Strutture di Controllo in C

Informatica A AA 2024/2025

Giacomo Boracchi

30 Settembre, 2 Ottobre 2024

giacomo.boracchi@polimi.it

La lezione del Mercoledì è stata spostata in Aula T.1.3.

Laboratorio

Sul sito trovate il link alla pagina del laboratorio:

https://polimi365-my.sharepoint.com/:x:/g/personal/10545327_polimi_it/EUMp605w0_xFilxobIijlvgBWoNq097mxESMsUys0ZNQqQ?e=wYheu6&CID=9b1e2539-62cf-74e7-7805-ea51dbe95dce

Gli orari esatti per ogni squadra sono nel calendario del corso

L'aula risulta tipicamente disponibile prima o dopo lo slot di laboratorio. Potete andare in aula prima della lezione per iniziare a lavorare o trattenetevi per concludere.

I Laboratori

Nei laboratori vi sarà richiesto di **programmare autonomamente**

Sarete divisi in due squadre

- **'Squadra 1' CP DISPARI**
- **'Squadra 2' CP PARI**

Il Laboratorio è **molto utile** per

- prendere familiarità con l'ambiente di sviluppo
- **consolidare la conoscenza** dei linguaggi, dei metodi e degli strumenti introdotti a lezione.

I Laboratori

Nei laboratori vi sarà richiesto di **programmare autonomamente**

Sarete divisi in due squadre

- **'Squadra 1' CP DISPARI:**
- **'Squadra**

Il Laboratorio è

- prendere
- **consolida** nti introdotti a lezione.

Non si impara a programmare su carta!

Durante il lab fate domande e chiedete chiarimenti, non potete risolvere tutti gli esercizi.

I Laboratori, Consigli

Arrivare preparati e con una certa familiarità con l'ambiente di sviluppo prima del laboratorio

I testi degli esercizi di laboratorio verranno pubblicati sul sito del docente già da Lunedì:

- incominciate a lavorarci, userete il lab per fare domande e chiedere chiarimenti

I responsabili sono disponibili per fornire assistenza, chiamateli alla vostra postazione e chiedete consigli/aiuto!

Prima di chiamare un responsabile/tutor:

1. Sistemare il codice
2. Leggere tutti i messaggi del compilatore
3. Formulare una domanda chiara

I Laboratori

Vi invitiamo ad installare l'ambiente di programmazione

- Installare Code::Blocks per il C, versione con compilatore (http://www.codeblocks.org/)
- Potete anche usare alternative come Xcode (utenti iOS) o DevC++
- Sul sito del corso troverete le istruzioni per installare su Windows e Mac OS

https://boracchi.faculty.polimi.it/teaching/InfoA/install_code_blocks_Win10-OSX-AA2019-20.pdf oppure https://boracchi.faculty.polimi.it/teaching/InfoA/Install_code_blocks%20_win11.pdf

- Potrete usare quelli del laboratorio, accedendo con le vostre credenziali a **virtualdesktop**. Trovate qui tutte le indicazioni https://www.ict.polimi.it/software/virtual-desktop-software-for-study-and-teaching/?lang=en
- Guardate sempre il calendario per trovare gli orari e l'aula del vostro turno.

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.

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

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

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.

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)`

Tabelle di Verità

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

Una Tabella di Verità rappresenta tutti i valori che un'espressione booleana composta assume al variare delle espressioni che la compongono

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`

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.

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 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.

disgiunzione (OR)		
A	**B**	**A || B**
0	0	0
0	1	1
1	0	1
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, come spesso accade nella lingua parlata

disgiunzione (OR)		
`A`	`B`	`A || B`
0	0	0
0	1	1
1	0	1
1	1	1

Altri operatori logici (derivati) e loro tabelle di verità

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

A	B	A xnor B
0	0	1
0	1	0
1	0	0
1	1	1

…o uno o l'altro ma non entrambi

A xor B = (A and (not B)) or ((not A) and B)

A xnor B = not((A and (not B)) or (B and (not A)))

Precedenze nelle espressioni Booleane

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`

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

Tautologie e Contraddizioni

Tautologia

- Una espressione logica che è sempre vera, per qualunque combinazione di valori delle variabili
  - Esempio: principio del “terzo escluso”: A or not A (tertium non datur, non si dà un terzo caso tra l’evento A e la sua negazione)

Contraddizione

- Una espressione logica che è sempre falsa, per qualunque combinazione di valori delle variabili
  - Esempio: principio di “non contraddizione”: A and not A (l’evento A e la sua negazione non possono essere entrambi veri)

Altro Esempio di Tabella di Verità

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

Altro Esempio di 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			

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`

Es: `!((a >= 5) && (a <= 10))` -> [De Morgan]

`!(a >= 5)||!(a <= 10)` -> [proprietà `>=` e `<=`]

`!!(a < 5)|| !!(a > 10)` -> [doppia negazione] `((a < 5) || (a > 10))`

Esempio

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

Esempio

Dimostrare che le seguenti espressioni sono equivalenti

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

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

Esempio

Dimostrare che le seguenti espressioni sono equivalenti

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

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

Esempio

Dimostrare che le seguenti espressioni sono equivalenti

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

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

Esempio

Dimostrare che le seguenti espressioni sono equivalenti

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

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

Esempio

Dimostrare che le seguenti espressioni sono equivalenti

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

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

Esempio

Dimostrare che le seguenti espressioni sono equivalenti

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

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

Esempio

Dimostrare che le seguenti espressioni sono equivalenti

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

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

Ancora sulle proprietà

Alcune proprietà somigliano a quelle dell'algebra numerica tradizionale:

- Proprietà *associativa*: A or (B or C) = (A or B) or C (idem per AND)
- Proprietà *commutativa*: A or B = B or A (idem per AND)
- Proprietà *distributiva* di AND rispetto a OR:
  A and (B or C) = A and B or A and C
- … e altre ancora

Ma parecchie altre sono alquanto insolite…

- Proprietà *distributiva* di OR rispetto a AND:
  A or B and C = (A or B) and (A or C)
- Proprietà di *assorbimento* (A assorbe B):
  A or A and B = A
- *Legge dell'elemento* 1: not A or A = 1
- … e altre ancora

Uso delle proprietà

Trasformare un'espressione logica in un'altra, differente per aspetto ma equivalente:

not A and B or A =	(assorbimento)
= not A and B or (A or A and B) =	(togli le parentesi)
= not A and B or A or A and B =	(commutativa)
= not A and B or A and B or A =	(distributiva)
= (not A or A) and B or A =	(legge dell'elemento 1)
= ***true*** and B or A =	(vero and B = B)
= B or A è più semplice dell'espressione originale !	

Si *verifichi* l'equivalenza con le tabelle di verità!

Occorre conoscere un'ampia lista di proprietà e si deve riuscire a "vederle" nell'espressione (qui è il difficile)

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 ⇔ falso**
- **qualsiasi valore `!=0` ⇔ 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`

Esempio: Mondiali 2014 – Gruppo D

	Squadra	Punti	Differenza reti	Reti segnate
	Costa Rica	6	+3	4
	Italia	3	0	2
	Uruguay	3	-1	3
	Inghilterra	0	-2	2

Criteri di passaggio del turno: passano le prime due squadre con più:

- punti
- maggiore differenza reti
- maggiori reti segnate

Credits: Francesco Trovò

Passa l'italia se:

L'italia passa solo se pareggia o vince con un qualunque risultato

Dobbiamo controllare i predicati:

VI: «vince l'Italia»
PI: «pareggia l'Italia»

Passa l'Italia		
VI	PI	VI || PI
0	0	0
0	1	1
1	0	1
1	1	1

Passa l'italia se:

L'italia passa solo se pareggia o vince con un qualunque risultato

Dobbiamo controllare i predicati:

VI: «vince l'Italia»
PI: «pareggia l'Italia»

Passa l'Italia		
VI	PI	VI || PI
0	0	0
0	1	1
1	0	1
~~1~~	~~1~~	~~1~~

ATTENZIONE!!!
I due eventi sono mutualmente esclusivi. Non si possono verificare contemporaneamente!!!

