Lezione 3

Assegnamento
Compendio sintassi
Tipo booleano ed operatori logici
Espressioni aritmetiche e logiche
Programmazione strutturata
Istruzioni condizionali

Assegnamento

Istruzione di assegnamento

- Espressione di assegnamento:

  nome_variabile **=** <*espressione*>

- Istruzione di assegnamento

  <*espressione di assegnamento*> **;**

- Viene utilizzata per assegnare ad una variabile (non ad una costante!) il valore di un’espressione

lvalue e rvalue

- Viene preso l'indirizzo della variabile individuata mediante l'identificatore a sinistra dell'assegnamento
  - tale indirizzo è detto **lvalue** (left value)
- Il valore dell'espressione che compare a destra fornisce il nuovo valore
  - tale valore è detto **rvalue** (right value)

Assegnamento e memoria

Esempio

```
int  N;
```

simbolo	*indirizzo*
N	**1600**

	...
1600	?
	...

L'esecuzione di una **definizione** provoca l'allocazione di uno spazio in memoria pari a quello necessario a contenere un dato del tipo specificato

```
N = 150;
```

simbolo	*indirizzo*
N	**1600**

	...
1600	**150**
	...

L'esecuzione di un **assegnamento** provoca l'inserimento nello spazio relativo alla variabile del valore indicato a destra del simbolo **=**

Ordine di esecuzione

- L'esecuzione di un'istruzione di assegnamento comporta **prima** la valutazione di tutta l'espressione a destra dell'assegnamento.
  Esempi:

```
int c, d;
c = 2;
d = (c+5)/3 - c;
d = (d+c)/2;
```

- Solo **dopo** si inserisce il valore risultante (rvalue) nella spazio di memoria dedicato alla variabile

Risultato assegnamento 1/2

- Come tutte le espressioni, anche l'espressione di assegnamento ha un proprio valore
- In particolare ha per valore *l'indirizzo della variabile a cui si è assegnato il nuovo valore* (quindi l'lvalue) Esempio: l'espressione **`a = 3`** ha per valore l'indirizzo di **`a`**
- Uno dei modi in cui si può sfruttare tale indirizzo è per effettuare *assegnamenti multipli*, ad esempio:

```
int c, d;
c = d = 2;
```

  - L'effetto della seconda istruzione, che, come si vedrà meglio in seguito, è equivalente a
    `c = (d = 2) ;`
    è il seguente:

Risultato assegnamento 2/2

- L'espressione `d = 2` produce come valore l'indirizzo della variabile `d`
- L'espressione `c =` ... si aspetta a destra un valore da assegnare a `c`
  - Siccome si ritrova invece l'indirizzo di una variabile, tale indirizzo viene utilizzato per accedere al (nuovo) valore della variabile `d` ed utilizzarlo per assegnare il nuovo valore a `c`
- In definitiva dopo l'istruzione
  `c = (d = 2) ;`
  sia `c` che `d` hanno il valore 2

Esercizio: numero al contrario

- Specifiche (nel nostro caso le specifiche sono una semplice traccia)

  Leggere da *stdin* un numero intero positivo, non multiplo di 10 e compreso tra 101 e 999, e memorizzare in una variabile intera un numero intero le cui cifre siano in ordine inverso rispetto al numero letto da *stdin*; stampare infine il numero ottenuto

- Esempi:
  103 → 301
  234 → 432
  527 → 725

Procediamo con ordine

- Per fare un buon lavoro, rispettiamo le fasi di sviluppo viste nella seconda esercitazione!
- Quindi analizziamo prima di tutto con calma il problema
- Quindi cerchiamo di farci venire un'idea (chiara) su come risolverlo ...

Idea!

