Corso di Laurea triennale in Ingegneria Chimica

in condivisione con

Corso di Laurea triennale in

Ingegneria Navale e Scienze dei Materiali

Elementi di Informatica

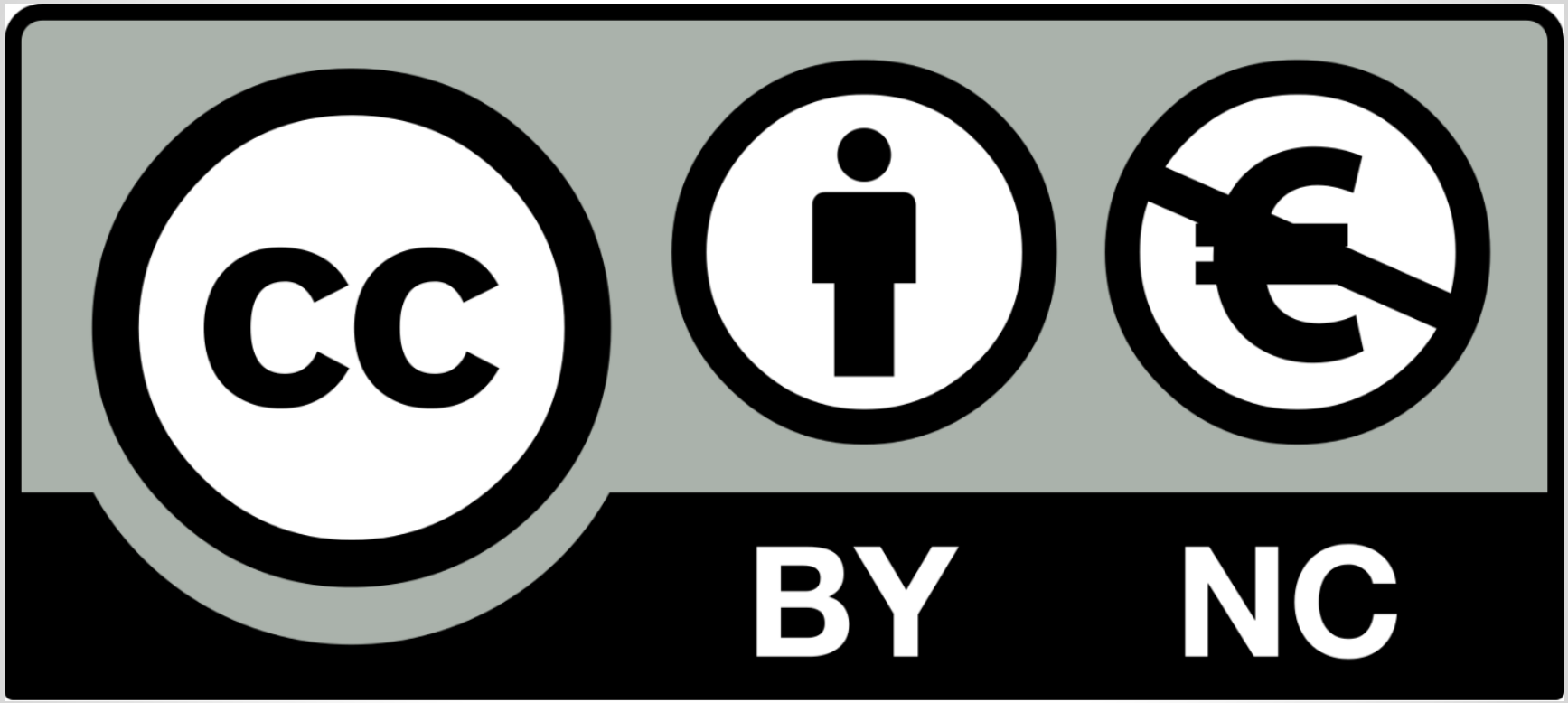
A.A. 2016/17

prof. Mario Barbareschi

Linguaggi di programmazione

Informazioni di Licenza

- Questo lavoro è licenziato con la licenza Creative Commons BY-NC



- Per consultare una copia della licenza visita: http://creativecommons.org/licenses/by-nc/3.0/legalcode

Linguaggio di Programmazione

- Il linguaggio di programmazione è una **notazione formale** per descrivere algoritmi ed è dotato di un **alfabeto, un lessico, una grammatica, una sintassi ed una semantica.**
- Un limitato insieme di regole definisce il lessico del programma, ovvero le parole del linguaggio.
- L'organizzazione delle parole in frasi è invece guidata da regole della grammatica. I periodi sono costruiti in base alla **sintassi.**
- Infine l'attribuzione di un significato alle frasi è oggetto delle **regole semantiche**.

Generazioni di linguaggi (1/2)

- Il **linguaggio macchina** è il linguaggio compreso dalla CPU: esso ha uno stretto legame con l'hardware ma è difficilmente usabile, essendo composto da istruzioni formate da sequenze di bit.
  - Ogni architettura di calcolo possiede il proprio linguaggio macchina.

- Per evitare di scrivere programmi direttamente in linguaggio macchina sono stati introdotti i **linguaggi assemblativi** (o linguaggi di basso livello, o di seconda generazione) che usano un linguaggio elementare per rappresentare istruzioni in linguaggio macchina.

Linguaggio Assemblativo	Linguaggio Macchina
STOR OP1, 2	01000110
ADD OP1, OP1	01100111

Generazioni di linguaggi (2/2)

- I **linguaggi di alto livello** (o di terza generazione) sono concepiti per essere più comprensibili agli uomini (piuttosto che all'hardware) e usano parole dei linguaggi naturali (inglese) e concetti affini alla matematica.
  - I linguaggi di seconda e terza generazione necessitano di un processo di traduzione in linguaggio macchina affinché gli algoritmi possano essere eseguiti dal calcolatore.

