

# Corso “Programmazione 1”

## Capitolo 03: Istruzioni

Docente: Marco Roveri - [marco.roveri@unitn.it](mailto:marco.roveri@unitn.it)

Esercitatori: Giovanna Varni - [giovanna.varni@unitn.it](mailto:giovanna.varni@unitn.it)

Andrea E. Naimoli - [andrea.naimoli@unitn.it](mailto:andrea.naimoli@unitn.it)

TBC -

C.D.L.: Informatica (INF)

A.A.: 2025-2026

Luogo: DISI, Università di Trento

URL: <https://shorturl.at/cxAAS>



Ultimo aggiornamento: 24 settembre 2025

# Terms of Use and Copyright

## USE

This material (including video recording) is intended solely for students of the University of Trento registered to the relevant course for the Academic Year 2025-2026.

## SELF-STORAGE

Self-storage is permitted only for the students involved in the relevant courses of the University of Trento and only as long as they are registered students. Upon the completion of the studies or their abandonment, the material has to be deleted from all storage systems of the student.

## COPYRIGHT

The copyright of all the material is held by the authors. Copying, editing, translation, storage, processing or forwarding of content in databases or other electronic media and systems without written consent of the copyright holders is forbidden. The selling of (parts) of this material is forbidden. Presentation of the material to students not involved in the course is forbidden. The unauthorised reproduction or distribution of individual content or the entire material is not permitted and is punishable by law.

The material (text, figures) in these slides is authored mostly by Roberto Sebastiani, with contributions by Marco Roveri, Alessandro Armando, Enrico Giunchiglia e Sabrina Recla.

# Struttura di un programma

- Un programma consiste in un insieme di funzioni, eventualmente suddivise in più file
  - La funzione che costituisce il programma principale si deve necessariamente chiamare `main`
- Ogni programma contiene una lista di istruzioni di ogni tipo:  
**istruzioni semplici o istruzioni strutturate**

Esempio:

```
(...)
int main() {
    int x=2, y=6, z;
    z=x*y;
    return 0;
}
```

# Istruzioni semplici

- Le **istruzioni semplici** sono la base delle istruzioni più complesse (**istruzioni strutturate**)
- Sono sempre terminate da un punto-e-virgola “;”
- Si distinguono in:
  - definizioni/dichiarazioni** (declaration-statement) di
    - variabili, es. `int x, y, z;`
    - costanti, es. `const int kilo=1024;`
  - espressioni** (expression-statement)
    - di input, es: `cin >> x`
    - di output, es: `cout << 3*x`
    - di assegnamento, es. `x=2*(3-y)`
    - matematiche, es. `(x-3)*sin(x)`
    - logiche, es. `x==y && x!=z`
    - costanti, es. `3*12.7`
    - condizionali (a seguire)

ogni espressione seguita da un “;” è anche un’istruzione

# L'espressione condizionale

- Sintassi: `exp1 ? exp2 : exp3 :`  
Se `exp1` è vera equivale a `exp2`, altrimenti equivale a `exp3`

## Esempio

```
prezzo = valore * peso * (peso>10) ? 0.9 : 1;
```

- se il peso è maggiore di 10, equivale a:

```
prezzo = valore * peso * 0.9;
```

- altrimenti, equivale a

```
prezzo = valore * peso * 1;
```

## Esempio di uso di espressione condizionale:

```
{ ESEMPI_ITE/espressione_condizionale.cc }
```

# Istruzioni strutturate

- Le **istruzioni strutturate** consentono di specificare azioni complesse
- Si distinguono in
  - **istruzione composta** (compound-statement)
  - **istruzioni condizionali** (conditional-statement)
  - **istruzioni iterative** (iteration-statement)
  - **istruzioni di salto** (jump-statement)