- Utilizzare le operazioni di modulo e di divisione fra numeri interi
- Dato un numero, valgono le seguenti relazioni:
  - Unità = (numero/10^0)%10;
    - Esempio: (234/1)%10 = 4
  - Decine = (numero/10^1)%10;
    - Esempio: (234/10)%10 = 3
  - Centinaia = (numero/10^2)%10;
    - Esempio: (234/100)%10 = 2

Algoritmo 1/3

- Dato il numero letto da *stdin* (ad esempio 234)
  - Memorizzare in una variabile **`unita`** il risultato di **`numero % 10`** (che ci restituisce proprio le unità)
    Nel nostro esempio otteniamo: **`unita = 4`**
  - Memorizzare in una variabile decine il risultato di **`(numero/10) % 10`**
    Nel nostro esempio: **`decine = 23%10 = 3`**
  - Memorizzare in una variabile **`centinaia`** il risultato di **`(numero/100) % 10`**
    Nel nostro esempio: **`centinaia = 2%10 = 2`**

Algoritmo 2/3

- A questo punto, nel risultato, la cifra memorizzata dentro `unita` deve indicare le centinaia, quindi va moltiplicata per 100
  Nel nostro esempio, 4 deve diventare 400
- La cifra memorizzata dentro `decine` deve indicare le decine, quindi va moltiplicata per 10
  Nel nostro esempio, 3 deve diventare 30
- La cifra memorizzata dentro `centinaia` deve indicare le unità, quindi non va moltiplicata per nulla
  Nel nostro esempio, 2 deve rimanere 2

Algoritmo 3/3

- In definitiva il risultato va calcolato come:

```
unita*100 + decine*10 + centinaia
```

Programma

```
main()
{
  int numero;
  int unita, decine, centinaia, risultato;

  cin>>numero;

  unita = (numero)%10;
  decine = (numero/10)%10;
  centinaia = (numero/100)%10;

  risultato = unità*100 + decine*10 + centinaia;
  cout<<risultato<<endl;
}
```

Esercizio per casa

Leggere da *stdin* un numero intero compreso tra 100 e 999, e ristamparlo al contrario

- Esempio: 100 va ristampato come 001
- Notare che in questa traccia non è richiesto di memorizzare alcun risultato in alcuna variabile

Ancora sulla sintasssi del C/C++

Sintassi del C/C++ 1/2

- Ora che abbiamo più familiarità col linguaggio, fissiamo un po' meglio la sintassi ...
- Un programma C/C++ è una sequenza di parole (**token**) delimitate da spazi bianchi (**whitespaces**)
  - *Spazio bianco*: carattere spazio, tabulazione, a capo
  - *Parola*: sequenza di lettere o cifre non separate da spazi bianchi
- Token possibili: *identificatori, parole chiave (riservate), espressioni letterali, operatori, separatori*
  - Operatore: denota una operazione nel calcolo delle espressioni
  - Separatore: `( ) , ; : { }`

Sintassi del C/C++ 2/2

IDENTIFICATORI

<Identificatore> ::= *<Lettera>* { *<Lettera>* | *<Cifra>* }

- *<Lettera>* include tutte le lettere, maiuscole e minuscole, e l'underscore "_"
- La notazione { A | B } indica una sequenza indefinita di elementi A o B
- Maiuscole e minuscole sono considerate diverse (il linguaggio C/C++ è *case-sensitive*)

PAROLE CHIAVE (RISERVATE)

- **`int, float, double, char, if, for, do, while, switch, break, continue,`** …

  `{ }` delimitatore di blocco

COMMENTI

- **`// commento, su una sola riga`**
- **`/* commento,`**

  `anche su più righe */`

Uso degli spazi bianchi

- Una parola chiave ed un identificatore **vanno separati da almeno uno spazio bianco**
- Esempio:
  `int a; // inta sarebbe un identificatore !`
- In tutti gli altri casi gli spazi bianchi <u>non sono obbligatori</u>
  - Li si utilizza però per migliorare la leggibilità del programma per un 'umano'
- Si può separare una coppia di token consecutivi col <u>numero ed il tipo di spazi bianchi che si preferisce</u> (va messo almeno uno spazio bianco solo nel caso si tratti di una parola chiave seguita da un identificatore)

Tipo booleano

Tipo booleano

- Disponibile in C++, ma non in C
- Nome del tipo: `bool`
- Valori possibili: vero (**true**), falso (**false**)
  - `true` e `false` sono due letterali booleani
- Esempio di definizione:

```
bool u, v = true ;  // la seconda variabile
                    // è inizializzata a vero
```

- Operazioni possibili: ...