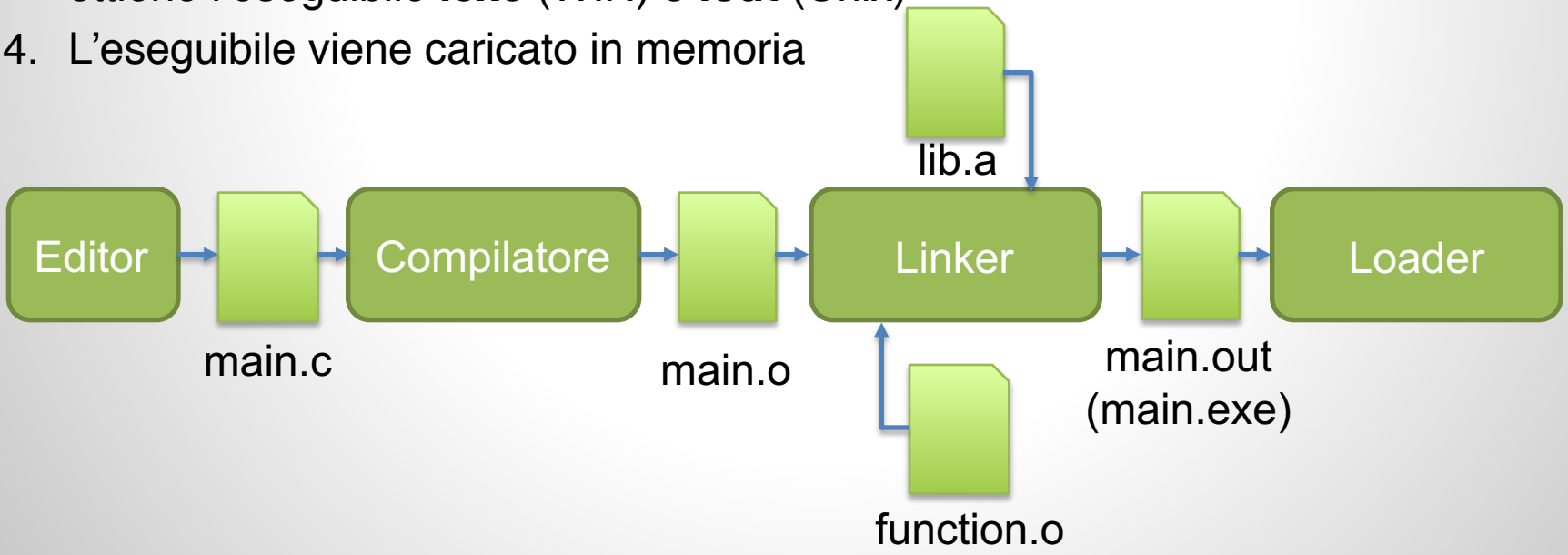
Linguaggio di alto livello	Linguaggio Assemblativo	Linguaggio Macchina
x = 2	STOR OP1, 2	01000110
x = x + x	ADD OP1, OP1	01100111

Il linguaggio C/C++

- Il **linguaggio C** è un linguaggio **general-purpose** sviluppato nel tra il 1969-1973 da Dennis Ritchie per re-codificare il sistema operativo Unix.
  - Il C è stato progettato come un linguaggio di alto livello, ma include anche elementi di linguaggi di basso livello: questo rende il C altamente flessibile nel suo impiego, e i programmi C si traducono efficientemente in istruzioni macchina.
- Il C è stato standardizzato dall'ANSI nel 1989 (ANSI C), ed è attualmente il linguaggio più usato di tutti i tempi.
- Per estendere il linguaggio C con nuovi paradigmi di programmazione, nel 1983 Bjarne Stroustrup sviluppò il **linguaggio C++**.

Generazione di un programma eseguibile in C/C++

1. Si inizia con la creazione di un **file sorgente** con un'applicazione di scrittura testi (es. Notepad) oppure un editor integrato (es. IDE Dev C++): il file creato deve avere estensione **.cpp** (C++) o **.c** (C)
2. Una volta scritto il programma seguendo le regole del linguaggio, esso viene compilato (a riga di comando o tramite l'IDE); Se non ci sono errori, viene generato un file oggetto con estensione **.obj** (WIN) oppure **.o** (Unix)
3. Il linker assembla questo file oggetto con eventuali programmi esterni, sempre creati dal programmatore, oppure con delle funzioni di libreria (**.lib** o **.a**) e si ottiene l'eseguibile **.exe** (WIN) o **.out** (Unix)
4. L'eseguibile viene caricato in memoria



Compilazione Vs interpretazione

- I linguaggi C e C++ sono linguaggi di alto livello, dunque necessitano di un processo di traduzione in linguaggio macchina per l'esecuzione:
  - La **compilazione** (ad opera di un compilatore C/C++) traduce i **file sorgente** (i file che contengono i programmi scritti in C/C++) in **file oggetto**, che contengono la loro traduzione in linguaggio macchina.
  - I file oggetto passano poi ad una fase di collegamento, necessaria quando si riusano parti di programma scritte da altri programmatori.

Collegamento

- Il programma potrebbe aver bisogno di funzionalità definite esternamente (ad esempio, da altri programmatori ed offerte in forma di "librerie software").
- Un programma di collegamento detto **Linker** (ovvero, collegatore) completa il processo di creazione del file eseguibile, unendo (collegando) tutti i file oggetto e le librerie, per formare un programma eseguibile che include tutto ciò che è necessario per l'esecuzione.
- Alla fine di questo processo il **Loader** (ovvero, caricatore) può caricare il programma in memoria ed avviarne l'esecuzione.