# Istruzione Composta

- Trasforma una sequenza di istruzioni in una singola istruzione
  - sequenza delimitata per mezzo della coppia di delimitatori '{' e '}'
  - la sequenza così delimitata è detta **blocco**
- Le definizioni possono comparire in qualunque punto del blocco
  - sono visibili solo all'interno del blocco
  - possono accedere ad oggetti definiti esternamente
  - in caso di identificatori identici, prevale quello più interno

```
{  
    int a=4;  
    a*=6;  
    char b='c';  
    b+=3;  
}
```

Esempio di blocchi e visibilità:  
{ ESEMPI\_ITE/visibilita.cc }

# L'Istruzione Condizionale if-then

- Istruzione “**if**” semplice (if-then):

- Sintassi:

- if** (exp)  
    istruzione1

- Significato: se exp è vera, viene eseguita istruzione1, altrimenti non viene eseguito nulla

- istruzione1 può a sua volta essere un'istruzione complessa

## Esempio

```
if (x!=0)
    y=1/x;
```

## Esempio di if-then:

```
{ ESEMPI_ITE/divisibilita.cc }
```

# L'Istruzione Condizionale if-then-else

- Istruzione “**if**” composta (if-then-else):
  - Sintassi: **if** (exp) istruzione1 **else** istruzione2
  - Significato: se exp è vera, viene eseguita istruzione1, altrimenti viene eseguita istruzione2
- istruzione1 e istruzione2 possono essere a loro volta istruzioni complesse (un blocco, un’altro if-then-else, ...)

## Esempio

```
if (x<0)
    y=-x;
else
    y=x;
```

## Esempio di if-then-else:

```
{ ESEMPI_ITE/divisibilita2.cc }
```

# If annidati

- Nei costrutti if-then e if-then-else, istruzione1 e istruzione2 possono essere a loro volta istruzioni complesse  
(un blocco, un'altro if-then-else, ...)
- L'annidamento di if-then-else e l'uso di operatori logici permettono di costruire strutture decisionali complesse
  - Uso di if-then-else annidati,...:  
  { ESEMPI\_ITE/eq\_1grado.cc }
  - ..., con diverso ordine,...:  
  { ESEMPI\_ITE/eq\_1grado2.cc }
  - ... e con operatori logici:  
  { ESEMPI\_ITE/eq\_1grado3.cc }

L'indentazione del codice è importantissima!!!

- Individua a colpo d'occhio l'inizio e la fine del codice/blocco
- Contribuisce alla leggibilità del codice

## If-then-else annidati: esempi

- Esempi di alternative all'utente:  
  { ESEMPI\_ITE/conversione2.cc }
- Esempio di “Dangling else”:  
  { ESEMPI\_ITE/dangling\_else.cc }
- Esempio di “Dangling else” (2):  
  { ESEMPI\_ITE/dangling\_else2.cc }
- Errore tipico con “if”:  
  { ESEMPI\_ITE/ifeq\_err.cc }
- Versione corretta:  
  { ESEMPI\_ITE/ifeq\_corr.cc }
- Minimo tra due numeri:  
  { ESEMPI\_ITE/minimo.cc }
- Minimo tra tre numeri:  
  { ESEMPI\_ITE/minimo2.cc }
- Scelte tra valori multipli:  
  { ESEMPI\_ITE/simple\_calc.cc }

# L'Istruzione Condizionale **switch**

- Sintassi

```
switch (exp) {  
    case const-exp1: istruzione1 break;  
    case const-exp2: istruzione2 break;  
    ...  
    default: istruzione-default  
}
```

- L'esecuzione dell'istruzione **switch** consiste

- nel calcolo dell'espressione `exp`
- nell'esecuzione dell'istruzione corrispondente all'alternativa specificata dal valore calcolato
- se nessuna alternativa corrisponde, se esiste, viene eseguita `istruzione-default`

Scelte tra valori multipli con **switch**:

{ ESEMPI\_ITE/simple\_calc2.cc }

# Scelte multiple con **switch**

- Se dopo l'ultima istruzione di un'alternativa non c'è un **break**, viene eseguita anche l'alternativa successiva
- Questo comportamento è **sconsigliato** ma può essere giustificato in alcuni casi