Operatori logici

operatore logico	*numero argomenti*	**sintassi (posizione)**	**esempi**
not logico (negazione)	*uno (unario)*	**!** (prefisso)	`bool b, a = !true ;` `b = !a ;`

and logico (congiunzione)	*due (binario)*	**&&** (infisso)	`bool b, a, c ;`
			`c = a && b ;`
			`b = true && a ;`

or logico (disgiunzione)	*due (binario)*	**||** (infisso)	`bool b, a, c ;`
			`c = a || b ;`
			`b = true || a ;`

Che valori ritornano questi operatori?

La loro semantica è definita dalle cosiddette tabelle di verità

Tabella di verità

AND			*Ris.*	OR			*Ris.*	NOT	*Ris.*
V	&&	V	V	V	||	V	V	!V	F
V	&&	F	F	V	||	F	V	!F	V
F	&&	V	F	F	||	V	V		
F	&&	F	F	F	||	F	F		

Tipo booleano e tipi numerici

- Se un oggetto di tipo booleano è usato dove è atteso un valore numerico
  - **`true`** è convertito a 1
  - **`false`** è convertito a 0
- Viceversa, se un oggetto di tipo numerabile è utilizzato dove è atteso un booleano
  - ogni valore diverso da 0 è convertito a **`true`**
  - il valore 0 è convertito a **`false`**

Esercizio

- *stampa_bool.cc* della terza esercitazione

Tipo booleano e linguaggio C

- In C, non esistendo il tipo `bool`, gli operatori logici
  - operano su interi
    - il valore 0 viene considerato falso
    - ogni valore diverso da 0 viene considerato vero
  - e restituiscono un intero:
    - il risultato è 0 o 1
- Esempi di espressioni con operatori logici (che in C++ ritornerebbero **true** o **false**)

```
5 && 7            0 || 33          !5
```

Operatori di confronto

Operatori di confronto

== Operatore di confronto di uguaglianza
(il simbolo = denota invece l'operazione di assegnamento!)

!= Operatore di confronto di diversità

> Operatore di confronto di maggiore stretto

< Operatore di confronto di minore stretto

>= Operatore di confronto di maggiore-uguale

<= Operatore di confronto di minore-uguale

- Restituiscono un valore di tipo **booleano**: **`true`** oppure **`false`**

Esercizio

- *stampa_logica_semplice.cc* della terza esercitazione

Espressioni

Espressioni

- Costrutto sintattico formato da letterali, identificatori, operatori, parentesi tonde, ...
- Operatori binari
  - Moltiplicativi: *   /   %
  - Additivi: +   -
  - Traslazione: <<   >>
  - Relazione (confronto): <   >   <=   >=
  - Eguaglianza (confronto): ==   !=
  - Logici: && ||
  - Assegnamento: =   +=   -=   *=   /=
- Abbiamo già visto quasi tutti questi operatori parlando del tipo **int** e del tipo **bool**

Altri operatori

- Assegnamento abbreviato: +=, -=, *=, /=, ...
  `a += b ;` ↔ **`a = a + b ;`**

- Incremento e decremento: ++ --

  - Prefisso: prima si effettua l'incremento/decremento, poi si usa la variabile. Restituisce un **lvalue** (l'indirizzo della variabile incrementata)

    ```
    int a = 3; cout<<++a; // stampa 4
    (++a) = 4; // valido
    ```

  - Postfisso: prima si usa il valore della variabile, poi si effettua l'incremento/decremento. Restituisce un **rvalue**

    ```
    int a = 3; cout<<a++; // stampa 3
    (a++) = 4; // ERRORE !!!
    ```

Tipi di espressioni

- Un'espressione si definisce
  - aritmetica: produce un risultato di tipo aritmetico
  - logica: produce un risultato di tipo booleano
- Esempi:

Espressioni aritmetiche

2 + 3

(2 + 3) * 5

4 > 2

true || (2 > 5)

Espressioni logiche

Proprietà degli operatori

- <u>**Posizione**</u> rispetto ai suoi operandi (o argomenti): prefisso, postfisso, infisso
- <u>**Numero di operandi (arietà)**</u>
- <u>**Precedenza**</u> (o <u>**priorità**</u>) nell'ordine di esecuzione
  - Es: **`1 + 2 * 3`** è valutato come **`1 + (2 * 3)`**
    `k<b+3` è valutato come **`k<(b+3)`**, e non **`(k<b) +3`**
- <u>**Associatività**</u>: ordine con cui vengono valutati due operatori con la stessa precedenza.
  - <u>Associativi a sinistra</u>: valutati da sinistra a destra
    - Es: **`/`** è associativo a sinistra, quindi **`6/3/2`** ⇔ **`(6/3)/2`**
  - <u>Associativi a destra</u>: valutati da destra a sinistra
    - Es: **`=`** è associativo a destra ...

