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> ***;***

  - E' cioè una espressione di assegnamento **seguita da un *;***
  - Viene utilizzata per assegnare ad una variabile (non ad una costante!) il valore di un'espressione

Domanda

- Quali operazioni bisogna effettuare sulla memoria per assegnare un nuovo valore ad una variabile?

Risposta

- Bisogna scrivere il nuovo valore all'interno delle celle di memoria in cui è memorizzata la variabile
- Più a basso livello: bisogna scrivere una nuova configurazione di *bit*
  - quella che rappresenta il nuovo valore

Assegnamento e memoria

Esempio

```
int  N=10;
```

simbolo	indirizzo
N	1600

	...
1600	10
	...

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 **=**

Domanda

- Quale informazione bisogna avere per poter modificare il valore di una variabile in memoria?

Risposta

- Bisogna sapere dove si trova in memoria!
- Occorre sapere cioè il suo **indirizzo**
  - Ossia l'indirizzo della prima delle celle consecutive in cui è memorizzata la variabile

lvalue e rvalue

- Come si effettua quindi l'assegnamento? Consideriamo, per esempio, il precedente assegnamento:
  `N = 150;`
- Viene preso l'indirizzo della variabile individuata dall'identificatore a sinistra dell'assegnamento (l'identificatore è **`N`** nel nostro esempio)
  - tale indirizzo è detto **lvalue** (left value)
- Viene calcolato il valore dell'espressione che compare a destra (150 nell'esempio) ed assegnato all'oggetto memorizzato all'indirizzo (lvalue) ottenuto nel passo precedente
  - tale valore è detto **rvalue** (right value)

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` (ossia 2), 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

Stato conoscenze

- A questo punto conosciamo due istruzioni del linguaggio
  - Definizione
  - Assegnamento
- Inoltre abbiamo imparato un po' ad usare gli oggetti *cout* e *cin* senza indagare sul meccanismo interno con cui funzionano

Esercizio: numero al contrario

- Specifica del problema
  - come sempre accadrà d'ora in poi, nel nostro caso la specifica è una semplice traccia:

- Leggere da *stdin* un numero intero positivo, che si assume essere compreso tra 100 e 999 (lo si dà per scontato senza effettuare controlli), e stamparlo al contrario (con le cifre in ordine inverso)

- Esempi:
  103 → 301
  230 → 032
  527 → 725

Procediamo con ordine

- Questo è un problema per risolvere il quale dobbiamo riflettere bene ...
- Per fare un buon lavoro, rispettiamo le fasi di sviluppo viste nella seconda esercitazione!
- Quindi analizziamo prima di tutto con calma il problema
- Solo dopo di essere sicuri di aver chiaro il problema anche nei dettagli, cerchiamo di farci venire un'idea (chiara) su come risolverlo ...

Suggerimento 1/3

- Prendiamo un numero qualsiasi, per esempio 573
- Quanto vale 573 % 10 ?

Suggerimento 2/3

- Vale 3
- Ossia proprio il valore della sola cifra delle unità ...
- E se volevamo ottenere il valore della cifra delle decine?
  - Purtroppo 573%100 non è uguale alla cifra delle decine, cioè 7, ma è uguale a 73
  - Per poter estrarre le decine, potrei utilizzare %10 se riuscissi prima a trasformare 573 in 57 ...
  - Come potrei fare?

Suggerimento 3/3

- Lo divido semplicemente per 10 (divisione intera)
  573 / 10 = 57
- Quindi quanto farà (573/10) % 10 ?
- Una volta capito come ottenere anche le decine, dovreste essere pronti per l'idea completa

Idea!

