

Corso “Programmazione 1”

Capitolo 03: Istruzioni

Docente: **Roberto Sebastiani** - roberto.sebastiani@unitn.it
Esercitori: **Mario Passamani** - mario.passamani@unitn.it
Alessandro Tomasi - alessandro.tomasi@unitn.it
C.D.L.: Informatica (INF)
Ing. Informatica, delle Comunicazioni ed Elettronica (ICE)
Studenti con numero di matricola pari
A.A.: 2019-2020
Luogo: DISI, Università di Trento
URL: disi.unitn.it/rseba/DIDATTICA/prog1_2020/

Outline

1 Istruzioni Semplici

2 Istruzioni Strutturate

3 Istruzioni Condizionali

4 Istruzioni Iterative

5 Istruzioni di Salto

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:

```
{ IF_THEN_ELSE_SWITCH/espressione_condizionale.cc }
```

Outline

- 1 Istruzioni Semplici
- 2 Istruzioni Strutturate**
- 3 Istruzioni Condizionali
- 4 Istruzioni Iterative
- 5 Istruzioni di Salto

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:

```
{ IF_THEN_ELSE_SWITCH/visibilita.cc }
```

Outline

- 1 Istruzioni Semplici
- 2 Istruzioni Strutturate
- 3 Istruzioni Condizionali**
- 4 Istruzioni Iterative
- 5 Istruzioni di Salto

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:

```
{ IF_THEN_ELSE_SWITCH/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`:

```
{ IF_THEN_ELSE_SWITCH//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, ...:
{ IF_THEN_ELSE_SWITCH/eq_1grado.cc }
 - ..., con diverso ordine, ...:
{ IF_THEN_ELSE_SWITCH/eq_1grado2.cc }
 - ... e con operatori logici:
{ IF_THEN_ELSE_SWITCH/eq_1grado3.cc }

L'indentazione del codice è importantissima!!!

- individua a colpo d'occhio l'inizio e la fine del codice
- contribuisce grandemente alla leggibilità del codice

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,...:
{ IF_THEN_ELSE_SWITCH/eq_1grado.cc }
 - ..., con diverso ordine,...:
{ IF_THEN_ELSE_SWITCH/eq_1grado2.cc }
 - ... e con operatori logici:
{ IF_THEN_ELSE_SWITCH/eq_1grado3.cc }

L'indentazione del codice è importantissima!!!

- individua a colpo d'occhio l'inizio e la fine del codice
- contribuisce grandemente alla leggibilità del codice

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,...:**
{ `IF_THEN_ELSE_SWITCH/eq_1grado.cc` }
 - ..., con diverso ordine,...:
{ `IF_THEN_ELSE_SWITCH/eq_1grado2.cc` }
 - ... e con operatori logici:
{ `IF_THEN_ELSE_SWITCH/eq_1grado3.cc` }

L'indentazione del codice è importantissima!!!

- individua a colpo d'occhio l'inizio e la fine del codice
- contribuisce grandemente alla leggibilità del codice

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,...:**
`{ IF_THEN_ELSE_SWITCH/eq_1grado.cc }`
 - **..., con diverso ordine,...:**
`{ IF_THEN_ELSE_SWITCH/eq_1grado2.cc }`
 - **... e con operatori logici:**
`{ IF_THEN_ELSE_SWITCH/eq_1grado3.cc }`

L'indentazione del codice è importantissima!!!

- individua a colpo d'occhio l'inizio e la fine del codice
- contribuisce grandemente alla leggibilità del codice

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,...:**
{ `IF_THEN_ELSE_SWITCH/eq_1grado.cc` }
 - **..., con diverso ordine,...:**
{ `IF_THEN_ELSE_SWITCH/eq_1grado2.cc` }
 - **... e con operatori logici:**
{ `IF_THEN_ELSE_SWITCH/eq_1grado3.cc` }

L'indentazione del codice è importantissima!!!