Associatività assegnamento

- L’operatore di assegnamento può comparire più volte in un’istruzione.
- L’associatività dell’operatore di assegnamento è a **destra**
  Esempio:
  ```
  k = j = 5;
  ```
  equivale a
  ```
  j = 5;
  k = j;
  ```
- Invece:
  ```
  k = j+2 = 5; // ERRORE !!!!!
  ```
  perché **j+2** non può fornire un **lvalue**, ossia l'indirizzo di una variabile!

Ordine valutazione espressioni

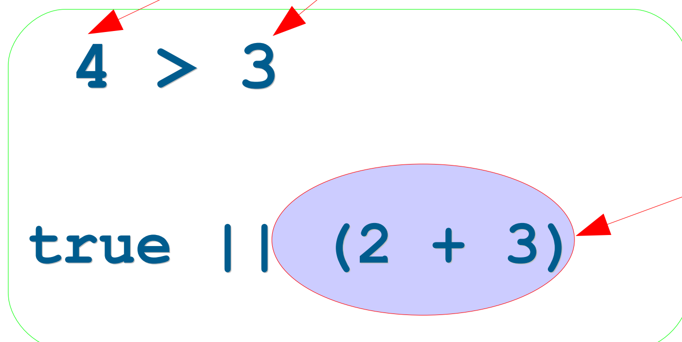
- Si calcolano prima i fattori, quindi i termini
  - **Fattori**: ottenuti dalle espressioni letterali e calcolo delle funzioni e degli operatori unari
  - **Termini**: ottenuti dal calcolo degli operatori binari
    - Moltiplicativi: * / %
    - Additivi: + -
    - Traslazione: << >>
    - Relazione: < > <= >=
    - Eguaglianza: == !=
    - Logici: && ||
    - Assegnamento: = += -= *= /=
- Con le parentesi possiamo modificare l'ordine di valutazione dei termini

Esempi

Espressioni aritmetiche

Fattori

Termini



Espressioni logiche

Sintesi priorità degli operatori

Fattori

! ++ --

Termini

* / %

+ -

> >= < <=

== !=

&&

||

? :

Assegnamento

=

Esercizi

- *stampa_logica_composta.cc* e *stampa_1_se_in_intervallo.cc* della terza esercitazione

Programmazione strutturata

Programmazione strutturata

- Si parla di **programmazione strutturata** [Dijkstra, 1969] se si utilizzano solo i seguenti costrutti per determinare l'ordine di esecuzione delle istruzioni (detto anche flusso di controllo):
  - **concatenazione** e composizione
    - conosciamo già la concatenazione, mentre la composizione permette di 'trattare' una sequenza di istruzioni come se fosse una sola istruzione
  - **selezione** (**istruzione condizionale**)
    - fa proseguire il flusso di controllo tra due possibili rami in base al valore vero o falso di una espressione detta *condizione di scelta*
  - **iterazione**
    - permette all’esecuzione ripetuta di un’istruzione o di una sequenza di istruzioni finché permane vera una espressione detta “condizione di iterazione”

Scopo e possibili limiti

- Rendere i programmi più leggibili e facili da manutenere
- Perdiamo qualcosa se utilizziamo solo i costrutti della programmazione strutturata nei nostri programmi?
- Ossia, rischiamo di non essere in grado di codificare qualche algoritmo?
- Ci vuole un pizzico di teoria ...

