ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

COMANDI CONDIZIONALI E ITERATIVI

COSIMO LANEVE
cosimo.laneve@unibo.it

CORSO 00819 – PROGRAMMAZIONE

ARGOMENTI (SAVITCH, SEZIONE 2.4, CAPITOLO 3)

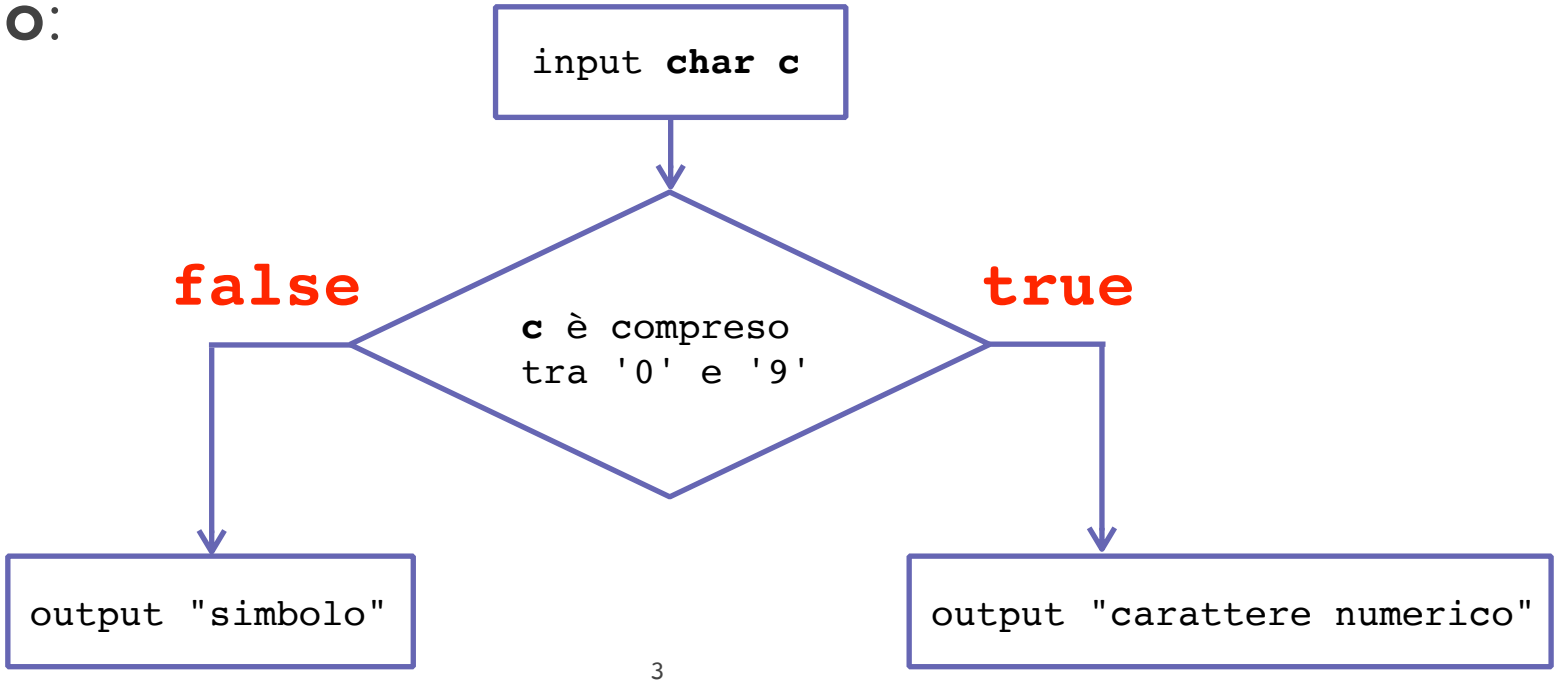
1. i comandi `if-then-else` e `if-then`
2. il comando `while`
3. il comando `for`
4. esempi/esercizi

CASO DI STUDIO: TIPOLOGIA DI CARATTERE

scrivere un programma che prende in input un carattere e stampa

* "`carattere numerico`" se è una cifra
* "`simbolo`" altrimenti

algoritmo:



COMANDI CONDIZIONALI/IF-THEN-ELSE

permette di effettuare una scelta tra diversi *comandi alternativi* da eseguire

- la scelta viene fatta calcolando il valore di una espressione booleana, detta *condizione*

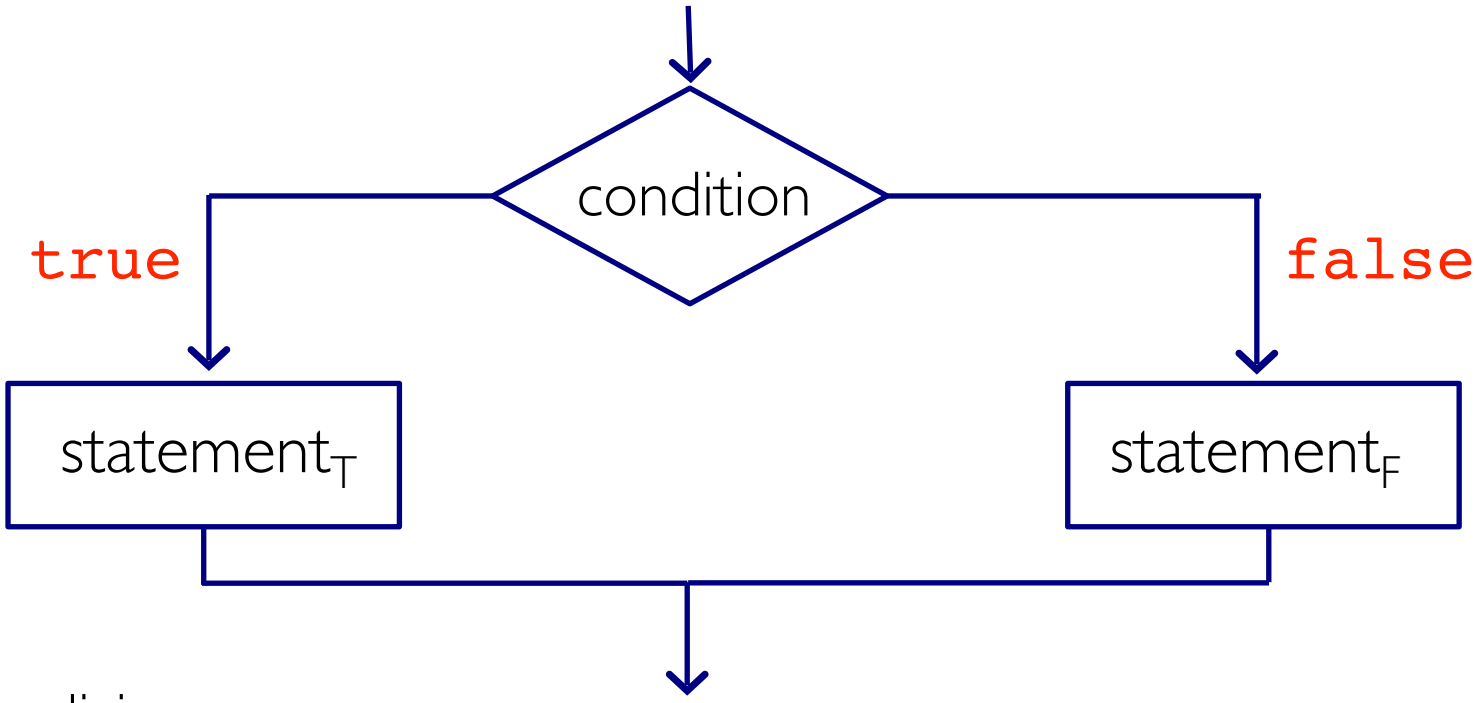
sintassi: `if (`*condition*`)` *statement*$_T$ `else` *statement*$_F$