- individua a colpo d'occhio l'inizio e la fine del codice
- contribuisce grandemente alla leggibilità del codice

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,...:**
{ `IF_THEN_ELSE_SWITCH/eq_1grado.cc` }
 - **..., con diverso ordine,...:**
{ `IF_THEN_ELSE_SWITCH/eq_1grado2.cc` }
 - **... e con operatori logici:**
{ `IF_THEN_ELSE_SWITCH/eq_1grado3.cc` }

L'indentazione del codice è importantissima!!!

- individua a colpo d'occhio l'inizio e la fine del codice
- contribuisce grandemente alla leggibilità del codice

If-then-else annidati: esempi

- Esempi di alternative all'utente:
{ IF_THEN_ELSE_SWITCH/conversione2.cc }
- Esempio di "Dangling else":
{ IF_THEN_ELSE_SWITCH/dangling_else.cc }
- Esempio di "Dangling else" (2):
{ IF_THEN_ELSE_SWITCH/dangling_else2.cc }
- Errore tipico con "if":
{ IF_THEN_ELSE_SWITCH/ifeq_err.cc }
- Versione corretta:
{ IF_THEN_ELSE_SWITCH/ifeq_corr.cc }
- Minimo tra due numeri:
{ IF_THEN_ELSE_SWITCH/minimo.cc }
- Minimo tra tre numeri:
{ IF_THEN_ELSE_SWITCH/minimo2.cc }
- Scelte tra valori multipli:
{ IF_THEN_ELSE_SWITCH/simple_calc.cc }

If-then-else annidati: esempi

- Esempi di alternative all'utente:
{ IF_THEN_ELSE_SWITCH/conversione2.cc }
- Esempio di "Dangling else":
{ IF_THEN_ELSE_SWITCH/dangling_else.cc }
- Esempio di "Dangling else" (2):
{ IF_THEN_ELSE_SWITCH/dangling_else2.cc }
- Errore tipico con "if":
{ IF_THEN_ELSE_SWITCH/ifeq_err.cc }
- Versione corretta:
{ IF_THEN_ELSE_SWITCH/ifeq_corr.cc }
- Minimo tra due numeri:
{ IF_THEN_ELSE_SWITCH/minimo.cc }
- Minimo tra tre numeri:
{ IF_THEN_ELSE_SWITCH/minimo2.cc }
- Scelte tra valori multipli:
{ IF_THEN_ELSE_SWITCH/simple_calc.cc }

If-then-else annidati: esempi

- Esempi di alternative all'utente:
{ IF_THEN_ELSE_SWITCH/conversione2.cc }
- Esempio di "Dangling else":
{ IF_THEN_ELSE_SWITCH/dangling_else.cc }
- Esempio di "Dangling else" (2):
{ IF_THEN_ELSE_SWITCH/dangling_else2.cc }
- Errore tipico con "if":
{ IF_THEN_ELSE_SWITCH/ifeq_err.cc }
- Versione corretta:
{ IF_THEN_ELSE_SWITCH/ifeq_corr.cc }
- Minimo tra due numeri:
{ IF_THEN_ELSE_SWITCH/minimo.cc }
- Minimo tra tre numeri:
{ IF_THEN_ELSE_SWITCH/minimo2.cc }
- Scelte tra valori multipli:
{ IF_THEN_ELSE_SWITCH/simple_calc.cc }

If-then-else annidati: esempi

- Esempi di alternative all'utente:
{ IF_THEN_ELSE_SWITCH/conversione2.cc }
- Esempio di "Dangling else":
{ IF_THEN_ELSE_SWITCH/dangling_else.cc }
- Esempio di "Dangling else" (2):
{ IF_THEN_ELSE_SWITCH/dangling_else2.cc }
- Errore tipico con "if":
{ IF_THEN_ELSE_SWITCH/ifeq_err.cc }
- Versione corretta:
{ IF_THEN_ELSE_SWITCH/ifeq_corr.cc }
- Minimo tra due numeri:
{ IF_THEN_ELSE_SWITCH/minimo.cc }
- Minimo tra tre numeri:
{ IF_THEN_ELSE_SWITCH/minimo2.cc }
- Scelte tra valori multipli:
{ IF_THEN_ELSE_SWITCH/simple_calc.cc }

