Strutture di Controllo in C

Informatica B a.a. 2020/2021

Francesco Trovò

francesco1.trovo@polimi.it

22 Settembre 2020

Operatori ed Espressioni Logiche

Algebra di Boole

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

Operatoori Aritmetici

Operatoori Relazionali

Algebra di Boole

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

Es.

- `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 `||`

Tavole 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)	
A	***!A***
0	1
1	0

Tavole 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

Tavole 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 || B` è vero se almeno una delle due è vera

NB: non è un or esclusivo (xor)

disgiunzione (OR)		
`A`	`B`	`A || 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

Esercizio: Tabella di Verità

- `A && (!B || C)`

A	B	C	!B	!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			

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`

Es.

```
!((a >= 5) && (a <= 10))
!(a >= 5) || !(a <= 10)      [De Morgan]
!!(a < 5) || !!(a > 10)      [proprietà >= e <= ]
((a < 5) || (a > 10))        [doppia negazione]
```

Esercizio

- Dimostrare che le seguenti espressioni sono equivalenti:
  - `A || C && !B`
  - `!((B || !C) && !A)`

- Due possibili soluzioni:
  - applicando le leggi di De Morgan cerco di passare da una all'altra
  - calcolo entrambe le tabella di verità e mostro che coincidono (provate a farlo a casa)

Esercizio

- Dimostrare che le seguenti espressioni sono equivalenti
  - `A || C && !B`
  - `!((B || !C) && !A)`
- Dimostrazione:
  - `!((B || !C) && !A)`
  - `(!(B || !C) || !!A)`
  - `!(B || !C) || A`
  - `(!B && C) || A`
  - `A || (!B && C)`
  - `A || (C && !B)`
  - `A || C && !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**:
  - 0 ⇔ **falso**
  - **qualsiasi valore** `!=0` ⇔ **vero**
- Ciò viene utilizzato in pratica (anche se non bello dal punto di vista concettuale) per:
  - **memorizzare in variabili intere risultati di condizioni** (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

istruzione composta: `if else`

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` sono keywords ovvero non possono essere usate in altri ambiti
- **`expression`** espressione booleana (vale 0 o 1)
- **`statement1`** è la sequenza di istruzioni da eseguire (corpo) quando **`expression`** è vera, **`statement0`** quando è falsa

```
if(expression)
    statement1
```

```
if(expression)
    statement1
else
    statement0
```

Costrutto Condizionale: `if`, l'esecuzione

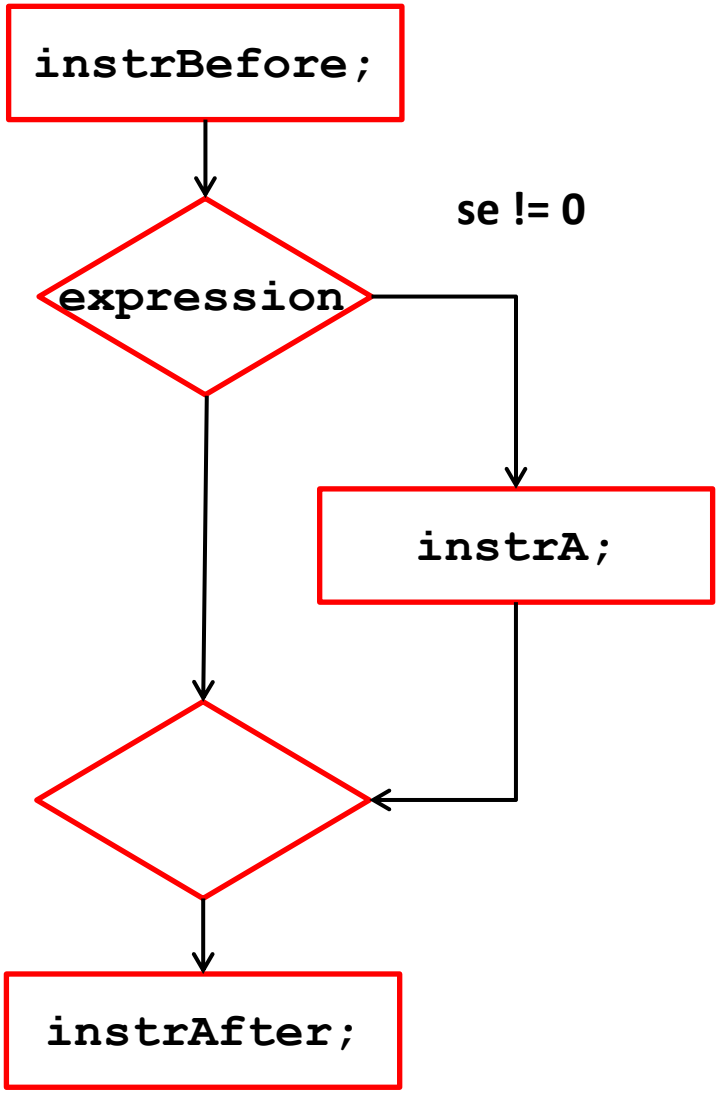
1. Terminata **`instrBefore`**, valuto **`expression`**
2. Se **`expression`** è vera (**`!=0`**), allora eseguo **`statement1`**, altrimenti eseguo **`statement0`** (se è presente **`else`**)
3. Terminato lo statement dell'**`if`**, procedi con **`instrAfter`**, la prima istruzione fuori dall'**`if`**

NB: **`else`** è opzionale

NB: **`if (expression)`** **non** richiede il **`;`** perché l'istruzione non termina dopo **la parentesi**