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

Algoritmo

- 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`**

Programma

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

   cin>>numero;

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

   cout<<unita<<decine<<centinaia<<endl;
}
```

Esercizio per casa

- Leggere da *stdin* un numero intero positivo, che si assume non essere multiplo di 10 ed essere compreso tra 101 e 999 (senza effettuare controlli), 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

- Idea, algoritmo e soluzione nelle prossime due slide

Idea

- In aggiunta all'idea di base già vista per l'esercizio precedente, nella variabile intera in cui va memorizzato il numero al contrario, 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

- Si può esprimere l'algoritmo con una semplice formula algebrica
- Il numero da memorizzare nella variabile intera andrà calcolato come:

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

Soluzione

```
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;
}
```

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 (**whitespace**)
  - *Spazio bianco*: carattere spazio, tabulazione, a capo
  - *Parola*: sequenza di lettere o cifre non separate da spazi bianchi

Token possibili: *operatori, separatori, identificatori, parole chiave (riservate), commenti, espressioni letterali*

  - 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 non sono obbligatori
  - Li si utilizza però per migliorare la leggibilità del programma per un 'umano'
- Si può separare una coppia di *token* consecutivi col numero ed il tipo di spazi bianchi che si preferisce (ripetiamo che 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: sintassi

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 && F F && V F && F	V F F F	V || V V || F F || V F || F	V V V F	!V !F	F V

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
```

Booleani e valori interi in C++

- Per compatibilità col C, anche in C++ si possono utilizzare gli interi dove sono attesi dei booleani
- Tali valori sono convertiti nel modo seguente:
  - il valore 0 viene convertito a **false**
  - ogni valore diverso da 0 viene convertito a **true**

Operatori di confronto

Operatori di confronto 1/2

`==` 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`**

Operatori di confronto 2/2

- Gli operatori di confronto si possono applicare agli oggetti di tipo **int**
- Gli operatori di uguaglianza e diversità si possono applicare anche agli oggetti di tipo **bool**
  - Gli altri operatori di confronto hanno poco senso coni booleani

Esercizio

- *stampa_logica_semplice.cc* della terza esercitazione

Gruppo facebook

facebook.com/groups/informaticaunimore2012

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**

Domanda

- Data una variabile booleana **`x`**, che differenza c'è tra i valori delle espressioni per ciascuna delle seguenti coppie?

```
x == true          x

x == false         !x
```

Risposta

- **Nessuna**
- Un programmatore utilizza sempre la forma sintatticamente e concettualmente più semplice per una data espressione
- Quindi nelle precedenti coppie di espressioni sono da preferire:

```
x

!x
```