esempio:

```
if ((x >= '0') && (x <= '9'))
        cout << "carattere numerico" ;
else cout << "simbolo" ;
```

COMANDI CONDIZIONALI/IF-THEN-ELSE

semantica: `if (`*condition*`)` *statement*$_T$ `else` *statement*$_F$



1. valuta la condizione
2. se il risultato è **`true`** esegue l'istruzione che segue la condizione (*ramo then*)
3. se il risultato è **`false`** esegue l'istruzione che segue la parola **`else`** (*ramo else*)

CONDIZIONI

operatori relazionali:

`<`	(minore di)	`>`	(maggiore di)
`<=`	(minore o uguale a)	`>=`	(maggiore o uguale a)

operatori di uguaglianza:

`==`	(uguale a)	`!=`	(diverso da)

operatori logici: `&&` (and) `||` (or) `!` (not)

tabella di verità di `&&`

op1	op2	op1 && op2
false	false	false
false	true	false
true	false	false
true	true	true

tabella di verità di `||`

op1	op2	op1 || op2
false	false	false
false	true	true
true	false	true
true	true	true

tabella di verità di `!`

op1	!op1
false	true
true	false

CONDIZIONI - PROBLEMI

* la seguente espressione esprime la condizione "`x` e' compreso tra `min` e `max`"?

```
min <= x <= max
```

* la seguente espressione esprime la condizione "`x` e `y` sono maggiori di `z`"?

```
x && y > z
```

* la seguente espressione esprime la condizione "`x` è uguale a `1.0` oppure a `3.0`" ?

```
x == 1.0 || 3.0
```

CONDIZIONI - COMPLEMENTO

- il complemento (`!`) di una condizione formata da un solo operatore relazionale (o di uguaglianza) può essere effettuato cambiando l'operatore:
  - `!(x==y)` equivale a `x!=y`
  - `!(x <= y)` equivale a `x > y`
- il complemento di condizioni con `&&` e `||` si ottiene usando le leggi di De Morgan:
  - `!(expr1 && expr2)` equivale a `!expr1 || !expr2`
  - `!(expr1 || expr2)` equivale a `!expr1 && !expr2`
- l'operatore `!` può rendere le espressioni difficili da comprendere: **usarlo solamente quando è necessario**!

LA SEQUENZA DI COMANDI

un comando è la combinazione di singoli comandi in un'unica struttura logica

- assegnamento
- sequenza
- condizionale
- iterazione

sequenza di comandi: gruppo di istruzioni racchiuse tra graffe ed eseguite sequenzialmente

```
{       statement-1 ;
        statement-2 ;
        ...
        statement-n ;
}
```

le parentesi graffe definiscono il **raggruppamento di comandi**

il controllo passa da ***statement-i*** a ***statement-(i+1)*** quando ***statement-i*** **termina**

SEQUENZA DI COMANDI E CONDIZIONALE

è possibile scrivere

```
if (condition) {
        statement1
        . . .
        statementm
} else {
        statement1
        . . .
        statementn
}
```

il cui significato è:

- **se** la condizione è vera **allora** esegui la **sequenza di comandi** nel ramo `then`
- **altrimenti** esegui la **sequenza di comandi** nel ramo `else`

IF: ESERCIZI

1. qual e' il complemento di

```
(x>y) && (ch=='a' || ch=='b') ?
```

2. scrivere un programma che prende in input due interi e restituisce true se sono uguali e false altrimenti

3. scrivere un programma che prende in input tre numeri e un carattere a scelta tra '-' e '/' e stampa in output una tabella con la differenza o il rapporto dei numeri a due a due

   esempio: se i valori in input sono 2, 7, 3, e ' - ' stampa

```
     2    7    3
2    0   -5   -1
7    5    0    4
3    1   -4    0
```