If-then-else annidati: esempi

- Esempi di alternative all'utente:
{ IF_THEN_ELSE_SWITCH/conversione2.cc }
- Esempio di "Dangling else":
{ IF_THEN_ELSE_SWITCH/dangling_else.cc }
- Esempio di "Dangling else" (2):
{ IF_THEN_ELSE_SWITCH/dangling_else2.cc }
- Errore tipico con "if":
{ IF_THEN_ELSE_SWITCH/ifeq_err.cc }
- Versione corretta:
{ IF_THEN_ELSE_SWITCH/ifeq_corr.cc }
- Minimo tra due numeri:
{ IF_THEN_ELSE_SWITCH/minimo.cc }
- Minimo tra tre numeri:
{ IF_THEN_ELSE_SWITCH/minimo2.cc }
- Scelte tra valori multipli:
{ IF_THEN_ELSE_SWITCH/simple_calc.cc }

If-then-else annidati: esempi

- Esempi di alternative all'utente:
{ IF_THEN_ELSE_SWITCH/conversione2.cc }
- Esempio di "Dangling else":
{ IF_THEN_ELSE_SWITCH/dangling_else.cc }
- Esempio di "Dangling else" (2):
{ IF_THEN_ELSE_SWITCH/dangling_else2.cc }
- Errore tipico con "if":
{ IF_THEN_ELSE_SWITCH/ifeq_err.cc }
- Versione corretta:
{ IF_THEN_ELSE_SWITCH/ifeq_corr.cc }
- Minimo tra due numeri:
{ IF_THEN_ELSE_SWITCH/minimo.cc }
- Minimo tra tre numeri:
{ IF_THEN_ELSE_SWITCH/minimo2.cc }
- Scelte tra valori multipli:
{ IF_THEN_ELSE_SWITCH/simple_calc.cc }

If-then-else annidati: esempi

- Esempi di alternative all'utente:
{ IF_THEN_ELSE_SWITCH/conversione2.cc }
- Esempio di "Dangling else":
{ IF_THEN_ELSE_SWITCH/dangling_else.cc }
- Esempio di "Dangling else" (2):
{ IF_THEN_ELSE_SWITCH/dangling_else2.cc }
- Errore tipico con "if":
{ IF_THEN_ELSE_SWITCH/ifeq_err.cc }
- Versione corretta:
{ IF_THEN_ELSE_SWITCH/ifeq_corr.cc }
- Minimo tra due numeri:
{ IF_THEN_ELSE_SWITCH/minimo.cc }
- Minimo tra tre numeri:
{ IF_THEN_ELSE_SWITCH/minimo2.cc }
- Scelte tra valori multipli:
{ IF_THEN_ELSE_SWITCH/simple_calc.cc }

If-then-else annidati: esempi

- Esempi di alternative all'utente:
{ IF_THEN_ELSE_SWITCH/conversione2.cc }
- Esempio di "Dangling else":
{ IF_THEN_ELSE_SWITCH/dangling_else.cc }
- Esempio di "Dangling else" (2):
{ IF_THEN_ELSE_SWITCH/dangling_else2.cc }
- Errore tipico con "if":
{ IF_THEN_ELSE_SWITCH/ifeq_err.cc }
- Versione corretta:
{ IF_THEN_ELSE_SWITCH/ifeq_corr.cc }
- Minimo tra due numeri:
{ IF_THEN_ELSE_SWITCH/minimo.cc }
- Minimo tra tre numeri:
{ IF_THEN_ELSE_SWITCH/minimo2.cc }
- Scelte tra valori multipli:
{ IF_THEN_ELSE_SWITCH/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`:

```
{ IF_THEN_ELSE_SWITCH/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:

```
{ IF_THEN_ELSE_SWITCH/ESERCIZI_PROPOSTI.txt }
```

Outline

- 1 Istruzioni Semplici
- 2 Istruzioni Strutturate
- 3 Istruzioni Condizionali
- 4 Istruzioni Iterative**
- 5 Istruzioni di Salto

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` (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` (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` (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` (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):
{ LOOPS/stampaciao.cc }
- ... (contatore decrescente):
{ LOOPS/stampaciao2.cc }
- ..., con loop infinito:
{ LOOPS/stampaciao_inflloop.cc }
- somma con accumulatore:
{ LOOPS/sommainteri_while.cc }
- prodotto con accumulatore:
{ LOOPS/fact_while.cc }
- condizione di uscita diversa da conteggio:
{ LOOPS/divisibile.cc }

L'Istruzione Iterativa `while`: Esempi I

- ripetizione pedissequa di un'operazione (contatore crescente):
{ `LOOPS/stampaciao.cc` }
- ... (contatore decrescente):
{ `LOOPS/stampaciao2.cc` }
- ..., con loop infinito:
{ `LOOPS/stampaciao_inflloop.cc` }
- somma con accumulatore:
{ `LOOPS/sommainteri_while.cc` }
- prodotto con accumulatore:
{ `LOOPS/fact_while.cc` }
- condizione di uscita diversa da conteggio:
{ `LOOPS/divisibile.cc` }

L'Istruzione Iterativa `while`: Esempi I

- ripetizione pedissequa di un'operazione (contatore crescente):
{ LOOPS/stampaciao.cc }
- ... (contatore decrescente):
{ LOOPS/stampaciao2.cc }
- ..., con loop infinito:
{ LOOPS/stampaciao_inffloop.cc }
- somma con accumulatore:
{ LOOPS/sommainteri_while.cc }
- prodotto con accumulatore:
{ LOOPS/fact_while.cc }
- condizione di uscita diversa da conteggio:
{ LOOPS/divisibile.cc }

L'Istruzione Iterativa `while`: Esempi I

- ripetizione pedissequa di un'operazione (contatore crescente):
{ `LOOPS/stampaciao.cc` }
- ... (contatore decrescente):
{ `LOOPS/stampaciao2.cc` }
- ..., con loop infinito:
{ `LOOPS/stampaciao_inflloop.cc` }
- **somma con accumulatore:**
{ `LOOPS/sommainteri_while.cc` }
- prodotto con accumulatore:
{ `LOOPS/fact_while.cc` }
- condizione di uscita diversa da conteggio:
{ `LOOPS/divisibile.cc` }

L'Istruzione Iterativa `while`: Esempi I

- ripetizione pedissequa di un'operazione (contatore crescente):
{ `LOOPS/stampaciao.cc` }
- ... (contatore decrescente):
{ `LOOPS/stampaciao2.cc` }
- ..., con loop infinito:
{ `LOOPS/stampaciao_inflloop.cc` }
- somma con accumulatore:
{ `LOOPS/sommainteri_while.cc` }
- prodotto con accumulatore:
{ `LOOPS/fact_while.cc` }
- condizione di uscita diversa da conteggio:
{ `LOOPS/divisibile.cc` }

L'Istruzione Iterativa `while`: Esempi I

- ripetizione pedissequa di un'operazione (contatore crescente):
{ `LOOPS/stampaciao.cc` }
- ... (contatore decrescente):
{ `LOOPS/stampaciao2.cc` }
- ..., con loop infinito:
{ `LOOPS/stampaciao_inffloop.cc` }
- somma con accumulatore:
{ `LOOPS/sommainteri_while.cc` }
- prodotto con accumulatore:
{ `LOOPS/fact_while.cc` }
- condizione di uscita diversa da conteggio:
{ `LOOPS/divisibile.cc` }