L'Italia passa come prima se:

L'italia passa come prima se vince, la Costa Rica perde ed è verificata una delle seguenti:

- la sua differenza reti è maggiore della Costa Rica
- la sua differenza reti è uguale a quella della Costa Rica e ha segnato più reti della Costa Rica

L'Italia passa come prima se:

L'italia passa come prima se vince, la Costa Rica perde ed è verificata una delle seguenti:

- la sua differenza reti è maggiore della Costa Rica
- la sua differenza reti è uguale a quella della Costa Rica e ha segnato più reti della Costa Rica

Dobbiamo controllare i predicati:

- VI: «vince l'Italia»
- CRP: «perde la Costa Rica»
- DRM: «l'Italia ha una differenza reti maggiore della Costa Rica»
- DRU: «l'Italia ha una differenza reti uguale alla Costa Rica»
- GSM: «l'Italia ha una segnato più reti della Costa Rica»

L'Italia passa come prima se:

L'italia passa come prima se vince, la Costa Rica perde ed è verificata una delle seguenti:

- la sua differenza reti è maggiore della Costa Rica
- la sua differenza reti è uguale a quella della Costa Rica e ha segnato più reti della Costa Rica

Dobbiamo controllare i predicati:

- VI: «vince l'Italia»
- CRP: «perde la Costa Rica»
- DRM: «l'Italia ha una differenza reti maggiore della Costa Rica»
- DRU: «l'Italia ha una differenza reti uguale alla Costa Rica»
- GSM: «l'Italia ha una segnato più reti della Costa Rica»

```
VI && CRP && ( DRM || (DRU && GSM))
```

La Tavola di Verità

VI	CRP	DRM	DRU	GSM	DRU && GSM	DRM ||(DRU && GSM)	VI && CRP && (DRM||(DRU && GSM))
1	1	1	1	1			
1	1	1	1	0			
1	1	1	0	1			
1	1	1	0	0			
1	1	0	1	1			
1	1	0	1	0			
1	1	0	0	1			
1	1	0	0	0			
1	0	1	1	1			
1	0	1	1	0			
1	0	1	0	1			
1	0	1	0	0			
1	0	0	1	1			
1	0	0	1	0			
1	0	0	0	1			
1	0	0	0	0			

La Tavola di Verità

VI	CRP	DRM	DRU	GSM	DRU && GSM	DRM ||(DRU && GSM)	VI && CRP && (DRM||(DRU && GSM))
1	1	1	1	1			
1	1	1	1	0			
1	1	1	0	1			
1	1	1	0	0			
1	1	0	1	1			
1	1	0	1	0			
1	1	0	0	1			
1	1	0	0	0			
1	0	1	1	1			
1	0	1	1	0			
1	0	1	0	1			
1	0	1	0	0			
1	0	0	1	1			
1	0	0	1	0			
1	0	0	0	1			
1	0	0	0	0			

Sappiamo come continua la tabella… se `VI==0`, il predicato è falso, lo stesso vale per `CRP==0`

La Tavola di Verità

VI	CRP	DRM	DRU	GSM	DRU && GSM	DRM ||(DRU && GSM)	VI && CRP && (DRM||(DRU && GSM))
1	1	1	1	1	1		
1	1	1	1	0	0		
1	1	1	0	1	0		
1	1	1	0	0	0		
1	1	0	1	1	1		
1	1	0	1	0	0		
1	1	0	0	1	0		
1	1	0	0	0	0		
1	0	1	1	1	1		
1	0	1	1	0	0		
1	0	1	0	1	0		
1	0	1	0	0	0		
1	0	0	1	1	1		
1	0	0	1	0	0		
1	0	0	0	1	0		
1	0	0	0	0	0		

La Tavola di Verità

VI	CRP	DRM	DRU	GSM	DRU && GSM	DRM ||(DRU && GSM)	VI && CRP && (DRM||(DRU && GSM))
1	1	1	1	1	1	1	
1	1	1	1	0	0	1	
1	1	1	0	1	0	1	
1	1	1	0	0	0	1	
1	1	0	1	1	1	1	
1	1	0	1	0	0	0	
1	1	0	0	1	0	0	
1	1	0	0	0	0	0	
1	0	1	1	1	1	1	
1	0	1	1	0	0	1	
1	0	1	0	1	0	1	
1	0	1	0	0	0	0	
1	0	0	1	1	1	1	
1	0	0	1	0	0	0	
1	0	0	0	1	0	0	
1	0	0	0	0	0	0	

La Tavola di Verità

VI	CRP	DRM	DRU	GSM	DRU && GSM	DRM ||(DRU && GSM)	VI && CRP && (DRM||(DRU && GSM))
1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1
1	1	1	0	1	0	1	1
1	1	1	0	0	0	1	1
1	1	0	1	1	1	1	1
1	1	0	1	0	0	0	0
1	1	0	0	1	0	0	0
1	1	0	0	0	0	0	0
1	0	1	1	1	1	1	0
1	0	1	1	0	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	0	0	0	0
1	0	0	1	1	1	1	0
1	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0

Eventi impossibili

VI	CRP	DRM	DRU	GSM	DRU && GSM	DRM ||(DRU && GSM)	VI && CRP && (DRM||(DRU && GSM))
1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1
1	1	1	0	1	0	1	1
1	1	1	0	0	0	1	1
1	1	0	1	1	1	1	1
1	1	0	1	0	0	0	0
1	1	0	0	1	0	0	0
1	1	0	0	0	0	0	0
1	0	1	1	1	1	1	0
1	0	1	1	0	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	0	0	0	0
1	0	0	1	1	1	1	0
1	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0

DRU e DRM non possono essere entrambe vere

Tre possibili scenari

Italia-Uruguay 2-0, Costa Rica-Inghilterra 0-2

Italia-Uruguay 2-0, Costa Rica-Inghilterra 1-2

Italia-Uruguay 4-2, Costa Rica-Inghilterra 0-1

	Squadra	Punti	Differenza reti	Reti segnate
	Costa Rica	6	+3	4
	Italia	3	0	2
	Uruguay	3	-1	3
	Inghilterra	0	-2	2

Italia-Uruguay 2-0, Costa Rica-Inghilterra 0-2

VI	CRP	DRM	DRU	GSM	DRU && GSM	DRM ||(DRU && GSM)	VI && CRP && (DRM||(DRU && GSM))
1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1
1	1	1	0	1	0	1	1
1	1	1	0	0	0	1	1
1	1	0	1	1	1	1	1
1	1	0	1	0	0	0	0
1	1	0	0	1	0	0	0
1	1	0	0	0	0	0	0
1	0	1	1	1	1	1	0
1	0	1	1	0	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	0	0	0	0
1	0	0	1	1	1	1	0
1	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0

Tre possibili scenari

Italia-Uruguay 2-0, Costa Rica-Inghilterra 0-2

- Italia passa come prima

Italia-Uruguay 2-0, Costa Rica-Inghilterra 1-2

Italia-Uruguay 4-2, Costa Rica-Inghilterra 0-1

Italia-Uruguay 2-0, Costa Rica-Inghilterra 1-2

VI	CRP	DRM	DRU	GSM	DRU && GSM	DRM ||(DRU && GSM)	VI && CRP && (DRM||(DRU && GSM))
1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1
1	1	1	0	1	0	1	1
1	1	1	0	0	0	1	1
1	1	0	1	1	1	1	1
1	1	0	1	0	0	0	0
1	1	0	0	1	0	0	0
1	1	0	0	0	0	0	0
1	0	1	1	1	1	1	0
1	0	1	1	0	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	0	0	0	0
1	0	0	1	1	1	1	0
1	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0

Tre possibili scenari

Italia-Uruguay 2-0, Costa Rica-Inghilterra 0-2

- Italia passa come prima

Italia-Uruguay 2-0, Costa Rica-Inghilterra 1-2

- Italia non passa prima ma come seconda (considera tabella precedente)

Italia-Uruguay 4-2, Costa Rica-Inghilterra 0-1

Italia-Uruguay 4-2, Costa Rica-Inghilterra 0-1

VI	CRP	DRM	DRU	GSM	DRU && GSM	DRM ||(DRU && GSM)	VI && CRP && (DRM||(DRU && GSM))
1	1	1	1	1	1	1	1
1	1	1	1	0	0	1	1
1	1	1	0	1	0	1	1
1	1	1	0	0	0	1	1
1	1	0	1	1	1	1	1
1	1	0	1	0	0	0	0
1	1	0	0	1	0	0	0
1	1	0	0	0	0	0	0
1	0	1	1	1	1	1	0
1	0	1	1	0	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	0	0	0	0
1	0	0	1	1	1	1	0
1	0	0	1	0	0	0	0
1	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0

Tre possibili scenari

Italia-Uruguay 2-0, Costa Rica-Inghilterra 0-2

- Italia passa come prima

Italia-Uruguay 2-0, Costa Rica-Inghilterra 1-2

- Italia non passa prima ma come seconda (considera tabella precedente)

Italia-Uruguay 4-2, Costa Rica-Inghilterra 0-1

- Italia passa come prima

Tre possibili scenari

Italia-Uruguay 2-0, Costa Rica-Inghilterra 0-2

- Italia passa come prima

Italia-Uruguay 2-0, Costa Rica-Inghilterra 1-2

- Italia non passa prima ma come seconda (considera tabella precedente)

Italia-Uruguay 4-2, Costa Rica-Inghilterra 0-1

- Italia passa come prima

L'amara verità

Italia-Uruguay 0-1, Costa Rica-Inghilterra 0-0

- Italia eliminata

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 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

```c
#include<stdio.h>
int 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!");
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

