\ninserire b: ");
    scanf("%d", &b);
    printf("\ninserire c: ");
    scanf("%d", &c);

    max = a;
    if(max < b)
        max = b;
    if(max < c)
        max = c;

    printf("\nmax = %d", max);
    return 0;
}
```

### Osservazioni:

1. L'uso della variabile ausiliaria facilita le cose
2. Non ricorda niente questa soluzione?



Vi ricordate?

## Algoritmo per ricercare il prodotto migliore

1. Prendi in mano il primo prodotto: assumi che sia il migliore
2. Procedi fino al prossimo prodotto
3. Confrontalo con quello che hai in mano
4. **Se** il prodotto davanti a te è migliore: abbandona il prodotto che hai in mano e prendi quello sullo scaffale
5. **Ripeti** i passi **2 - 4 fino a** raggiungere la fine della corsia
6. Hai in mano il prodotto migliore.

**Algoritmo per trovare il massimo di una sequenza numerica**



# Linguaggio C: Costrutti Iterativi

Istruzioni composte: `while`, `do while`, `for`





## Costrutto Iterativo: **while**, la sintassi

- Il costrutto iterativo permette di ripetere l'esecuzione di istruzioni finché una condizione è valida
- **while** è una keyword
- **expression** espressione booleana, condizione che determina la **permanenza** nel ciclo
- **statement** sequenza di istruzioni da eseguire (corpo del ciclo)
- **NB**: come per **if**, se **statement** contiene più istruzioni, va delimitato tra { }

```
while (expression)  
    statement
```

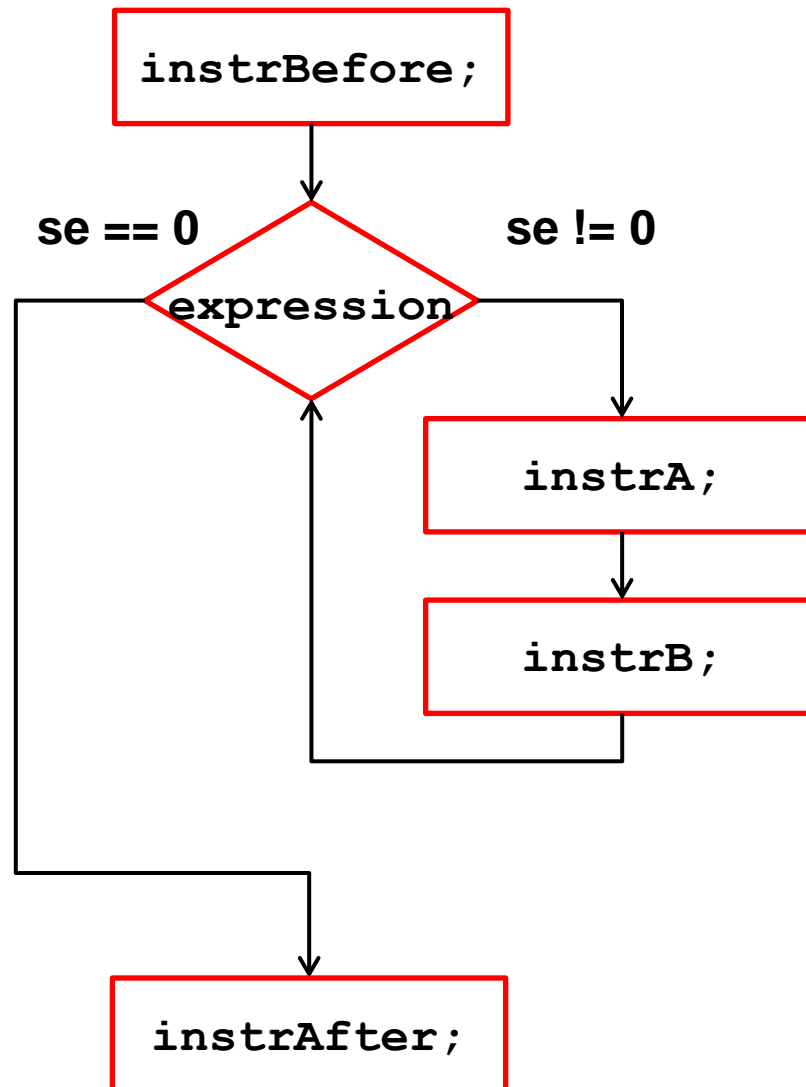


## Costrutto Iterativo: **while**, l'esecuzione

1. Terminata **instrBefore** viene valutata **expression**  
**instrBefore;**
2. Se **expression** è vera viene eseguito **statement**  
**while (expression)**  
**statement;**
3. Al termine, viene valutata nuovamente **expression** e la procedura continua finché **expression** è falsa  
**instrAfter;**
4. Uscito dal ciclo, eseguo **instrAfter**



## Costrutto Iterativo: **while**, l'esecuzione



```
instrBefore;  
while (expression) {  
    instrA;  
    instrB;  
}  
instrAfter;
```



## Esempio

```
/* stampa i primi 100 numeri naturali */
```



## Esempio

```
/* stampa i primi 100 numeri naturali */  
# include<stdio.h>  
int main()  
{  
    int a = 1;  
    while (a <= 100)  
    {  
        printf("\n%d" , a) ;  
        a++;  
    }  
    return 0;  
}
```



## Esempio

```
/* stampa i primi 100 numeri naturali pari*/
```



## Esempio

```
/* stampa i primi 100 numeri pari */  
# include<stdio.h>  
int main()  
{  
    int a = 1;  
    while (a <= 100)  
    {  
        printf("\n%d" , 2*a) ;  
        a++;  
    }  
    return 0;  
}
```



## Costrutto Iterativo: `while`, Avvertenze

```
/* Esempio: stampa i primi 100 numeri pari */  
# include<stdio.h>  
int main()  
{  
    int a = 101;  
    while (a <= 100)  
    {  
        printf("\n%d" , 2*a) ;  
        a++;  
    }  
    return 0;  
}
```





## Costrutto Iterativo: `while`, Avvertenze

- Il corpo del `while` non viene mai eseguito quando `expression` risulta falsa al primo controllo

```
/* Esempio: stampa i primi 100 numeri pari */  
# include<stdio.h>  
int main()  
{  
    int a = 101;  
    while (a <= 100)  
    {  
        printf("\n%d" , 2*a) ;  
        a++;  
    }  
    return 0;  
}
```



## Costrutto Iterativo: `while`, Avvertenze

```
/* Esempio: stampa i primi 100 numeri pari */
# include<stdio.h>
int main()
{
    int a = 1;
    while (a < 100)
    {
        printf("\n%d" , 2*a) ;
    }
    return 0;
}
```



## Costrutto Iterativo: `while`, Avvertenze

- Se **expression** è vera ed il corpo non ne modifica mai il valore, allora abbiamo un loop infinito (l'esecuzione del programma **non** termina)

