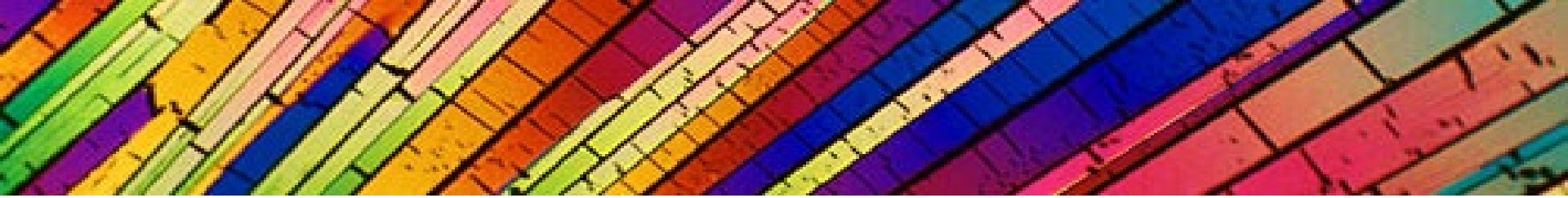


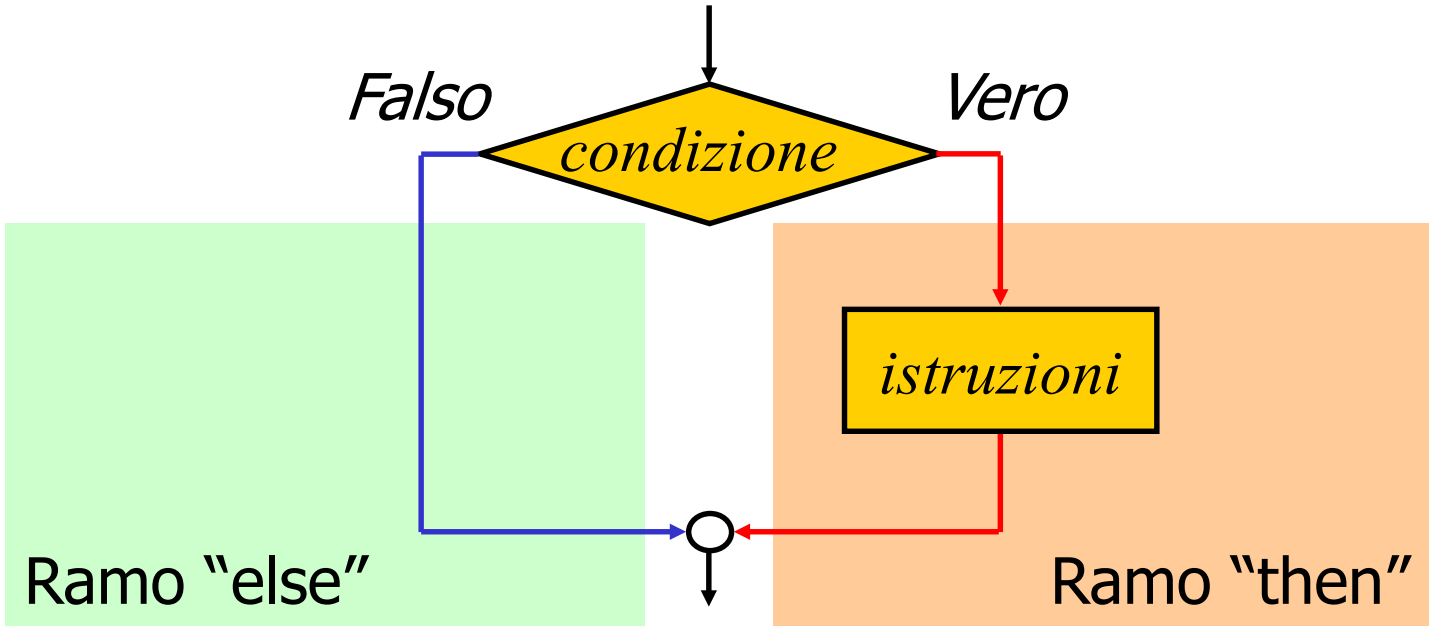
Esecuzione condizionale

Ver. 3

Esecuzione condizionale

- Permette l'esecuzione di un blocco di codice solo se si verifica una certa *condizione*