Gestione degli errori

- Gli errori che si possono commettere quando si scrive un programma possono essere di tipo lessicale, sintattico o logico
  - Gli errori lessicali e sintattici sono segnalati durante il processo di compilazione.
  - Gli errori logici possono verificarsi a tempo di esecuzione.

Errori di compilazione

- Gli errori sintattici in fase di compilazione sono dovuti al fatto che non si rispetta la grammatica del linguaggio e possono essere rimossi modificando opportunamente il codice sorgente nell'editor.
- Alcune proposizioni rispettano comunque la grammatica, ma possono indicare errori semantici, pertanto il compilatore li segna come WARNING.
  - Es: tentativo di confrontare una variabile numerica con una stringa di caratteri

Errori di collegamento

- Ci possono essere errori di collegamento quando per esempio di utilizza un oggetto in maniera diversa rispetto a come è stato definito nel sorgente di origine
  - Es. una funzione con 3 parametri viene chiamata senza parametri

Errori di caricamento ed esecuzione

- Un tipico errore di caricamento si ha quando il programma eseguibile ha bisogno di più memoria di quella disponibile
- Un errore in esecuzione (anche detto bug o eccezione) mostra i suoi effetti solo quando il programma viene eseguito
  - Es: un'istruzione richiede la divisione per zero

Errori logici

- Gli errori logici solo legati all'algoritmo che il programma esegue e possono essere individuati solo **testando** il programma, fornendogli degli input opportuni e verificando che il risultato ottenuto è corretto
  - Es: il programma non termina e l'elaboratore va in stallo

Le frasi di un linguaggio di programmazione

- Tutti i linguaggi di terza generazione (alto livello) prevedono quattro tipologie di frasi diverse:
  - le **istruzioni** di calcolo ed assegnazione, che tradotte in linguaggio macchina indicano al processore le operazioni da svolgere;
  - le **strutture di controllo**, che definiscono l'ordine di esecuzione delle istruzioni;
  - le **frasi di commento**, che permettono l'introduzione di frasi in linguaggio naturale utili a rendere più comprensibili i programmi ad un lettore umano; le frasi di commento non vengono tradotte in linguaggio macchina
  - le **dichiarazioni**, con le quali il programmatore da ordini al traduttore del linguaggio di programmazione; anche le dichiarazioni non vengono tradotte in linguaggio macchina poiché servono solo a guidare il processo di traduzione.

Dichiarazioni di variabili: identificatori

- In C++, una variabile viene **dichiarata** con la sintassi:

  Tipo <nome_variabile>;

  - <*nome_variabile*> è un nome scelto dal programmatore formato da una qualunque sequenza di lettere (maiuscole o minuscole) e di cifre numeriche, senza altri caratteri o spazi bianchi a parte l'underscore (che è considerato una lettera), che deve cominciare con una lettera;
  - Il C++ è case-sensitive: significa che il compilatore considera le lettere maiuscole e minuscole come caratteri distinti;
  - L'identificatore scelto rappresenta un'area di memoria la cui dimensione dipende dal tipo della variabile;
  - E' conveniente usare nomi significativi per le variabili;
- Esempi di dichiarazioni

```
int x2;        //x2 è un intero
char c;        //c è un carattere
```

Definizione di una variabile

- La variabile è **definita** quando le è assegnato un valore, (che viene quindi scritto nella cella di memoria associata)
- Una variabile deve essere **dichiarata** prima di essere definita.
- La definizione di una variabile viene effettuata con *istruzioni di assegnazione*

variabile = espressione;

- Esempi di definizioni

  x2=3;

  c='a';
- Dichiarazione e definizione possono avvenire in un sol colpo:

  int x2=3;

Costanti

- Le costanti esprimono dei valori prefissati non alterabili e possono essere di tipo numerico ed alfanumerico
- Le costanti numeriche si distinguono in:
  - Costanti intere: sequenze di cifre eventualmente precedute da un segno
  - Costanti reali: si esprimono sia in virgola fissa esplicitando la posizioni del punto che in virgola mobile indicando l'esponente con la lettera e (minuscola o maiuscola)
    - 10.23, -0.11
    - 0.1e-23, -10.0003E10
- Le costanti alfanumeriche si distinguono in:
  - Costanti carattere: caratteri ASCII racchiusi tra apici
    - 'A', 'c', '?', '2'
  - Costanti stringhe di caratteri: sequenze di caratteri ASCII racchiusi tra doppie virgolette
    - "questa è una stringa"

Costanti stringhe di caratteri

- La stringa "" è una stringa di lunghezza nulla
- Con il backslash \ si può inserire qualsiasi carattere ASCII specificandone il codice in ottale o esadecimale
  - \101 è la lettera A codificata come 101 in ottale
  - \x41 è la lettera A codificata come 41 in esadecimale
- Una stringa di caratteri può essere spezzata su più linee usando il backslash
  - " questa è una stringa \

    molto lunga "
- E' possibile inserire caratteri speciali sia all'interno delle stringhe di caratteri che nelle costanti carattere

Carattere speciale	Descrizione
\n	Newline o anche ritorno a capo
\r	Carriage return o ritorno a inizio rigo
\t	Tab
\v	Vertical tab
\b	Backspace
\f	Form feed
\a	Alert o beep
\'	Singolo apice
\"	Singole virgolette
\?	Punto interrogativo
\\	Backslash

Dichiarazione di costanti

- Il linguaggio consente di assegnare alle costanti un nome da usare al loro posto nel programma per migliorare la leggibilità e la parametricità
- E' possibile anche dichiarare costanti tipizzate, usando il prefisso const

const nome_tipo *nome_costante* = valore;