```
/* Esempio: stampa i primi 100 numeri pari */
# include<stdio.h>
int main()
{
    int a = 1;
    while (a < 100)
    {
        printf("\n%d" , 2*a) ;
    }
    return 0;
}
```



## Costrutto Iterativo: `while`, Avvertenze

```
/* Esempio: stampa i primi 100 numeri pari */  
# include<stdio.h>  
int main()  
{  
    int a = 1;  
    while (a > 0)  
    {  
        printf("\n%d" , 2*a) ;  
        a++;  
    }  
    return 0;  
}
```



## Costrutto Iterativo: `while`, Avvertenze

- Se **expression** è vera ma è sbagliata potremmo avere un loop infinito (l'esecuzione del programma **non** termina)

```
/* Esempio: stampa i primi 100 numeri pari */
# include<stdio.h>
int main()
{
    int a = 1;
    while(a > 0)
    {
        printf("\n%d" , 2*a) ;
        a++;
    }
    return 0;
}
```



## Costrutto Iterativo: `while`

```
/* eseguire la somma di una sequenza di numeri  
naturali inseriti dall'utente (continuare fino a  
quando l'utente inserisce 0)*/
```

```
# include<stdio.h>
```

```
int main()
```

```
{
```

```
}
```



## Costrutto Iterativo: `while`

```
/* eseguire la somma di una sequenza di numeri
naturali inseriti dall'utente (continuare fino a
quando l'utente inserisce 0)*/
```

```
# include<stdio.h>
int main()
{
    int a , somma;
    somma = 0;
    printf("\nInserire a:");
    scanf("%d" , &a);
    while (a != 0)
    {
        somma += a; //somma = somma + a;
        printf("\nInserire a:");
        scanf("%d" , &a);
    }
    printf("\nSomma = %d" , somma);
    return 0;}
```



## Costrutto Iterativo: `while`

```
/* eseguire la somma e la media di una sequenza di numeri  
naturali inseriti dall'utente (continuare fino a quando  
l'utente inserisce 0)*/
```

```
# include<stdio.h>
```

```
int main()
```

```
{
```

```
}
```





## Costrutto Iterativo: `while`

```
/* eseguire la somma e la media di una sequenza di numeri
naturali inseriti dall'utente (continuare fino a quando
l'utente inserisce 0)*/

# include<stdio.h>
int main()
{
    int a , somma , n; float media;
    somma = 0; n = 0;
    printf("\nInserire a:");
    scanf("%d" , &a);
    while (a != 0)
    {
        somma += a;
        n++; //n = n + 1;
        printf("\nInserire a:");
        scanf("%d" , &a);
    }
    media = (1.0 * somma) / n;
    printf("\nSomma = %d , media = %f", somma , media);
    return 0;}
```



## Esercizio per casa

Preparare un programma C per giocare in due a Carta / Sasso / Forbice, richiedendo a ciascun giocatore di inserire uno fra i caratteri 'c', 's', 'f', controllando anche che il carattere inserito sia ammissibile. Il programma termina quando uno dei due giocatori inserisce 'x'



## Costrutto Iterativo: **do-while**, l'esecuzione

1. Viene eseguito **statement**
2. Viene valutata **expression** se è vera viene eseguito **statement** e la procedura continua finché **expression** diventa falsa ( $0 == 0$ )
3. Viene eseguita l'istruzione successiva al ciclo

```
do  
    statement  
while  
(expression) ;
```



## do-while

- Il costrutto **do-while** garantisce l'esecuzione del corpo del **while** almeno una volta.

```
do  
    statement  
while (expression) ;
```

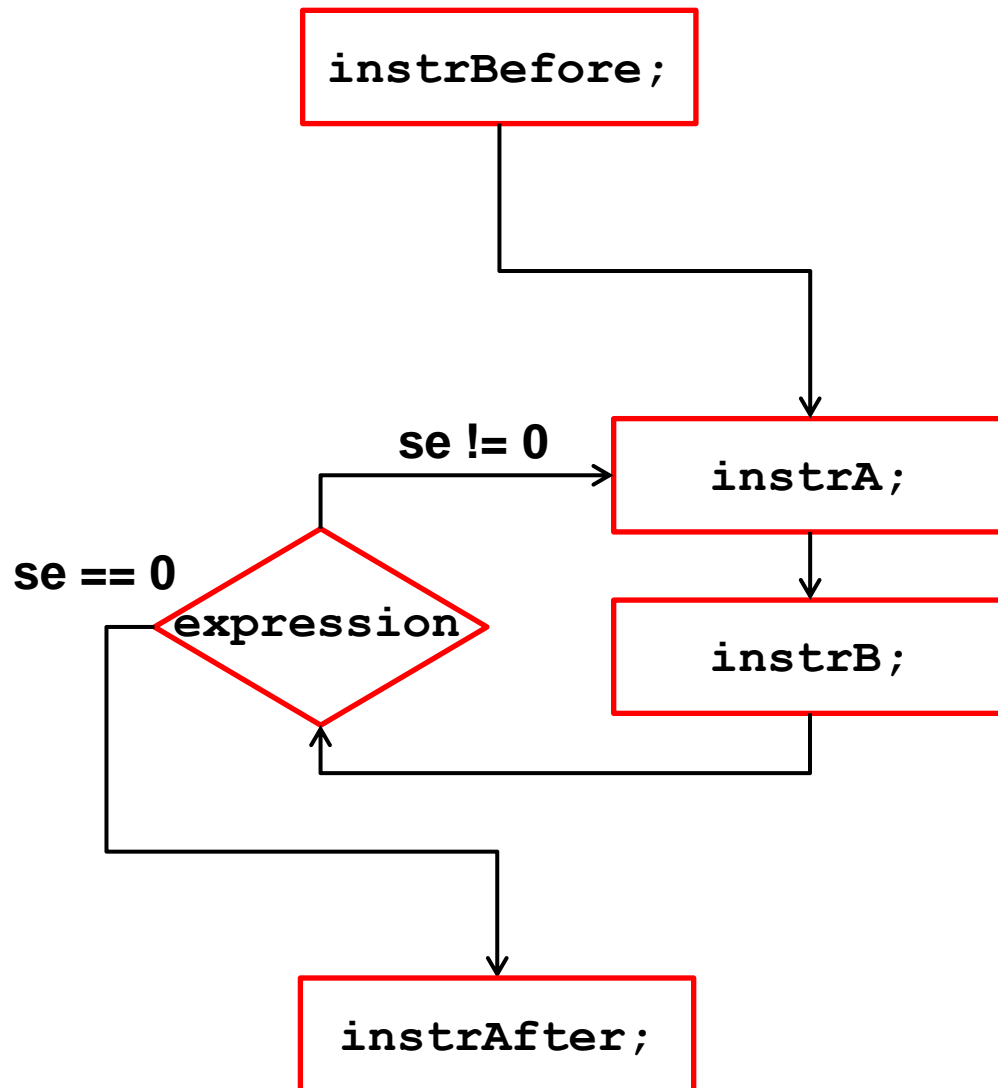