  SE (*condizione* è vera)
  ALLORA esegui *istruzioni*



Costrutto if

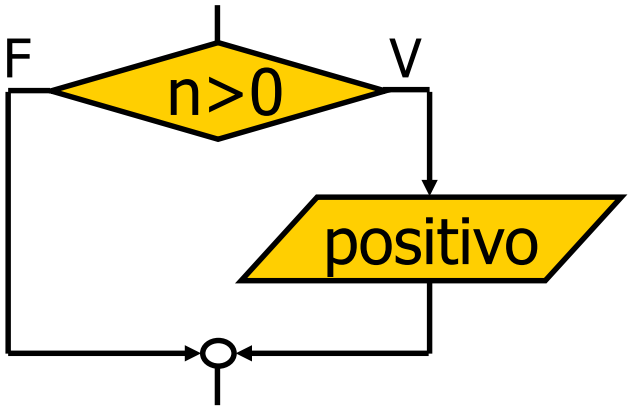
- Sintassi (minimale):
  `if (`*condizione*`)` ← *senza il `;`*
  blocco istruzioni
- Parentesi necessarie prima e dopo *condizione*
- Non deve esserci il `;` alla fine
- Il *blocco istruzioni* è racchiuso da parentesi graffe, opzionali se è composto da una sola istruzione; il codice del blocco (non le eventuali parentesi) deve essere scritto indentato (rientrato)
- Come dopo tutte le parole chiave, dopo `if` generalmente si preferisce mettere uno spazio

Costrutto if

- Esempio 1
  Programma che chiede un numero e se è positivo scrive "positivo" altrimenti nulla.

```
scanf("%d", &n);
if (n > 0)
{
   printf("positivo\n");
}
```

Le parentesi graffe sono opzionali in quanto il blocco condizionato è composto dalla sola `printf`

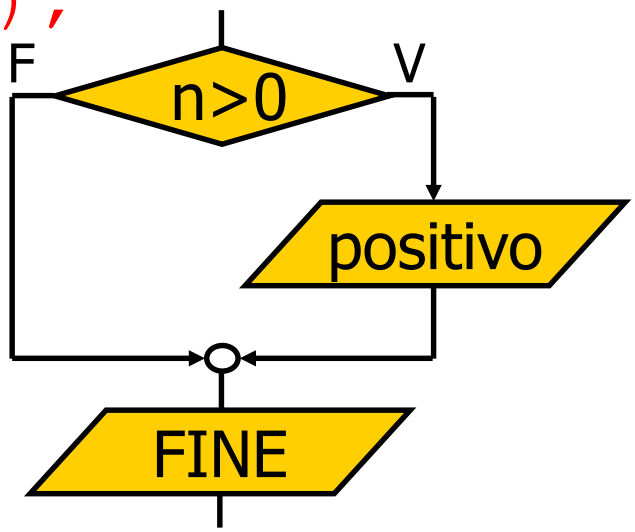


Costrutto if

- Esempio 2
  Programma che chiede un numero e se è positivo scrive “positivo” altrimenti nulla, *prima di uscire deve sempre scrivere "FINE”.*