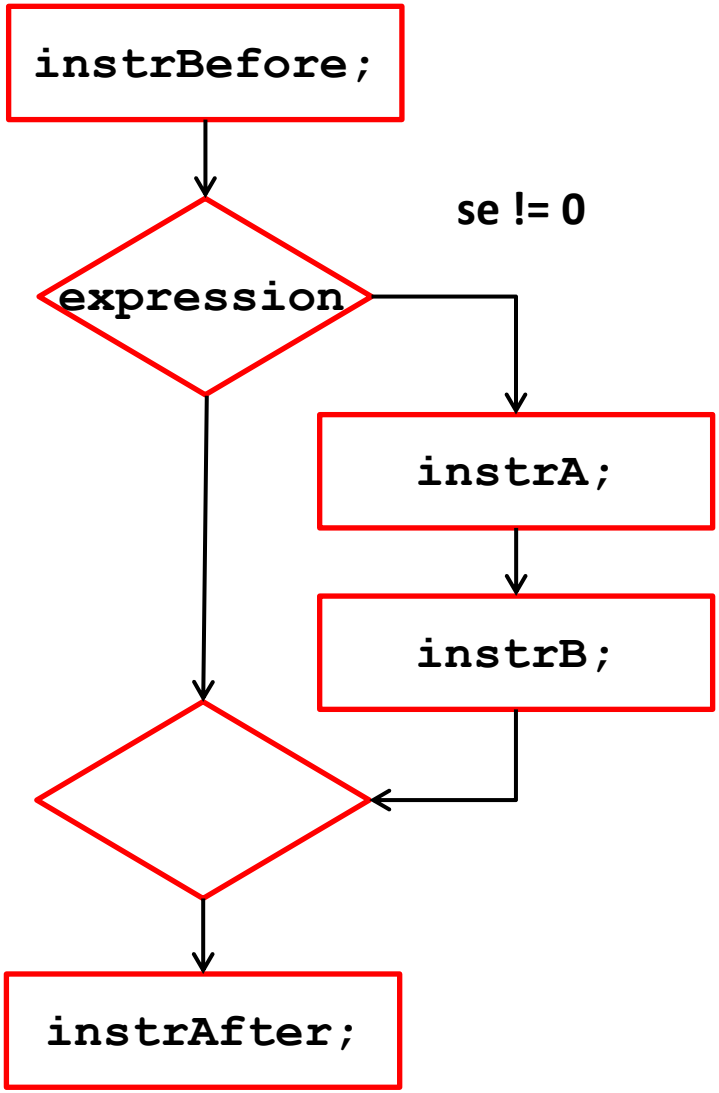
```
instrBefore;
if(expression)
        statement1;
else
        statement0;
instrAfter;
```

Costrutto Condizionale: `if`, l'esecuzione




```
instrBefore;
if(expression)
        instrA;
instrAfter;
```

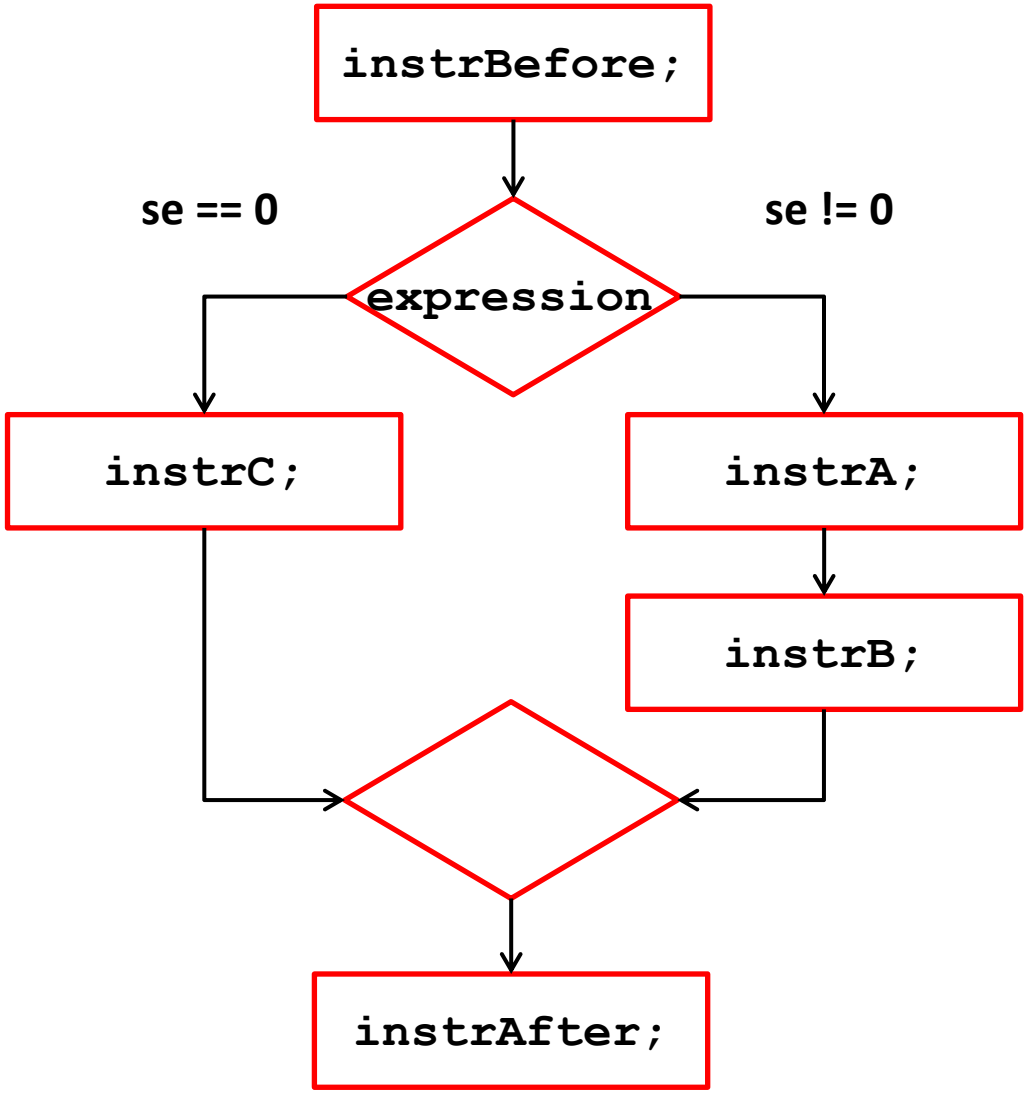
Costrutto Condizionale: `if`, l'esecuzione