```
statement  
while (expression)  
    statement
```

- Utile per garantire che valori acquisiti con **scanf** soddisfino certi prerequisiti
- **NB:** **do-while** richiede il **;** in **while(expression) ;** il **while** no
- **NB:** come per **if**, se **statement** contiene più istruzioni, va delimitato tra **{ }**



## do-while



```
instrBefore;
```

```
do {
```

```
    instrA;
```

```
    instrB;
```

```
}
```

```
while (expression);
```

```
instrAfter;
```



## Quando utilizzare `while` e quando `do-while` ?

`while` e quando `do-while` sono in teoria interscambiabili ma:

- Va sempre preferita la soluzione che evita di ripetere codice

```
/* Costrutto INAPPROPRIATO */  
  
scanf("%d" , &a);  
while (a >= min && a<= MAX) {  
    scanf("%d" , &a);  
}
```

```
/* Costrutto APPROPRIATO */  
  
do {  
    scanf("%d" , &a);  
} while (a >= min && a<= MAX);
```



## Quando utilizzare `while` e quando `do-while` ?

`while` e quando `do-while` sono in teoria interscambiabili ma:

- Va sempre preferita la soluzione che evita di ripetere codice

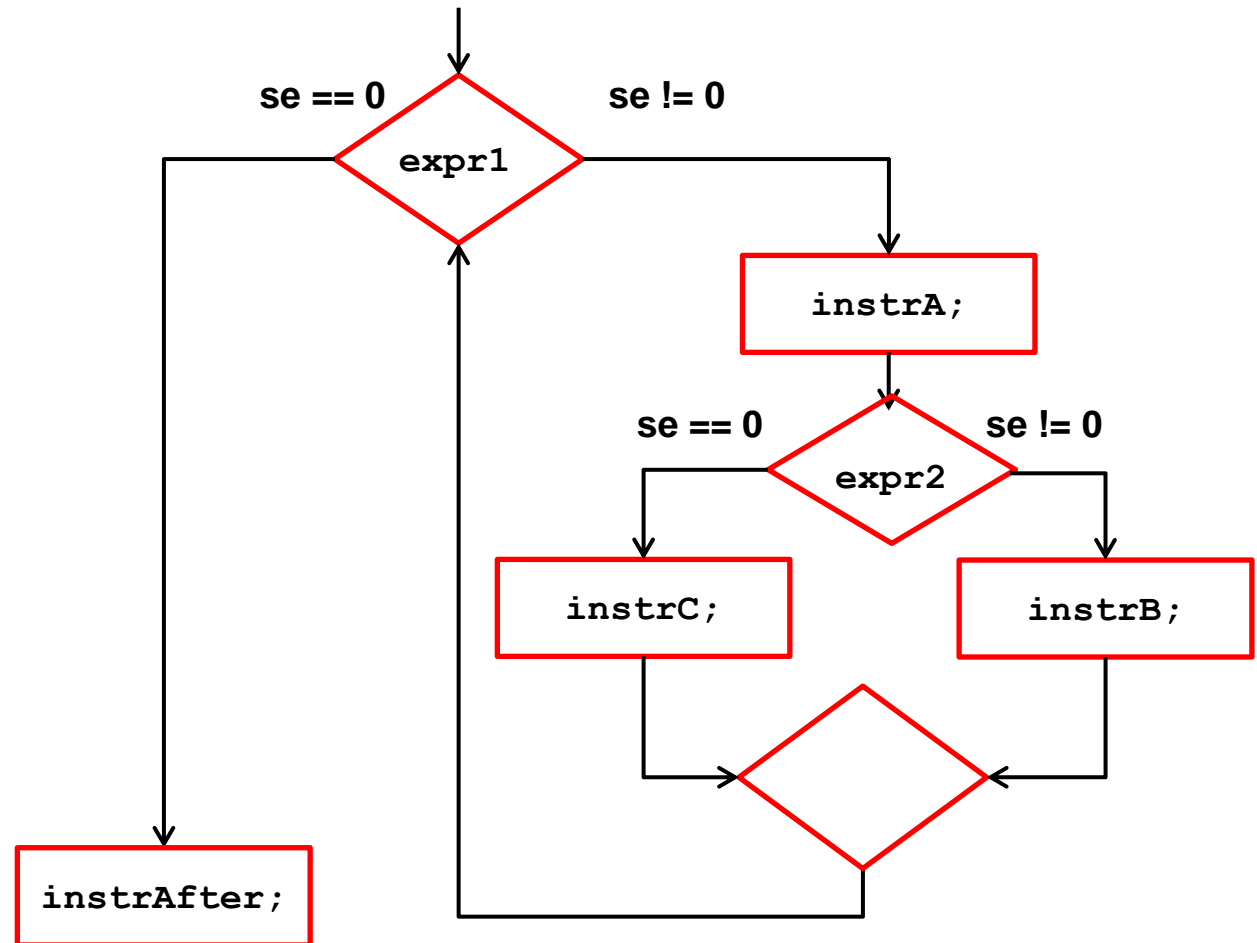
```
/* Costrutto INAPPROPRIATO */  
do {  
    scanf("%d" , &a) ;  
    if(a != STOP)  
        <OPERAZIONI>  
} while (a != STOP) ;
```