COMANDI CONDIZIONALI/IF-THEN

permette di effettuare una scelta tra un *comando* da eseguire e non far nulla

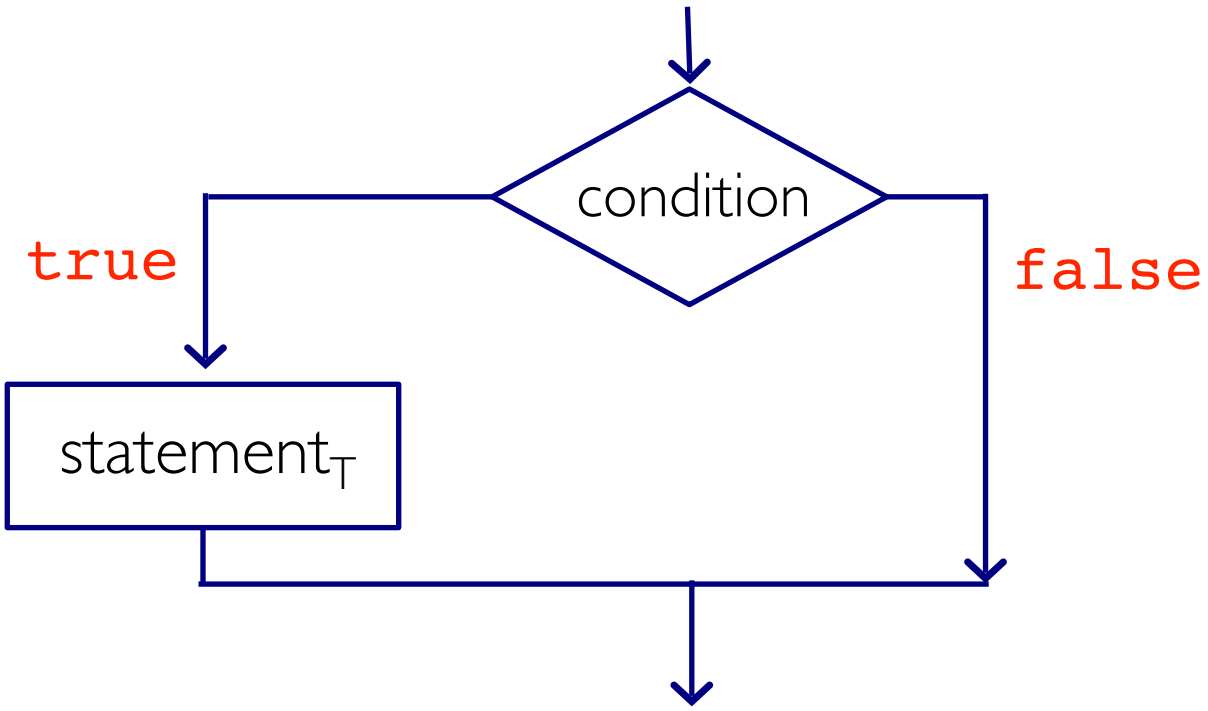
- la scelta viene fatta calcolando il valore di una espressione booleana, detta *condizione*

sintassi: `if (`*condition*`)` *statement*$_T$

esempio: `if (x != 0) cout << 25/x ;`

COMANDI CONDIZIONALI/`IF-THEN`

semantica: `if (`*condition*`)` *statement*$_T$



1. valuta la condizione tra parentesi
2. se il risultato è **`true`** esegue l'istruzione che segue la condizione (*ramo then*)
3. se il risultato è **`false`** va all'istruzione successiva

COMANDI CONDIZIONALI/`IF` ANNIDATI

comando **`if`** in cui $statement_T$ (o $statement_F$) è a sua volta un'istruzione

`if` è utilizzata per codificare decisioni con più di due alternative

formato standard:

`if` ($condition_1$) $statement_1$
`else if` ($condition_2$) $statement_2$
. . .
`else if` ($condition_n$) $statement_n$
`else` $statement_e$

semantica: le condizioni sono testate in sequenza, finchè si trova **una condizione vera**

se viene trovata una condizione vera, **viene eseguita l'istruzione corrispondente** e il resto dell'**`if`** a più alternative viene saltato

se nessuna condizione è vera, viene eseguita l'istruzione $statement_e$

esempio:

```
if (x > 0) num_pos = num_pos + 1;
else if (x < 0)  num_neg = num_neg + 1;
else  num_zero = num_zero + 1;
```

`if` ANNIDATI E SEQUENZE DI `if`

sequenze di **`if`**:

```
if (x > 0)   num_pos = num_pos + 1;
if (x < 0)   num_neg = num_neg + 1;
if (x == 0)   num_zero = num_zero + 1;
```

- meno leggibile (è più difficile capire che verrà eseguito uno solo dei tre assegnamenti per ogni valore di **`x`**)
- meno efficiente (se **`x`** è positivo, si eseguono tutti e tre i confronti)
- **la semantica è differente**: i comandi possono modificare il valore di **`x`**

IF: TRAPPOLE

* tipici errori di programmazione (non di compilazione) legati all'**`if`**:

```
if (x=0)...
```

* errori nelle condizioni come quelli visti in precedenza

```
x<=y<=z
```

* associatività dell'if

```
if (cond1) com1;
      if (cond2) com2;
else com3;
```

provate a inserire questo tipo di errori per vedere cosa succede al programma

ESERCIZI

1. scrivere un “firewall” che prende in input 5 caratteri e stampa solamente le lettere minuscole
2. scrivere un programma che prende 3 interi e stampa 1 se uno dei tre è divisore degli altri due, 0 altrimenti (fare due versioni: una con l’if e una senza)
3. scrivere un programma che simula una calcolatrice tascabile con le operazioni “+”, “-”, “*”, “/” e “%”; cioè prende un intero, uno dei simboli precedenti, e un altro intero e calcola il risultato dell’operazione relativa
4. scrivere un programma che prende 3 interi e li stampa in maniera ordinata
5. scrivere un programma che prende 4 interi e stampa l’intero tra i quattro più vicino al valor medio

COMANDI ITERATIVI

un problema molto usuale negli algoritmi/programmi è la **ripetizione** della stessa operazione tante volte

esempio: stampare 10 volte "`ciao`"

```
int main(){
    cout << "ciao" << endl ;
    cout << "ciao" << endl ;
    cout << "ciao" << endl ;
    cout << "ciao" << endl ;
    cout << "ciao" << endl ;
    cout << "ciao" << endl ;
    cout << "ciao" << endl ;
    cout << "ciao" << endl ;
    cout << "ciao" << endl ;
    cout << "ciao" << endl ;
    return(0) ;
}
```

. . . e se volete stampare "ciao" 100 volte?

COMANDI ITERATIVI

esempio: stampare 10 volte "ciao"

* usare i comandi iterativi

```
int main(){
    int conta ;
    conta = 100 ;
    while (conta > 0) {
        cout << "ciao" << endl ;
        conta = conta - 1 ;
    }
    return(0) ;
}
```

. . . e se volete stampare "**ciao**" 100 volte?