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

- Se nel corpo ho più di una istruzione utilizzo le { } per raggrupparle

Costrutto Condizionale: `if`, l'esecuzione




```
instrBefore;
if(expression)          {
        instrA;
        instrB;
        }
else
        instrC;
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);
//Si può farlo senza else?


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

`if` Annidati

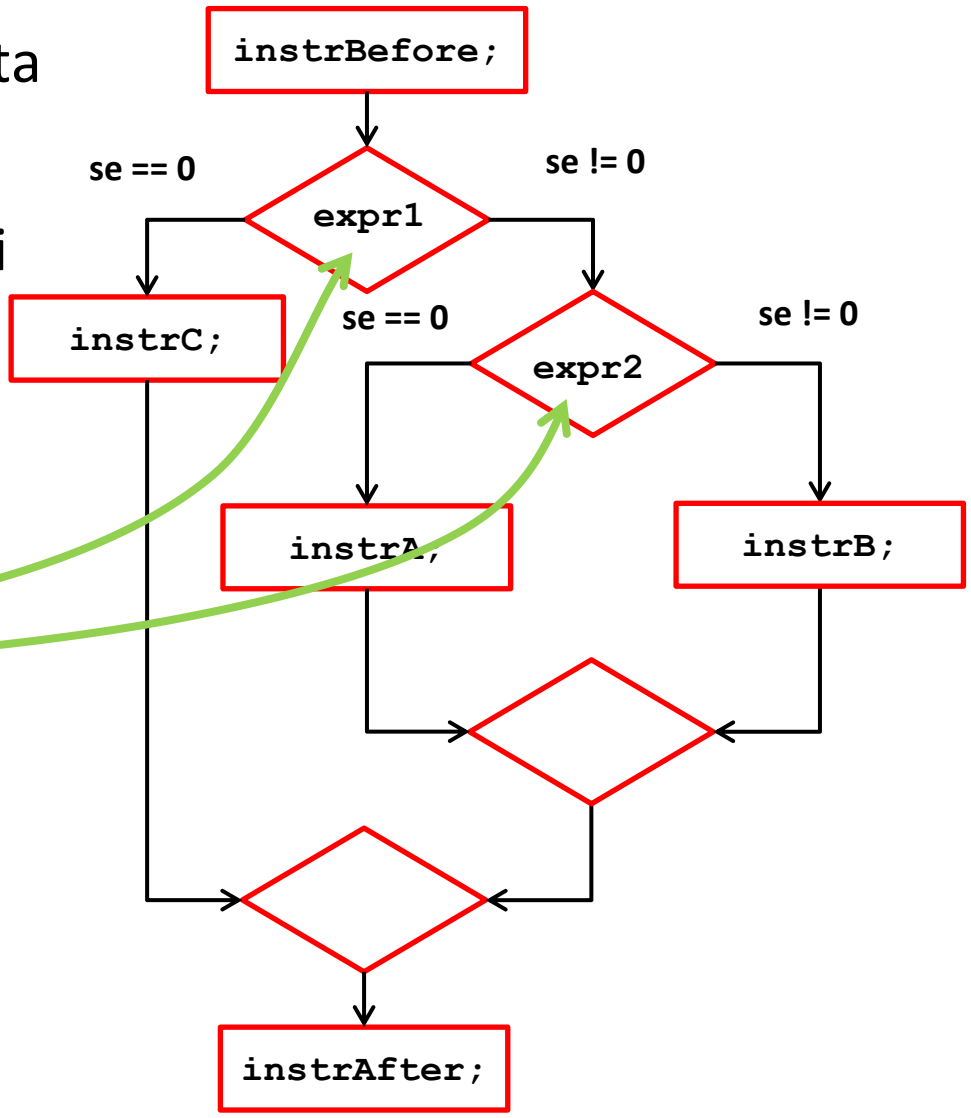
Le istruzioni condizionali possono essere annidate, inserendo un ulteriore **`if`** all'interno di **`statement1`** o **`statement0`**

Es.

```
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");
else
    printf("x non multiplo di 5");
```

`if` Annidati

- Il corpo di un **`if`** (cioè gli **`statement`**) può a loro volta contenere costrutti altri **`if`**
- Si realizzano quindi istruzioni condizionali **annidate**




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

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

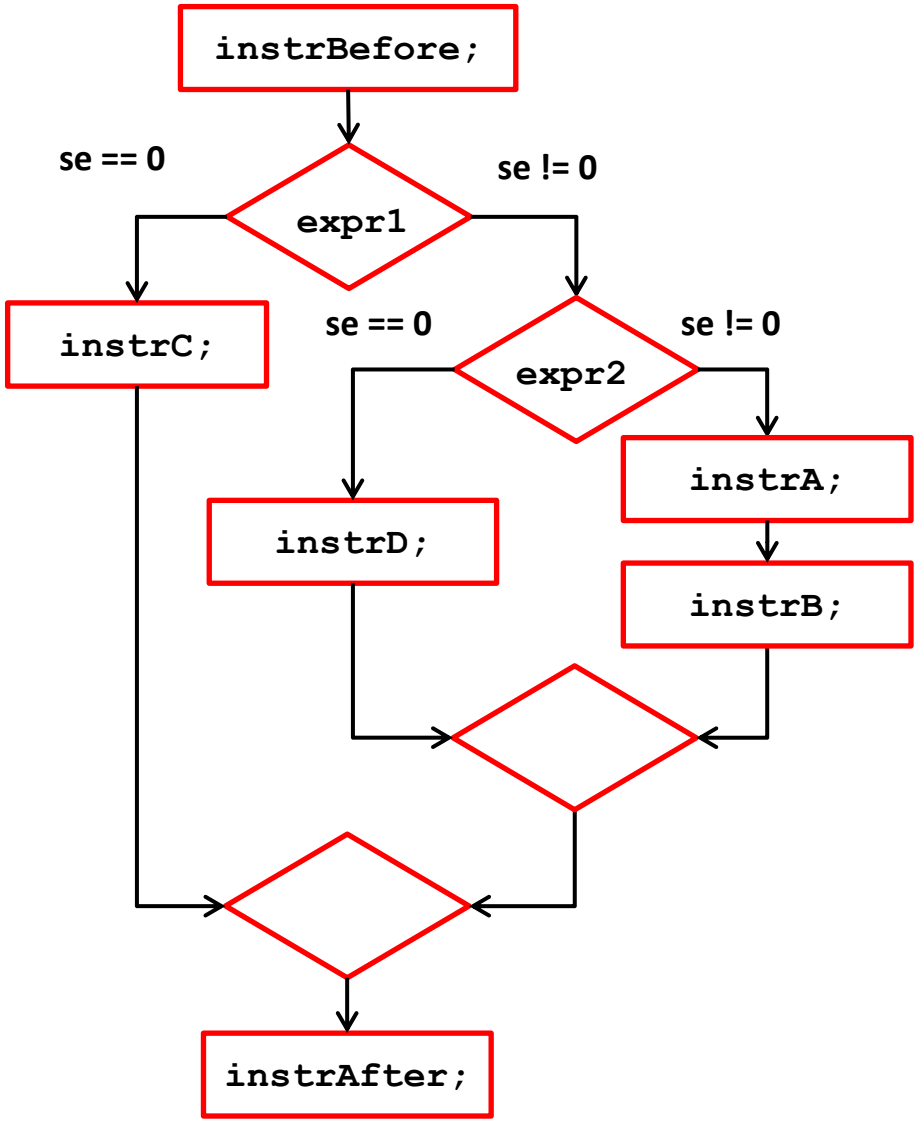
Regola: In caso di costrutti annidati, ed in assenza di parentesi che indichino diversamente, ogni **`else`** viene associato all' **`if`** più vicino