```
/* Costrutto APPROPRIATO */  
scanf("%d" , &a) ;  
while (a != STOP) {  
    <OPERAZIONI>  
    scanf("%d" , &a) ;  
}
```



## Cicli Annidati

Ovviamente anche il corpo del **while** può contenere altri costrutti (**while** / **if** o altri che vedremo poi)



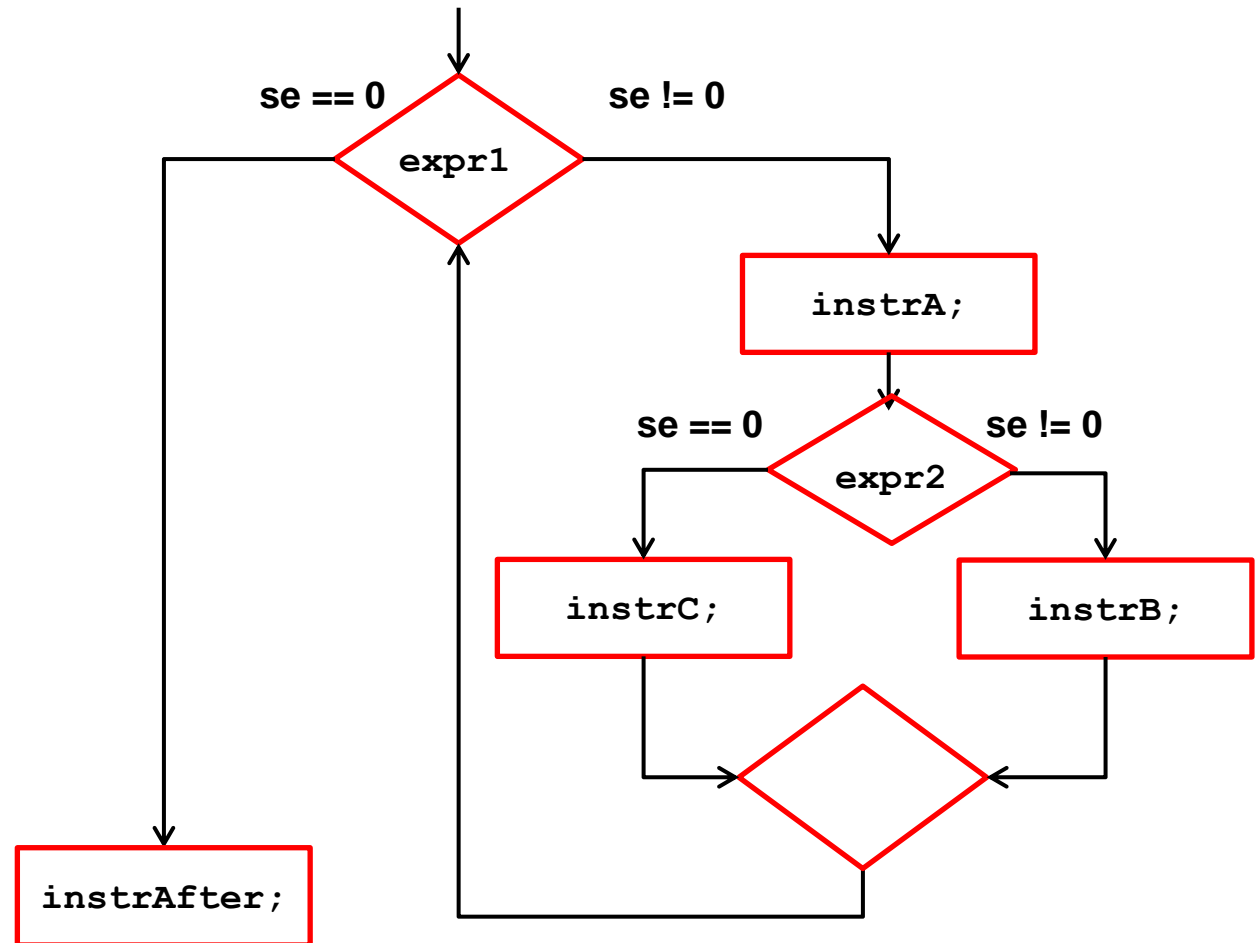




## Cicli Annidati

Ovviamente anche il corpo del **while** può contenere altri costrutti (**while** / **if** o altri che vedremo poi)

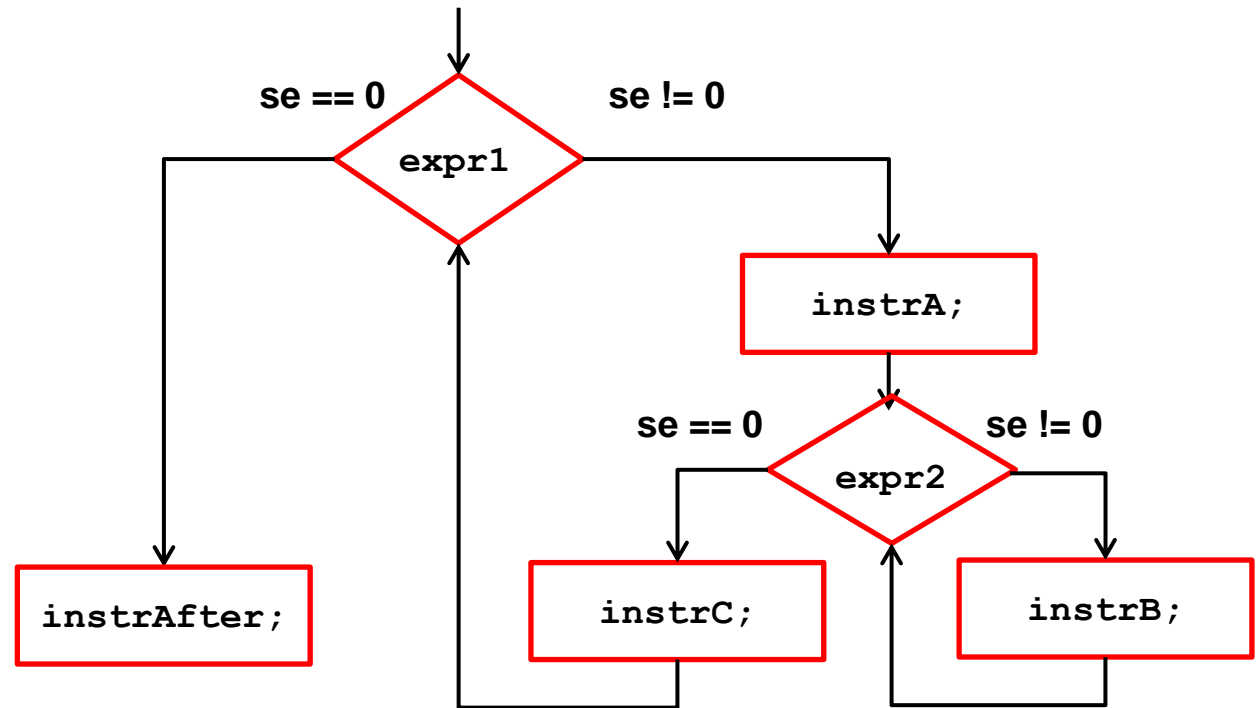
```
while (expr1)
{
    instrA;
    if (expr2)
        instrB;
    else
        instrC;
}
instrAfter;
```





## Cicli Annidati

Ovviamente anche il corpo del **while** può contenere altri costrutti (**while** / **if** o altri che vedremo poi)

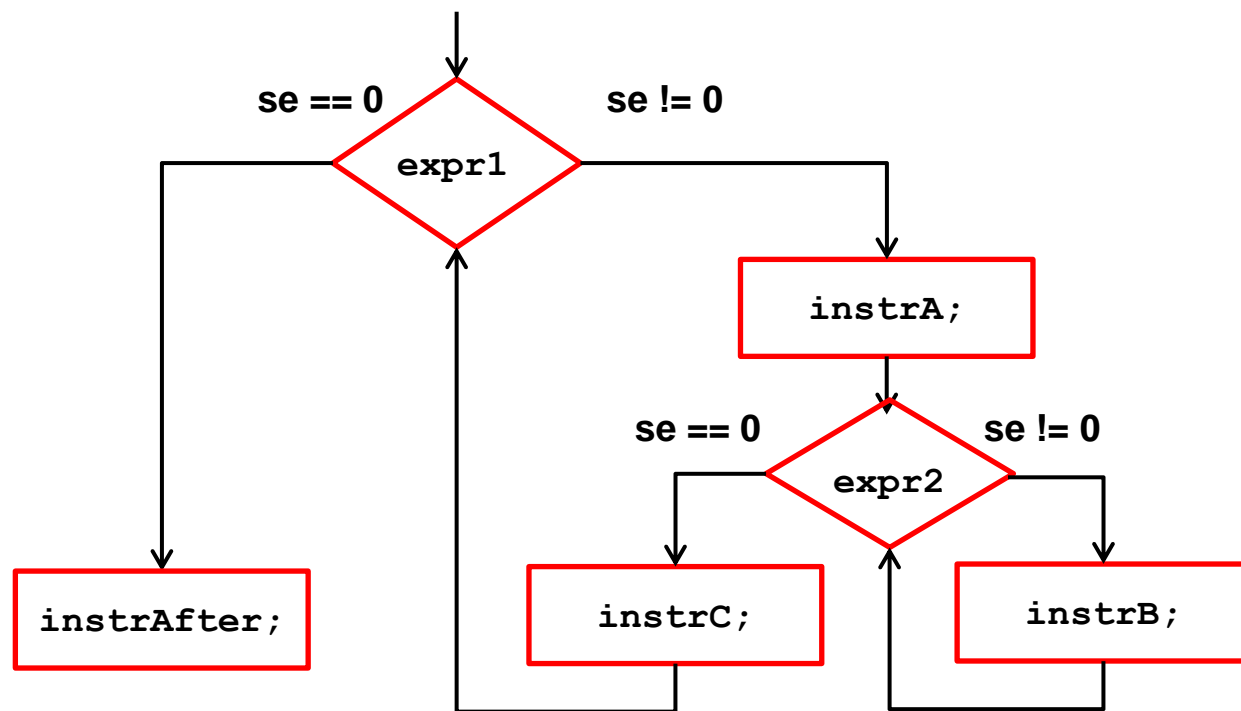




## Cicli Annidati

Ovviamente anche il corpo del **while** può contenere altri costrutti (**while** / **if** o altri che vedremo poi)