- Nota: in questo caso a nome_costante viene attribuito uno spazio di memoria non modificabile

Tipi delle variabili C++

- I tipi delle variabili C++ possono essere:
  - **Semplici:** sono atomici ovvero non suddividibili e ci permettono di definire alcune tipologie base di variabili.
    - Esempi : interi, booleani, caratteri
  - **Strutturati:** sono composti da più tipi semplici o strutturati stessi.
    - Tipi strutturati omogenei vengono chiamati **array.**
    - Tipi strutturati eterogenei vengono chiamati **record (struct).**

Tipi semplici C++

- Tra i tipi semplici del C++ elenchiamo i seguenti:

Qualificatoe di accesso	Qualificatore di segno	Qualificatore di lunghezza	Tipi di dati	Uso
const	signed / unsigned	short/ long/ long long	int	Numeri interi o naturali
		long	float	Virgola mobile (IEEE 754 prec. Singola)
			double	Virgola mobile (IEEE 754 prec. doppia)
	signed / unsigned		char	Carattere ASCII
			bool	Valore booleano (logico)

- **Tutti i qualificatori sono opzionali.**
- C++ non definisce il numero di byte da usare per ogni tipo, ma dipende dal compilatore e dall'architettura utilizzata: l'istruzione **sizeof(type)** restituisce il numero di byte usati per un tipo.

Commenti

- In C++ le frasi di commento si indicano anteponendo alla frase la sequenza // quando il commento è su un solo rigo;
- Quando la frase occupa più righe la si può racchiudere fra /* e */
- I commenti possono cominciare subito dopo un'istruzione (dopo il «;»)

Operatori (1/4)

Operatori aritmetici

Operatore	Descrizione
+	Somma
-	Sottrazione
*	Moltiplicazione
/	Divisione
%	Modulo che restituisce il resto della divisione tra due interi

- Gli operatori aritmetici forniscono risultati dello stesso tipo delle variabili in ingresso, se la coppia ha lo stesso tipo, altrimenti del tipo che preserva maggiore accuratezza tra i due tipi della coppia.
  - Es.: float + float → float
    int + int → int
    double + double → double
    double + float → double
    int + double → double
    long int + short int → long int

Operatori (2/4)

Operatori logici

Operatore	Descrizione
&&	AND o anche prodotto logico
||	OR o anche somma logica
!	NOT o anche negazione logica

- Gli operatori logici forniscono risultati booleani (logici).
  - Anche se possono essere applicati a qualsiasi tipo di dato, si consiglia di usarli solamente per tipi di dati bool.
  - C++ usa le costanti “true” e “false” per riferirsi ai due valori di verità o falsità (ma si possono anche usare i numeri 1/0).

C++

```
bool propLogica_1 = true;
bool propLogica_2 = false;
bool propOr = propLogica_1 || propLogica_2 ;
bool propAnd = propLogica_1&& propLogica_2 ;
cout << propOr << “ “ << propAnd ;
```

Output:

```
1 0
```

Operatori (3/4)

Operatori relazionali

Operatore	Descrizione
==	Uguaglianza, per determinare se due valori sono uguali
!=	Diversità, per determinare se due valori sono diversi
>	Per determinare se il valore a sinistra precede quello di destra (o è più grande)
<	Per determinare se il valore a sinistra segue quello di destra (o è più piccolo)
>=	Per determinare se il valore a sinistra è più grande o uguale a quello di destra
<=	Per determinare se il valore a sinistra è più piccolo o uguale a quello di destra

- Si applicano a qualsiasi tipo di dato e restituiscono un booleano.
- **Attenzione a non confondere l'operatore di uguaglianza "==" con l'operatore di assegnazione "="!!**

C++

```
bool propLogica =  5 < 2;
bool propLogica2 =  !(5 == 2);

cout << propLogica << " " << propLogica2;
```

Output:

```
0 1
```

Operatori (4/4)

Operatori bitwise

Operatore	Descrizione
&	AND bit a bit
|	OR bit a bit
^	OR esclusivo bit a bit
~	Complementazione bit a bit o anche inversione dei bit (zero al posto di uno e viceversa)
<<	Shift left
>>	Shift right

- Questi operatori effettuano operazioni logiche bit-a-bit tra le stringhe contenute nei registri.
- **In applicazioni matematiche sono poco utilizzati, ma occorre fare attenzione a non confonderli con gli operatori logici!**

  & → Operatore bitwise

  && → Operatore logico

  | → Operatore bitwise

  || → Operatore logico

Operatori e tipi semplici

- Gli **operatori del linguaggio** hanno effetti diversi al seconda del tipo delle variabili sui quali sono applicati.

C++

```
int dividendo = 5;
int divisore = 2;

int risultato =
    dividendo / divisore;

cout << risultato;
```

Output:

```
2
```

C++

```
double dividendo = 5;
double divisore = 2;

double risultato =
    dividendo / divisore;

cout << risultato;
```

Output:

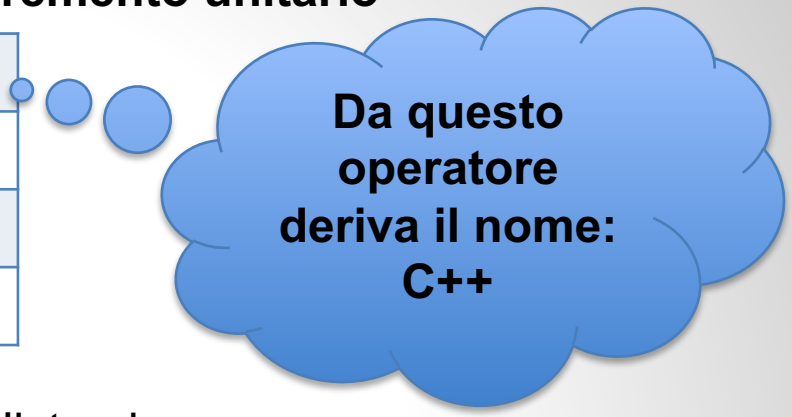
```
2.5
```

- Esempio: la divisione tra tipi interi darà un risultato che è ancora un intero; la divisione tra numeri a virgola mobile darà un risultato che è a virgola mobile.