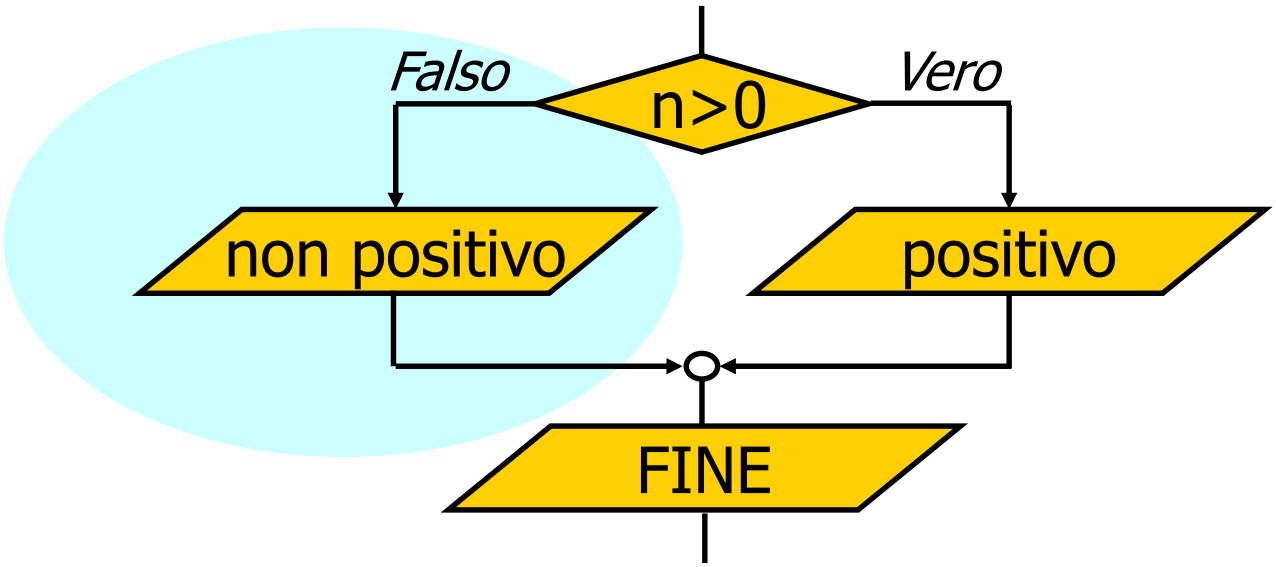
```
if (n > 0)
{
   printf("positivo\n");
}
printf("FINE\n");
```

FINE non è dentro il blocco e quindi la `printf` viene eseguita comunque



Costrutto if-else

- Per indicare un'azione da eseguire quando invece la condizione NON è vera, la soluzione è collocarla nel ramo else



Costrutto if-else

- Il ramo else è introdotto dalla keyword `else` (“altrimenti”), senza il ‘`;`’ alla fine
- Esempio
  Programma che chiede un numero, se positivo scrive “positivo”, altrimenti “non positivo”

```
scanf("%d", &n);
if (n > 0)
  printf("positivo\n");
else
  printf("non positivo\n");
printf("FINE\n");
```

Nota: la `printf` con FINE non è dentro il blocco e quindi viene eseguita sempre, le graffe sono facoltative perché i blocchi sono composti da una sola istruzione

Costrutto if-else

- Si noti che TUTTO il costrutto

  `if` (*condizione*)
  blocco$_{then}$
  `else`
  blocco$_{else}$

  costituisce un'UNICA istruzione di selezione
- Per ciascun blocco, le graffe sono opzionali se il blocco è composto da una sola istruzione eseguibile