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, cosa assegna ad a?
```

  - 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++) = 7; // 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` è uguale a `(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 `k = (j = 5);` ossia:
  ```
  j = 5;
  k = j;
  ```

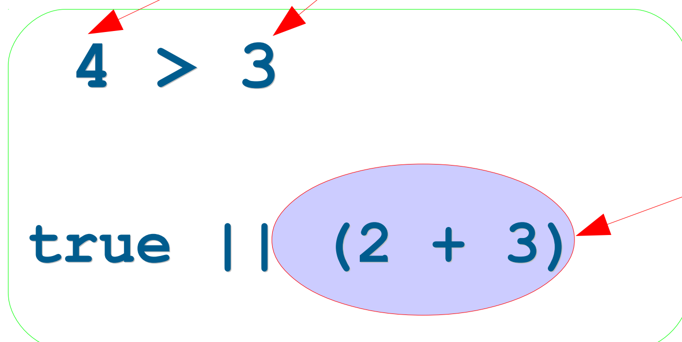
- Invece:
  ```
  k = j + 2 = 5; // ERRORE !!!!!
  ```
  perché `j + 2` non può fornire un **lvalue**, ossia l'indirizzo di una variabile!

Ordine valutazione espressioni

- Si calcolano prima i fattori, quindi i termini
  - **Fattori**: ottenuti dalle espressioni letterali e dal 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



Espressioni logiche

Sintesi priorità degli operatori

Fattori
- ! ++ --

Termini
- * / %
- + -
- > >= < <=
- == !=
- &&
- ||
- ? :

Assegnamento
- =

Priorità decre-scente

Esercizi

- *stampa_logica_composta.cc* e *stampa_1_se_in_intervallo.cc* della terza esercitazione
- Leggere con cura i consigli forniti nella slide della terza esercitazione successiva alla slide in cui è riportato il secondo dei precedenti due esercizi

Programmazione strutturata

Problema 1/2

- Supponiamo di dover svolgere il seguente esercizio, utilizzando solo le istruzioni apprese finora
- Scrivere un programma che legge un numero intero da standard input (*cin*, *stdin*) che rappresenta il voto preso in Programmazione I e, se il voto è superiore a 27, stampa

  Programmazione I è veramente uno dei migliori corsi di Informatica!!

  Altrimenti stampa:

  Quel def ... di docente di quel noioso ...

Problema 2/2

- E' praticamente impossibile
- Come mai?

Risposta e nuova domanda

- Perché, siccome le istruzioni sono eseguite l'una dopo l'altra, ogni volta che si fa partire il programma si eseguono sempre le stesse istruzioni
- Noi invece vorremmo che in un caso si stampasse qualcosa, e nell'altro caso qualcos'altro
  - Ossia che in un caso si eseguisse una certa istruzione, e nell'altro caso un'altra
- Supponendo di poter aggiungere nuove istruzioni a quelle che conosciamo, come potremmo riuscire a raggiungere questo scopo?

Risposta e nuova domanda

- Una soluzione sarebbe la seguente
  - All'interno del programma inseriamo sia le istruzioni da eseguire in un caso che le istruzioni da eseguire nell'altro caso
  - Dopo l'istruzione di lettura del valore da stdin
    - inseriamo un controllo, in cui valutiamo quale dei due casi sia vero
    - e **saltiamo** al pezzo di codice relativo a quel caso

Salto: goto, jump, ...

- Il salto è la prima tecnica adottata, storicamente, per eseguire passi diversi a seconda dei dati passati in ingresso
- I tipici nomi delle istruzioni di salto sono:
  - **`goto`** nei linguaggi ad alto livello
  - **`jump`** nel linguaggio macchina