```
while (expr1)
{
    instrA;
    while (expr2)
        instrB;
    instrC;
}
instrAfter;
```





## Esercizi per casa

- Scrivere un programma che richiede all'utente una sequenza di caratteri minuscoli e ne stampa il corrispettivo maiuscolo (fino a quando l'utente non inserisce '#')
- Suggerimento: 'a' = 97 e 'A' = 65 ..... 'z' = 122 e 'Z' = 90...la differenza è sempre 32!!!



## Esercizi per casa

- Scrivere un programma che richieda all'utente di inserire due interi e ne calcola il massimo comune divisore.
- Suggerimento:  $x$  è divisore di  $y$  se e solo se  $y \% x == 0$



## Costrutto Iterativo: **for** sintassi

```
for(init_instr; expression; loop_instr)  
    statement
```

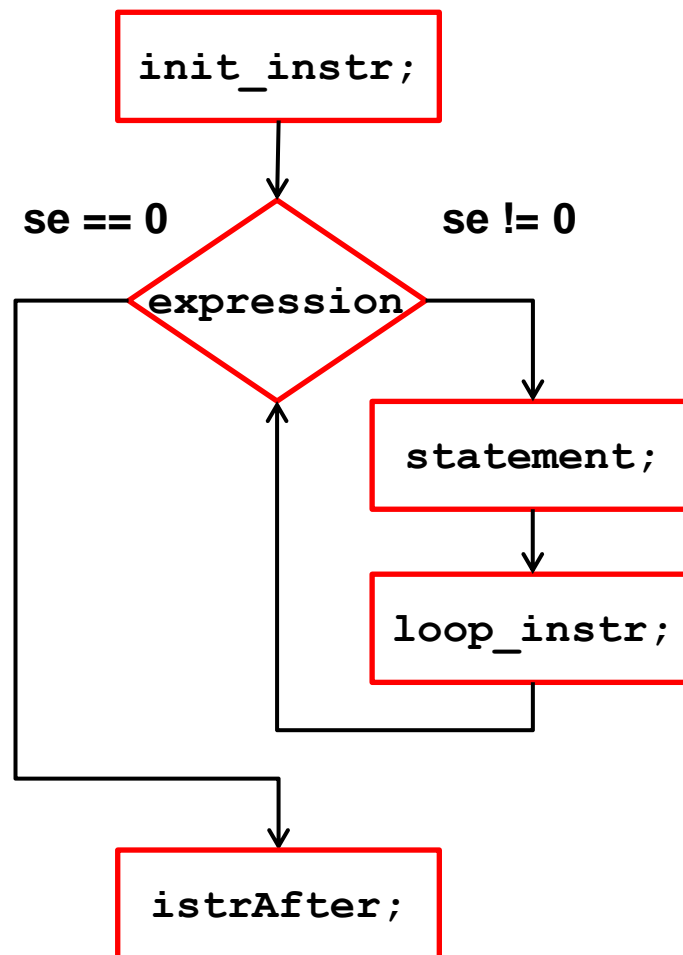
- **for** è un costrutto iterativo, equivalente al **while**
  - **for** keyword
  - **init\_instr** istruzione (di inizializzazione)
  - **expression** espressione booleana
  - **loop\_instr** istruzione (di loop)
  - **statement** corpo del ciclo
- **NB:** se **statement** contiene più istruzioni, richiede { }



## Costrutto Iterativo: **for** esecuzione

**for**(init\_instr; expression; loop\_instr)  
statement

1. Esegue **init\_instr**
2. Valuta **expression**
3. Se vera, esegue **statement**, se falsa, termina il loop.
4. Al termine di **statement** esegue **loop\_instr**
5. Valuta **expression**





## for vs while

```
for (init_instr; expression; loop_instr)  
    statement
```



```
init_instr;  
while (expression)  
{  
    statement;  
    loop_instr;  
}
```

Utile per cicli regolati da una «**variabile di loop**»

- inizializzata con `init_instr`
- Incremento regolato da `loop_instr`





## for vs while

```
...  
i = 0;  
while (i < 100) {  
    //statement  
    i++;  
}  
...
```



## for vs while

```
...  
i = 0;  
while (i < 100) {  
    //statement  
    i++;  
}  
...
```

```
...  
for (i=0; i<100; i++)  
    //statement  
...
```

Il **for**, nei cicli regolati da variabile di loop

- ha una stesura più compatta
- mette in evidenza la variabile di loop e come questa evolve



## for vs while

...