Ulteriori operatori (1/2)

Operatori di incremento e decremento unitario

variabile++
variabile--
++variabile
--variabile



- L'effetto di questi operatori è traducibile con l'istruzione:

  variabile = variabile op 1;
- Tuttavia, quando il risultato di questi operatori sono assegnati a variabili:

risultato = variabile++;	Assegna **e dopo** incrementa
risultato = variabile--;	Assegna **e dopo** decrementa
risultato = ++variabile;	Incrementa **e dopo** assegna
risultato = --variabile;	Decrementa **e dopo** assegna

Ulteriori operatori (2/2)

Operatori composti

```
var = var op espressione;
```

```
var op= espressione;
```

Assegnazione	Equivalente a
B = B * 4	B *= 4
C = C – (D * 4)	C -= D * 4
X = X + (++I)	X += ++I
X = X + 1	X += 1

Operatore condizionale ?

```
condizione ? risulato1 : risultato2
```

Regole di precedenza tra operatori

Regole di precedenza degli operatori

Priorità	Operatore	Descrizione
1	++ --	Incremento/ decremento postfisso
2	++ --	Incremento/decremento prefisso
2	+ - !	Operatori unari di segno e complemento
3	(tipo)	Conversione di tipo
4	* / %	Moltiplicativi
5	+ -	Additivi
6	<< >>	Shift
7	== != < > <= >=	Relazionali
8	&	AND bitwise
9	^	XOR bitwise
10	|	OR bitwise
11	&&	AND logico
12	||	OR logico
13	? :	Condizionale
14	= *= /= %= += -= >>= <<= &= ^= |=	Assegnazione semplice o composta

Istruzioni di output

- Permettono di porre un valore dallo standard output
- In C++ si usa l'istruzione: **cout<<** x;
- Tale istruzione fa parte della libreria **iostream**, che va quindi inclusa nel programma con la direttiva #include
- **L'istruzione cout scrive il valore memorizzato nella variabile x sullo standard output (console mostrata a video)**

```
1.  #include <iostream>
2.  …
3.  using namespace std;
4.  ...
5.  {
6.   …
7.  char x ='c';
8.  cout << x << endl;
9.  …
10. }
```

Tramite l'operatore **shift a sinistra** si esegue la stampa su schermo del valore della variabile posta in ingresso.

Per spostarsi a linea nuova si usa la parola "endl", oppure si usa il carattere "\n".

Istruzioni di input

- Permettono di prelevare un valore dallo standard input e di assegnarlo ad una variabile
- In C++ si usa l'istruzione : **cin>>** x;
- Tale istruzione fa parte della libreria **iostream**, che va quindi inclusa nel programma con la direttiva #include
- **L'istruzione cin legge il primo valore in coda dallo standard input (tastiera) e lo assegna alla variabile x**

```
1. #include <iostream>
2. …
3. using namespace std;
4. ...
5. {
6.  …
7. int x;
8. cin >> x;
9. …
10. }
```

Nota: la variabile in cui viene salvato lo standard input deve essere prima dichiarata.

A seconda del tipo della variabile, il valore letto sarà interpretato di conseguenza (intero, double, carattere, …).

STRUTTURE DI CONTROLLO

Strutture di controllo

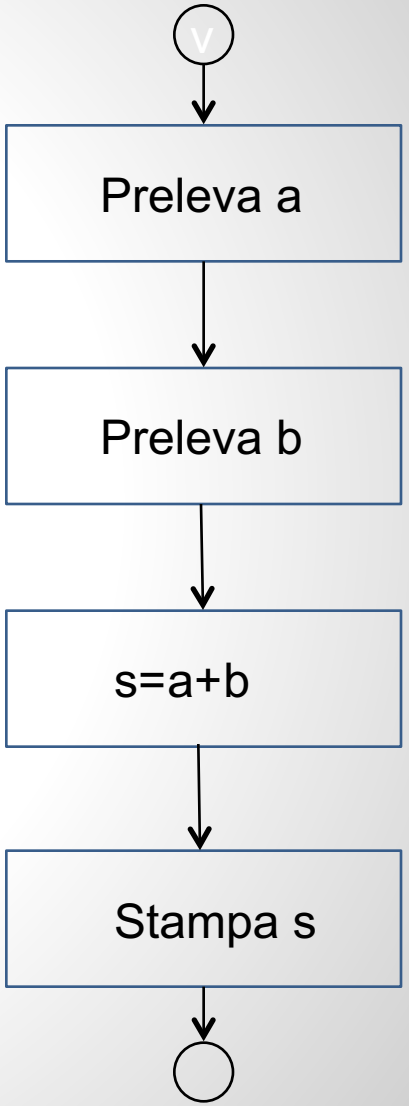
- Per **paradigma di programmazione** si intende il modello adottato da un linguaggio di programmazione per definire il concetto di programma e programmazione.
- Nel **paradigma imperativo** i programmi sono intesi come sequenze di istruzioni di trasformazione di dati che vengono impartite sequenzialmente ad un esecutore.
- Per sviluppare un programma in tale paradigma è necessario:
  - definire i dati che saranno oggetto delle trasformazioni.
  - definire il flusso trasformazionale su di esse (**flusso di controllo**).