COMANDI ITERATIVI/`WHILE`

il comando while consente di ripetere un gruppo di comandi, detto **corpo del ciclo**

* la scelta viene fatta calcolando il valore di una espressione booleana, detta **guardia del ciclo**

sintassi: `while` (*condition*) *statement*

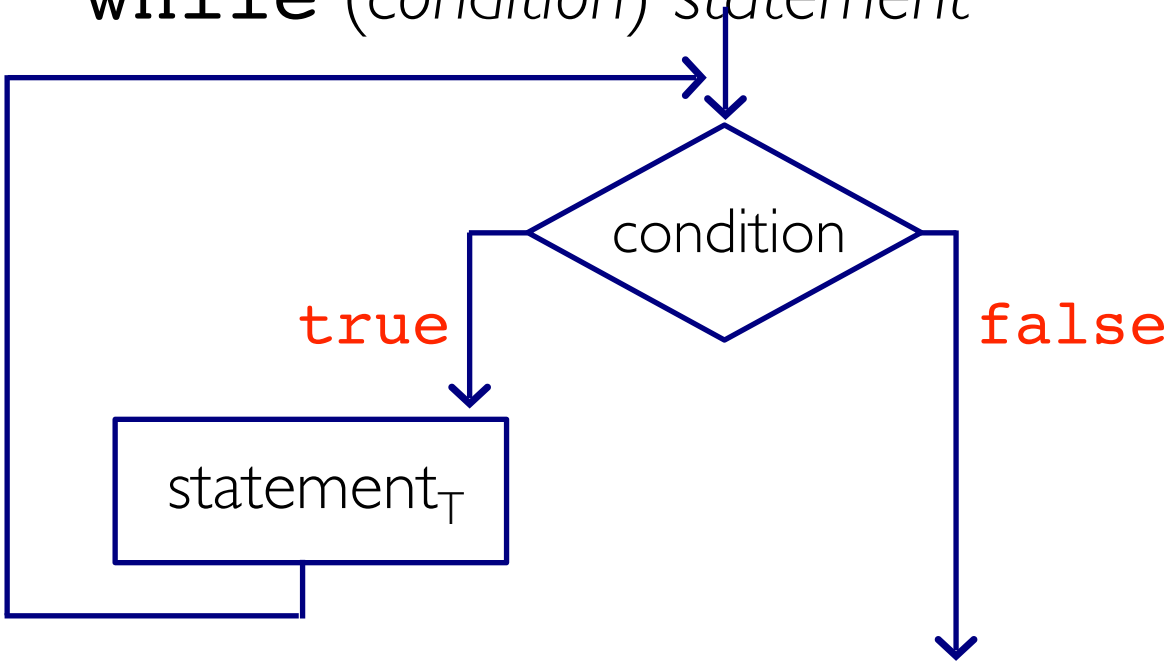
* *condition* è la struttura decisionale per controllare il numero di iterazioni (**guardia del ciclo**)
* *statement* è il gruppo di istruzioni che vengono ripetute (**corpo del ciclo**)

esempio:

```
while (x >= y) { x = x-y ; quoziente = quoziente+1 ;}
```

COMANDI ITERATIVI/`WHILE`

semantica: **`while`** (*condition*) *statement*



il salto è all'indietro!

1. valuta la guardia del ciclo
2. se il risultato è **`true`** esegue il corpo del ciclo e ritorna al punto 1
3. se il risultato è **`false`** il comando **`while`** termina (e si esegue quello successivo)

COMANDI ITERATIVI/WHILE/WHEELER

il primo programma eseguito su un calcolatore e memorizzato in memoria (David Wheeler, Cambridge, 6 Maggio1949) [riscritto in C++]

```
// calcola e stampa la tabella dei quadrati di 0-99
int main(){
    int i = 0;
    while (i<100) {
        cout << i << '\t' << i*i << '\n' ;
        i = i+1 ;
    }
    return(0) ;
}
```

COMANDI ITERATIVI/WHILE

il programma di Wheeler, come quello della stampa di "ciao", sono esempi di ***cicli controllati da contatore***

- ciclo il cui numero di iterazioni è **conosciuto prima dell'inizio** dell'esecuzione del ciclo
- la ripetizione del ciclo è gestita da una **variabile di controllo**, il cui valore rappresenta un contatore
- **formato**:
  1. inizializza la *variabile_di_controllo* ad un *valore iniziale*
  2. la condizione diventa *variabile_di_controllo* < *valore finale*
  3. esegue il corpo e alla fine del corpo incrementa/decrementa la *variabile_di_controllo*

COMANDI ITERATIVI/WHILE/ESEMPIO

scrivere un programma che prende un intero **n**, prende altri **n** interi e stampa la loro somma

```
int main(){
      int res, n , val;
      res = 0 ;
      cin >> n ;
      while (n>0) {
            cin >> val ;
            res = res + val ;
            n = n-1 ;
      }
      cout << res ;
      return(0) ;
}
```

COMANDI ITERATIVI/`WHILE`/OSSERVAZIONI

nel comando **`while`** (*condition*) *statement*

* se la *condition* è **`false`** la prima volta che viene testata, lo *statement* non viene mai eseguito

  esempio:
  ```
  while (false) cout << "Hello" ;
  cout << "Hello" ;
  ```

  stampa una sola volta **`"Hello"`** quindi è equivalente a

  ```
  cout << "Hello" ;
  ```

* se il corpo del **`while`** è eseguito, il valore della condizione può cambiare (perchè cambiano le variabili che vi occorrono)

  esempio:

  ```
  while (x < 10) { cout << "Hello" ; x = x+1 ;}
  ```

COMANDI ITERATIVI/WHILE/OSSERVAZIONI

nel comando **`while`** (*condition*) *statement*

- se la condition rimane sempre `true` l'iterazione non termina (ciclo **infinito**)

  esempio: `while (true) cout << "Hello" ;`

  stampa una sequenza infinita di **`"Hello"`**

- ogni iterazione deve contenere una istruzione che invalida la guardia del **`while`**

- **esempio**: stampare i numeri dispari minori di `12`

```
while (x != 12){
        cout << x << endl;
        x = x + 2;
}
```

- meglio usare la condizione `while (x < 12)`

ESERCIZI

1. scrivere un programma che prende un intero **`n`**, e calcola la somma dei numeri minori di **`n`**,

2. scrivere un programma che prende un intero **`n`**, e calcola il fattoriale di **`n`** (**`!0 =1`** ; **`!n = 1*2*…*n`**)

3. scrivere un programma che prende n e ritorna la parte intera della radice quadrata di n

4. scrivere un programma che prende n e ritorna la parte intera del logaritmo in base 2 di n

5. scrivere un programma che prende n e ritorna la radice quadrata di n calcolata con il metodo di Bombelli