```
scanf("%d", &a);
```

```
while (a < 0)
```

```
    scanf("%d", &a);
```

...



## for vs while

```
...  
scanf("%d", &a);  
while (a < 0)  
    scanf("%d", &a);  
...
```

```
...  
scanf("%d", &a);  
for( ; a < 0; )  
    scanf("%d", &a);  
...
```

Nei cicli senza variabile di loop è comunque possibile usare il **for**, lasciando vuote `loop_instr` e `init_instr`



## for vs while

```
...  
scanf("%d", &a);  
while (a < 0)  
    scanf("%d", &a);  
...
```

```
...  
scanf("%d", &a);  
for( ; a < 0; )  
    scanf("%d", &a);  
...
```

Nei cicli senza variabile di loop è comunque possibile usare il **for**, lasciando vuote `loop_instr` e `init_instr`

Altra soluzione (tecnicamente possibile ma inusuale)

```
...  
for(scanf("%d", &a); a < 0; scanf("%d", &a))  
...  
...
```



## Esempio `for`

- Stampare i primi 100 numeri



## Esempio for

- Stampare i primi 100 numeri  

```
for ( j = 0;  j < 100;  j++)  
    printf( "%d", j );
```



## Esempio for

- Stampare i primi 100 numeri  

```
for ( j = 0; j < 100; j++)  
    printf( "%d", j );
```
- Stampare i quadrati perfetti minori di L





## Esempio for

- Stampare i primi 100 numeri  
`for ( j = 0; j < 100; j++)`  
`printf( "%d", j );`
- Stampare i quadrati perfetti minori di L  
`for( n = 1; n*n < L; n++ )`  
`printf("%d", n*n);`



## Esempio for

- Stampare i primi 100 numeri  

```
for ( j = 0; j < 100; j++)  
    printf( "%d", j );
```
- Stampare i quadrati perfetti minori di L  

```
for( n = 1; n*n < L; n++ )  
    printf("%d", n*n);
```
- Scrivere una soluzione basata su **for** per gli esercizi
  - Le tabelline,
  - Le tabelline il triangolo inferiore



## Costrutto `switch-case` sintassi

- `switch`, `case`, `default` keywords
- `int_expr` espressione a valori integral (char o int)
- `constant-expr1` numero o carattere
- `default` opzionale

```
switch (int_expr) {  
    case constant-expr1:  
        statement1  
    case constant-expr2:  
        statement2  
    ...  
    case constant-exprN:  
        statementN  
    default : statement  
}
```



## Costrutto `switch-case` sintassi

- **NB:** `int_expr` può contenere variabili
- **NB:** `constant-expr1` non può contenere una variabile,
- **NB:** a differenza di `if`, `while` e `for`,
  - `int_expr` non è un'espressione booleana
  - Non occorre delimitare gli `statement` tra `{ }`, anche nel caso contengano più istruzioni. Questi sono delimitati dal case seguente

```
switch (int_expr) {  
    case constant-expr1:  
        statement1  
    case constant-expr2:  
        statement2  
    ...  
    case constant-exprN:  
        statementN  
    default : statement  
}
```



## Costrutto `switch-case` esecuzione

1. Viene valutata `int_expr`
2. Si controlla se `int_expr` è **uguale** a `constant-expr1`
3. Se sono **uguali** eseguo **`statement1`**, ed in **cascata**, **tutti gli `statement` dei `case` seguenti** (senza verifiche, incluso lo `statement` di **`default`**)
4. Altrimenti controllo se **`expression` è uguale a `constant-expr2`** ...
5. Eseguo lo `statement` di **`default`** [se presente]

```
switch (int_expr) {  
    case constant-expr1:  
        statement1  
    case constant-expr2:  
        statement2  
        ...  
    case constant-exprN:  
        statementN  
    default : statement  
}
```



## Note switch-case

Esempio di utilizzo di **switch**

```
.....  
scanf ("%c" , &a) ;  
switch (a) {  
    case 'A' : nA++ ;  
    case 'E' : nE++ ;  
    case 'O' : nO++ ;  
    default : nCons++ ;  
  
}
```

- ```
.....
```
- Se `a == 'A'` , verranno incrementate `nA` , `nE` , `nO` , `nCons` ;
  - Se `a == 'E'` , verranno incrementate `nE` , `nO` , `nCons` ;
  - Se `a == 'O'` , verranno incrementate `nO` , `nCons` ;
  - Se `a == 'K'` , verranno incrementa `nCons` ;



## Note switch-case

Per evitare l'**esecuzione in cascata** alla prima corrispondenza trovata, occorre inserire negli statements opportuni la keyword **break**

```
scanf ("%c" , &a) ;  
switch (a) {  
    case 'A' : nA++; break;  
    case 'E' : nE++; break;  
    case 'O' : nO++; break;  
    default : nCons++;  
}
```

- Se `a == 'A'` , verrà incrementata `nA` ;
- Se `a == 'E'` , verrà incrementata `nE` ;
- Se `a == 'O'` , verrà incrementata `nO` ;
- Se `a == 'K'` , verrà incrementa `nCons` ;



## Esercizi per casa

- Scrivere un programma che opera come una calcolatrice: richiede due operandi ed un operatore  $+$   $-$   $*$   $/$   $\%$  e restituisce il risultato a schermo





## break e continue

- L'istruzione **break** termina l'esecuzione dei seguenti costrutti
  - **while**, **do while** e **for** (costrutti iterativi)
  - **switch** (evita l'esecuzione di tutti i casi in cascata)
- L'istruzione **continue** all'interno di un costrutto iterativo passa direttamente all'iterazione seguente, interrompendo quella corrente.
  - **continue** può essere utilizzato solo nei cicli iterativi, i.e.: **while**, **do while**, **for**.



## Cosa fa?

```
for (i=0; i<10; i++) {  
    scanf ("%d" , &x) ;  
    if (x < 0)  
        break ;  
    printf ("%d" , x) ;  
}
```



## Cosa fa?

```
for (i=0; i<10; i++) {  
    scanf ("%d" , &x) ;  
    if (x < 0)  
        break ;  
    printf ("%d" , x) ;  
}
```

Richiede **al massimo** 10 numeri e ne stampa il valore inserito. Le acquisizioni terminano anticipatamente se viene inserito un valore negativo. Il valore negativo non viene stampato a schermo.

**N.B** il **break** interrompe comunque il costrutto iterativo (**for**) anche se si trova all'interno dell' **if**



## Cosa fa?