Tesi di Church-Turing

- Ogni algoritmo può essere eseguito (calcolato) da una **Macchina di Turing**
  - Macchina dotata di una testina e di un nastro costituito da un numero, concettualmente infinito, di celle adiacenti
  - La testina può: spostarsi da una cella all'altra, leggere/scrivere la cella su cui si trova
- Questa tesi è indimostrabile, o perlomeno mai dimostrata, ma è ormai universalmente accettata

Teorema di Jacopini-Boem

- Assumendo la tesi di Church-Turing per vera, tale teorema afferma che ogni algoritmo può essere tradotto in un programma scritto con un linguaggio caratterizzato <u>solo</u> da
  - Tipi di dato: Naturali con l'operazione di somma (+)
  - Istruzioni: assegnamento
istruzione composta
istruzione condizionale
istruzione di iterazione
- Quindi con la programmazione strutturata si può esprimere <u>qualsiasi algoritmo</u>

- In questa prima presentazione vedremo la selezione (ossia le istruzioni condizionali) e la composizione (ossia le istruzioni composte)

Istruzioni condizionali

Istruzioni condizionali

- In C/C++ disponiamo di due tipi di istruzioni condizionali:
  - Istruzione di SCELTA SEMPLICE o ALTERNATIVA
  - Istruzione di SCELTA MULTIPLA
    Non è essenziale, ma migliora l'espressività del linguaggio

Scelta semplice

- Consente di scegliere fra due istruzioni alternative in base al verificarsi di una data *condizione*

<*istruzione-di-scelta*> ::=
`if` (<*condizione*>) <*istruzione1*>
[**`else`** <*istruzione2*>]

- <*condizione*> è un'espressione logica che viene valutata al momento dell'esecuzione dell'istruzione **`if`**
- Se <*condizione*> risulta vera si esegue <*istruzione1*>, altrimenti si esegue <*istruzione2*>
- In entrambi i casi l'esecuzione continua poi con l'istruzione che segue l'istruzione **`if`**.
- NOTA: Se <*condizione*> è falsa e la parte **`else`** (opzionale) è omessa, si passa subito all'istruzione che segue l'istruzione **`if`**

Esempio

```
int a=3, n=-6, b=0;
if (n <= 0)
      a = b + 5;
```

- Alla fine dell'esecuzione
  - a == ?
  - b == ?
  - n == ?

Esempio

```
int a=3, n=-6, b=0;
if (n > b)
      a = b + 5;
else
      n = b*5;
```

- Alla fine dell'esecuzione
  - a == ?
  - b == ?
  - n == ?

Esercizi

- Svolgere gli esercizi della terza esercitazione fino all'esercizio sulla divisione intera incluso (slide 10-29)

Problema

- E se vogliamo eseguire più di una istruzione in uno dei due rami o in entrambi?
- Esempio:

```
if (<condizione>)
        <varie istruzioni>
else
        <varie istruzioni>
```

Abbiamo bisogno delle *istruzioni composte* ...

Istruzioni composte

Istruzione composta

- Sequenza di istruzioni racchiuse tra parentesi graffe:

```
{
        <istruzione1>
        <istruzione2>
        ...
}
```

- Ovunque la sintassi preveda una istruzione si può inserire tanto una istruzione *semplice* (ossia non composta) che una istruzione composta
- Ai fini della sintassi e della semantica una istruzione composta è trattata come una istruzione semplice
- L'esecuzione di una istruzione composta implica l'esecuzione ordinata di tutte le istruzioni della sequenza tra parentesi graffe

Completamento istruzioni di scelta semplice

Forma completa

- Identica a quella già vista:
  <*istruzione-di-scelta*> ::=
  `if` (<*condizione*>) <*istruzione-ramo-if*>
  [**`else`** <*istruzione-ramo-else*>]
- Sia l'istruzione del ramo **`if`** che quella del ramo **`else`** possono essere una qualsiasi istruzione semplice (istruzione espressione, istruzione condizionale, istruzione iterativa) o composta
- Le istruzioni alternative da eseguire sono spesso chiamate anche *corpo del ramo if* o *corpo del ramo else*

Esempio

```
if (n > 0)
    { /* inizio blocco */
        a = b + 5;
        c = x + a - b;
    } /* fine  blocco */
else
    n = b*5;
```

Esercizio

- Svolgere gli esercizi della terza esercitazione dalla slide 30 alla 46

Istruzioni di scelta annidate

- Come caso particolare, <*istruzione-ramo-if*> o <*istruzione-ramo-else*> potrebbero essere a loro volta un’istruzione di scelta
- In questo caso occorre fare attenzione ad associare i rami **`else`** (opzionali) agli **`if`** corretti
- In base alla sintassi del linguaggio C/C++, un ramo **`else`** è sempre associato all’**`if`** più interno (vicino)
- Se questa non è l'associazione desiderata, occorre racchiudere l'**`if`** più interno in un blocco **`{ }`**