COMANDI ITERATIVI/WHILE/FIBONACCI

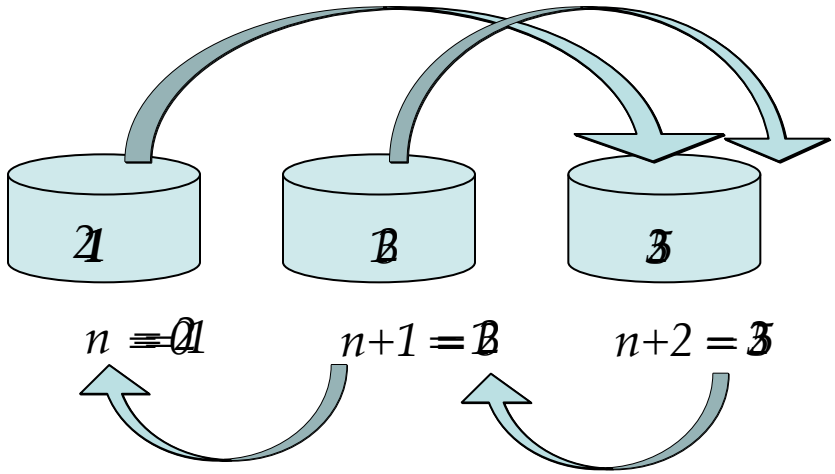
la funzione di fibonacci è definita da:

$f(0) = 0 \quad f(1) = 1 \quad f(n+2) = f(n+1) + f(n)$

scrivere un programma che prende **n** e ritorna $f(\mathbf{n})$

algoritmo: usare tre contenitori

$$f(0) = 0$$
$$f(1) = 1$$
$$f(2) = f(1) + f(0) = 2$$
$$f(3) = f(2) + f(1) = 3$$
$$f(4) = f(3) + f(2) = 5$$
$$f(5) = f(4) + f(3) = 8$$
$$f(6) = f(5) + f(4) = 13$$
$$f(7) = f(6) + f(5) = 21$$
$$\cdots$$



COMANDI ITERATIVI/WHILE/FIBONACCI

```
int main(){
        int x, n_oldold, n_old , n;
        cin >> x ;              // x>0 [per x=0 ritorna 1]
        n_oldold = 0 ;
        n_old = 1 ;
        n = 1 ;
        while (x-2 >= 0) {
                n = n_old + n_oldold ;
                n_oldold = n_old ;
                n_old = n ;
                x = x-1 ;
        }
        cout << n ;
        return(0) ;
}
```

COMANDI ITERATIVI/FOR

sintassi: `for (`*variable* `=` *expression* `;` *condition* `;` *variable* `=` *expression'*`)` *statement*

- *variable* = *expression* è l'assegnamento di inizializzazione della variabile di controllo

una alternativa molto usata è
type variable = *expression*
cioè dichiarazione+inizializzazione

- *condition* è la struttura decisionale per controllare il numero di iterazioni, detta guardia del ciclo for
- *statement* `;` *variable* `=` *expression'* è il gruppo di istruzioni che vengono ripetute, detto corpo del ciclo for

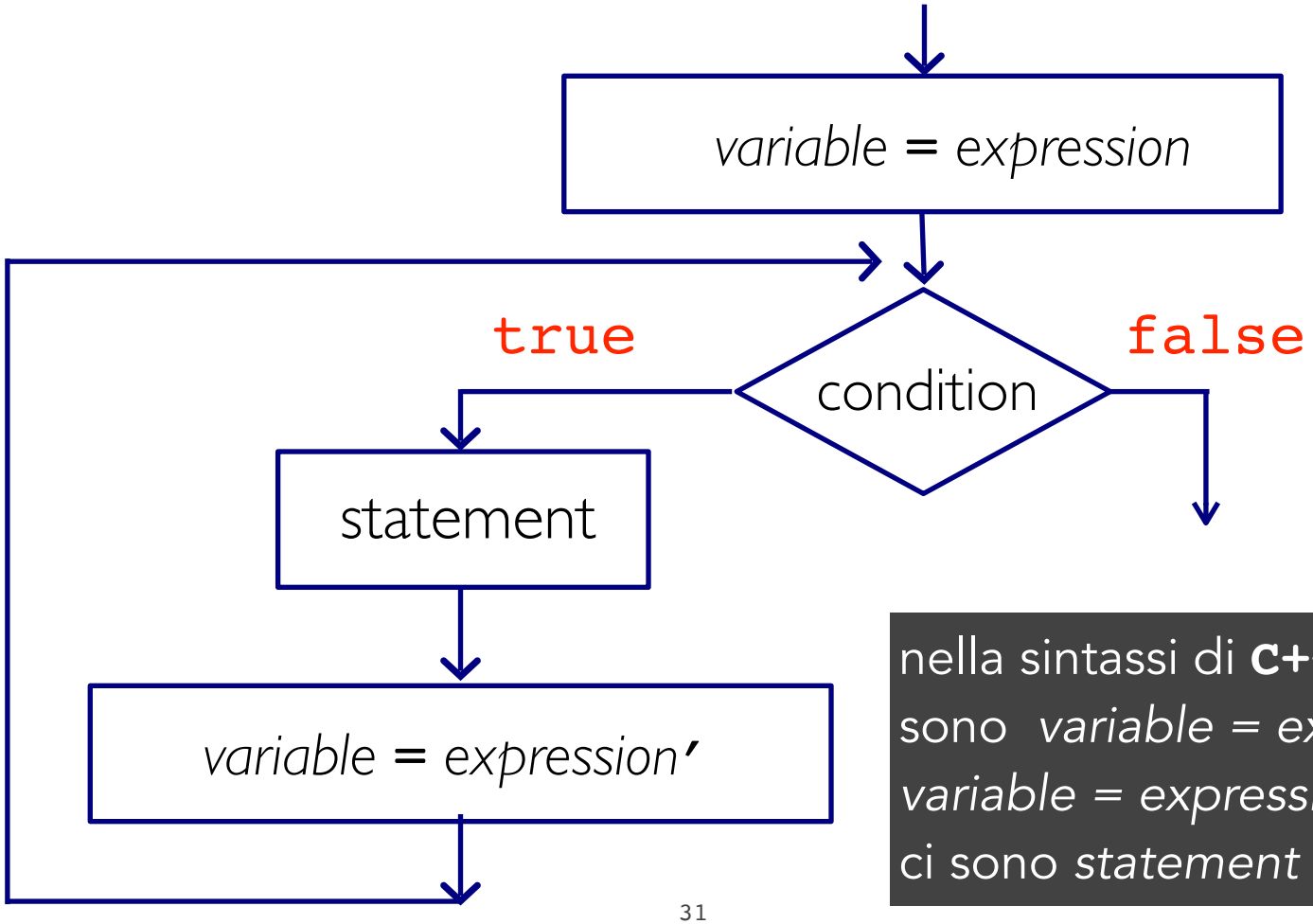
esempio: stampare `10` volte "`ciao`"

```
for (conta = 10 ; conta > 0 ; conta = conta-1)
        cout << "ciao" << endl ;
```