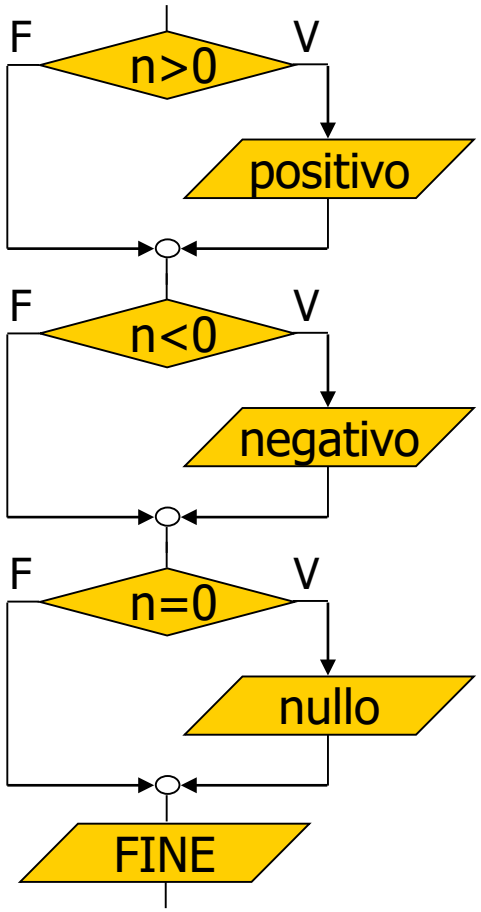
Selezione a più rami

- È richiesta quando si hanno più di due casi che necessitano di elaborazioni diverse
- Esempio
  Programma che chiede un numero, se positivo scrive "positivo", se negativo "negativo", se zero "nullo"
- Soluzione 1
  Si possono avere più costrutti `if` completi ***in sequenza*** (detti anche *in cascata*), ossia uno di seguito all'altro

Selezione a più rami

- Soluzione 1 - codice

```
scanf("%d", &n);
if (n > 0)
   printf("positivo\n");
if (n < 0)
   printf("negativo\n");
if (n == 0)
   printf("nullo\n");
printf("FINE\n");
```



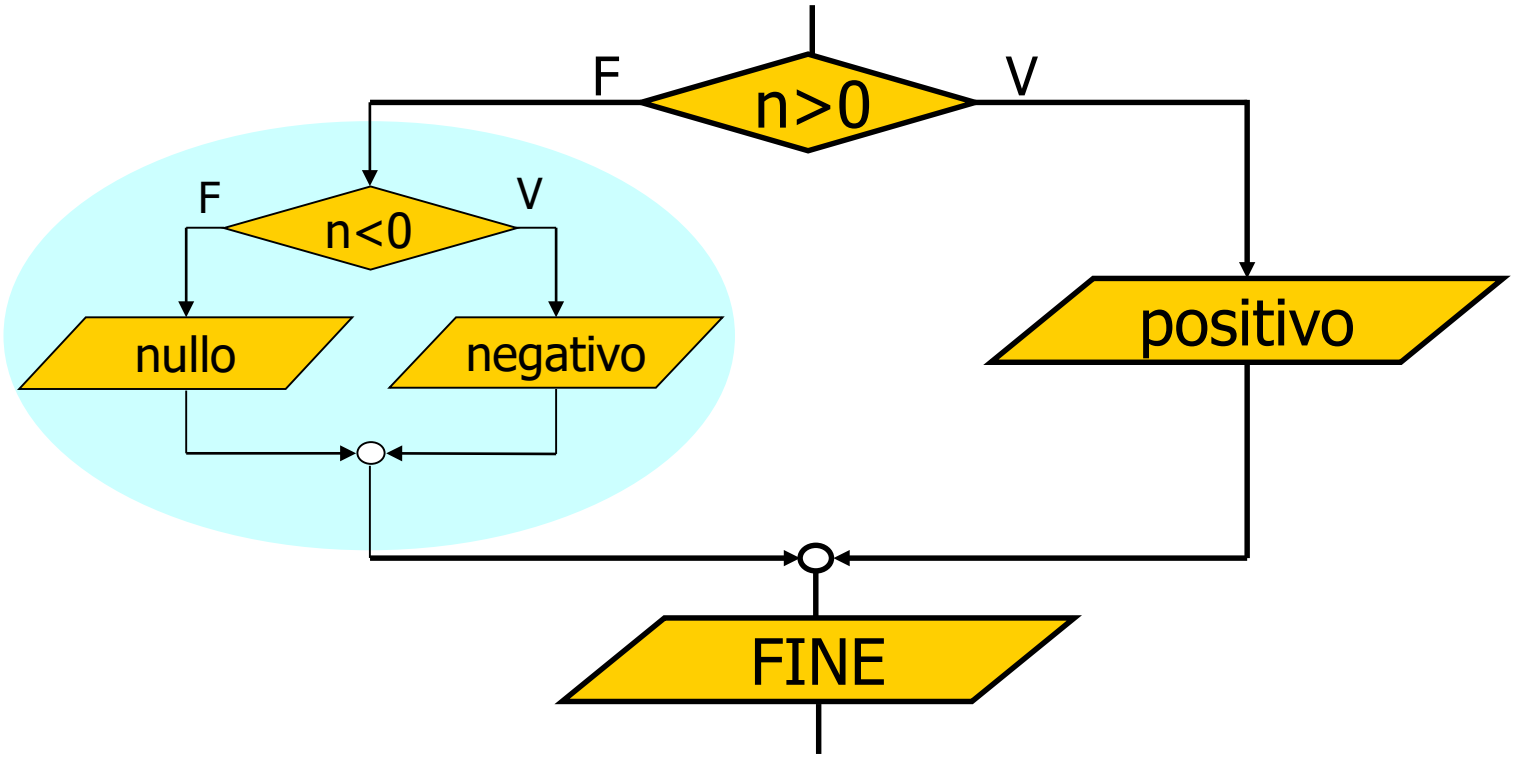
  - **Inefficiente** perché le condizioni vengono SEMPRE valutate TUTTE, anche quando non è necessario (ad esempio quando `n` > 0)