  programma = dati + flusso di controllo (istruzioni)
- I linguaggi di alto livello offrono **costrutti per definire il flusso di controllo**, ovvero l'ordine delle istruzioni da eseguire in esecuzione:
  - Strutture di sequenza
  - Strutture di selezione
  - Strutture di iterazioni

Blocco sequenziale

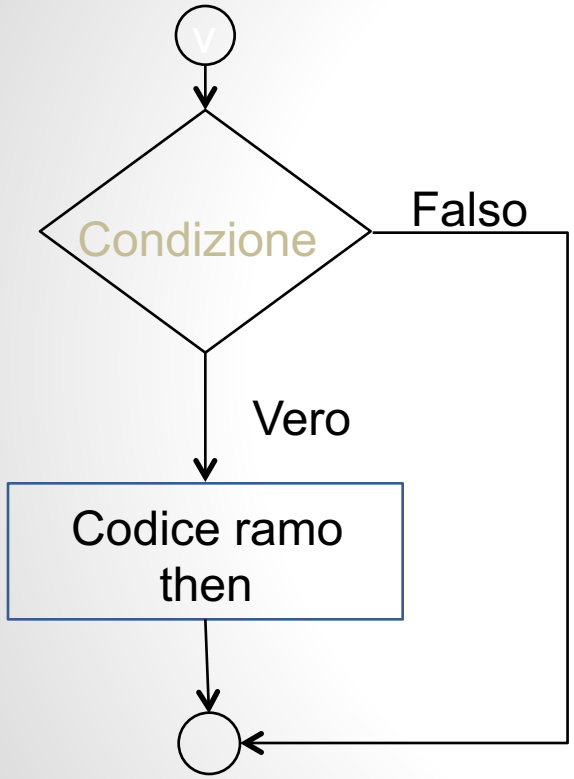
- Indica un insieme di istruzioni da eseguire sequenzialmente.
- Ogni istruzione può semplice o complesse (es.: costituita da altri blocchi).
- E' delimitata dalle parentesi graffe { }

```
{
int a;
int b;
int s;
cin>>a;
cin>>b;
s=a+b;
cout<<s;
}
```



Istruzione di selezione selettivo: if

Permette di decidere quale azione eseguire tra varie alternative



Può essere un predicato logico (secondo l'algebra dei predicati)

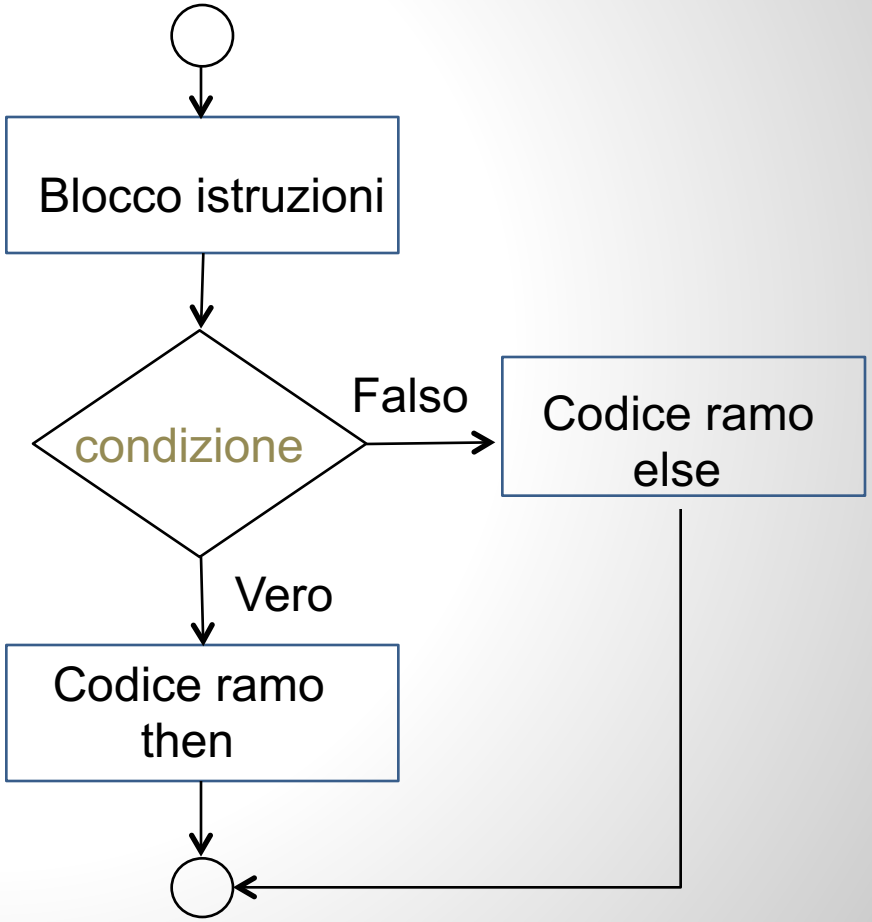
```
1. if (condizione) {
2.    codice ramo then;
3. }
```

➢Esempio:
(**if**) Se il voto dello studente è maggiore o uguale a 18 **(then)** allora stampa "Promosso"

Istruzione di selezione if-then-else

Fornisce un'azione alternativa se la condizione dell'if non è vera

```
1. Blocco istruzioni;
2. if (condizione) {
3.    codice ramo then;
4. } else{
5.    codice ramo else;
6. }
```