COMANDI ITERATIVI/FOR

semantica:

`for (`*variable* `=` *expression* `;` *condition* `;` *variable* `=` *expression* `')`
statement



nella sintassi di **C++** non ci sono *variable = expression* e *variable = expression'* ma ci sono *statement*

COMANDI ITERATIVI/FOR/WHEELER

il primo programma eseguito su un calcolatore e memorizzato in memoria (David Wheeler, Cambridge, 6 Maggio1949) [riscritto in **C++** con il comando **for**]

```
// calcola e stampa la tabella dei quadrati di 0-99
int main(){
      for (int i = 0; i<100 ; i = i+1 )
            cout << i << '\t' << i*i << '\n' ;
      return(0) ;
}
```

COMANDI ITERATIVI/FOR/COMMENTI

il comando **for** permette di raggruppare in un'unica posizione tutte e tre le componenti tipiche di una iterazione:

1. *inizializzazione della variabile di controllo del ciclo*
2. *test della condizione di ripetizione del ciclo*
3. *aggiornamento della variabile di controllo del ciclo*

osservazione: un comando **for** può essere riscritto in un comando **while**:

```
for (variable = expression ; condition ; variable = expression')  statement
        ≡       variable = expression ;
                while (condition) {   statement ;
                                      variable = expression' ;
                }
```

in C++ vale anche il viceversa: ogni comando **while** può essere riscritto in comando **for**

COMANDI ITERATIVI/STILI

il comando `for` va utilizzato

* quando le fasi di inizializzazione e aggiornamento sono semplici (es. azzeramento e incremento);
* **esempio**: assegnamento iniziale `i = 0` e aggiornamento `i = i + 1`
* quando staticamente sono note il numero di iterazioni che il comando deve fare

osservazioni: (1) in altri linguaggi il `for` può avere assegnamenti iniziali e aggiornamenti solo "semplici", il `C++` consente più libertà

(2) storicamente il `for` poteva essere usato solo per cicli il cui numero di iterazioni era noto a priori, altrimenti si doveva usare il `while` (o il `do-while`)

`for i = 1 to 10 do` *comando*

rispettate questa convenzione, altrimenti potreste indispettire gli informatici più tradizionalisti

COMANDI ITERATIVI: ESEMPIO

scrivere un programma che prende in input un carattere e se il carattere ‘0’, ‘1’, …, ‘9’ stampa l’intero successivo, se diverso da "`q`" (quit) non stampa niente, se "`q`" termina (il numero delle iterazioni del ciclo non è noto a priori)

attenzione a questo passo!
l’input è un `char`, l’output è un `int`
come calcolarlo?

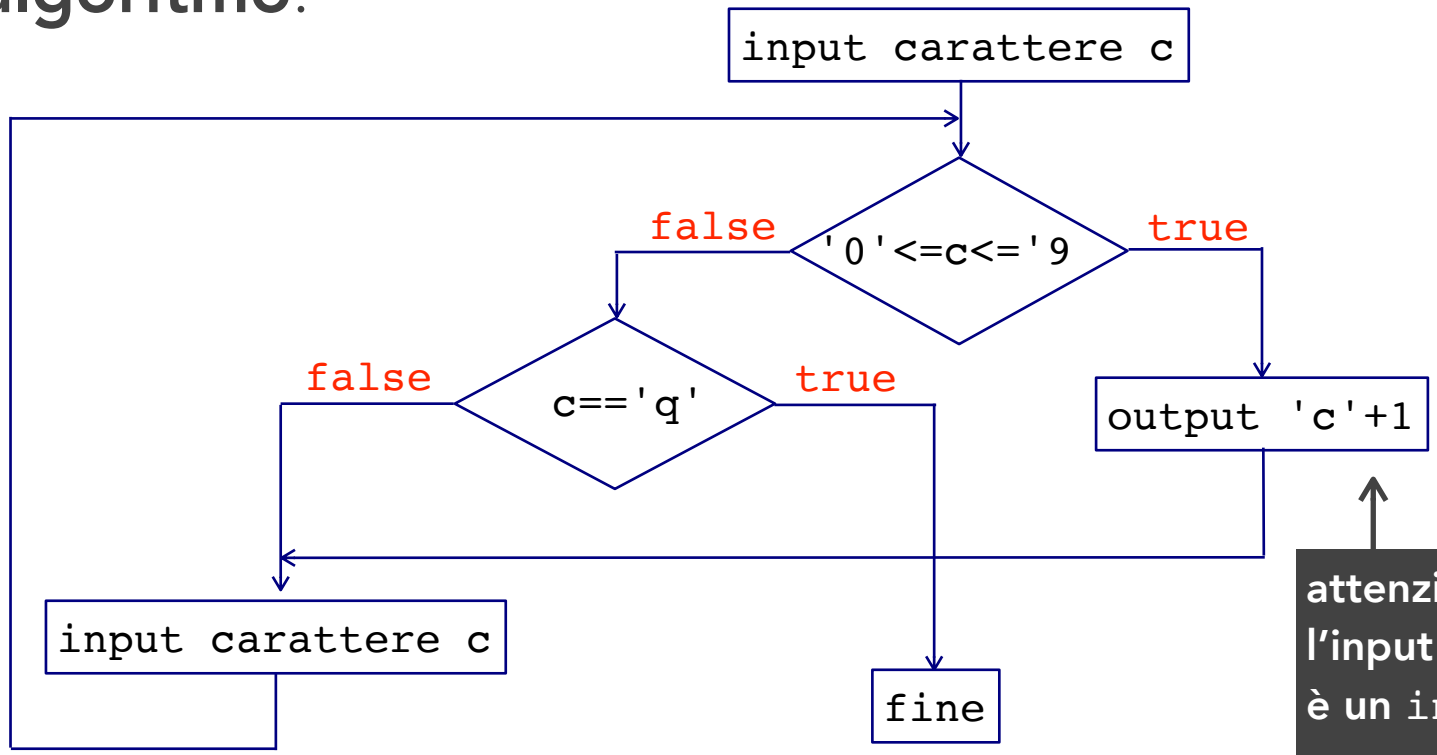
algoritmo:

1. prendi un carattere in input
2. se è compreso tra ‘0’ e ‘9’ ritorna l’intero successivo
3. altrimenti, se diverso da ‘`q`’ ritorna a 1
4. altrimenti (= ‘`q`’) termina

COMANDI ITERATIVI: ESEMPIO

scrivere un programma che prende in input un carattere e se il carattere '0', '1', …, '9' stampa l'intero successivo, se diverso da "`q`" (quit) non stampa niente, se "`q`" termina (il numero delle iterazioni del ciclo non è noto a priori)

algoritmo:



attenzione a questo passo! l'input è un `char`, l'output è un `int`
come calcolarlo?