```
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

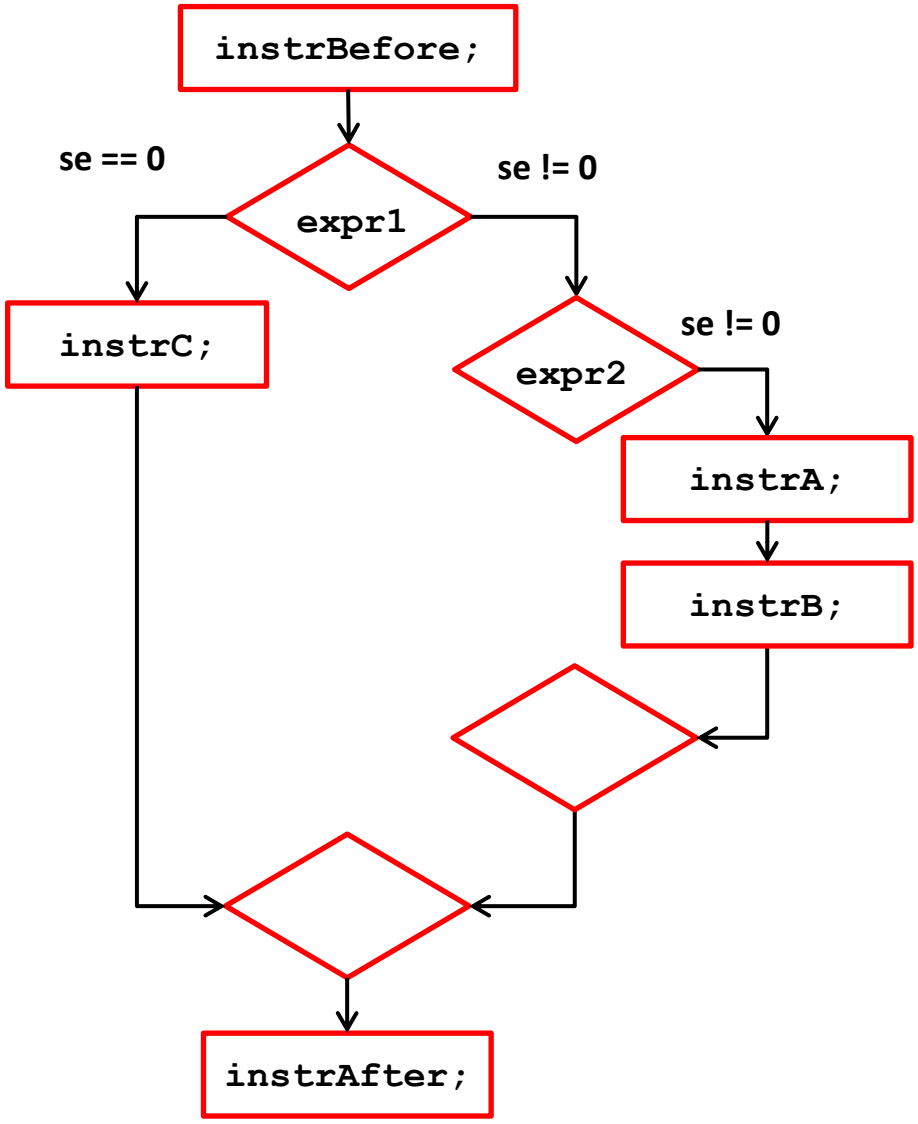
```
instrBefore;
if(expr1)
 if(expr2)
      {instrA;
       instrB;}
  else
      instrD;
else
 instrC;
instrAfter;
```



`if` Annidati

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

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



Esempio: Anno Bisestile

- Scrivere un programma che, inserito un intero positivo, determina se corrisponde ad un anno bisestile, dove un anno è bisestile se:
  - è multiplo di 4, ma non di 100
  - oppure se è multiplo di 400

Soluzione (1): if Annidati

```
#include<stdio.h>
void main(){
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!");
}
```

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

Soluzione (2): Condizioni Composte e Predicati

```
#include<stdio.h>
void main(){
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!");
}
```

Osservazione:

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

Esempio: Massimo

- Scrivere un programma che determina il massimo tra tre numeri inseriti da tastiera e lo scrive a schermo

Soluzione (1): if Annidati

```c
#include<stdio.h>
void main(){
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)
        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);
}
```

Osservazioni:

1. Il numero di annidamenti è n, pari a quanti numeri occorre controllare
2. Le parentesi negli if non sono necessarie in questo caso

Soluzione (2): Condizioni Composte

```
#include<stdio.h>
void main(){
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)

}
```

Osservazioni:

1. Condizioni composte si allungano quando si aggiungono numeri da controllare
2. Il numero di condizioni da valutare per n numeri è n
3. If usati in sequenza
4. 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>
void main()
{
int a,b,c;
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,%d,%d) = %d", a, b, c, max);
}
```

Osservazioni:

1. L'uso della variabile ausiliaria facilita le cose

NB: Non ricorda niente questa soluzione?

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

NB: Analogo all'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)

```
while (expression)
    statement
```

NB: come per **`if`**, se **`statement`** contiene più istruzioni, va delimitato tra **{ }**

Costrutto Iterativo: `while`, l'esecuzione

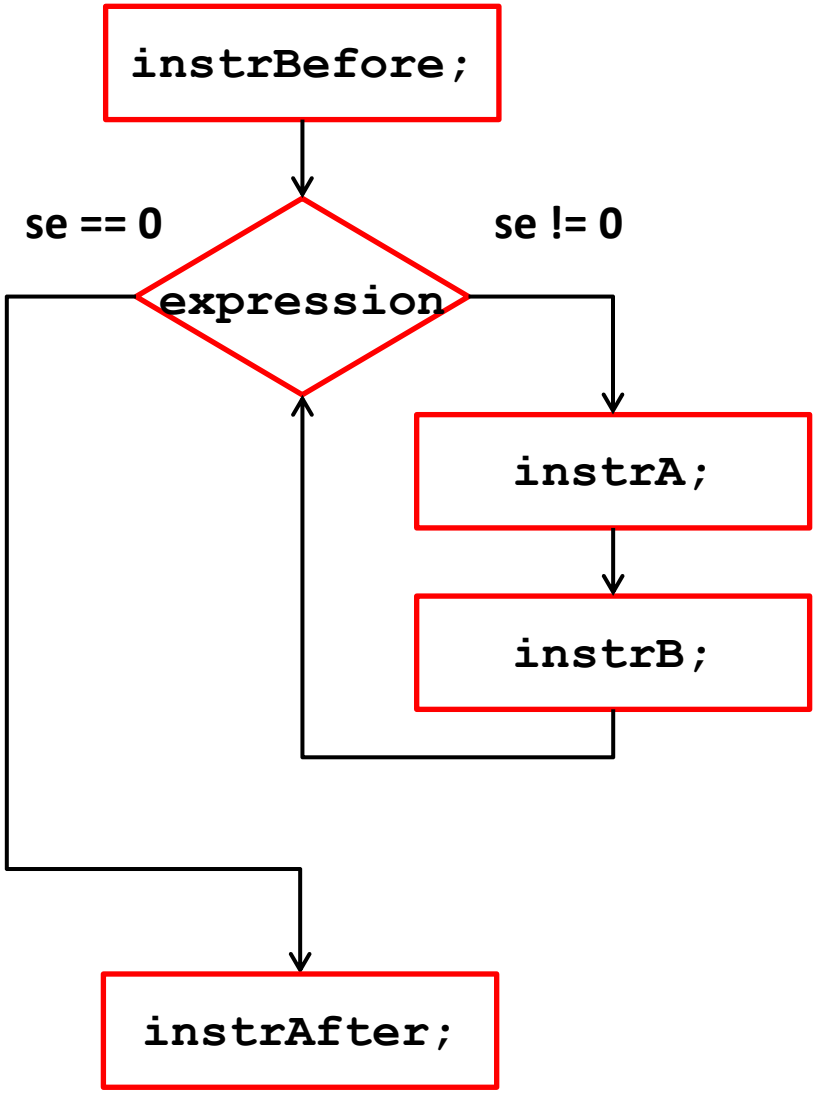
1. Terminata **`instrBefore`** viene valutata **`expression`**
2. Se **`expression`** è vera (o != 0) viene eseguito **`statement`**
3. Al termine, viene valutata nuovamente **`expression`** e la procedura continua finché **`expression`** è falsa (== 0)
4. Uscito dal ciclo, eseguo **`instrAfter`**