Selezione a più rami

- Soluzione 2
  Si utilizzano più costrutti `if` ***annidati,*** *ossia* uno *completamente* dentro *un ramo* dell'altro (in questo esempio il blocco del ramo `else` contiene un altro costrutto `if` completo)
  - **Efficiente** perché le condizioni NON vengono valutate sempre TUTTE, ma si ferma alla prima che viene soddisfatta

Selezione a più rami

- Soluzione 2 – Flow-chart



Quando `n` > 0 non valuta tutto il ramo `else`

Selezione a più rami

- Soluzione 2 – codice (1ª forma)

```
scanf("%d", &n);
if (n > 0)
  printf("positivo\n");
else
    if (n < 0)
      printf("negativo\n");
    else
      printf("nullo\n");
printf("FINE\n");
```

Costrutto `if` interno

Si noti che il *costrutto* `if` *interno* è un'unica istruzione (essendo unica non servono graffe)

Selezione a più rami

- Soluzione 2 – codice (2ª forma, MIGLIORE)

```
scanf("%d", &n);
if (n > 0)
    printf("positivo\n");
else if (n < 0)
    printf("negativo\n");
else
    printf("nullo\n");
printf("FINE\n");
```

Dato che per il compilatore ritorni a capo e spazi sono equivalenti, questa forma e la precedente sono identiche (stessa sequenza di operazioni), ma le keyword sono *allineate*, i blocchi sono *allineati*.

Selezione a più rami

- La struttura generale dell' **`if`** è dunque:

`if` $(condizione_1)$
    $blocco_1$
`else if` $(condizione_2)$
    $blocco_2$
`else if` $(condizione_3)$
    $blocco_3$
…
`else`
    $blocco_{else}$

0 o più

0 o 1

Dove: **`if`** deve esserci, possono essere presenti più **`else if`** (anche nessuno), può essere presenta al massimo un **`else`**

Corrispondenza dell'else

- Ogni `else` si riferisce sempre all'ultimo `if`
- Se è necessaria una corrispondenza diversa, si usino le graffe per isolare il blocco interno:

```
if (condizione1)
{
   if (condizione2)
      blocco_then_IF2;
}
else
   blocco_else_IF1;
```

Senza le graffe precedenti, l'**`else`** sarebbe associato all'`if` interno e costituirebbe il *blocco_else_IF2*

Espressioni relazionali

- Confrontano due valori, se sono di tipo diverso vengono promossi come nelle espressioni aritmetiche (incluse le promozioni integrali)
- I valori possono essere variabili, costanti o generiche espressioni
- Il risultato è un valore di tipo `int`, pari a `1` se vero, a `0` se falso
- Attenzione al confronto tra valori `signed` e `unsigned` (vedere slide sulle espressioni)
- Operatori:

`==`	→	uguale	`!=`	→	diverso
`<`	→	minore	`<=`	→	minore o uguale
`>`	→	maggiore	`>=`	→	maggiore o uguale

Espressioni relazionali

- Hanno priorità inferiore alle operazioni aritmetiche

```
if (a-5 >= 12*x) ...
```

  prima calcola le espressioni `a-5` e `12*x`
  poi ne fa il confronto

- Hanno priorità maggiore degli operatori di assegnamento

```
int x;
x = 5 > 2;
```

  prima fa il confronto tra 5 e 2
  poi ne assegna il risultato (`1`) a `x`

Confronto di valori f.p.