## Esempio

```
switch (giorno)
{ case lun: case mar:
  case mer: case gio:
  case ven: orelavorate+=8; break;
  case sab: case dom: break;
}
```

# Esercizi Proposti

Esercizio su istruzioni condizionali:  
{ ESEMPI\_ITE/ESERCIZI\_PROPOSTI.txt }

# L'Istruzione Iterativa **while** (while-do)

- Sintassi: **while** (exp) istruzione
  - exp è un'espressione Booleana
  - istruzione può essere un'istruzione complessa
- L'esecuzione dell'istruzione **while** comporta
  1. il calcolo dell'espressione exp
  2. se exp è vera, l'esecuzione di istruzione e la ripetizione dell'esecuzione dell'istruzione **while**
- istruzione potrebbe non essere mai eseguita
- È possibile generare loop infiniti.

## Nota

exp tipicamente contiene almeno una variabile (**variabile di controllo** del ciclo), che viene modificata in istruzione per far convergere exp verso uno stato in cui diventi falsa.

# L'Istruzione Iterativa **while**: Esempi I

- Ripetizione pedissequa di un'operazione (contatore crescente):  
  { ESEMPI\_LOOP/stampaciao.cc }
- ... (contatore decrescente):  
  { ESEMPI\_LOOP/stampaciao2.cc }
- ..., con loop infinito:  
  { ESEMPI\_LOOP/stampaciao\_infloop.cc }
- Somma con accumulatore:  
  { ESEMPI\_LOOP/sommainteri\_while.cc }
- Prodotto con accumulatore:  
  { ESEMPI\_LOOP/fact\_while.cc }
- Condizione di uscita diversa da conteggio:  
  { ESEMPI\_LOOP/divisibile.cc }

# L'Istruzione Iterativa **while**: Esempi II

- Ripetizione di comando a menu:  
  { ESEMPI\_LOOP/conversione3\_while.cc }
- Somma con accumulatore, con conteggio:  
  { ESEMPI\_LOOP/serie\_while.cc }
- Somma con accumulatore, con cond. uscita :  
  { ESEMPI\_LOOP/serie1\_while.cc }
- Uso di “cin loops”:  
  { ESEMPI\_LOOP/cin\_loop.cc }
- Stessa cosa, ma con fail:  
  { ESEMPI\_LOOP/cin\_loop\_equivalent.cc }

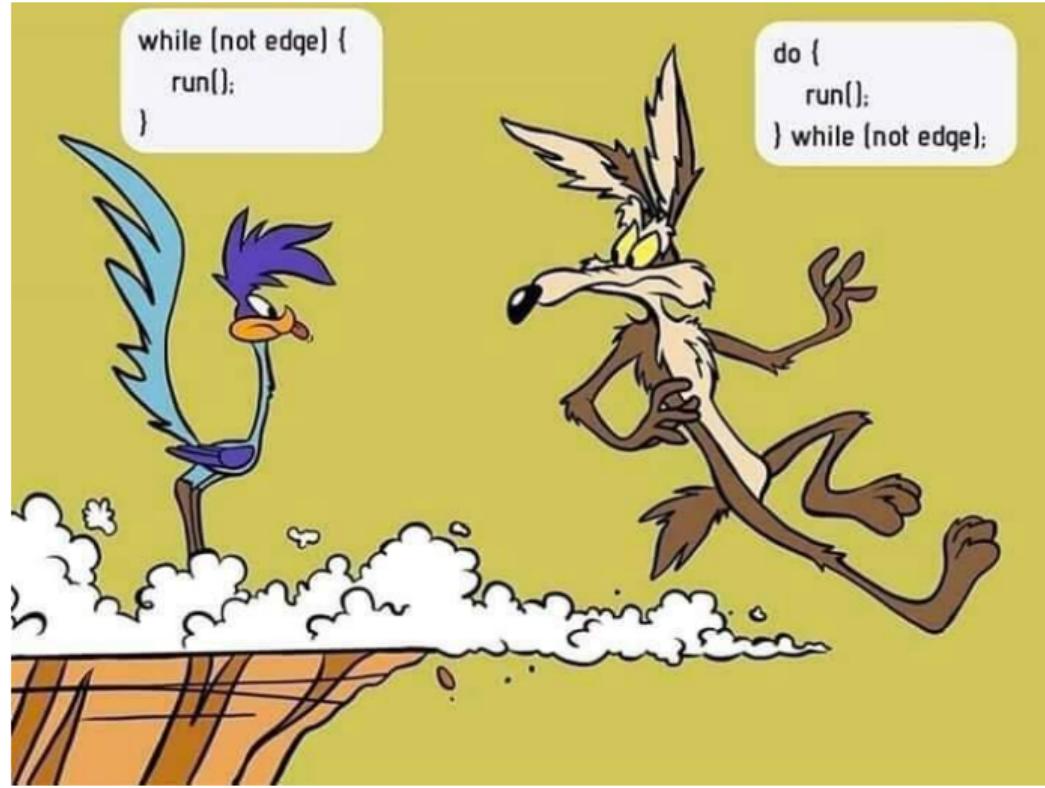
# L'Istruzione Iterativa do (do-while)