L'Istruzione Iterativa `while`: Esempi II

- ripetizione di comando a menu:
{ `LOOPS/conversione3_while.cc` }
- somma con accumulatore, con conteggio:
{ `LOOPS/serie_while.cc` }
- somma con accumulatore, con cond. uscita :
{ `LOOPS/serie_while1.cc` }
- uso di "cin loops":
{ `LOOPS/cin_loop.cc` }
- stessa cosa, ma con fail:
{ `LOOPS/cin_loop_equivalent.cc` }

L'Istruzione Iterativa `while`: Esempi II

- ripetizione di comando a menu:
{ `LOOPS/conversione3_while.cc` }
- somma con accumulatore, con conteggio:
{ `LOOPS/serie_while.cc` }
- somma con accumulatore, con cond. uscita :
{ `LOOPS/serie_while1.cc` }
- uso di "cin loops":
{ `LOOPS/cin_loop.cc` }
- stessa cosa, ma con fail:
{ `LOOPS/cin_loop_equivalent.cc` }

L'Istruzione Iterativa `while`: Esempi II

- ripetizione di comando a menu:
{ `LOOPS/conversione3_while.cc` }
- somma con accumulatore, con conteggio:
{ `LOOPS/serie_while.cc` }
- somma con accumulatore, con cond. uscita :
{ `LOOPS/serie_while1.cc` }
- uso di "cin loops":
{ `LOOPS/cin_loop.cc` }
- stessa cosa, ma con fail:
{ `LOOPS/cin_loop_equivalent.cc` }

L'Istruzione Iterativa `while`: Esempi II

- ripetizione di comando a menu:
{ `LOOPS/conversione3_while.cc` }
- somma con accumulatore, con conteggio:
{ `LOOPS/serie_while.cc` }
- somma con accumulatore, con cond. uscita :
{ `LOOPS/serie_while1.cc` }
- uso di “cin loops”:
{ `LOOPS/cin_loop.cc` }
- stessa cosa, ma con fail:
{ `LOOPS/cin_loop_equivalent.cc` }

L'Istruzione Iterativa `while`: Esempi II

- ripetizione di comando a menu:
{ `LOOPS/conversione3_while.cc` }
- somma con accumulatore, con conteggio:
{ `LOOPS/serie_while.cc` }
- somma con accumulatore, con cond. uscita :
{ `LOOPS/serie_while1.cc` }
- uso di “cin loops”:
{ `LOOPS/cin_loop.cc` }
- stessa cosa, ma con fail:
{ `LOOPS/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):**
{ LOOPS/sommainteri_do.cc }
- **ripetizione di comando a menu (do):**
{ LOOPS/conversione3_do.cc }
- **conversione di base:**
{ LOOPS/base.cc }

L'Istruzione Iterativa `do`: Esempi

- **somma con accumulatore (`do`):**
{ `LOOPS/sommainteri_do.cc` }
- **ripetizione di comando a menu (`do`):**
{ `LOOPS/conversione3_do.cc` }
- **conversione di base:**
{ `LOOPS/base.cc` }

L'Istruzione Iterativa `do`: Esempi

- **somma con accumulatore (`do`):**
{ `LOOPS/sommainteri_do.cc` }
- **ripetizione di comando a menu (`do`):**
{ `LOOPS/conversione3_do.cc` }
- **conversione di base:**
{ `LOOPS/base.cc` }

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
⇒ miglior praticità e leggibilità del codice.

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
⇒ miglior praticità e leggibilità del codice.

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
⇒ miglior praticità e leggibilità del codice.

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
⇒ miglior praticità e leggibilità del codice.

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
⇒ 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++;  
  }; }
```