```
i = 0;
while (i < 10) {
    scanf ("%d", &x) ;
    if (x < 0)
        continue;
    printf ("%d", x) ;
    i++;
}
```



## Cosa fa?

```
i = 0;
while (i < 10) {
    scanf ("%d" , &x) ;
    if (x < 0)
        continue;
    printf ("%d" , x) ;
    i++;
}
```

Richiede numeri (**anche all'infinito**) fino a quando non ne vengono inseriti 10 positivi. Stampa a schermo il valore inserito di ogni positivo. Per i valori negativi non viene stampato il valore inserito e nemmeno incrementata `i` (il **continue** fa saltare tutte le successive istruzioni)



## Cosa fa?

```
for (i=0; i<10; i++) {  
    scanf ("%d" , &x) ;  
    if (x < 0)  
        continue ;  
    printf ("%d" , x) ;  
}
```



## Cosa fa?

```
for (i=0; i<10; i++) {  
    scanf ("%d" , &x) ;  
    if (x < 0)  
        continue ;  
    printf ("%d" , x) ;  
}
```

Richiede **esattamente** 10 numeri e ne stampa il valore inserito. Le acquisizioni **non** terminano se viene inserito un valore negativo, però non viene stampato il valore inserito (il **continue** fa saltare alla successiva esecuzione)

La **loop\_expr** non viene saltata dal **continue**.  
È una particolarità del **for**



## Alternative a **break** e **continue**

- Utilizzo di cicli con variabili **flag (o sentinella)** per terminare anticipatamente l'esecuzione del ciclo
  - Una variabile che assume un valore 0 / 1 a seconda che si verifichino o meno alcune condizioni durante l'esecuzione





## ESEMPIO: Alternative a `break` e `continue`

Es: Scrivere un ciclo che richiede una serie di valori interi e li associa alla variabile intera **a** e stampa a schermo

- non più di N richieste
- saltando i valori negativi inseriti
- interrompendo l'elaborazione al primo valore nullo incontrato



## ESEMPIO: soluzione con `break` e `continue`

```
for (i = 0 ; i < N ; i++ ) {  
    printf("immetti un intero > 0 ");  
    scanf("%d", &n) ;  
    if (n < 0)  
        continue;  
    if (n == 0)  
        break;  
    printf("%d", n) ;  
    .. /*elabora i positivi */  
}
```



## ESEMPIO: soluzione con variabile di flag

```
int n, i, flag;
flag = 0//diventa 1 quando inserisco un neg.
for(i =0; i <= N && flag == 0; i++)
{
    printf("\ninserire n: ");
    scanf("%d", &n);
    if(n==0)
        flag = 1;
    else
        if(n > 0) {// se n<0 il progr. non fa
nulla
                printf(" %d", n);
                .. /*elabora i positivi */
            }
}
```



## Importanza delle variabili di flag

Es: Scrivere un ciclo che richiede una serie di valori interi e li associa alla variabile intera **a** e stampa a schermo

- non più di 10 richieste
- saltando i valori negativi inseriti
- interrompendo l'elaborazione al primo valore nullo incontrato
- **Al termine, stampare un messaggio qualora fossero stati inseriti 10 numeri positivi**



## Importanza delle variabili di flag

```
int n, i, flag;
flag = 0; //diventa 1 quando inserisco un neg.
for(i =0; i <= 10 && flag == 0; i++)
{
    printf("\ninserire n: ");
    scanf("%d", &n);
    if (n==0)
        flag = 1;
    else
        if (n > 0)
            printf(" %d", n);
    /*eventuali altre istruzioni per i positivi*/
}
if(flag == 0)
    printf("\n tutti non nulli");
```



## Importanza delle variabili di flag

```
int n, i, flag;
flag = 0; //diventa 1 quando inserisco un neg.
for(i = 0; i <= 10 && flag == 0; i++)
{
    printf("\ninserire n: ");
    scanf("%d", &n);
    if (n==0)
        flag = 1;
    else
        if (n > 0)
            printf(" %d", n);
    /*eventuali altre istruzioni per i positivi*/
}
if(flag == 0)
    printf("\n tutti non nulli");
```

se flag è rimasto zero vuol dire che nel ciclo sopra non è mai stato inserito un valore nullo, altrimenti sarebbe diventato 1



## Importanza delle variabili di flag

```
int n, i, flag;
flag = 0; //diventa 1 quando inserisco un neg.
for(i = 0; i <= 10 && flag == 0; i++)
{
    printf("\ninserire n: ");
    scanf("%d", &n);
    if (n==0)
        flag = 1;
    else
        if (n > 0)
            printf(" %d", n);
    /*eventuali altre istruzioni per i positivi*/
}

if(flag == 0)
    printf("\n tutti non nulli");
```

Se avessi usato il break al posto della variabile di flag non avrei potuto determinare se il ciclo sopra si fosse interrotto per via del break o se fosse terminato normalmente



# Le costanti in C





## Le Costanti

- Dichiarando una costante viene associato **stabilmente** un valore ad un identificatore

- *Esempi:*