```
instrBefore;
while(expression)
        statement;
instrAfter;
```

NB: `while(expression)` **non** richiede il **;** perché l'istruzione non termina dopo la parentesi, ma con lo **`statement;`**

NB: `while(expression);` è un ciclo senza corpo

Costrutto Iterativo: `while`, l'esecuzione




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

Esempio

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

Esempio

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

Avvertenze per il Costrutto `while`

- Il corpo del **`while`** non viene mai eseguito quando **`expression`** risulta falsa al primo controllo
- 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>
void main()
{
        int a = 1;
        while(a > 0)
        {
            printf("\n%d" , 2*a);
            a++;
        }
}
```

Costrutto Iterativo: while

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

# include <stdio.h>
void 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);
}
```

Costrutto Iterativo: while

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

# include <stdio.h>
void 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);
}
```

Homeworks

- Preparare un programma C per giocare a Carta / Sasso / Forbice, richiedendo all'utente di inserire i caratteri `'c'`, `'s'`, `'f'`, controllando anche che il carattere inserito sia ammissibile

- Il controllo deve avvenire almeno una volta, ma in caso di errore dell'utente deve avvenire più volte

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)
3. Viene eseguita l'istruzione successiva al ciclo

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

Proprietà del Costrutto `do-while`

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

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

VS.

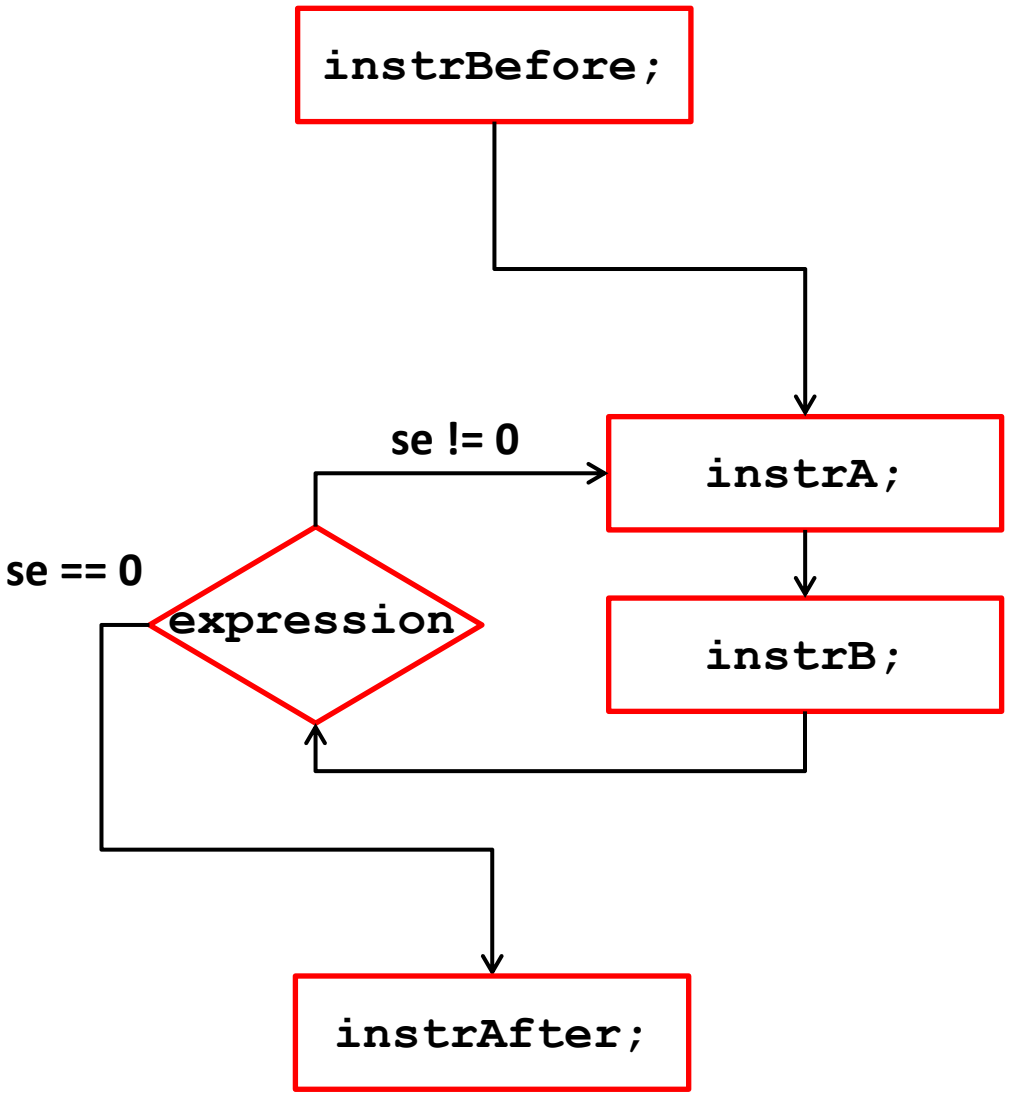
```
statement
while (expression)
   statement
```

- 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 **`{ }`**

Diagramma del `do-while`




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

Homeworks

- Preparare un programma C per giocare a Carta / Sasso / Forbice, richiedendo all'utente di inserire I caratteri `'c'`, `'s'`, `'f'`, controllando anche che il carattere inserito sia ammissibile

- Inserire un controllo nel programma dell'anno bisestile per assicurarsi che il numero inserito da tastiera sia un intero positivo