```c
#include<stdio.h>
int 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!");
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 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);
return 0;}
```

Soluzione 1: if Annidati

```
#include<stdio.h>
int 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);
return 0;}
```

Osservazioni:

1. Il numero di indentazioni è n, pari a quanti numeri occorre controllare
2. Le parentesi negli if non sono necessarie qua

Soluzione 2: Condizioni Composte

```c
#include<stdio.h>
int 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)

return 0;
}
```

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

```c
#include<stdio.h>
int main()
{
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,%d,%d) = %d", a, b, c, 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
```

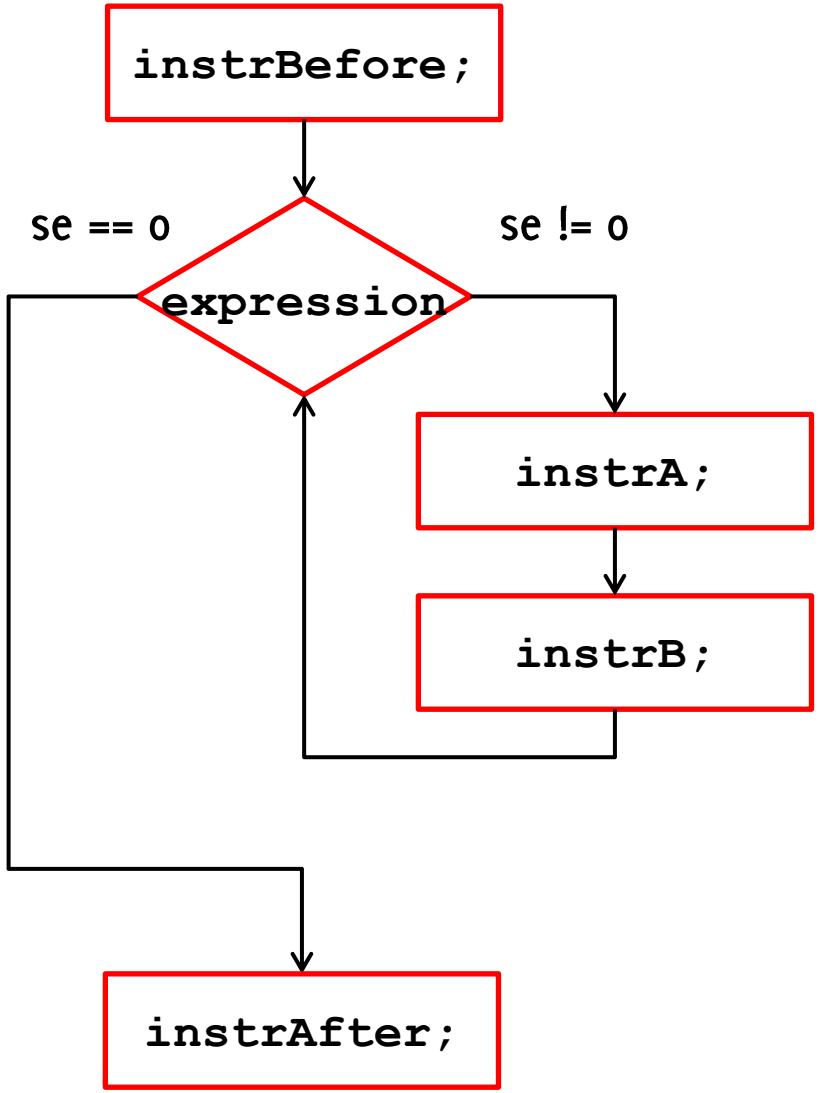
Costrutto Iterativo: `while`, l'esecuzione

1. Terminata **`instrBefore`** viene valutata **`expression`**
2. Se **`expression`** è vera (0 != 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;
```

N.B: **`while(expression)`** **non** richiede il **`;`** perché l'istruzione non termina dopo **`)`** ma con lo **`statement`**:
`while(expression);` è un ciclo senza corpo

Costrutto Iterativo: `while`, l'esecuzione




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

Esempio

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

Esempio

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

Esempio

```
/* stampa i primi 100 numeri 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++;
      }
}
```

Costrutto Iterativo: `while`, Avvertenze

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>
int main()
{
        int a = 1;
        while(a < 100)
        {
            printf("\n%d" , 2*a);
            a++;
        }
}
```

Costrutto Iterativo: `while`, Avvertenze

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>
int main()
{
        int a = 1;
        while(a > 0)
        {
            printf("\n%d" , 2*a);
            a++;
        }
}
```

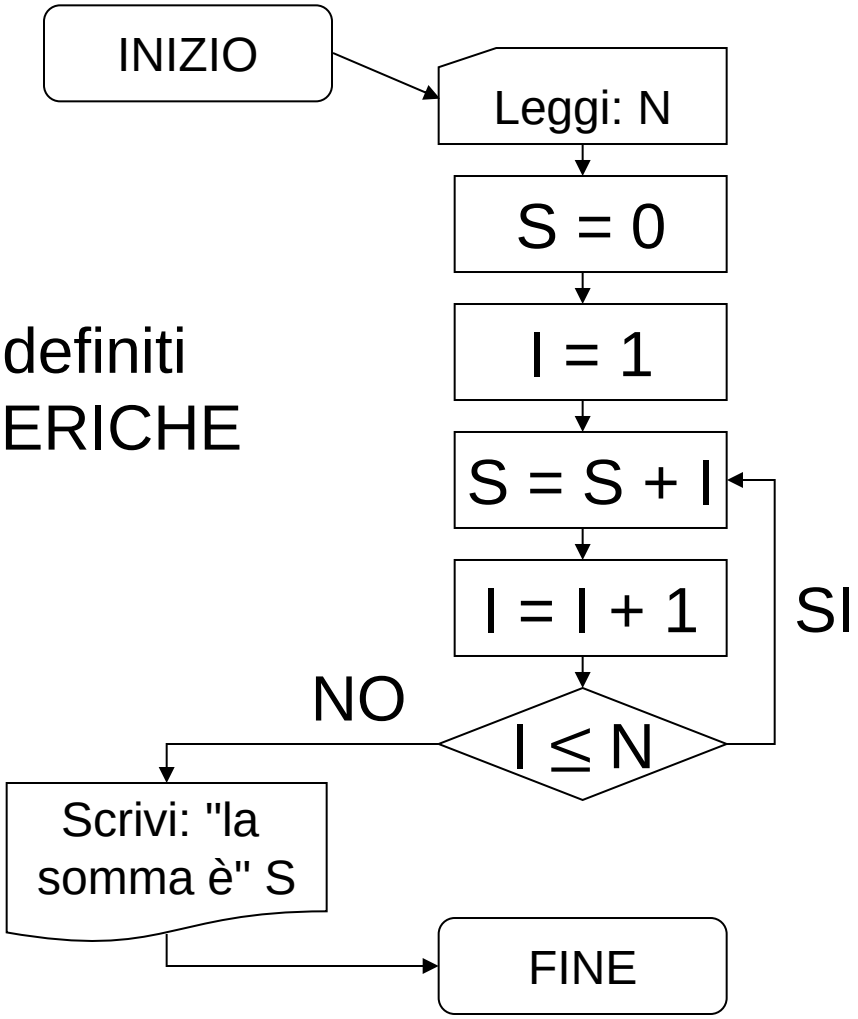
Somma dei primi N numeri naturali

L'utente specifica un numero N ed il programma calcola la somma dei primi N numeri naturali.