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`):
{ `LOOPS/fact_for.cc` }
- somma con accumulatore (numero iterazioni) (`for`):
{ `LOOPS/serie_for.cc` }
- somma con accumulatore (cond. uscita) (`for`):
{ `LOOPS/serie_for1.cc` }
- `for` annidati:
{ `LOOPS/doublefor.cc` }

L'Istruzione Iterativa `for`: Esempi I

- prodotto con accumulatore (`for`):
{ `LOOPS/fact_for.cc` }
- somma con accumulatore (numero iterazioni) (`for`):
{ `LOOPS/serie_for.cc` }
- somma con accumulatore (cond. uscita) (`for`):
{ `LOOPS/serie_for1.cc` }
- `for` annidati:
{ `LOOPS/doublefor.cc` }

L'Istruzione Iterativa `for`: Esempi I

- prodotto con accumulatore (`for`):
{ `LOOPS/fact_for.cc` }
- somma con accumulatore (numero iterazioni) (`for`):
{ `LOOPS/serie_for.cc` }
- somma con accumulatore (cond. uscita) (`for`):
{ `LOOPS/serie_for1.cc` }
- `for` annidati:
{ `LOOPS/doublefor.cc` }

L'Istruzione Iterativa `for`: Esempi I

- prodotto con accumulatore (`for`):
{ `LOOPS/fact_for.cc` }
- somma con accumulatore (numero iterazioni) (`for`):
{ `LOOPS/serie_for.cc` }
- somma con accumulatore (cond. uscita) (`for`):
{ `LOOPS/serie_for1.cc` }
- `for` annidati:
{ `LOOPS/doublefor.cc` }

L'Istruzione Iterativa `for`: Esempi II

- **condizione iniziale multipla con `for`:**
{ `LOOPS/serie_for1_2init.cc` }
- **cond. iniziale multipla & uscita multipla con `for`:**
{ `LOOPS/serie_for1_2init2.cc` }
- **incremento come input dato dall'utente:**
{ `LOOPS/minmax.cc` }
- **doppio incremento:**
{ `LOOPS/doublecontrol.cc` }

L'Istruzione Iterativa `for`: Esempi II

- **condizione iniziale multipla con `for`:**
{ `LOOPS/serie_for1_2init.cc` }
- **cond. iniziale multipla & uscita multipla con `for`:**
{ `LOOPS/serie_for1_2init2.cc` }
- **incremento come input dato dall'utente:**
{ `LOOPS/minmax.cc` }
- **doppio incremento:**
{ `LOOPS/doublecontrol.cc` }

L'Istruzione Iterativa `for`: Esempi II

- **condizione iniziale multipla con `for`:**
{ `LOOPS/serie_for1_2init.cc` }
- **cond. iniziale multipla & uscita multipla con `for`:**
{ `LOOPS/serie_for1_2init2.cc` }
- **incremento come input dato dall'utente:**
{ `LOOPS/minmax.cc` }
- **doppio incremento:**
{ `LOOPS/doublecontrol.cc` }

L'Istruzione Iterativa `for`: Esempi II

- **condizione iniziale multipla con `for`:**
{ `LOOPS/serie_for1_2init.cc` }
- **cond. iniziale multipla & uscita multipla con `for`:**
{ `LOOPS/serie_for1_2init2.cc` }
- **incremento come input dato dall'utente:**
{ `LOOPS/minmax.cc` }
- **doppio incremento:**
{ `LOOPS/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 dalla 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 **indecidibile** 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, ...)

Gli Invarianti di un Ciclo (Loop Invariant)

- Tecnica per la verifica di correttezza dei cicli (proprietà P)
- Idea: suddividere la proprietà desiderata P dalla 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 **indecidibile** 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, ...)

Gli Invarianti di un Ciclo (Loop Invariant)

- Tecnica per la verifica di correttezza dei cicli (proprietà P)
- Idea: suddividere la proprietà desiderata P dalla 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 **indecidibile** 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, ...)

Gli Invarianti di un Ciclo (Loop Invariant)

- Tecnica per la verifica di correttezza dei cicli (proprietà P)
- Idea: suddividere la proprietà desiderata P dalla 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 **indecidibile** 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, ...)

Gli Invarianti di un Ciclo (Loop Invariant)

- Tecnica per la verifica di correttezza dei cicli (proprietà P)
- Idea: suddividere la proprietà desiderata P dalla 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 **indecidibile** 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, ...)

Gli Invarianti di un Ciclo (Loop Invariant)

- Tecnica per la verifica di correttezza dei cicli (proprietà P)
- Idea: suddividere la proprietà desiderata P dalla 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 **indecidibile** 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° ciclo, $i = 4$).