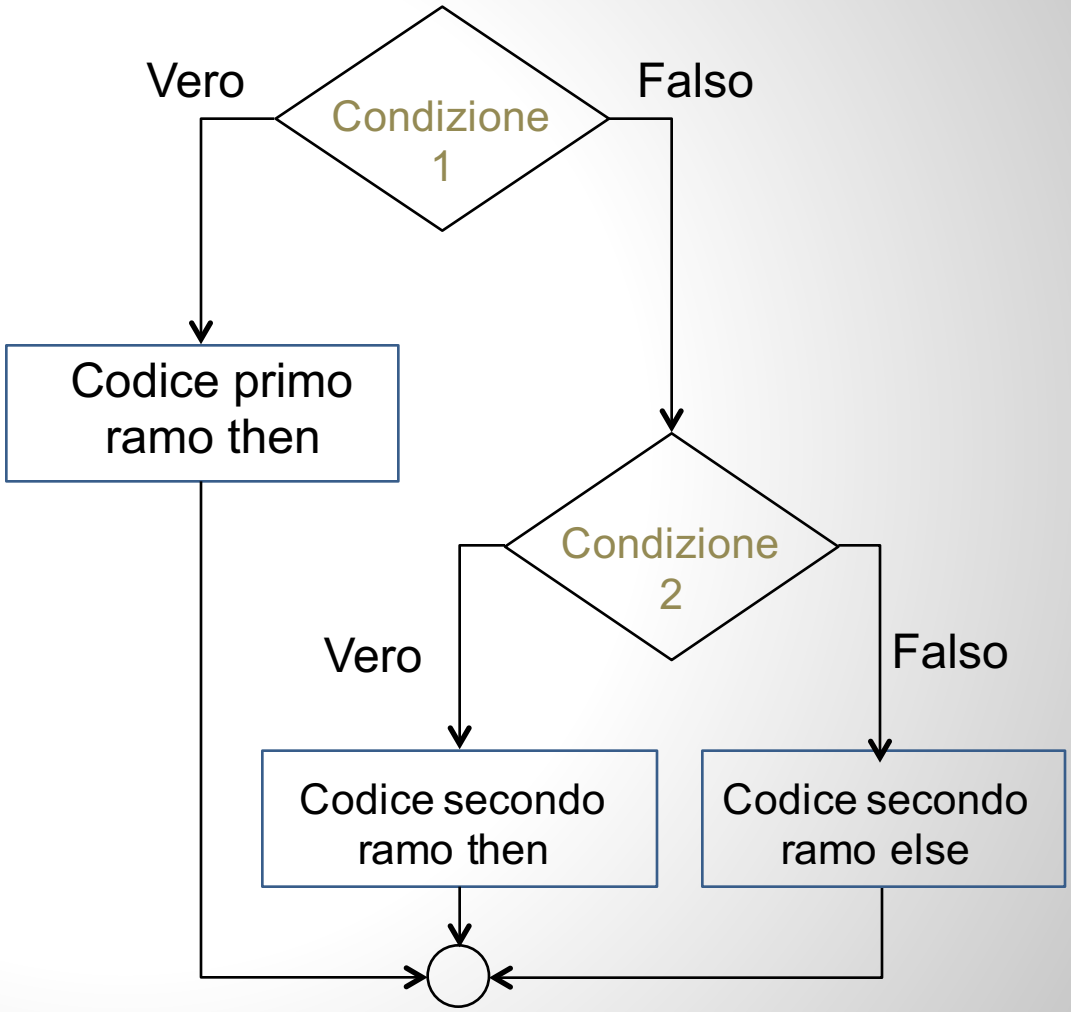


➢Esempio : (**if**)Se il raggio del cerchio è negativo **(then)** dà un messaggio di errore **(else)** calcola l'area

Nesting degli if-then-else

- La parte **then** e la parte **else** possono a loro volta contenere delle istruzioni di selezione
- Una volta che una condizione risulta vera, le rimanenti sono saltate

```
1. if(condizione 1) {
2.   codice primo ramo then;
3. }
4. else if (condizione 2){
5.   codice secondo ramo then;
6. }else {
7.   codice secondo ramo else;
8. }
```



Strutture di controllo iterative

- Con una struttura di controllo di tipo iterativo si determina la ripetizione dell'esecuzione di un blocco di istruzioni:
  - Per un numero di volte determinato dalla verità (falsità) di un predicato (while/until).
  - Per un numero di volte determinato dalla verità (falsità) di un predicato, ma almeno uno (do-while/do-util).
  - Per un numero di volte predeterminato (strutture a conteggio, costrutti for).

Il ciclo while

Il ciclo **while** impone che l'esecuzione del blocco di istruzioni sia ripetuta fino a quando la condizione non diventa FALSE.

```
while (condizione)
{ S;
}
```

Inizialmente viene valutata la condizione:

- se è FALSE, la sequenza S non viene eseguita;
- Se è TRUE, si esegue S e al suo termine si ricalcola la condizione e si riesegue S se la condizione è ancora VERA
- **Assicurarsi che ci sia una condizione di terminazione!**

Il ciclo do… while

Con il ***do while*** la condizione viene scritta dopo la sequenza S per ottenere che l'esecuzione del blocco avvenga almeno una volta.

```
do
    { S;
    }
    while (condizione)
```

Inizialmente viene valutata la condizione:

- se è FALSE, la sequenza S non viene eseguita;
- Se è TRUE, si esegue S e al suo termine si ricalcola la condizione e si riesegue S se la condizione è ancora VERA
- **Assicurarsi che ci sia una condizione di terminazione!**

Il ciclo for

Il ciclo ***for*** opera come il while con una iterazione della sequenza che continua quando la condizione espressa risulta vera: appena la condizione è falsa la iterazione termina.

```
for (inizializzazioni; condizione; variazioni)
                        { S;
                        }
```

Il ciclo for prescrive:

1. l'esecuzione delle istruzioni indicate come *inizializzazione* (non obbligatorie);
2. il calcolo della *condizione*;
3. la esecuzione della sequenza S se risulta verificata la condizione (assume valore TRUE); in caso contrario l'iterazione termina;
4. l'esecuzione delle istruzioni di *variazione* (non obbligatorie) al termine di S;
5. la rivalutazione della condizione con il ripetersi dei passi precedenti fino alla determinazione della falsità della condizione.