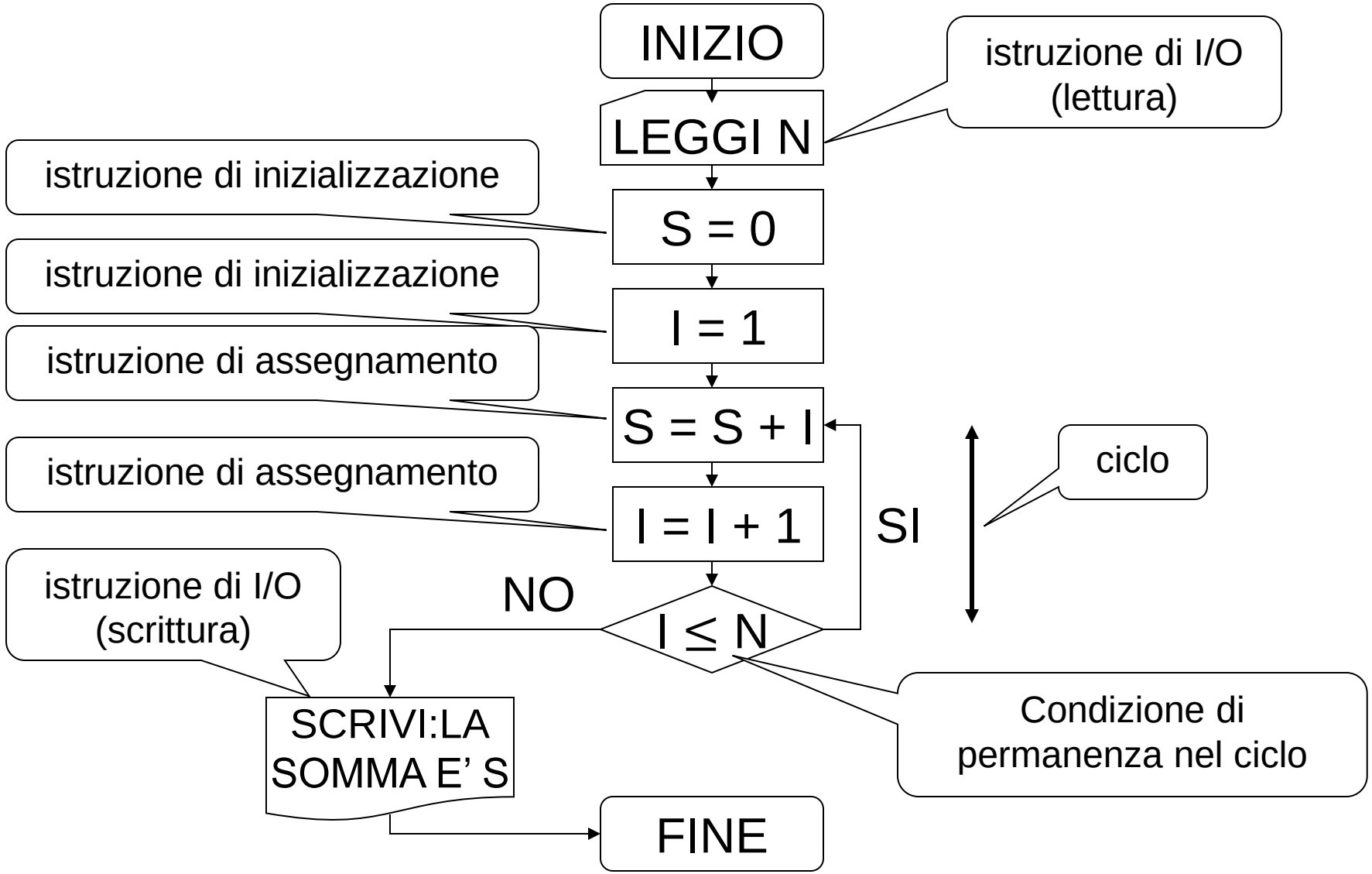
P.S. non usare le formula $\frac{N}{2}(N+1)$ ma progettare una soluzione iterativa

Somma dei primi N numeri naturali

I simboli S, I e N sono definiti come VARIABILI NUMERICHE (di tipo intero)



Somma dei primi N numeri naturali



Istruzioni iterative: ciclo a condizione iniziale - while

```
#include <stdio.h>
/* Somma dei primi N naturali */
int main ( ) {
   int N, S, I;
       printf ("Inserisci N: ");
       scanf ("%d", &N);
       if (N >= 0) {
              S = 0;
              I = 1;
              while (I <= N) {
                     S = S + I;
                     I = I + 1;
              }
              printf ("\nSum is: %d\n", S);
       }
   return 0;
}
```

condizione di permanenza nel ciclo

corpo del ciclo

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>
int main()
{









}
```

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>
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);
}
```

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>
int main()
{









}
```

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>
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);
}
```

Esercizio di warm up

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.

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);
```

do-while

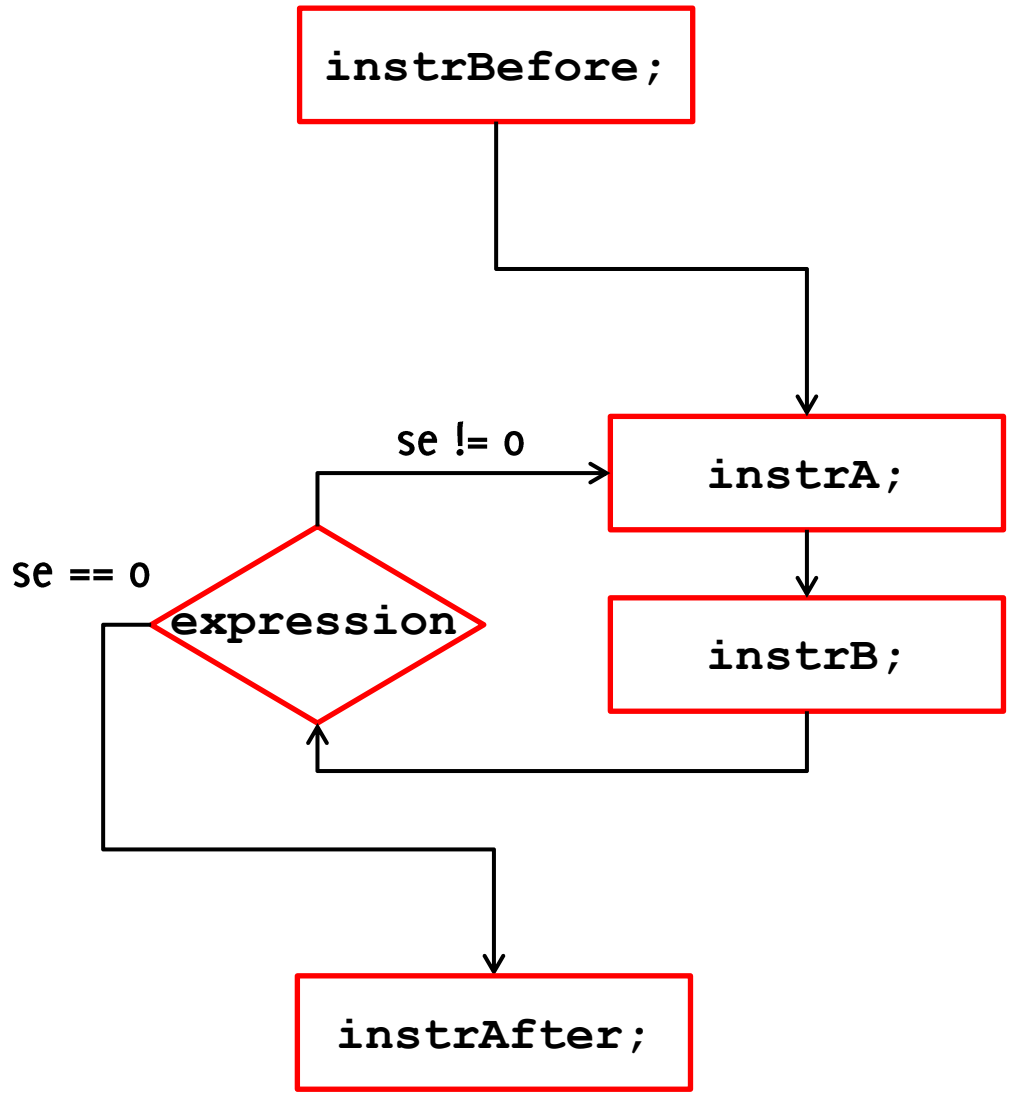
- Il costrutto **do-while** garantisce l'esecuzione del corpo del **while** almeno una volta.