IMPLEMENTAZIONE 1

```
int main() {
    char c ;
    cout <<  "carattere in input (q per terminare)?> " ;
    cin >> c ;
    while (c != 'q') {
        if (c>='0' && c <='9') {
                if (c == '0') cout << "> " << 1 << "\n" ;
                else if (c == '1') cout << "> " << 2 << "\n" ;
                else if (c == '2') cout << "> " << 3 << "\n" ;
                else if (c == '3') cout << "> " << 4 << "\n" ;
                else if (c == '4') cout << "> " << 5 << "\n" ;
                else if (c == '5') cout << "> " << 6 << "\n" ;
                else if (c == '6') cout << "> " << 7 << "\n" ;
                else if (c == '7') cout << "> " << 8 << "\n" ;
                else if (c == '8') cout << "> " << 9 << "\n" ;
                else cout << "> " << 10 << "\n" ;
                }
        else  cout << "\n" << "carattere non cifra; \n" ;
        cout <<  "carattere in input (q per terminare)>" ;
        cin >> c ;
    }
    return(0);
}
```

troppe ripetizioni!

IMPLEMENTAZIONE 2

```
int main(){
    char c ;
    cout <<  "carattere in input (q per terminare)?>" ;
    cin >> c ;
    while (c != 'q') {
        if (c>='0' && c <='9') {
                cout << "> " << c - '0' + 1 << "\n" ;
        }
        else  cout << "\n" << "carattere non cifra; \n" ;
        cout <<  "carattere in input (q per terminare)>" ;
        cin >> c ;
    }
    return(0);
}
```

l'espressione interessante è

`c - '0' + 1`

COMANDI ITERATIVI: ESERCIZI

1. scrivere un programma che prende in input un numero intero n e stampa n asterischi
2. scrivere un programma che chiede in input un numero: se è primo stampa "primo" altrimenti stampa "non_primo"
3. scrivere un programma che interroga uno studente sulla divisibilità: prende in input un numero intero e chiede allo studente di inserire un divisore. Il programma termina solo quando lo studente inserisce un divisore corretto.
4. scrivere un programma che prende un intero e stampa la somma delle sue cifre

COMANDO `DO-WHILE`

`do` *comando* **`while(`***condizione***`);`**

semantica:

- viene eseguito il *comando* e in seguito la *condizione* viene testata
- se è vera, il *comando* viene ripetuto
- se è falsa si esce dal ciclo e l'esecuzione continua con il comando successivo al **`while`**
- il *comando* viene chiamato corpo del **`do-while`**

osservazione: il corpo del **`do-while`** viene eseguito **almeno una volta** anche nel caso in cui la condizione sia falsa

esercizi: provare a scrivere i programmi precedenti col **`do-while`**: quando i programmi sono più semplici?

ESEMPIO

successore del carattere ‘0’, ‘1’, …, ‘9’ . . .

```
int main(){
      char c ;
      do {
          cout <<  "carattere in input (q per terminare)?>" ;
          cin >> c ;
          if (c>='0' && c <='9') {
                  cout << "> " << c - '0' + 1 << "\n" ;
          } else cout << "\n" << "carattere non cifra; \n" ;
      }
      while (c != 'q') ;
      return(0);
}
```

CICLI ANNIDATI

- spesso può essere utile usare un ciclo dentro l'altro
  - ad ogni iterazione del ciclo esterno corrispondono 0 o più iterazioni del ciclo interno
  - fate attenzione a
    1. non mischiare le variabili di controllo dei 2 cicli
    2. ripristinare i valori di inizializzazione del ciclo interno ad ogni iterazione del ciclo esterno
  - **attenzione**: il programma **diventa poco leggibile**: prendere in considerazione la possibilità di rimpiazzare il ciclo interno con una invocazione di funzione
- si possono anche annidare più livelli di cicli

CICLI ANNIDATI: ESERCIZI

1. scrivere un programma che stampa un triangolo isoscele dell'altezza desiderata presa in input. Ad esempio, se l'altezza è 4 stampa:

```
   *
  ***
 *****
*******
```

2. scrivere un programma che chiede all'utente dei numeri interi e scrive se sono primi o non primi. Il programma termina quando l'utente inserisce il numero 0

CICLI CHE NON TERMINANO

Il programma seguente termina?

```
int main(){
   int i;
   cout << "Inserire un numero intero:";
   cin >> i;
   while(i < 100){
      cout << i << "\n";
      i = i * 2;
   }
}
```

METODI PER CONTROLLARE I CICLI

ci sono **due** metodi per controllare le iterazioni dei cicli

1. controlli definiti da **contatori**
   - **è il tipo di controllo più semplice** perché prima che il ciclo inizi si conoscono il numero di iterazioni
   - vedi programma di Wheeler

2. cicli **controllati da flag**
   - una variabile il cui cambiamento di valore indica che un particolare evento è avvenuto è detta **flag**
   - vedi programmi successivi

CONTROLLO DEI CICLI: CONTATORI

considera il seguente ciclo (Collatz 1937)

```
while (n != 1){
        if (n%2 == 0) n = n/2 ;
        else  n = 3*n + 1 ;
}
```

siamo sicuri che termina?