- Sintassi: **do** { istruzione } **while** (exp);
  - exp è un'espressione Booleana
  - istruzione può essere un'istruzione complessa
- L'esecuzione dell'istruzione **do** comporta
  1. l'esecuzione di istruzione
  2. il calcolo dell'espressione exp
  3. se exp è vera, la ripetizione dell'esecuzione dell'istruzione **do**
- istruzione **viene sempre eseguita almeno una volta**
- È la meno usata tra le istruzioni iterative

# L'Istruzione Iterativa do: Esempi

- Somma con accumulatore (do):  
  { ESEMPI\_LOOP/sommainterido.cc }
- Ripetizione di comando a menu (do):  
  { ESEMPI\_LOOP/conversione3\_do.cc }
- Conversione di base:  
  { ESEMPI\_LOOP/base.cc }

# While-Do vs. Do-While



©Warner Bros Inc.

# L'Istruzione Iterativa **for**

- Sintassi: **for** ( init; exp; agg) istruzione
  - init è un'istruzione di **inizializzazione** delle variabili di controllo
  - exp è un'espressione Booleana
  - istruzione può essere un'istruzione complessa
  - agg è un'istruzione di **aggiornamento** delle variabili di controllo
- L'esecuzione dell'istruzione **for** comporta:
  1. l'esecuzione di init
  2. il calcolo dell'espressione exp
  3. se exp è vera, viene eseguita istruzione, poi agg, e si ricomincia dal passo 2.
- È la più usata tra le istruzioni iterative.
- Si possono definire variabili di controllo **interne al ciclo**:  
**for** (**int** i=0; i<MAXDIM; i++) {<i occorre solo qui>}

Consente di separare le istruzioni di controllo del ciclo e concentrarle tutte in un'unica riga  $\Rightarrow$  miglior praticità e leggibilità del codice.

# Cicli for e while

```
for ( init; exp; agg )
```

    istruzione

equivale a:

```
{ init;  
  while ( exp ) {  
    istruzione  
    agg;  
}; }
```

## Esempio

```
for (int i=1; i<10; i++)  
    x*=2;
```

↔

```
{ int i=1;  
  while (i<10) {  
    x*=2;  
    i++;  
  };  
}
```

# L'Istruzione Iterativa `for`: Esempi I

- Prodotto con accumulatore (`for`):  
  { ESEMPI\_LOOP/fact\_for.cc }
- Somma con accumulatore (numero iterazioni) (`for`):  
  { ESEMPI\_LOOP/serie\_for.cc }
- Somma con accumulatore (cond. uscita) (`for`):  
  { ESEMPI\_LOOP/serie\_for1.cc }
- For annidati:  
  { ESEMPI\_LOOP/doublefor.cc }