Teorema di Boehm-Jacopini

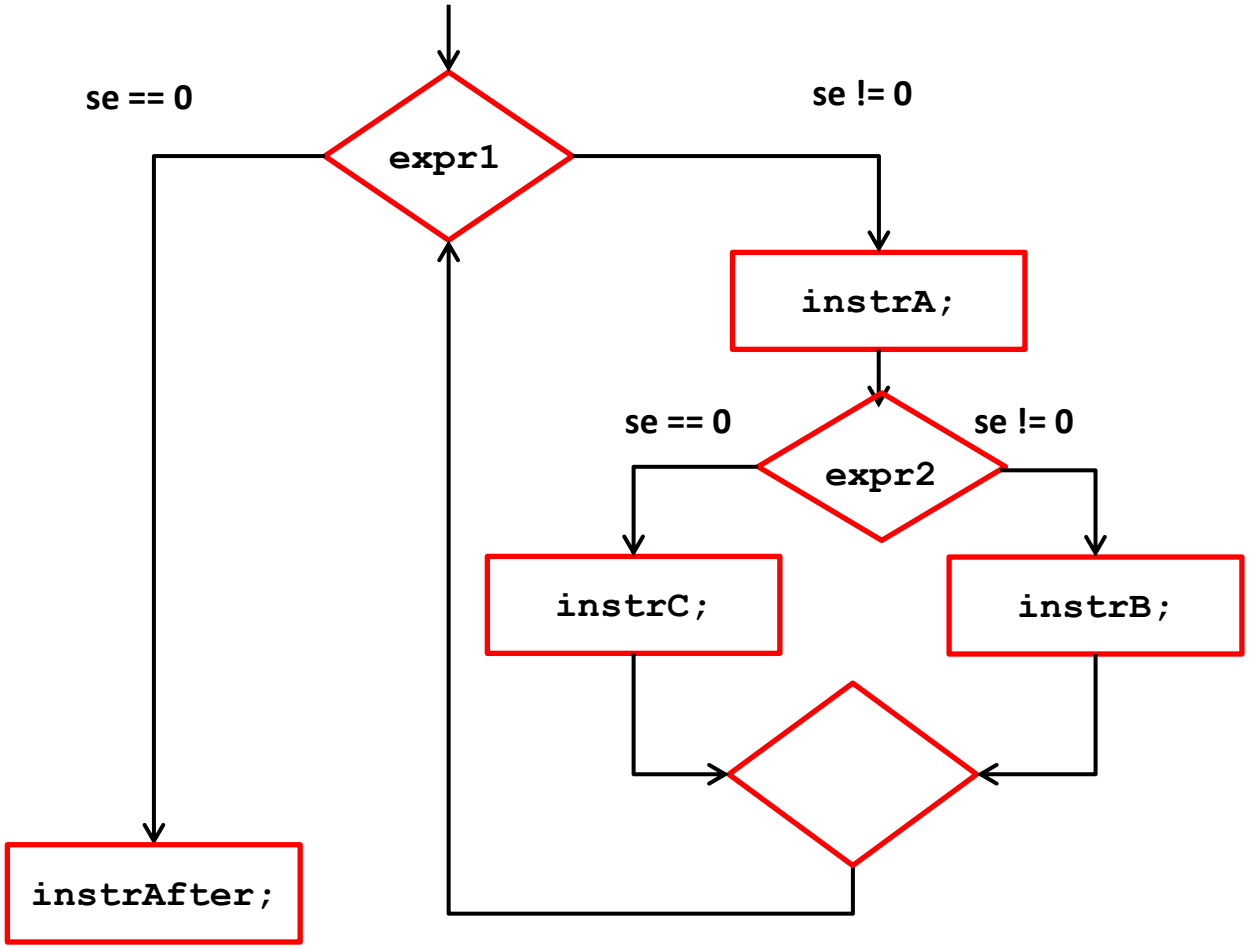
- Le istruzioni **`if`** e **`while`** (e la possibilità di eseguire istruzioni in sequenza) sono equivalenti alle istruzioni che la macchina di Von Neumann che può manipolare il registro Contatore di Programma
- Conseguenze: le istruzioni **`if`** e **`while`** sono complete, ovvero bastano per codificare qualsiasi algoritmo

NB: per praticità e convenienza si usano però molte altre strutture di controllo

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;
```



Homeworks

- 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 il carattere #
- Scrivere un programma che richieda all'utente di inserire due interi e ne calcola il Massimo Comune Divisore (MCD)

Homeworks

- Scrivere un programma che stampa la tabella pitagorica
  - Modificarlo per stampare solo la parte triangolare alta, solo la parte triangolare bassa, oppure solo la diagonale

TABELLINE

x	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	10
2	0	2	4	6	8	10	12	14	16	18	20
3	0	3	6	9	12	15	18	21	24	27	30
4	0	4	8	12	16	20	24	28	32	36	40
5	0	5	10	15	20	25	30	35	40	45	50
6	0	6	12	18	24	30	36	42	48	54	60
7	0	7	14	21	28	35	42	49	56	63	70
8	0	8	16	24	32	40	48	56	64	72	80
9	0	9	18	27	36	45	54	63	72	81	90
10	0	10	20	30	40	50	60	70	80	90	100

Costrutto Iterativo: `for` , la Sintassi

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

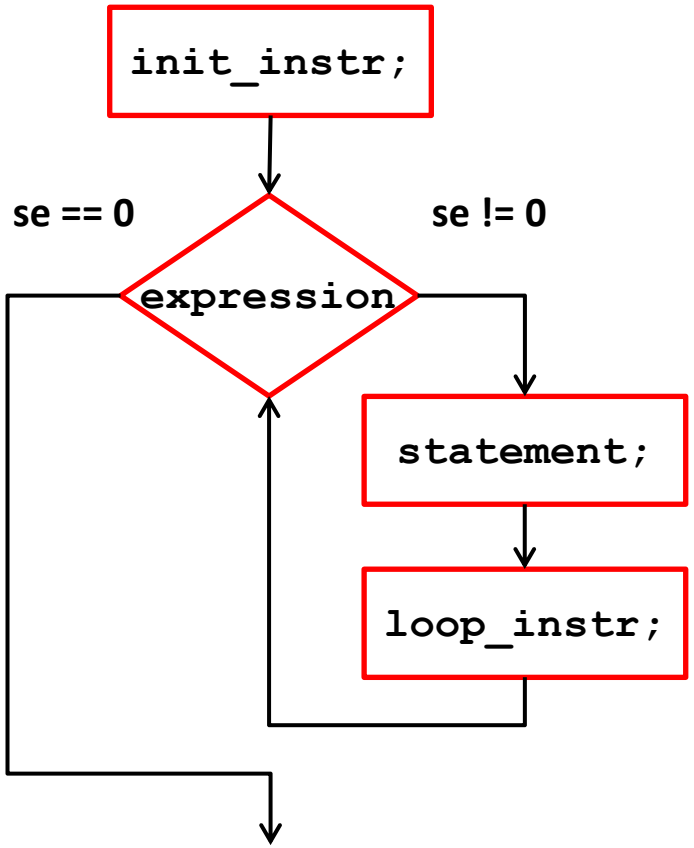
- **`for`** è un costrutto iterativo, equivalente al **`while`**
  - **`for`** è una keyword
  - **`init_instr`** istruzione (di inizializzazione)
  - **`expression`** espressione booleana
  - **`loop_instr`** istruzione (di loop)
  - **`statement`** corpo del ciclo

NB: se **`statement`** contiene più istruzioni, richiede **{ }**

Costrutto Iterativo: `for` , l'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`**
6. …



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 l'esercizio:
  - delle tabelline
  - delle tabelline solo per il triangolo inferiore

for vs. while

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

VS.