- Poiché i valori floating point possono essere memorizzati con un'approssimazione dovuta alla conversione da base 10 a base 2, i risultati dei calcoli sono anch'essi approssimati
- Ad esempio `(0.1+0.2 == 0.3)` dà risultato falso (differiscono internamente di circa 10^{-9})
- Quindi non si deve mai fare il confronto diretto tra numeri floating point, ma si deve verificare che la loro differenza in valore assoluto sia minore di un margine di errore accettabile per i dati previsi

Confronto di valori f.p.

- Esempio
  Non `if (a==b)` ma
  `if (fabs(a-b)<e)`
  con `e` pari all'errore tollerabile
- Ovviamente `e` dipende dai valori che si stanno utilizzando, se ad esempio i valori hanno al massimo 2 cifre dopo la virgola, la precisione assoluta di rappresentazione degli stessi è 1/100, quindi `e`=0.01 è accettabile
- Esistono diversi sistemi per valutare l'errore accettabile, oggetto di corsi più avanzati

Valori logici

- In un contesto logico (es. come condizione di una `if`) i valori numerici assumono un significato logico:
  - il valore 0 (di qualsiasi tipo) viene considerato equivalente a **falso**
  - ogni valore ≠0 è equivalente a **vero**
- Ad esempio

```
int test = a > b;
if (test) …
```

- Come già detto, gli operatori relazionali danno come risultato esattamente 1 e 0 per indicare vero e falso

Valori logici

- Per rendere il programma più leggibile si possono definire simboli con significato logico:

```
#define TRUE 1
#define FALSE 0
```

O anche:

```
#define SI 1
#define NO 0
```

- Oppure definire delle costanti con `enum`:

```
enum boolean {FALSE, TRUE};
enum {NO, SI};
```

- Se nella `enum` non si omette il tag si possono anche definire variabili di quel tipo

Valori logici

- Poiché un qualsiasi valore ≠0 equivale già a vero, nelle espressioni relazionali è possibile non scrivere un eventuale confronto '`!=0`'

  `if (trovato)` **...**

  equivale quindi a

  `if (trovato != 0)` **...**

- Si preferisca la forma che di volta in volta facilita la comprensione del programma (nell'esempio sopra, grazie all'uso di variabili significative del contenuto, è più chiaro leggere "se trovato" [sottinteso è vero] invece che "se la variabile trovato è diversa da 0")

Operatori logici

- Operano su valori logici e danno un risultato `int` esattamente pari a `1` se vero, `0` se falso
- Operatori (in ordine di priorità decrescente):

`!` → **NOT**, ha priorità **superiore** agli operatori relazionali e aritmetici (usare le parentesi se si deve applicare a un espressione)

```
if ( !trovato ) …
if ( !(a>b) ) …
```

`&&` → **AND**, ha priorità **inferiore** agli operatori relazionali e aritmetici (non servono le ())

```
if ( a>b && c!=0 ) …
```

`||` → **OR**, ha priorità **inferiore** agli operatori relazionali e aritmetici (non servono le ())

```
if ( a>b || c!=0 ) …
```

Espressioni logiche

- L'operatore `&&` ha precedenza sull'operatore `||` quindi

  ```
  if ( a>b || c<d && a!=0 ) ...
  ```

  equivale a:

  ```
  if ( a>b || (c<d && a!=0) ) ...
  ```

- Si possono usare le parentesi per cambiare l'ordine di valutazione delle operazioni

  ```
  if ( (a>b || c<d) && a!=0 ) ...
  ```

- **Attenzione:** è errato scrivere una condizione come `a < b < c`, occorre spezzarla in

  ```
  a<b && b<c
  ```

Espressioni logiche

- L'operatore `!` nega una condizione (converte vero in falso e falso in vero), numericamente:
  - valore ≠ 0 → 0
  - valore =0 → 1
- Esempi

```
int x = 12;
x = !x;  → 0
x = !x;  → 1
```