# L'Istruzione Iterativa `for`: Esempi II

- Condizione iniziale multipla con `for`:  
  { ESEMPI\_LOOP/serie\_for1\_2init.cc }
- Cond. iniziale multipla & uscita multipla con `for`:  
  { ESEMPI\_LOOP/serie\_for1\_2init2.cc }
- Incremento come input dato dall'utente:  
  { ESEMPI\_LOOP/minmax.cc }
- Doppio incremento:  
  { ESEMPI\_LOOP/doublecontrol.cc }

# Gli Invarianti di un Ciclo (Loop Invariant)

- Tecnica per la verifica di correttezza dei cicli (proprietà  $P$ )
- Idea: suddividere la proprietà desiderata  $P$  della correttezza del ciclo in una sequenza di affermazioni  $P_0, P_1, \dots, P_n$ , in modo che:
  - (1)  $P_0$  sia vera immediatamente prima che il ciclo inizi  
(dopo l'inizializzazione!)
  - (2) Per ogni indice di ciclo  $i \in \{1, \dots, n\}$ :  
se  $P_{i-1}$  è vera prima dell'inizio del ciclo  $i$ -esimo (ed è verificata la condizione di permanenza del ciclo), allora  $P_i$  è vera alla fine del ciclo  $i$ -esimo  
(e quindi immediatamente prima dell'inizio del ciclo  $(i+1)$ -esimo)
  - (3) Alla fine dell'ultimo ciclo ( $n$ -esimo),  $P_n$  (e la negazione della condizione di permanenza) implica la proprietà  $P$
- Tipicamente (2) è il passo più critico
- $P_i$  a volte ovvie, a volte molto complesse (or, if-then-else, ...)  
 $\implies$  problema **individabile** in generale
- Talvolta necessarie variabili ausiliarie addizionali
- Talvolta si adottano convenzioni per gestire il caso  $i = 0$ :  
(la somma di 0 elementi è 0, il prodotto di 0 elementi è 1, ...)

## Esempio: fattoriale

```
i = 1;  
fact = 1;  
while (i<=n) {  
    fact *= i;  
    i++;  
}
```

- Proprietà  $P$ : dopo il ciclo, `fact` vale il prodotto dei primi  $n$  numeri
- Invariante  $P_i$ : `fact` vale il prodotto dei primi  $i$  numeri
  - ✓(1) Prima del ciclo, `fact` vale il prodotto dei primi 0 numeri (cioè 1)
  - ✓(2) Prima dell' $i$ -esimo ciclo `fact` vale il prodotto dei primi  $i-1$  numeri  
    ⇒ dopo l' $i$ -esimo ciclo `fact` vale il prodotto dei primi  $i$  numeri
  - ✓(3) Alla fine dell'ultimo ciclo ( $n$ -esimo),  $P_n$  (più la negazione della condizione di permanenza del ciclo) implica la proprietà  $P$

### Nota

“Dopo l' $i$ -esimo ciclo”  $i$  è incrementato di 1. (Ex: dopo il 3<sup>o</sup> ciclo,  $i = 4$ ).

## Esempio: divisibilità per 2

```
ndiv2=0; tmp=num; // "tmp" ausiliaria
while ( tmp%2 == 0 ) {
    ndiv2++;
    tmp/=2;
}
```