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° ciclo, $i = 4$).

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° ciclo, $i = 4$).

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° ciclo, $i = 4$).

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° ciclo, $i = 4$).

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° ciclo, $i = 4$).

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° ciclo, $i = 4$).

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° ciclo, $i = 4$).

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° ciclo, $i = 4$).

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

Esempio: divisibilità per 2

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

- **Proprietà P** : dopo il ciclo, $tmp\%2\neq 0$ e $tmp * (2^{ndiv2}) == num$

- **Invariante P_i** : $tmp * (2^{ndiv2}) == num$

- (1) prima del ciclo, $tmp * (2^0) == num$

- (2) prima dell' i -esimo ciclo tmp è divisibile per due e $tmp * (2^{ndiv2}) == num$

- \implies dopo l' i -esimo ciclo $tmp * (2^{ndiv2}) == num$

- (infatti $tmp/2 * (2^{(ndiv2+1)}) == num$)

- (3) Alla fine dell'ultimo ciclo (n -esimo), P_n (più la negazione della condizione del ciclo) implica la proprietà P :

- $tmp * (2^{ndiv2}) == num$ e $tmp\%2\neq 0$

Esempio: divisibilità per 2

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

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

Esempio: divisibilità per 2

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

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

Esempio: divisibilità per 2

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

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

Esempio: divisibilità per 2

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

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

Esempio: divisibilità per 2

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

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

Esempio: divisibilità per 2

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

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

Esempio: divisibilità per 2

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

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

Esercizi Proposti

Esercizi sui cicli:

```
{ LOOPS/00ESERCIZI_PROPOSTI.txt }
```

Outline

- 1 Istruzioni Semplici
- 2 Istruzioni Strutturate
- 3 Istruzioni Condizionali
- 4 Istruzioni Iterative
- 5 Istruzioni di Salto**

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

- **Da evitare!** \implies si può sempre fare modificando la condizione

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

- **semplice break (while):**
{ `LOOPS/break_while.cc` }
- **come evitare un break (while):**
{ `LOOPS/nobreak_while.cc` }

L'Istruzione di Salto `break`

L'istruzione `break` termina direttamente tutto il ciclo

- **Da evitare!** \implies si può sempre fare modificando la condizione

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

- **semplice break (while):**
{ `LOOPS/break_while.cc` }
- **come evitare un break (while):**
{ `LOOPS/nobreak_while.cc` }

L'istruzione `return` in un loop (salto implicito)

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

- **Da evitare!** \implies si può sempre fare modificando la condizione

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

- **semplice return (while):**
{ [LOOPS/return_while.cc](#) }
- **come evitare un return (while):**
{ [LOOPS/noreturn_while.cc](#) }

L'Istruzione `return` in un loop (salto implicito)

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

- **Da evitare!** \implies si può sempre fare modificando la condizione

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

- **semplice return (while):**
{ [LOOPS/return_while.cc](#) }
- **come evitare un return (while):**
{ [LOOPS/noreturn_while.cc](#) }

L'Istruzione di Salto `continue`

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

- nel caso di ciclo `for` viene saltata l'istruzione di aggiornamento
- **Da evitare!** \implies si può sempre fare lo stesso con un "if"

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

- **semplice continue (while):**
{ `LOOPS/continue.cc` }
- **come evitare continue (while):**
{ `LOOPS/nocontinue.cc` }

L'Istruzione di Salto `continue`

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

- nel caso di ciclo `for` viene saltata l'istruzione di aggiornamento
- **Da evitare!** \implies si può sempre fare lo stesso con un "if"