```
const float PiGreco = 3.14;
```

```
const float PiGreco = 3.1415, e = 2.718;
```

```
const int N = 100, M = 1000;
```

```
const char CAR1 = 'A', CAR2 = 'B';
```

- Note:
  - Il compilatore segnala come errore ogni assegnamento a una costante nella parte eseguibile
    - Perché il valore di una costante DEVE ESSERE COSTANTE



## Le Costanti (cnt)

- Usare le costanti è utile perché:
  - Il nome della costante suggerisce il significato del suo valore
  - Permette di parametrizzare i programmi
    - riutilizzabili al cambiare di circostanze esterne
- *Es:* in un programma dichiaro:  
`const float PiGreco = 3.14;`  
poi uso `PiGreco` più volte nella parte esecutiva;  
Se viene richiesto una precisione diversa devo solo modificare la dichiarazione  
`const float PiGreco = 3.1415;`



# Errori più comuni



## Gli Errori

**Gli errori** possono essere di due tipi:

- Errori di sintassi, rilevabili a **compile-time**.
- Errori logici rilevabili a **run-time**.

Gli errori rilevabili a **compile-time** contengono **istruzioni che il compilatore non è in grado di risolvere**, per questo manda dei segnali di errore.

- Controllate sempre il log del compilatore!

Gli errori a **run-time** si **manifestano durante l'esecuzione** e possono causare:

- L'interruzione del programma
- Comportamenti inaspettati
- Sono più difficili da rilevare



## Errori Frequenti Rilevabili a Compile-Time

- Dimenticare un ; manda l'errore alla riga seguente (trova due istruzioni in una sola riga)

error: expected ';' before 'printf'

- Variabile non inizializzata?

error: 'iniz\_nome' undeclared (first use in this function)

- typo?

error: 'prin' undeclared (first use in this function)



## Errori Frequenti Rilevabili a Run-Time

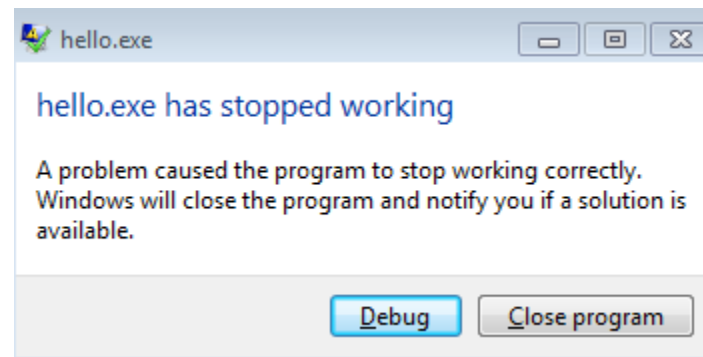
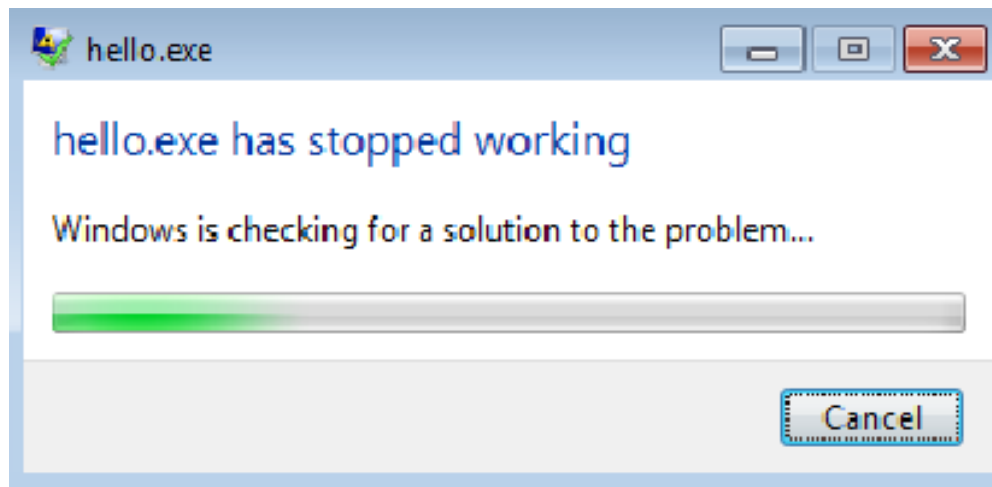
Dimenticare `&` nelle variabili della `scanf` :  
il compilatore non lo rileva! Errore rilevabile a **run-time**,  
quando il valore dallo `stdin` viene scritto in una cella di  
indirizzo «sbagliato»

```
int a = 7;
```

```
scanf ("%d" , a) ;
```

 scrive nella cella all'indirizzo 7 .

Tipicamente se ne accorge Microsoft appena ci provate.





## Errori Frequenti Rilevabili a Run-Time

Confondere l'assegnamento con il confronto

```
int a = 10;  
    if (a = 7)  
        printf("Vero");  
    else  
        printf("Falso");
```



## Errori Frequenti Rilevabili a Run-Time

Confondere l'assegnamento con il confronto

```
int a = 10;  
    if (a = 7)  
        printf("Vero");  
    else  
        printf("Falso");
```

Stampa sempre **Vero** perché `(a = 7)` è un assegnamento e l'operazione diventa **1** cioè, assegnamento eseguito con successo!





## Errori Frequenti Rilevabili a Run-Time

Sbagliare lo specificatore di formato in una scanf o printf

```
void main() {  
    int x;  
    printf("inserire x: ");  
    scanf("%f", &x);  
    printf("\nx = %d", x);  
}
```



## Errori Frequenti Rilevabili a Run-Time

Sbagliare lo specificatore di formato in una scanf o printf

```
void main() {  
    int x;  
    printf("inserire x: ");  
    scanf("%f", &x);  
    printf("\nx = %d", x);  
}
```

```
inserire x: 4  
x = 1082130432  
Process returned 15 (0xF)    execution time : 5.226 s  
Press any key to continue.
```



## Errori Frequenti Rilevabili a Run-Time

Sbagliare lo specificatore di formato in una scanf o printf

```
void main() {  
    float x;  
    printf("inserire x: ");  
    scanf("%f", &x);  
    printf("\nx = %d", x);  
}
```



## Errori Frequenti Rilevabili a Run-Time

Sbagliare lo specificatore di formato in una scanf o printf

```
void main() {  
    float x;  
    printf("inserire x: ");  
    scanf("%f", &x);  
    printf("\nx = %d", x);  
}
```

```
inserire x: 4  
x = 1082130432  
Process returned 15 (0xF)    execution time : 5.226 s  
Press any key to continue.
```



## E' invece possibile...

```
void main() {  
    char x;  
    printf("inserire x: ");  
    scanf("%c", &x);  
    printf("\nx = '%c' = %d", x, x);  
}
```



## E' invece possibile...

E' invece possibile stampare i char come interi

```
void main() {  
    char x;  
    printf("inserire x: ");  
    scanf("%c", &x);  
    printf("\nx = '%c' = %d", x, x);  
}
```

```
inserire x: a  
x = 'a' = 97  
Process returned 13 (0xD)    execution time : 3.579 s  
Press any key to continue.  
—
```



## Errori Frequenti Rilevabili a Run-Time: il ; nelle strutture di controllo.

```
int a = 10;  
if(a == 7) ;  
    printf("Vero") ;
```



## Errori Frequenti Rilevabili a Run-Time: il ; nelle strutture di controllo.

Il ; termina un'istruzione e quindi non va messo dopo **if**,  
**while**, **for**, **switch**,

- Se presente, il ; specifica che il costrutto non ha corpo e l'istruzione successiva viene eseguita

```
int a = 10;  
if(a == 7) ;  
    printf("Vero");
```

- Stampa vero perché `printf("Vero");` è fuori dall' **if**
- Il ; viene usato nel **do while** perché il costrutto termina con il **while(expression)** ;





## Errori Frequenti Rilevabili a Run-Time: il ; nelle strutture di controllo.

```
int a = 10;  
if (a == 7) ;  
    printf("Vero") ;  
else  
    printf("Sono nell'else") ;
```



## Errori Frequenti Rilevabili a Run-Time: il ; nelle strutture di controllo.

- Se ci fosse stato un **else** il compilatore avrebbe dato errore: esiste un **else** non associato ad un **if**

```
int a = 10;  
if (a == 7) ;  
    printf("Vero") ;  
else  
    printf("Sono nell'else") ;
```