```
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(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 vs. do-while

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

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

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

- In questo caso **do-while**, risulta più compatto e chiaro

`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, ovvero **`while`**, **`do-while`** e **`for`**

Interpretazione del Codice (a.k.a. Che Cosa fa?)

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

- Richiede fino a 10 numeri e ne stampa il valore inserito
- Le acquisizioni terminano anticipatamente se viene inserito un valore negativo

NB: il `break` interrompe comunque il costrutto iterativo (`for`) anche se si trova all'interno dell' `if`

Che Cosa fa?

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

- Richiede numeri fino a quando non ne inserisci 10 positivi
- Stampa a schermo il valore inserito di ogni positivo
- I valori negativi non vengno stampati e non viene incrementato il contatore `i`

NB: il `continue` fa saltare tutte le successive istruzioni

Che 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 e viene stampato solo se è positivo

NB: la **`loop_expr`** non viene saltata dal **`continue`**, è una particolarità del **`for`**

Alternative a `break` e `continue`

Alternativa al `break`

- Utilizzo di cicli con variabili **flag (o sentinella)** per terminare anticipatamente l'esecuzione del ciclo

Alternativa al `continue`

- Una variabile che assume un valore 0 / 1 a seconda che si verifichino o meno alcune condizioni durante l'esecuzione

NB: Fortemente consigliato utilizzare queste due alternative

Esempio

- Scrivere un ciclo che richiede una serie di valori interi e li associa alla variabile intera **`n`**:
  - non fa più di **`N`** richieste
  - li stampa a schermo, saltando i valori negativi inseriti
  - interrompendo l'elaborazione al primo valore nullo incontrato

Soluzione (1): 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 */
}
```

Soluzione (2): variabile flag

```
int n, i, flag;
flag = 0; //diventa 1 quando inserisco zero
for(i = 0; i <= N && 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*/
}
```

Esercizio

- Scrivere un ciclo che richiede una serie di valori interi e li associa alla variabile intera **`n`**:
  - non fa più di **`N`** richieste
  - li stampa a schermo, 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**

Soluzione (2): variabile flag

```
int n, i, flag;
flag = 0; //diventa 1 quando inserisco zero
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");
```

Alcune Precisazioni

Nota sugli Identificatori

- **Gli identificatori sono unici**: non è possibile associare due identificatori diversi alla stessa variabile o lo stesso identificatore a due variabili diverse
- In un programma, ogni riferimento alla variabile **`a`** rimanda alla stessa cella di memoria e non esistono altri identificatori per quella cella
- **Non si possono usare** alcune espressioni come identificatori perché fanno riferimento a parole **riservate, le keywords** tra cui:
  - **`if`**, **`for`**, **`switch`**, **`while`**, **`main`**, **`printf`**, **`scanf`**, **`int`**, **`float`**, etc.

Nota sulla Dichiarazione di Variabili

- Solo le variabili dichiarate possono essere utilizzate
- Il fatto che sia richiesta la dichiarazione delle variabili permette gli editor di riconoscere eventuali typos
- Ogni sequenza di caratteri in un codice di un programma C può essere:
  - Un nome di variabile
  - Un nome di funzione
  - Una keyword
- Se provate ad usare una variabile **a** senza averla dichiarata, il compilatore risponde:

```
'a' undeclared (first use in this function)
```

Nota sulla Dichiarazione di Variabili

- Sintassi per la dichiarazione:

```
nomeTipo nomeVariabile;
```

Es. `int N;`

- Le celle di memoria **non sono «vuote»**, ma tipicamente contengono valori non sensati
- La dichiarazione **non modifica** tali valori iniziali, sarà il primo assegnamento a farlo
- Provare per credere:

```
int N;
printf("%d",N);
scanf("%d",&N);
printf("%d",N);
```

Le Costanti

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

Es.

```
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';
```

- La differenza dalle variabili non costanti è che il compilatore segnala come errore ogni istruzione di assegnamento a una costante nella parte eseguibile

Le Costanti

- Usare le costanti è utile perché:
  - l'identificatore suggerisce il significato di un valore
  - permette di **parametrizzare** i programmi, e riutilizzare il codice 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;
```

Abbreviazioni nell'Assegnamento

- Istruzioni della forma:

  variabile = variabile ***operatore*** *espressione*

  si possono scrivere come:

  variabile ***operatore*** *= espressione*

Es.

`b = b + 7;` ⟹ `b += 7;` (Idem con altri operatori)

- Incrementare o decrementare una variabile di 1 è molto frequente, quindi c'è una notazione apposita

Es.

```
a = a + 1; ⟹ a++;
b = b - 1; ⟹ b--;
```

Caratteri di Conversione

- Nella **`stringaControllo`** di **`printf`** è possibile specificare la formattazione quando viene stampato un valore di una variabile

Es.

`"%5d"` dedica 5 caratteri alla stampa del numero intero

`"%.2f"` dedica due cifre dopo la virgola per un float

Linguaggio C: un Altro Costrutt0

istruzione composta: **`switch case`**

`switch-case`, la sintassi

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

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

`switch-case`, la sintassi

NB: **`constant-expr1`** non può contenere una variabile

NB: **`int_expr`** può contenere variabili

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

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

`switch-case`, l'Esecuzione

1. Viene valutata **`int_expr`** (eventualmente convertita)
2. Si controlla se **`int_expr`** è **uguale** a **`constant-expr1`**
3. Se sono **uguali** eseguo **`statement1`** , e 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`** e così via
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]
      }
```

Che Cosa fa?

```
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'**, verrà incrementa n**Cons;**

Come Eseguire un Solo 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 n**Cons;**

Esercizio

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

Homework

- Scrivere un programma che richiede all'utente di inserire dei numeri (controllando che questi siano positivi) fino a quando non viene inserito uno zero. Il programma conta quanti sono i:
  - multipli di 2,
  - multipli di 4,
  - multipli di 6,
  - multipli di 8,

  Tra i numeri inseriti e stampa a schermo un istogramma (tanti asterischi quanti i valori inseriti)

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

Errori Frequenti 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?