- c' è una congettura (di Collatz) secondo la quale, qualunque sia il numero iniziale questo programma **termina** sempre
- ma non è stata mai dimostrata

CONTROLLO DEI CICLI: CONTATORI

poiché

```
while (n != 1){
      if (n%2 == 0) n = n/2 ;
      else  n = 3*n + 1 ;
}
```

potrebbe non terminare, conviene imporre la terminazio-ne mediante un `flag`:

- utilizziamo una variabile `flag` inizializzata a un certo limite superiore
- ad ogni iterazione decrementiamo `flag`
- modifichiamo la guardia in

```
((n != 1) && (flag > 0))
```

CONTROLLO DEI CICLI: CONTATORI

il ciclo di Collatz diventa

```
int flag = upper_bound ;
while ((n != 1) && (flag > 0)){
      if (n%2 == 0) n = n/2 ;
      else  n = 3*n + 1 ;
      flag = flag - 1 ;
}
```

che termina sempre . . .

(farà al massimo **`upper_bound`** iterazioni)

CONTROLLO DEI CICLI: FLAG

programma che prende un numero e stampa "**primo**" o "**non primo**" a seconda se il numero è o meno primo

```
int main() {
    int n , i;
    bool flag ;
    cin >> n ;
    i = 2 ; flag = true ;
    while ((i < n) && flag) {
        if (n % i == 0) flag = false;
        else i = i+1 ;
    }
    if (flag) cout << "primo" ;
    else cout << "non primo";
    return (0) ;
}
```

il **flag** consente di separare i due casi!

DEBUG DI CICLI

gli errori comuni che riguardano i cicli sono

1. gli **errori "off-by-one"** in cui il ciclo esegue una volta di più o una volta di meno

2. i **cicli infiniti**, di solito dovuti a un errore progettuale della guardia del ciclo

DEBUG DI CICLI

per gli errori "**off-by-one**"

* verifica la guardia: deve esserci "<" oppure "<=" ?
* verifica che l'inizializzazione della variabile di controllo sia fatta correttamente
* il ciclo prende in considerazione il caso di 0 iterazioni?

per i **cicli infiniti**

* nelle guardie, verifica le diseguaglianze: deve esserci "<" oppure ">"?
* nelle guardie, utilizza "<" oppure ">" piuttosto che "==" (ricordare che i `double` sono approssimati . . .)

ALTRI SUGGERIMENTI

verifica accuratamente che il bug sia nel ciclo

tracciare la variabile di controllo per verificare come cambia durante le iterazioni

- "**tracciare**" significa controllare i valori della variabile durante l'esecuzione: molti supporti runtime forniscono tools per fare questa operazione (anche **`Eclipse`**)
- un modo semplice per tracciare una variabile è usare **`cout`**

ESEMPIO DI DEBUG DI CICLI

questo ciclo dovrebbe memorizzare nella variabile **`result`** il prodotto dei naturali minori o uguali a `5`

```
int next = 2, result = 1;
while (next < 5){
      next = next + 1 ;
      result = result * next ;
}
```

testiamolo aggiungendo un **`cout`** temporaneo per controllare le variabili (**tracciare le variabili**)

ESEMPIO DI DEBUG DI CICLI

```
int next = 2, result = 1;
while (next < 5){
      next = next + 1 ;
      result = result * next ;
      cout << "next = " << next << endl ;
      cout << "result = " << result << endl ;
}
```

i comandi `cout` rivelano che `result` non è mai moltiplicato per 2

soluzione: spostare l'assegnamento `next = next + 1 ;`

ESEMPIO DI DEBUG DI CICLI

```
int next = 2, result = 1;
while (next < 5){
        result = result * next ;
        next = next + 1 ;
        cout << "next = " << next << endl ;
        cout << "result = " << result << endl ;
}
```

c' è ancora un problema . . .

- se ri-testiamo il ciclo, ci accorgiamo che **`result`** non è mai moltiplicato per 5
- occorre rimpiazzare la condizione **`(next < 5)`** con `(next <= 5)`

TEST DI CICLI: GUIDA

ogni volta che **modificate** un programma occorre **ri-testarlo**

* il cambiamento in una parte di codice può causare problemi in un'altra

ogni ciclo deve essere testato con input che consentano di verificare (almeno) i casi

* **zero iterazioni** del corpo
* **una iterazione** del corpo
* un numero di iterazioni del corpo **immediatamente inferiore al massimo**
* il **numero massimo** di iterazioni

RICOMINCIARE DACCAPO

alcune volte **è meglio cestinare un programma con errori** e rifarne uno ex-novo piuttosto che ripararlo

* il nuovo programma sarà più leggibile
* difficilmente il nuovo programma avrà tanti errori come il primo
* ci si mette meno tempo a sviluppare un nuovo programma piuttosto che ripararne uno vecchio
* in ogni modo, imparerete la lezione che la fase cruciale per sviluppare codice è la **progettazione**

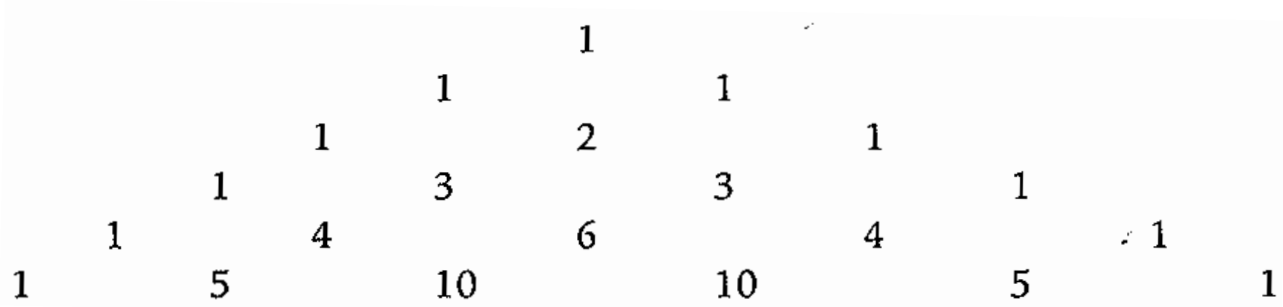
ESERCIZI

1. il valore e^x può essere approssimato dalla somma

$$1 + x + x^2/2! + x^3/3! + \ldots + x^n/n!$$

scrivere un programma che prende x, prende n e stampa l'approssimazione fino ad n di e^x. Il programma stampa anche il valore **`exp`**(x) dove **`exp`** è la funzione definita nella libreria cmath. [SUGGERIMENTO: dichiarare n **`double`** per evitare overflow]

2. scrivere un programma che prende **`n`** e stampa la seguente struttura di altezza **`n`**:



scrivere due versioni del programma: una che usa cicli **`for`**, l'altra che usa cicli **`while`** (c.f. **triangolo di Tartaglia**)