Salti e cicli

- Le istruzioni di salto permettono anche di ripetere più volte l'esecuzione di un dato pezzo di codice
  - Un pezzo di codice scritto in maniera tale che si ripeta più volte viene tipicamente chiamato *ciclo*
  - Per realizzarlo si può utilizzare una istruzione di salto per saltare all'indietro
    - saltare cioè all'inizio del ciclo quando lo si vuole ripetere, e proseguire invece dall'istruzione successiva alla fine del ciclo quando si vuole smettere

Problemi salto

- Stiamo quindi per studiare le istruzioni di salto?
- No
- Come mai?

Logica a spaghetti 1/2

- Perché, se si realizza la logica di un programma mediante salti avanti ed indietro, il programma stesso tende a diventare molto difficile da capire
- A meno che il programmatore non applichi delle regole rigide nell'utilizzo delle istruzioni di salto,
  - si tende alla cosiddetta **logica a spaghetti**

Logica a spaghetti 2/2

- La sequenza di esecuzione delle istruzioni tende cioè a diventare un groviglio

Roberto Gianferrari 16/7/2011

Quali regole?

- Quali sono le regole rigide a cui abbiamo accennato?
- Sono delle regole che fanno sì che l'ordine di esecuzione delle istruzioni coincida con l'ordine che si può ottenere utilizzando solo i costrutti della cosiddetta 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 ...

Macchina di Turing

- Macchina dotata di
  - una testina
  - un nastro costituito da un numero di celle adiacenti concettualmente infinito
- La testina può: spostarsi da una cella all'altra, leggere/scrivere la cella su cui si trova
- http://www.google.com/doodles/alan-turings-100th-birthday
  - Se siete curiosi cercatevi le istruzioni su quale è lo scopo del doodle e su come programmare la macchina di turing per cercare di raggiungere lo scopo

Tesi di Church-Turing

- Ogni algoritmo può essere eseguito (calcolato) da una **Macchina di Turing**
- Questa tesi è indimostrabile, o perlomeno mai dimostrata, ma è ormai universalmente accettata

Teorema di Jacopini-Boem

- Assumendo che la tesi di Church-Turing sia vera, tale teorema afferma che ogni algoritmo può essere tradotto in un programma scritto con un linguaggio caratterizzato <u>solo</u> da
  - Tipo 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>

Costrutti

- In questa prima presentazione vedremo
  - la selezione
    - ossia le istruzioni condizionali
  - 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

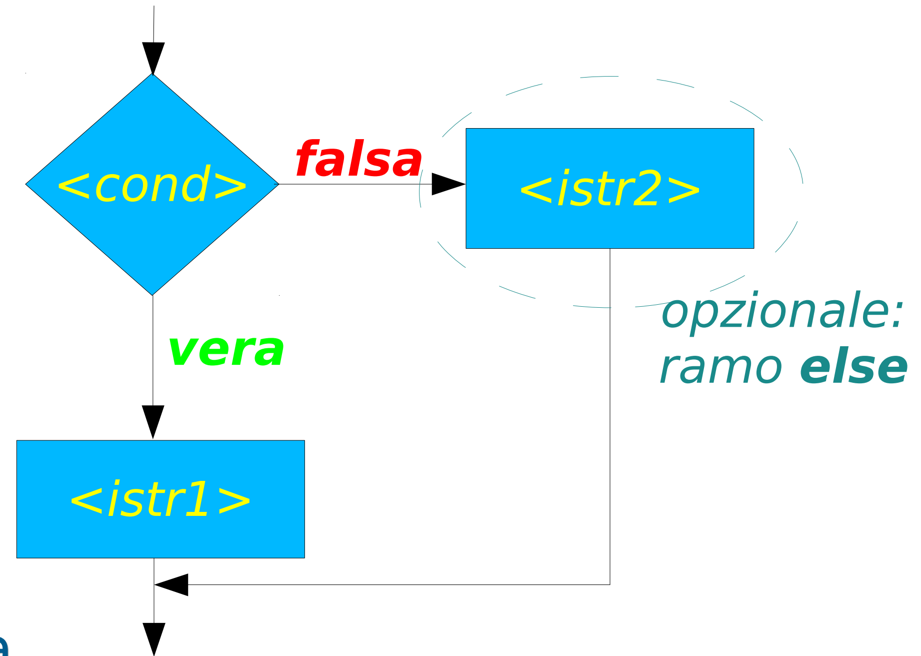
- Consente di scegliere fra due istruzioni alternative in base al verificarsi di una data *condizione*

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

- <*condizione*> è un'espressione logica che viene valutata al momento dell'esecuzione dell'istruzione **`if`**

Diagramma di flusso