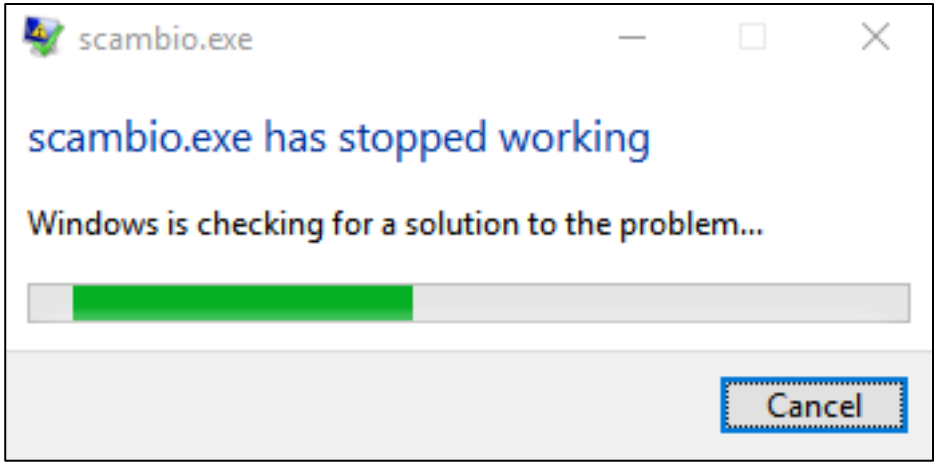
error: 'prinf' undeclared (first use in this function)

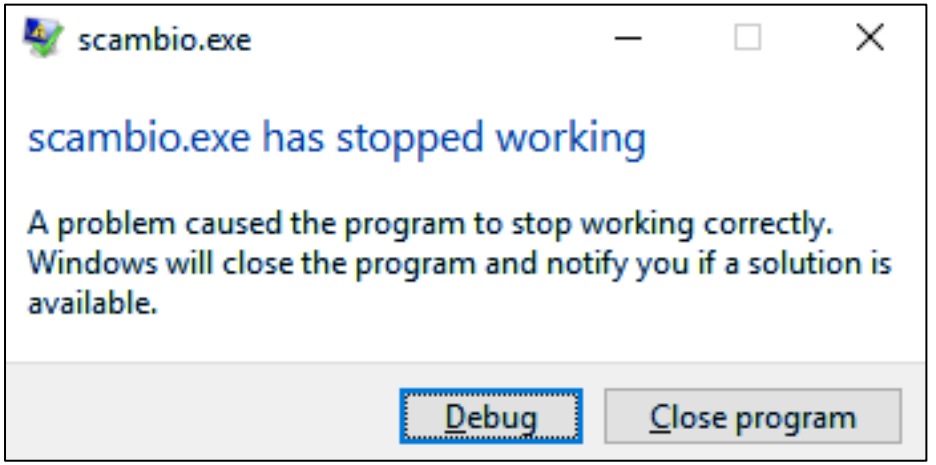
Errori Frequenti a Run-Time

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

```
int a = 7;
```

`scanf("%d",a);` (scrive nella cella all'indirizzo **`7`**)





Errori Frequenti a Run-Time

- Confondere l'assegnamento con il confronto

Es.

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

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

Errori Frequenti 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 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: 4.98

x = 536870912
Process returned 14 (0xE)   execution time : 9.780 s
Press any key to continue.
```

Sembra un Errore ma non è

- È invece possibile stampare i caratteri alfanumerici 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 a Run-Time

- L'istruzione `i++` corrisponde all'assegnamento `i = i + 1;`
- Non ha quindi senso `i = i++`
- Corrisponde a `i = (i = i + 1);`

Errori Frequenti a Run-Time

- Il `;` termina un'istruzione e quindi non va messo dopo **`if`**, **`while`**, **`for`** e **`switch`**
- Se presente, il `;` specifica che il costrutto non ha corpo

Es.

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

- Stampa vero perché **`printf("Vero");`** è fuori dall' **`if`**
- Se ci fosse stato un **`else`** il compilatore avrebbe dato errore
- Al contratrio, il `;` viene usato nel **`do-while`** perché il costrutto termina con il **`while(expression);`**

Acquisizione di caratteri da tastiera

- Le acquisizioni di caratteri consecutivi danno problemi
- In particolare, dopo uno **`scanf`**, l'invio di conferma rimane nel buffer di ingresso (stdin) e viene acquisto dal primo `scanf("%c", &variabile);` che segue

- Soluzioni, da mettere dopo il primo **`scanf`**:
  - `fflush(stdin);` pulisce il buffer stdin (per windows)
  - `scanf("% c");` acquisisce un secondo carattere che viene buttato via (non viene precisata la variabile di destinazione)

Esempi di scanf

- Acquisizione con `fflush(stdin);`

```
int main(void)
{ char a,b;
scanf("%c", &a);
fflush(stdin);// elimina tutto il buffer
scanf("%c", &b);// acquisisce da zero
printf("%c %c", a, b); }
```

- Acquisizione con `"% c";`

```
int main(void)
{ char a,b;
scanf("%c", &a);
scanf("% c");// acquisisce ed elimina l'invio
scanf("%c", &b);// acquisisce da zero
printf("%c %c", a, b); }
```

Confronto e Assegnamento

- L'operatore di confronto **==** non va confuso con l'operatore di assegnamento **=**
- Le loro sintassi sono simili

```
nomeVariabile == Espressione;

nomeVariabile = Espressione;
```

- In entrambi i casi **Espressione** è una variabile/una costante/un valore fissato o un'espressione che coinvolge gli elementi sopra
- Il risultato del confronto **nomeVariabile == Espressione** è **1** se **nomeVariabile** ed **Espressione** coincidono

Che Cosa fa?

```
#include <stdio.h>
void main()
{
        char a, b;
        b = '2';
        a = b == '0';
        printf("%d" , a);
}
```

- Associa ad **a** il valore **1** se **b** è **0**, **0** altrimenti
- Viene letto:

```
a = (b =='0');
```

- **b** è **'2'**, stampa 0
- Se **b** fosse **'0'**, stamperebbe 1
- Se **b** fosse **0**, stamperebbe 0

Cose da non fare

- Modificare la variabile di loop nel for a mano

Es.

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

- La variabile del ciclo for deve essere modificata unicamente dalla **`init_instr`** e dalla **`loop_instr`**
- Altrimenti il codice diventa di difficile interpretazione, ed è facile commettere errori
- Piuttosto usare **`while`**