- Poiché un valore pari a 0 equivale a falso

```
if (trovato == 0) ...
```

equivale a:

```
if (!trovato) ...
```

in quanto `trovato == 0` è vero

Valutazione minima di && e ||

- Gli operatori `&&` e `||` vengono valutati sempre da sinistra a destra
- La valutazione delle espressioni termina *non appena* è possibile stabilire se la condizione è complessivamente vera o falsa (*shortcircuit*):
  - `if (cond1 && cond2 && cond3 && ...)`
    Se `cond1` è falsa (0), non si valutano le $\texttt{cond}_x$ successive e complessivamente la condizione dà 0 (falso)
  - `if (cond1 || cond2 || cond3 || ...)`
    Se `cond1` è vera (!=0), non si valutano le $\texttt{cond}_x$ successive e complessivamente la condizione dà 1 (vero)

Valutazione minima di && e ||

- Questo comportamento è utile per evitare che le condizioni successive vengano valutate (ed eseguite) quando le precedenti ad esse non sono soddisfatte, ad esempio nella seguente `if (delta>=0 && sqrt(delta)>10)` se `delta<0` la seconda condizione non viene calcolata (darebbe errore)
- Gli operatori `&&` e `||` inseriscono un *sequence point* tra le espressioni che collegano (v.oltre)
- Quindi gli effetti degli operatori `++` e `--` in una condizione vengono portati a termine prima della valutazione della condizione successiva

Costrutto switch

- Per selezionare uno tra più rami alternativi *si può sempre* utilizzare una serie di costrutti `if` annidati (o in cascata)
- Ma se la selezione:
  - dipende dal risultato di un'unica espressione intera
  - e i risultati dell'espressione sono valori *interi costanti noti a priori*

  allora si *può anche* utilizzare il più efficiente costrutto `switch`

Costrutto switch

- Per costruire l'istruzione `switch`, la condizione dell'`if` viene "spezzata" spostando le parti che compongono
- Il costrutto `if` seguente:

if (*espressione* = *risultato*$_1$)
  blocco$_1$
else if (*espressione* = *risultato*$_2$)
  blocco$_2$
...
else
  blocco$_{default}$

utilizzando uno `switch` diventa...

Costrutto switch

- Sintassi di **`switch`**

```
switch (espressione)          ← senza il `;'
{
case risultato1:              ← notare il :
   blocco1
case risultato2:              ← notare il :
   blocco2
...
default:                      ← notare il :
   blocco default
}
```

- Per migliorare la leggibilità non si mettano spazi tra `switch` e la parentesi

Selezione multipla

- Funzionamento (di massima):
  - viene calcolata *espressione*
  - il valore risultante viene cercato tra i $risultato_x$ elencati a destra delle keyword `case`, quindi:
    - se viene trovato un $risultato_x$ uguale al risultato di *espressione*
      il $blocco_x$ corrispondente viene eseguito (ma non solo questo, vedere più avanti)
    - altrimenti, se esiste il caso `default`
      viene eseguito $blocco_{default}$
    - altrimenti nessun blocco viene eseguito

Costrutto switch

- Caratteristiche delle varie parti:
  - *espressione* è un espressione che produce un risultato di *tipo intero* (`char`, `short`, `int`, `long`)
  - $risultato_x$ sono valori interi costanti noti al compile time (numeri, `#define`, `enum`, non valori `const`)
  - $blocco_x$ sono blocchi di codice (le parentesi graffe per i blocchi sono opzionali e in genere sono omesse se non contengono la definizione di variabili locali al blocco)

Costrutto switch

- Alla fine di ciascun `case` e dell'eventuale `default` deve esserci il carattere `':'`
- L'ordine in cui sono elencati i vari $risultato_x$ è ininfluente, sia dal punto di vista dell'ordine di valutazione, sia da quello della performance (ossia i $risultato_x$ indicati all'inizio dello `switch` non è detto siano valutati prima dei successivi in elenco), è indefinito
- Il ramo `default` è opzionale, in genere viene collocato in fondo alla lista, ma per quanto detto sopra non è importante sia collocato lì
- `switch` non deve essere terminato da `';'`