Esempi 1/2

?

```
if (n > 0)
    if (a>b) n = a;
    else n = b*5; // associato all'if
                  // più interno
                  // (vicino)
```

Esempi 2/2

Per far sì che l'**else** si riferisca al primo **if**:

```
if (n > 0){
      if (a>b)
            n = a;
} else
      n = b*5;
```

Per maggiore leggibilità, si possono usare le parentesi anche nell'altro caso:

```
if (n > 0) {
      if (a>b) n = a;
     else n = b*5;
}
```

Esercizi

- Risolvere il problema alla slide 47 della terza esercitazione
- Passare alla quarta esercitazione e risolvere tutti gli esercizi fino al controllo di *overflow* in caso di prodotto

Istruzioni di scelta multipla

Istruzione di scelta multipla

- Consente di scegliere fra molti casi in base al valore di un'**espressione di selezione**

Sintassi e semantica 1/2

<istruzione-di-scelta-multipla> ::=

```
switch (<espressione di selezione>) {
        case <etichetta1> : <sequenza_istruzioni1> [ break; ]
        case <etichetta2> : <sequenza_istruzioni2> [ break; ]
        ...
        [ default : < sequenza_istruzioniN> ]
}
```

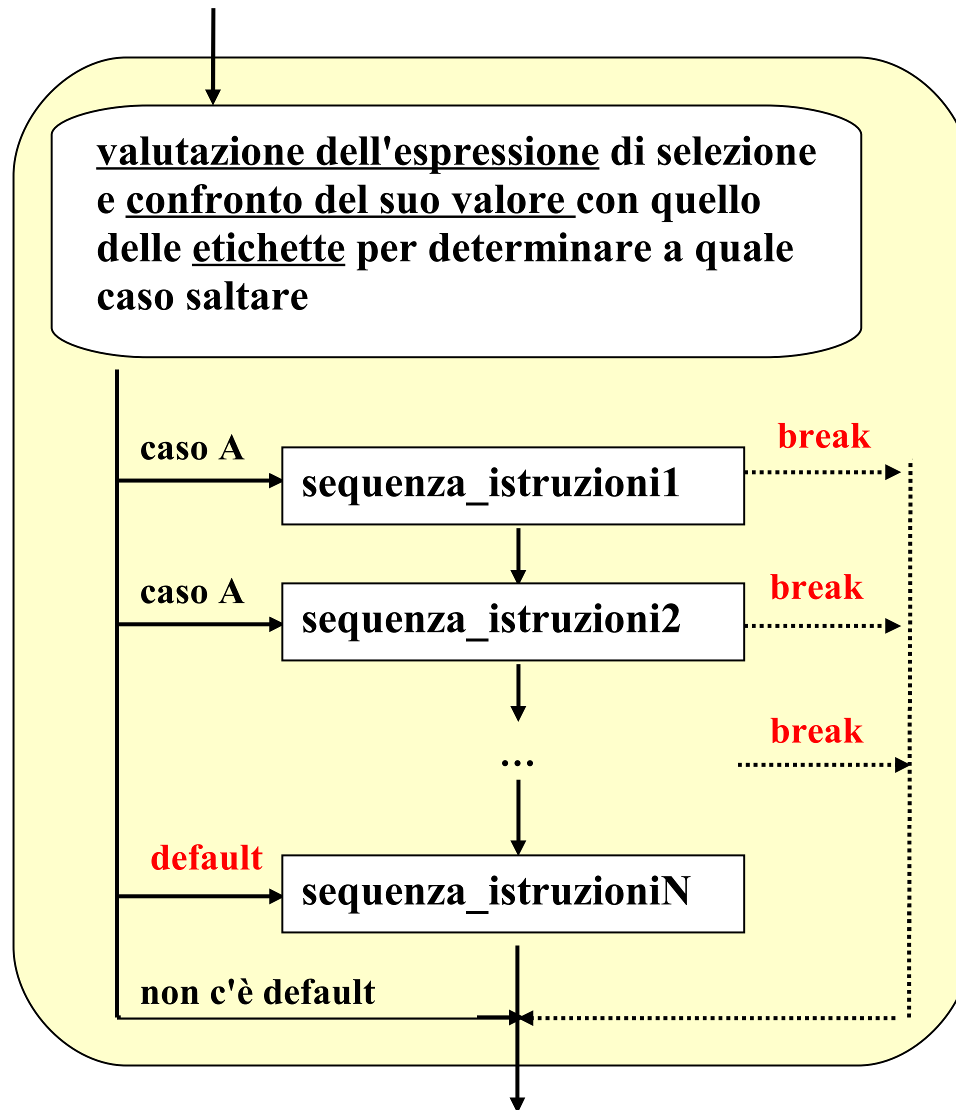
<espressione di selezione> è un'espressione che restituisce un valore **numerabile** (intero, carattere, enumerato, ...), e viene valutata al momento dell'esecuzione dell'istruzione switch