- **Proprietà  $P$ :** dopo il ciclo,  $\text{tmp} \% 2 \neq 0$  e  $\text{tmp} * (2^{\text{ndiv2}}) == \text{num}$
- **Invariante  $P_i$ :**  $\text{tmp} * (2^{\text{ndiv2}}) == \text{num}$ 
  - ✓(1) Prima del ciclo,  $\text{tmp} * (2^0) == \text{num}$
  - ✓(2) Prima dell'i-esimo ciclo  $\text{tmp}$  è divisibile per due e  $\text{tmp} * (2^{\text{ndiv2}}) == \text{num}$   
     $\Rightarrow$  dopo l'i-esimo ciclo  $\text{tmp} * (2^{\text{ndiv2}}) == \text{num}$   
(infatti  $\text{tmp}/2 * (2^{(\text{ndiv2}+1)}) == \text{num}$ )
  - ✓(3) Alla fine dell'ultimo ciclo ( $n$ -esimo),  $P_n$  (più la negazione della condizione del ciclo)  
implica la proprietà  $P$  :  
 $\text{tmp} * (2^{\text{ndiv2}}) == \text{num}$  e  $\text{tmp} \% 2 \neq 0$

# Esercizi Proposti

Esercizi sui cicli:

{ ESEMPI\_LOOP/ESERCIZI\_PROPOSTI.txt }

Istruzioni di salto (**break**, **continue**, **goto**):  
come non si deve programmare in C/C++ !!!

# L'Istruzione di Salto **break**

L'istruzione **break** termina direttamente tutto il ciclo

```
while (...) {  
    ...  
    break;      // --+  
    ...        // |  
}  
    // |  
    // <-----+  
    // |
```

- Da evitare!  $\Rightarrow$  si può sempre fare modificando la condizione
- Semplice break (while):  
`{ ESEMPI_LOOP/break_while.cc }`
- Come evitare un break (while):  
`{ ESEMPI_LOOP/nobreak_while.cc }`

# L'Istruzione **return** in un loop (salto implicito)

L'istruzione **return** termina direttamente il ciclo (e l'intera funzione)

```
int main () {  
...  
while (...) {  
    ...  
    return 0;      // --+  
    ...          //   |  
}                //   |  
}                // <-----+  
                  //           +-----|
```

- Da evitare!  $\Rightarrow$  si può sempre fare modificando la condizione
- Semplice return (while):  
`{ ESEMPI_LOOP/return_while.cc }`
- Come evitare un return (while):  
`{ ESEMPI_LOOP/noreturn_while.cc }`

# L'Istruzione di Salto **continue**

L'istruzione **continue** termina il ciclo attualmente in esecuzione e passa al successivo

```
while (...) {  
    ...  
    continue; // --+  
    ... // |  
    // <-----+  
}
```

- Nel caso di ciclo **for** si salta all'istruzione di aggiornamento ignorando quanto segue nel codice
- **Da evitare!**  $\Rightarrow$  si può sempre fare lo stesso con un “if”
- Semplice **continue** (while):  
`{ ESEMPI_LOOP/continue.cc }`
- Come evitare **continue** (while):  
`{ ESEMPI_LOOP/nocontinue.cc }`

# Esempio differenze istruzioni di Salto

- Semplice programmino per differenze **break**, **continue** e **return** :  
{ ESEMPI\_LOOP/bcr.cc }

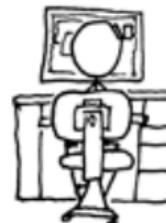
# L'Istruzione di Salto goto II

I COULD RESTRUCTURE  
THE PROGRAM'S FLOW  
OR USE ONE LITTLE  
'GOTO' INSTEAD.



EH, SCREW GOOD PRACTICE.  
HOW BAD CAN IT BE?

goto main\_sub3;  
\*COMPILE\*



© xkcd [www.xkcd.com](http://www.xkcd.com)

# L'Istruzione di Salto `goto` II

- Esempio di `goto`:  
{ ESEMPI\_LOOP/goto.cc }

Nota di servizio:

Nella soluzione di un testo di esame, **NON è ammesso** l'uso di `break`, `continue`, o `goto` (con l'importante eccezione dell'uso di `break` all'interno del costrutto `switch`), o di `return` all'interno di loop, pena l'annullamento dell'esercizio stesso.