```
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



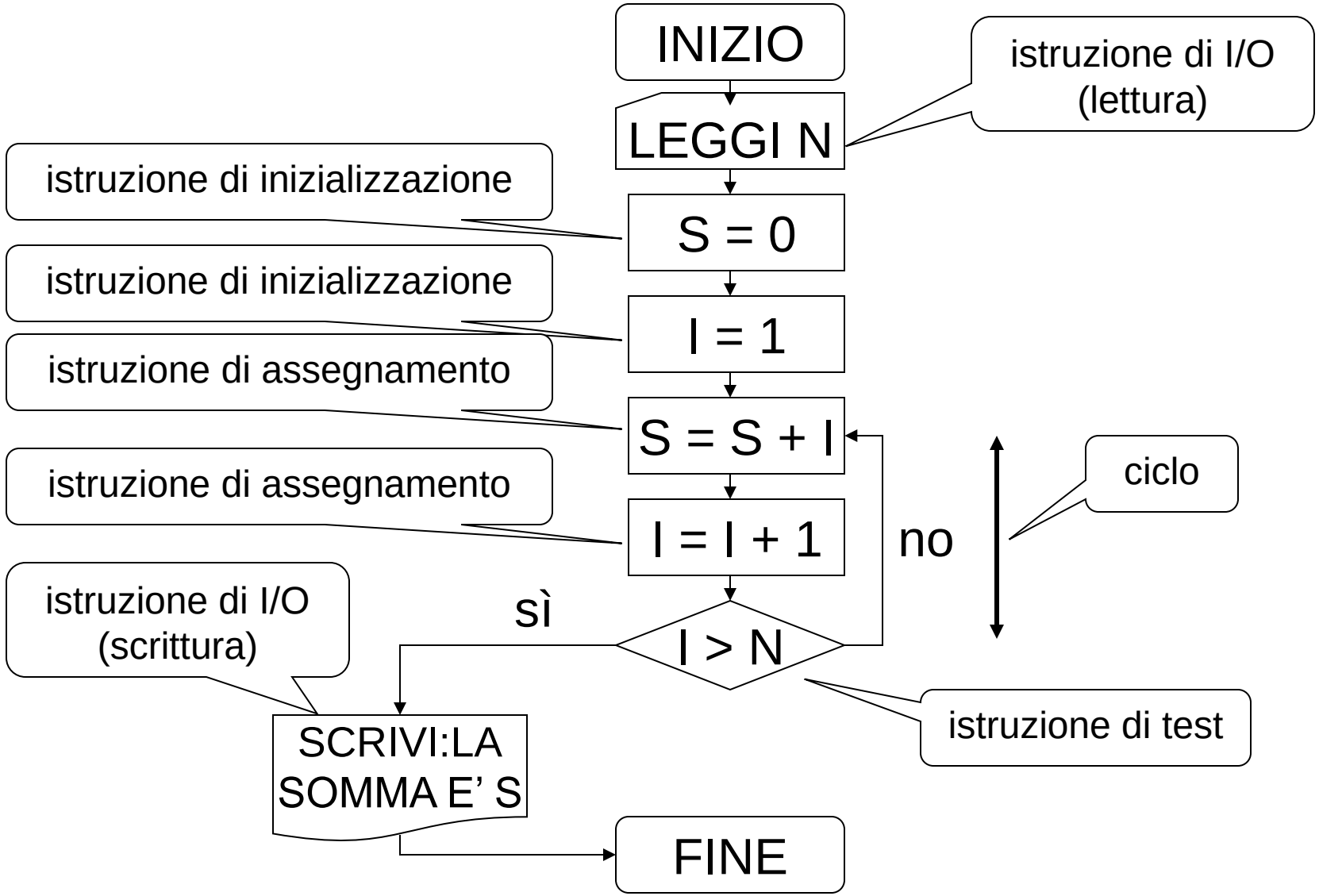

```
instrBefore;
do
   {
   instrA;
   instrB;
   }
while (expression);
instrAfter;
```

Esercizi TODO:

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

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.

Somma dei primi N numeri naturali



Istruzioni iterative: ciclo a condizione finale - do

```
#include <stdio.h>
/* Somma dei primi N numeri naturali */
int main ( ) {
      int N, S, I;                    /* dichiarazione variabili */
      printf ("Inserisci N: ");
      scanf ("%d", &N);               /* input */
      if ( N > 0 ) {                  /* accetto solo N positivo */
            S = 0;
            I = 1;
            do {                      /* ciclo a condizione finale */
                  S = S + I;
                  I = I + 1;
            } while (I <= N);
            /* il ciclo è eseguito almeno una volta ! */
            printf ("\nSum is: %d\n", S);   /* output */
      }
}
```

corpo del ciclo

condizione di permanenza nel ciclo

Teorema di Boehm-Jacopini

istruzioni **`if`** e **`while`** (e la possibilità di eseguire istruzioni in sequenza) sono equivalenti a istruzioni che la macchina di Von Neumann che può manipolare registro Contatore di Programma

➡ istruzioni **`if`** e **`while`** sono complete:
bastano per codificare qualsiasi algoritmo

Per praticità e convenienza si usano però molte altre strutture di controllo

TODO:

Scrivere un programma che calcola l'MCD di due numeri interi positivi inseriti dall'utente

Esempio: MCD

```c
#include<stdio.h>

int main()
{
    int x, a, b, min, mcd;
    scanf("%d",&a);
    scanf("%d",&b);
    x = 1;
    mcd = 1;
    if (a < b)
        min = a;
    else
        min = b;
    while (x < min) {
        x = x + 1;
        if ( (a % x)==0  &&  (b % x)==0 )
            mcd = x;
    }
    printf("massimo comune divisore: %d", mcd);
    return 0;
}
```

```c
#include<stdio.h>

int main()
{
    int x,a,b,min,mcd;
    printf("inserire a e b: ");
    scanf("%d",&a);
    scanf("%d",&b);
    x = 1;
    mcd = 1;
    if (a < b)
        min = a;
    else
        min = b;

    x = min;
    while (!((a % x)==0  &&  (b % x)==0))
        x = x - 1;

    mcd = x;
    printf("massimo comune divisore: %d", mcd);
    return 0;
}
```

Si tratta della *negazione* della condizione precedente (termina perché x=1 rende falsa la condizione)

```c
#include<stdio.h>

int main()
{
    int x,a,b,min,mcd;
    printf("inserire a e b: ");
    scanf("%d",&a);
    scanf("%d",&b);
    x = 1;
    mcd = 1;
    if (a < b)
        min = a;
    else
        min = b;
    x = min;
    while ((a % x)!= 0  ||  (b % x) != 0)
        x = x - 1;

    mcd = x;
    printf("massimo comune divisore: %d", mcd);
    return 0;
}
```

Equivalente al precedente:
rispetta la legge di De Morgan
!(C && D) ⇔ !C || !D

Esempio: MCD

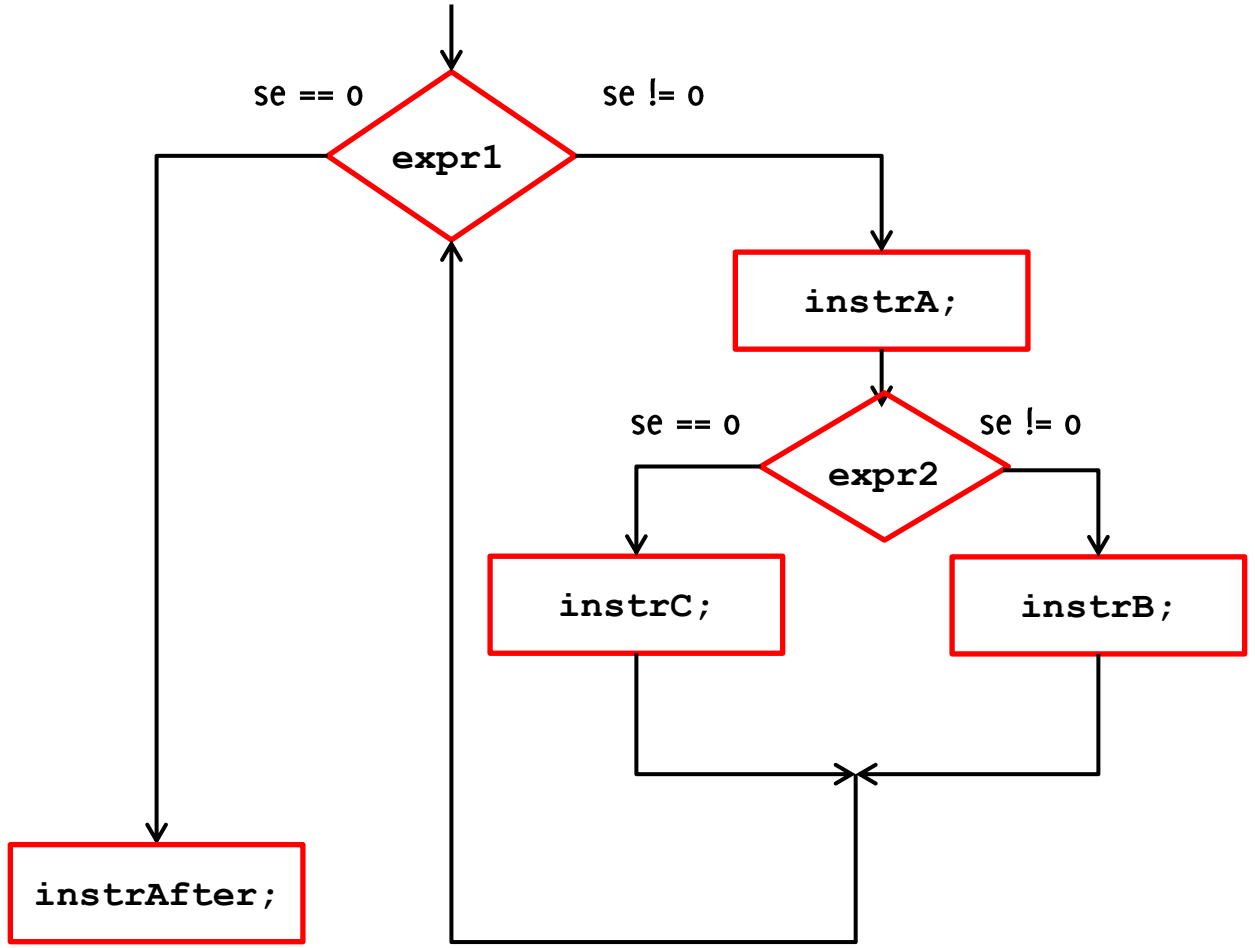
```
/* Codifica dell'algoritmo di Euclide */

printf("dammi due valori interi positivi : ");
scanf("%d", &a);
scanf("%d", &b);
while ( a != b ) {
      if ( a > b )
            a = a - b;
      else
            b = b - a;
}
mcd = a;
….
```

Scegliere a o b come MCD è indifferente: al momento dell'uscita dal ciclo while, infatti, a e b sono **certamente** uguali.