Le etichette *<etichetta1>*, *<etichetta2>*, ... devono essere delle costanti dello stesso tipo dell'espressione di selezione

Sintassi e semantica 2/2

- Definiamo ***corpo dell'istruzione switch***, la parte del costrutto compresa tra le parentesi graffe
- Il valore dell'espressione di selezione viene confrontato con le **costanti** che etichettano i vari casi: l'esecuzione salta al ramo dell'etichetta corrispondente, se esiste (vedi diagramma di flusso nella prossima slide)
  - L'esecuzione prosegue poi **sequenzialmente** fino alla fine del corpo dell'istruzione **`switch`**
    - **A meno che non si incontri un'istruzione `break`**, nel qual caso si esce dal corpo dello **`switch`**: ossia l'esecuzione prosegue dall'istruzione successiva all'istruzione **`switch`**
- Se nessuna etichetta corrisponde al valore dell'espressione, si salta al ramo **`default`** (se specificato)
  - Se tale ramo non esiste, l'esecuzione prosegue con l'istruzione successiva all'istruzione **`switch`**

Diagramma di flusso



Esempio/esercizio

```
int a = 2, n ;
cin>>n; // considerare separatamente i casi in cui
        // l'utente immetta 1, 2, 3, 4, oppure 0
switch (n){
      case 1:
            cout<<"Ramo A"<<endl;
            break;
      case 2:
            cout<<"Ramo B"<<endl;
            a = a*a;
            break;
      case 3:
            cout<<"Ramo C"<<endl;
            a = a*a*a;
            break;
      default:
            a=1;
}
cout<<a<<endl; // cosa viene stampato ?
```

Osservazioni

- <*sequenza_istruzioni*> denota una sequenza di istruzioni, quindi non è necessaria una istruzione composta
  - L'idea è che si salta all'inizio di uno dei rami
- In accordo al punto precedente, i vari rami non sono mutuamente esclusivi: una volta saltato all'inizio di un ramo, l'esecuzione prosegue in generale con le istruzioni dei rami successivi fino alla fine del corpo dello **`switch`**
- Per avere rami mutuamente esclusivi occorre forzare esplicitamente l'uscita mediante l'istruzione **`break`**

Esercizio

- Svolgere l'esercizio *primo_menu.cc* della quarta esercitazione
- Non dimenticare di inserire, dove necessaria, l'istruzione **`break;`**

Esempio/esercizio

```
int a = 2, n, b = 1;
cin>>n; // considerare separatamente i casi in cui
        // l'utente immetta 0, 1, 2, 3
switch (2 - n) {
      case 0:
            b *= a;
      case 1:
            b *= a;
      case 2:
            break;
      default:
            cout<<"Valore non valido per n\n" ;
}
cout<<b<<endl; // cosa viene stampato ?
```

Esercizio

- Svolgere gli esercizi *menu_multiplo.cc* e *calcolatrice.cc* della quarta esercitazione

Pro e contro scelta multipla

- L'istruzione **`switch`** garantisce maggiore leggibilità rispetto all'**`if`** quando c'è da scegliere tra più di due alternative
- Altrimenti è ovviamente un costrutto più ingombrante
- Ulteriori limitazioni dell'istruzione **`switch`**:
  - è utilizzabile solo con espressioni ed etichette di tipo numerabile (intero, carattere, enumerato, ...)
  - non è utilizzabile con numeri reali (**`float`**, **`double`**) o con tipi strutturati (stringhe, vettori, strutture...)