```
if (<condizione>)
        <istruzione1>
[ else
        <istruzione2> ]
```



- 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

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 singola 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 la terza esercitazione fino alla slide 48

Istruzioni di scelta annidate

- Come caso particolare, <*istruzione-ramo-if*> o <*istruzione-ramo-else*> potrebbero essere a loro volta un'istruzione di scelta
- Esempio:

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

- A quale **`if`** è associato il ramo **`else`**, il primo o il secondo?

Regola

- 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 **`{ }`**
- Cerchiamo di capire meglio con degli esempi

Esempi 1/2

NO → / SI →

```
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 49 della terza esercitazione
- Svolgere gli esercizi per casa della terza esercitazione

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 1/3

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

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

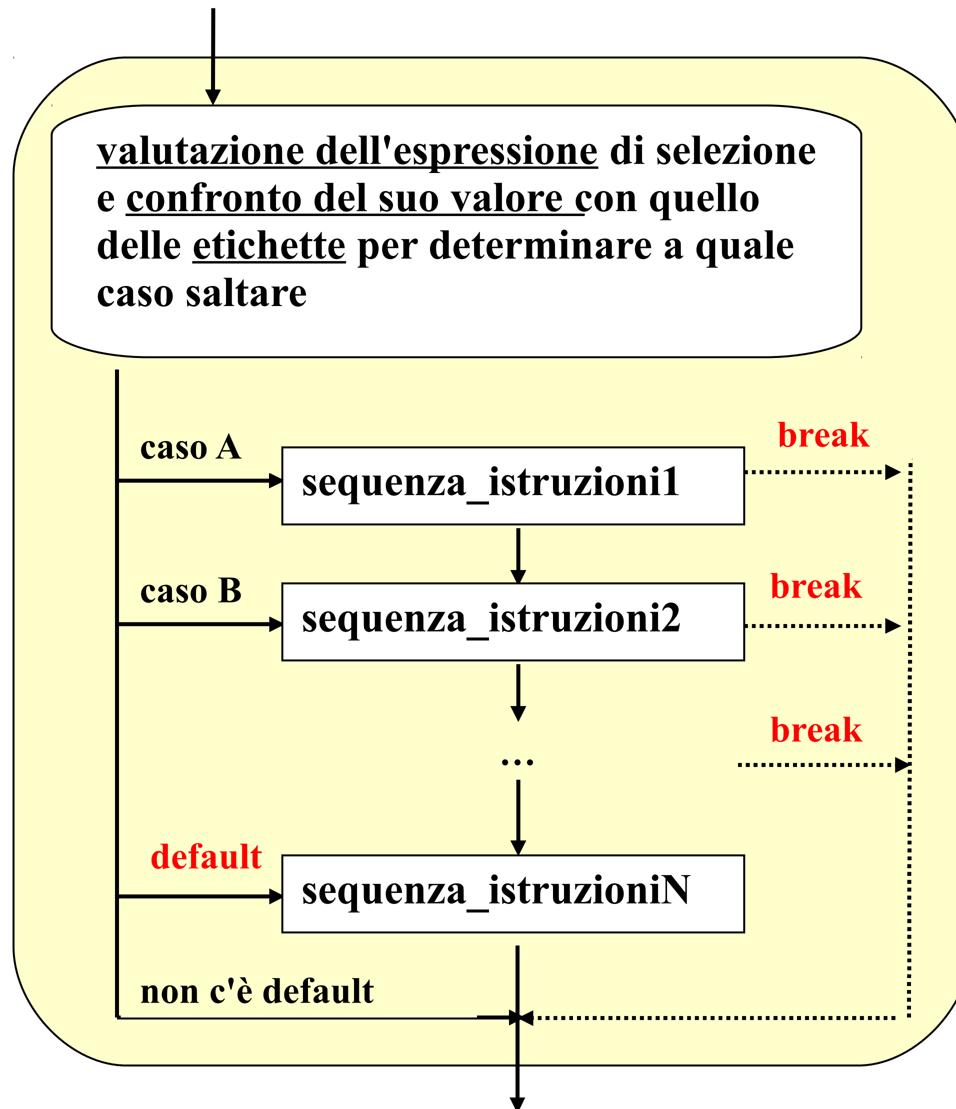
<espressione di selezione> è un'espressione che restituisce un valore **numerabile** (intero, carattere, enumerato, ...), e viene valutata al momento dell'esecuzione dell'istruzione switch

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

Sintassi e semantica 2/3

- 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 (la spiegazione continua poi nella slide ancora successiva)

Diagramma di flusso



Sintassi e semantica 3/3

- Dopo il salto al ramo di una delle etichette
  - 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`**

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 nei vari casi ?
```

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...)

Esercizi

- Terminare la quarta esercitazione
- Svolgere la quinta esercitazione fino all'esercizio *stampa_hex.cc*