Cicli Annidati…

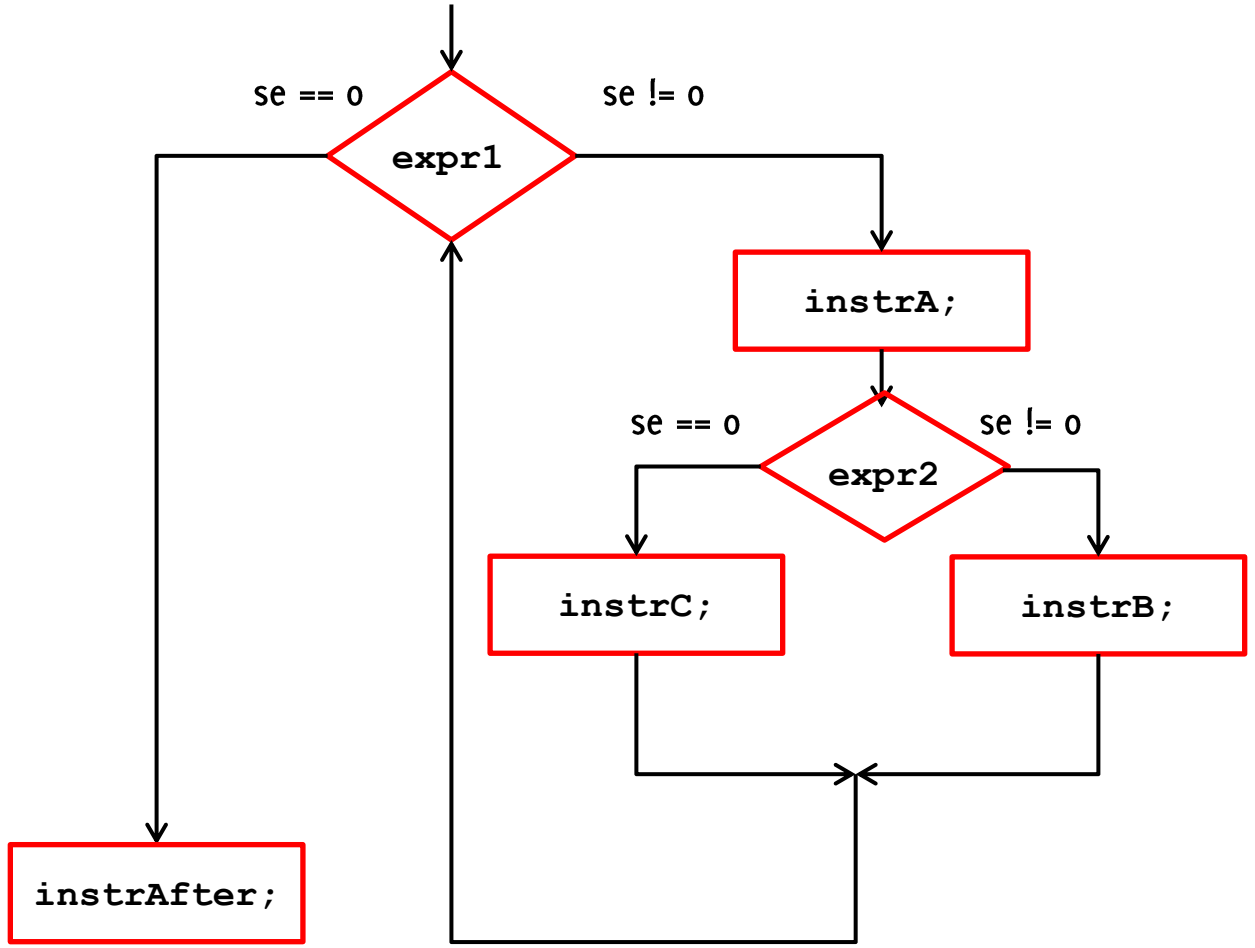
Ovviamente anche il corpo del **`while`** può contenere altri costrutti (**`while`** / **`if`** o altri che vedremo poi)



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;
```



Esercizi (TODO)

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 #)

Scrivere un programma che richieda all'utente di inserire due interi e ne calcola il massimo comune divisore.

Modificarlo per provare meno divisori del minimo tra gli input, utilizzando variabili di flag.

Scrivere un programma che stampa la tabella pitagorica

- **Modificarlo per stampare solo la parte triangolare alta/ solo la parte triangolare bassa / 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

Esercizio

Fare un programma che stampa l'intera tabella ASCII a 8 bit

```
"C:\Users\Giacomo\Dropbox (DEIB)\Didattica\2023_Informatica_A_Boracchi\Lez5_codes\tabellaAscii.exe"
  1) ☺   2) ☻   3) ♥   4) ♦   5) ♣   6) ♠   7)    8)   9)        10)
  14) ♫  15) ☼  16) ►  17) ◄  18) ↕  19) !!  20) ¶
 21) §  22) ▬  23) ↨  24) ↑  25) ↓  26) →  27) ←  28) ∟  29) ↔  30) ▲    31) ▼  32)    33) !  34) "  35) #  36) $  37) %  38) &  39) '  40) (
 41) )  42) *  43) +  44) ,  45) -  46) .  47) /  48) 0  49) 1  50) 2    51) 3  52) 4  53) 5  54) 6  55) 7  56) 8  57) 9  58) :  59) ;  60) <
 61) =  62) >  63) ?  64) @  65) A  66) B  67) C  68) D  69) E  70) F    71) G  72) H  73) I  74) J  75) K  76) L  77) M  78) N  79) O  80) P
 81) Q  82) R  83) S  84) T  85) U  86) V  87) W  88) X  89) Y  90) Z    91) [  92) \  93) ]  94) ^  95) _  96) `  97) a  98) b  99) c 100) d
101) e 102) f 103) g 104) h 105) i 106) j 107) k 108) l 109) m 110) n   111) o 112) p 113) q 114) r 115) s 116) t 117) u 118) v 119) w 120) x
121) y 122) z 123) { 124) | 125) } 126) ~ 127) ⌂ 128) Ç 129) ü 130) é   131) â 132) ä 133) à 134) å 135) ç 136) ê 137) ë 138) è 139) ï 140) î
141) ì 142) Ä 143) Å 144) É 145) æ 146) Æ 147) ô 148) ö 149) ò 150) û   151) ù 152) ÿ 153) Ö 154) Ü 155) ø 156) £ 157) Ø 158) × 159) ƒ 160) á
161) í 162) ó 163) ú 164) ñ 165) Ñ 166) ª 167) º 168) ¿ 169) ® 170) ¬   171) ½ 172) ¼ 173) ¡ 174) « 175) » 176) ░ 177) ▒ 178) ▓ 179) │ 180) ┤
181) Á 182) Â 183) À 184) © 185) ╣ 186) ║ 187) ╗ 188) ╝ 189) ¢ 190) ¥   191) ┐ 192) └ 193) ┴ 194) ┬ 195) ├ 196) ─ 197) ┼ 198) ã 199) Ã 200) ╚
201) ╔ 202) ╩ 203) ╦ 204) ╠ 205) ═ 206) ╬ 207) ¤ 208) ð 209) Ð 210) Ê   211) Ë 212) È 213) ı 214) Í 215) Î 216) Ï 217) ┘ 218) ┌ 219) █ 220) ▄
221) ¦ 222) Ì 223) ▀ 224) Ó 225) ß 226) Ô 227) Ò 228) õ 229) Õ 230) µ   231) þ 232) Þ 233) Ú 234) Û 235) Ù 236) ý 237) Ý 238) ¯ 239) ´ 240) -
241) ± 242) ‗ 243) ¾ 244) ¶ 245) § 246) ÷ 247) ¸ 248) ° 249) ¨ 250) ·   251) ¹ 252) ³ 253) ² 254) ■
Process returned 0 (0x0)   execution time : 0.147 s
Press any key to continue.
```

Intermezzo: Cosa fa questo algoritmo?

1. Alzatevi tutti in piedi
2. Ognuno di voi vale 1
3. **Ripeti:** Ciascuno cerca un compagno/a ancora in piedi
4. **Se** non avete trovato un compagno, il vostro valore non cambia e dovete restare in piedi
5. **Altrimenti** ogni coppia somma i loro valori, il risultato è il nuovo valore di ciascuno
6. Uno dei due si deve sedere e l'altro deve restare in piedi
7. Ricominciate dal punto 3, **finchè** non resta in piedi una sola persona in tutta la stanza
8. Il valore dell'ultima persona rimasta in piedi è

Costrutto Iterativo: `for`, la sintassi

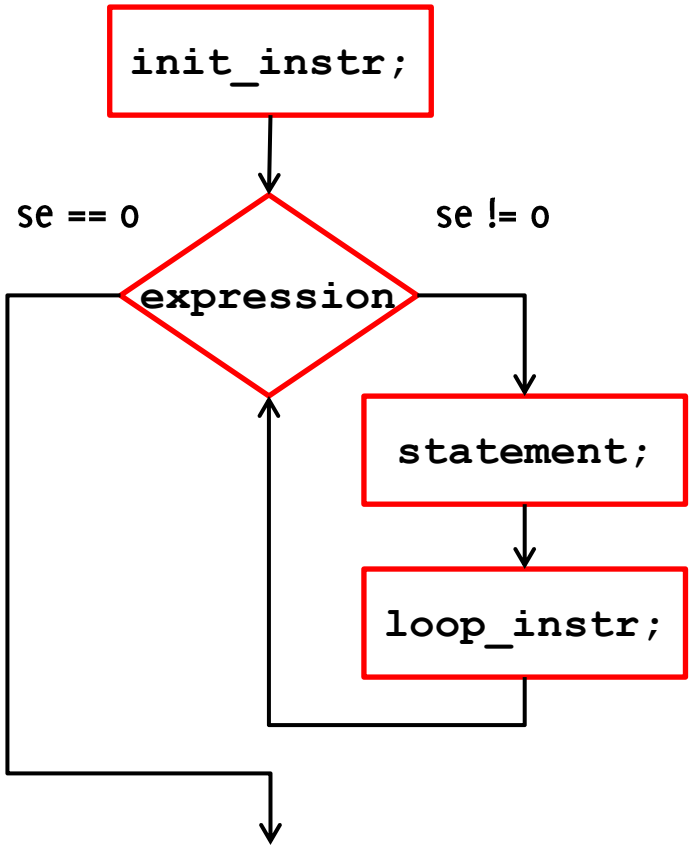
```
for(init_instr; expression; loop_instr)
      statement
```

- **`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`, l'esecuzione

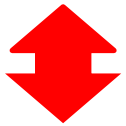
```
for(init_instr; expression; loop_instr)
    statement
```

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`**



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));
…
```