Costrutto switch

- Quando si entra ad eseguire un blocco, vengono eseguiti *in cascata* (*fall-through*) anche **tutti i blocchi successivi**
- Esempio (scorretto)

```
switch (cifra) {
case 0:
   cont0++;
case 1:
   cont1++;
case 2:
   cont2++;
...
default:
   altri++;
}
```

Se ad esempio `cifra` contiene il valore 1, vengono incrementati TUTTI i contatori, incluso `altri` ed escluso `cont0`

Costrutto switch

- Per eseguire il solo blocco corrispondente all'espressione e poi uscire dallo `switch` si deve utilizzare l'istruzione `break` collocata come ultima istruzione **di ciascun blocco**
- È buona norma aggiungere l'istruzione `break` anche nell'ultimo blocco, in modo che se si aggiungono successivamente altri `case` i precedenti sono già terminati correttamente
- *Dopo aver visto i cicli:*
  Un'istruzione `break` in un blocco `switch` collocato all'interno di un ciclo fa uscire dallo `switch` e NON dal ciclo

Costrutto switch

- Esempio (corretto)

```
switch (cifra) {
case 0:
   cont0++;
   break;
case 1:
   cont1++;
   break;
case 2:
   cont2++;
   break;
...
default:
   altro++;
   break;
}
```

Se ad esempio `cifra` contiene il valore 1, viene incrementato solo `cont1`

Costrutto switch

- L'esecuzione in cascata è indispensabile per associare più *risultato*$_x$ allo stesso *blocco*$_x$

```
switch (cifra)
{
case 0:  ← blocco vuoto senza break
case 1:  ← blocco vuoto senza break
case 2:  ← blocco vuoto senza break
case 3:
   cifre03++;
   break;
default:
   cifre49++;
   break;
}
```

Esercizi

1. Scrivere un programma che chieda due numeri da tastiera e dei due visualizzi il maggiore (es. se vengono inseriti 12 e 27 visualizza 27).
2. Scrivere un programma che chieda un numero da tastiera e indichi a video se è pari o dispari (consiglio: calcolare il resto).
3. Scrivere un programma che chieda tre numeri da tastiera e dei tre visualizzi il maggiore.
4. Scrivere un programma che chieda tre numeri da tastiera e li visualizzi in ordine decrescente.

Esercizi

5. Si vogliono dividere gli allievi di un corso in tre squadre denominate ROSSA, VERDE e BLU secondo il loro numero di matricola.
   L'assegnazione avviene con il seguente criterio: l'allievo con matricola 1 va nella squadra ROSSA, quello con matricola 2 nella VERDE, quello con matricola 3 nella BLU, quello con matricola 4 nella ROSSA, quello con 5 nella VERDE ecc.
   Il programma deve chiedere il numero di matricola dell'allievo e indicare a quale squadra è assegnato. Usare il costrutto `if`.
6. Come il precedente, utilizzare uno `switch`.

Esercizi

7. Scrivere un programma che chieda da tastiera di introdurre un numero intero corrispondente ad un voto e stampi a video
   "`Insufficiente`" se è inferiore a 18,
   "`Appena sufficiente`" (18),
   "`Basso`" (19-20),
   "`Medio`" (21-23),
   "`Buono`" (24-26),
   "`Alto`" (27-29),
   "`Massimo`" (30)
   "`Impossibile`" (tutti gli altri)
   Usare il costrutto `if`.
8. Come il precedente, utilizzare uno `switch`.

Esercizi

9. Si scriva un programma che chieda i tre coefficienti `a`, `b` e `c` di un'equazione di secondo grado, calcoli e visualizzi i valori delle soluzioni se questi sono reali; nel caso non lo siano deve semplicemente visualizzare "`Valori non reali`".
10. Un anno secolare (divisibile per 100) è bisestile se è divisibile per 400, un anno non secolare è bisestile se è divisibile per 4. Ad esempio l'anno 1900 non era bisestile, il 1996 era bisestile, il 2000 lo era, il 2002 non lo era. Si scriva un programma che chieda all'utente di introdurre l'anno e indichi se è bisestile.