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

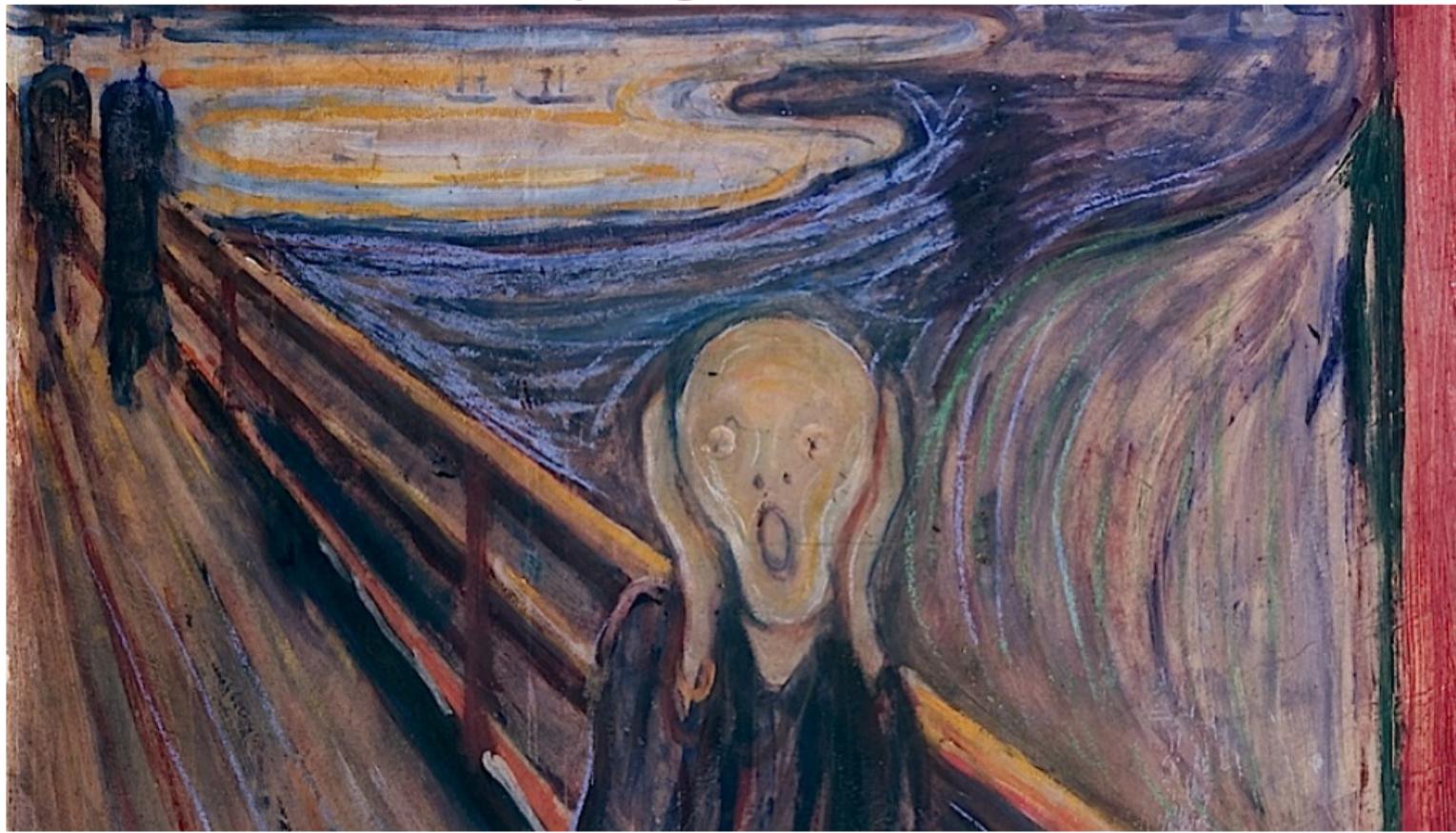
- **semplice continue (while):**
{ `LOOPS/continue.cc` }
- **come evitare continue (while):**
{ `LOOPS/nocontinue.cc` }

L'Istruzione di Salto goto

©Munch Museum, Oslo.

L'Istruzione di Salto goto

Un goto in un programma C++ ???!!??!!



L'Istruzione di Salto goto II



© xkcd www.xkcd.com

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.