for vs while

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

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

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

Esempio `for`

Stampare i primi 100 numeri

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);
```

Scrivere una soluzione basata su **`for`** per gli esercizi

- Le tabelline,
- Le tabelline il triangolo inferiore

`break` e `continue`

L'istruzione **`break`** termina l'esecuzione dei seguenti costrutti

- **`while`**, **`do while`** e **`for`** (costsrutti 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 fino a 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 infiniti) 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: Alternativa e break e continue

Es: Scrivere un ciclo con che richiede una serie di valori interi e e stampa a schermo

- Non più di **N** richieste
- Solo i valori positivi inseriti
- Interrompendo l'elaborazione al primo valore nullo incontrato

Esempi con continue e break

```
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 */
}
```

MOLTO IMPORTANTE: come farne a meno

```
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*/
}
```

Importanza delle variabili di flag

Es: Scrivere un ciclo con che richiede una serie di valori interi e e stampa a schermo

- Non più di **N** richieste
- Solo i valori positivi inseriti
- Interrompendo l'elaborazione al primo valore nullo incontrato
- **Al termine, stampare un messaggio qualora fossero stati inseriti 10 numeri positivi**

MOLTO IMPORTANTE: come farne a meno

```
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");
```

MOLTO IMPORTANTE: come farne a meno

```
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");
```

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

MOLTO IMPORTANTE: come farne a meno

```
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");
```

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

Alcune precisazioni…

Riprendiamo dei dettagli e vediamo gli errori più frequenti

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. Non esistono altri identificatori per quella cella.

Non si possono usare alcune espressioni come identificatori perché fanno riferimento a parole **riservate, le keywords**.

- Es: **`if`**, **`for`**, **`switch`**, **`while`**, **`main`**, **`printf`**, **`scanf`**, **`int`**, **`float`**, etc…

Note: 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 typo

Ogni sequenza di caratteri in un codice di un programma C può essere:

- Un nome di variabile
- Un nome di funzione
- Una Keyword

Provare ad usare una variabile **a** senza averla dichiarata. Il compilatore risponde:

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

Note: 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

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:

- come per le variabili si possono raggruppare dichiarazioni di più costanti dello stesso tipo
- Il compilatore segnala come errore ogni assegnamento a una costante nella parte eseguibile

Le Costanti (cnt)

Usare le costanti è utile perché:

- L'identificatore suggerisce il significato di un 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;
```

Note: Abbreviazioni nell'Assegnamento

Istruzioni della forma

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

si possono scrivere come

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

`b = b + 7;` $\Longrightarrow$ `b +=7;` (Idem con altri operatori)

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

`a = a + 1;` $\Longrightarrow$ `a++;`

`b = b - 1;` $\Longrightarrow$ `b--;`

Gli operatori ++ e --

Attenzione all'uso degli operatori incremento e decremento: **++** e **--**

++i è un'espressione che prima incrementa i e poi ne fornisce il valore (**pre-**incremento)

i++ è un'espressione che prima fornisce il valore di i e poi la incrementa (**post-**incremento)

- Esempio: sia la dichiarazione

```
int c = 5;
printf ("%d", ++c);      stampa 6
printf ("%d", c++);      stampa 5
```

in entrambi i casi al termine **c** ha valore **6**

Quando la variabile non è parte di una espressione, il pre-incremento e il post-incremento hanno lo stesso effetto

All'interno di una espressione, però, è determinante la regola del pre- e del post-

Quindi **if (i++ > 0)**... è **diverso** da

```
if ( ++i > 0 )...
```

Note: caratteri di conversione

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

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

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

`switch-case`, la 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]
     }
```

`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. Questi sono delimitati dal case seguente

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

`switch-case`, l'esecuzione

1. Viene valutata **`expression`** (eventualmente convertita)
2. Si controlla se **`expression`** 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 n**Cons;**

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 n**Cons;**

TODO: `switch-case`

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

```c
#include<stdio.h>

int main ()
{
    float op1, op2, tot;
    char operazione;
    int errore = 0;
    printf ("Operazione (+,-,*,/): "); scanf ("%c", &operazione);
    printf ("Primo operando: "); scanf ("%f", &op1);fflush(stdin);
    printf ("Secondo operando: "); scanf ("%f", &op2);fflush(stdin);

    switch (operazione)
    {
        case '+': tot = op1 + op2;
            break;
        case '-': tot = op1 - op2;
            break;
        case '*': tot = op1 * op2;
            break;
        case '/': if (op2 != 0)
            {
                tot = op1 / op2;
            } else {
                errore = 1;
            }
            break;
        default: printf ("Operazione?\n");
            errore = 1;
    }
    if (errore == 0)
          printf ("Risultato: %f\n", tot);

    else
          printf ("Errore!\n");

    return 0;
}
```

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 e a volte è necessario entrare in modalità debug per trovarli

Errori 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?

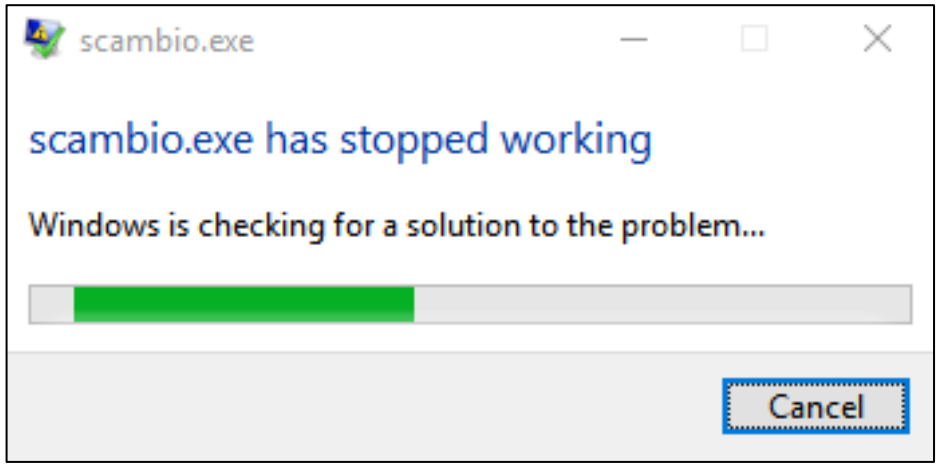
error: 'prinf' undeclared (first use in this function)

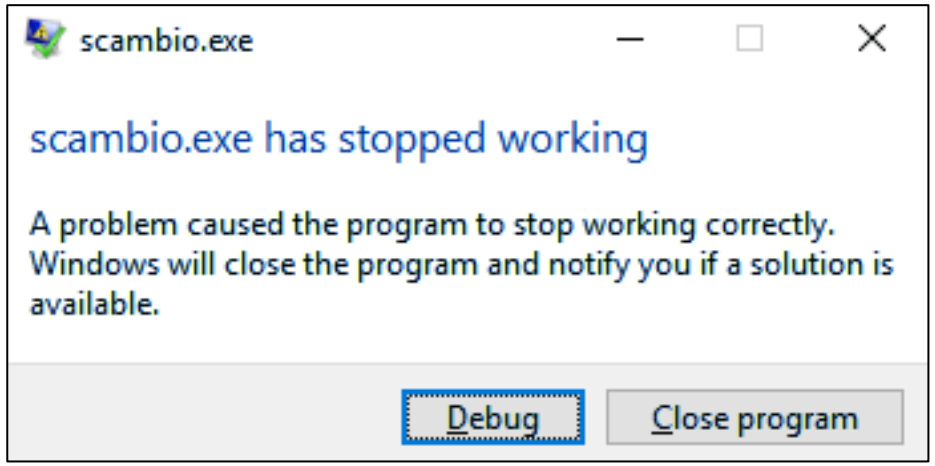
Errori a Run-Time

Dimenticare **`&`** nelle variabili della **`scanf`** :
il compilatore non lo rileva! Errore rilevabile a **run-time**, quando il valore dallo stdInput 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.





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!

Sbagliare lo specificatore di formato in una scanf o printf

```
int main(){
int x;
   printf("inserire x: ");
   scanf("%f", &x);
   printf("\nx = %d",x);
   return 0;
}
```

```
inserire x: 4

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

Sbagliare lo specificatore di formato in una scanf o printf

```
int main(){
float x;
  printf("inserire x: ");
  scanf("%f", &x);
  printf("\nx = %d",x);
  return 0;
}
```

```
inserire x: 4.98

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

E' invece possibile…

E' invece possibile stampare i char come interi

```
int 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.
```

Error a Run-Time `i = i++;`

L'istruzione **`i++`**

corrisponde all'assegnamento **`i = i + 1;`**

Non ha quindi senso **`i = i++;`** che corrisponde a

```
i = (i = i + 1);
```

Il comportamento del compilatore non è ben definito per queste istruzioni.

Error 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
- *Es*

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

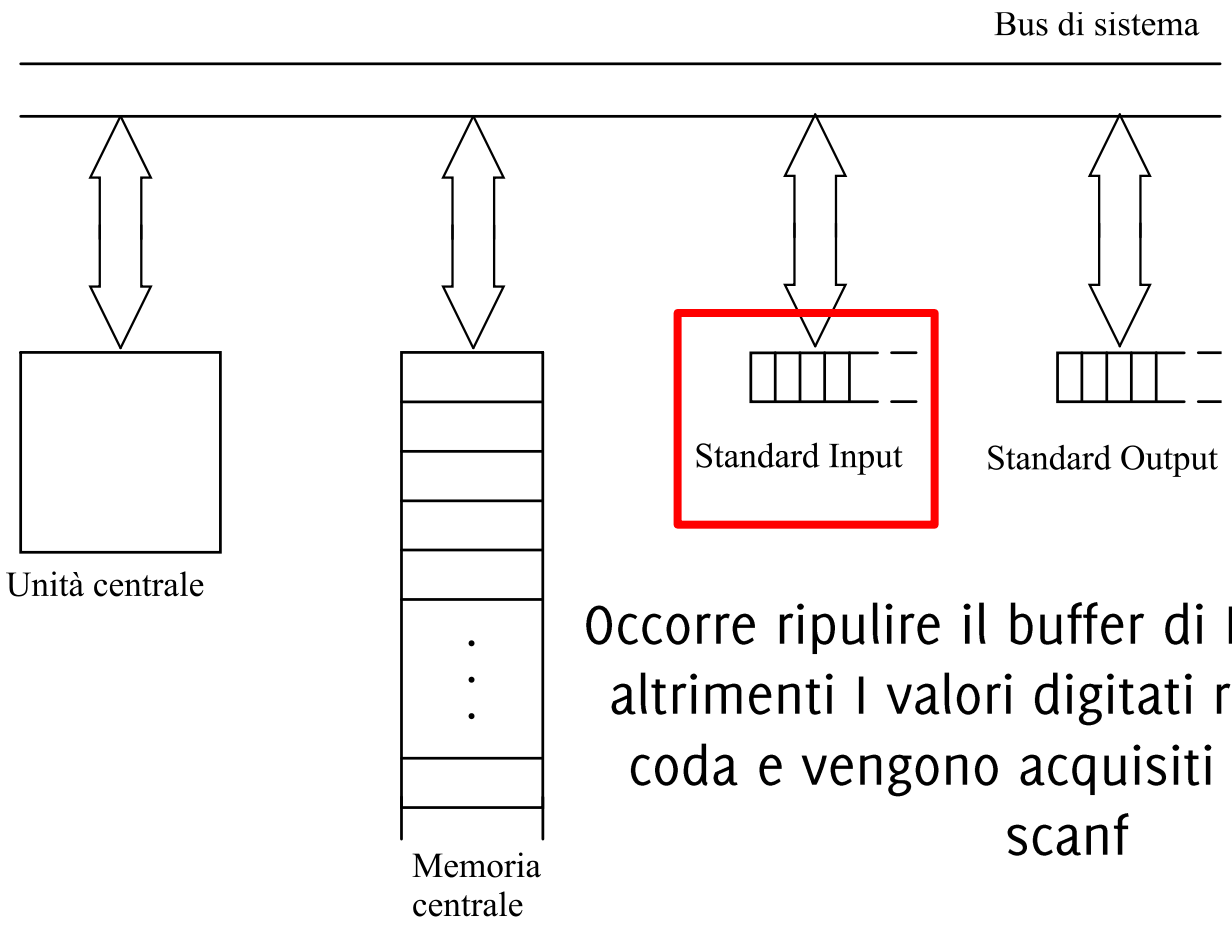
  - Stampa vero perché **printf("Vero");** è fuori dall' **if**
  - Se ci fosse stato un **else** il compilatore avrebbe dato errore: esiste un **else** non associato ad un **if**

- Il ; viene usato nel **do while** perché il costrutto termina con il **while(expression);**

Note sull'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.

Descrizione del linguaggio C mediante la macchina C che esegue i programmi codificati.



Occorre ripulire il buffer di Ingresso altrimenti I valori digitati rimangono in coda e vengono acquisiti al prossimo scanf

Lettura da Standard Input: `scanf`

Acquisizione di caratteri: richiede un ulteriore comando

```
char x;

scanf("%c", &x); fflush(stdin);
```

Altrimenti l'invio inserito per terminare l'acquisizione di **x** rimane in un buffer (**stdin**) e viene acquisito nella successiva acquisizione di caratteri, i.e., in eventuali **scanf("%c", &altraVariabile)** che seguono!

Lettura da Standard Input: `scanf`

Acquisizione di caratteri: richiede un ulteriore comando

```
char x;
scanf("%c", &x); fflush(stdin);
```

Altrimenti l'invio inserito per terminare l'acquisizione di **x** rimane in un buffer (**stdin**) e viene acquisito nella successiva acquisizione di caratteri, i.e., in eventuali **scanf("%c", &altraVariabile)** che seguono!

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

fflush(stdin) e scanf("%*c");

Un'alternativa a **`fflush(stdin)`** è aggiungere un'acquisizione di un carattere "a vuoto"

```
scanf("%*c")
```

il `%*c` indica che verrà acquisito un carattere e buttato via

fflush(stdin) e scanf("%*c");

Un'alternativa a **fflush(stdin)** è aggiungere un'acquisizione di un carattere "a vuoto"

```
scanf("%*c")
```

il `%*c` indica che verrà acquisito un carattere e buttato via

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

fflush(stdin) e scanf("%*c") e scanf(" %c") ;

Un'alternativa a **fflush(stdin)** è aggiungere un'acquisizione di un carattere "a vuoto"

```
scanf(" %c")
```

lo spazio che precede il `%c` indica che verranno scartati tutti gli spazi e l'invii che sono nel buffer e che precedono gli altri caratteri

```
int main()
{ char a,b;
scanf("%c", &a);
scanf(" %c", &b);// butta via tutti gli spazi
                 // ed invio quindi acquisisce
printf("%c %c", a, b); }
```

Confronto e Assegnamento

L'opratore 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.

Esempio

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

Cosa fa?

Esempio

```
#include<stdio.h>

int main()

{

        char a, b;

        b = '2';

        a = b == '0';

        printf("%d" , a);

}
```

Cosa fa?

Associa ad **a** il valore **1** se **b** è **0**, **1** altrimenti.

Viene letto

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

Se **b = '2';** Stampa 0

Se **b = '0';** Stampa 1

Se **b = 0;** Stampa 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`**