ESEMPI DI RIEPILOGO

Esempio di blocco sequenziale

- Tutte le istruzioni delimitate da un blocco sequenziale (tra parentesi graffe) sono eseguite una dopo l'altra in maniera imperativa:

C++

```
{
   int n = 4;
   cout << "Blocco sequenziale!" << endl;
   cout << "Istruzioni in sequenza" << endl;
   n = n + n;
   cout << "n = " << n << endl;
}
```

Output:

```
Blocco sequenziale!
Istruzioni in sequenza
n = 8
```

Esempio di blocco di selezione

- Il blocco if esegue le istruzioni nel blocco solo per il valore di verità del predicato:

C++

```
{
    int n;
    cout << “Inserisci numero intero: ”;
    cin >> n;

    if (n < 0) {
        cout << “Inserito numero negativo!”;
    }
}
```

Output:

```
Inserisci numero intero: -30
Inserito numero negativo.
```

Esempio di blocco di selezione (1/3)

- Il blocco else è eseguito se non **nessun predicato** del blocco di selezione è vero:

Se l'utente non inserisce 'y', allora…

C++

```
{
    char rispostaUtente;
    cout << "Vuoi proseguire? (y/n): ";
    cin >> rispostaUtente;

    if (rispostaUtente == 'y') {
        cout << "L'utente ha inserito y";
    } else {
        cout << "L'utente non ha inserito y";
    }
}
```

Output:

```
Vuoi proseguire? (y/n): k
L'utente non ha inserito y
```

Esempio di blocco di selezione (2/3)

- È possibile comporre più predicati uno dopo l'altro aggiungendo "else if": in questo caso verrà eseguito **solo il primo** predicato che risulta vero.

Se inseriamo -30 SOLO il primo ramo if è percorso, perché il suo predicato è vero.

(la valutazione è sequenziale!)

C++

```
{
    int n;
    cout << "Inserisci numero intero: ";
    cin >> n;

    if (n < 0) {
        cout << "Inserito numero negativo!";
    } else if (n <= 0) {
        cout << "Inserito zero!";
    }
}
```

Output:

```
Inserisci numero intero: -30
Inserito numero negativo.
```

Esempio di blocco di selezione (3/3)

- Il ramo **else** è opzionale è viene seguito **solo se** nessun predicato del costrutto if-else if risulta vero (se nessun altro ramo viene eseguito).

C++

```
{
    int n;
    cout << “Inserisci numero intero: ”;

    if (n < 0) {
        cout << “Inserito numero negativo!”;
    } else if (n == 0) {
        cout << “Inserito zero!”;
    } else {
        cout << “Inserito numero positivo!”;
    }
}
```

Il ramo **else** è viene eseguito se nessun altro ramo è VERO

Output:

```
Inserisci numero intero: -30
Inserito numero negativo.
```

Attenzione: valutazione sequenziale

- Le condizioni dei rami **if** sono valutate **sequenzialmente**: se un predicato è vero, il ramo viene percorso e si l'esecuzione prosegue dalla prima istruzione dopo il costrutto di selezione.

C++

```
int n = 0;
if (n < 0) {
    cout << "Inserito numero negativo!\n";
} else if (n == 0) {
    cout << "Inserito zero!";
} else if (n <= 100) {
    cout << "Numero tra 1 e 100 incluso\n";
} else {
    cout << "Numero maggiore di 100\n";
}
cout << "Costrutto if terminato\n";
```

Output:

```
Inserito zero!
Costrutto if terminato
```

Se n = 0,
il ramo di confronto con lo zero è percorso.
L'esecuzione prosegue **dopo** il blocco di selezione e **nessun altro predicato è valutato.**

Blocco while (1/2)

- Il ciclo while è eseguito fin quando il suo predicato risulta Vero.

C++

```
cout << “Numeri tra 0 e 10: ”;

int n = 0;
while (n <= 10) {
    cout << n << “ “;
    n = n + 1;
}
```

Output:

```
Numeri tra 0 e 10: 0 1 2 3 4 5 6 7 8 9 10
```

Blocchi while e variabili di conteggio

- I blocchi while sono in generale da preferire quando non è esplicita alcuna **"variabile di conteggio"**.

C++

```
int n = 55;
while (n > 1) {
    cout << n << endl;
    n = n / 2;
}
```

In generale, quando non so dire a priori il numero di iterazioni…

Output:

```
55
27
13
6
3
```

Blocco for a passo unitario

- Nel caso esista una variabile di conteggio, o se il numero di volte da eseguire un blocco è “predeterminato”, il ciclo for risulta più naturale da essere impiegato.

C++

```
cout << “Numeri tra 0 e 10: ”;

for (int n = 0; n <= 10; n++) {
    cout << n << “ “;
}
```

Output:

```
Numeri tra 0 e 10: 0 1 2 3 4 5 6 7 8 9 10
```

Blocco for a passo non unitario

- È possibile anche modificare il costrutto while per incrementare non a passi unitari, modificando la sezione di variazione.

C++

```
cout << “Numeri tra 0 e 10 passo di 3: ”;

for (int n = 0; n <= 10; n = n + 3) {
    cout << n << “ “;
}
```

Output:

```
Numeri tra 0 e 10 passo di 3: 0 3 6 9
```

Predicati con input utente

- Anche nel for i predicati delle iterazioni possono dipendere da variabili, piuttosto che da valori costanti.

C++

```
int n_max;
cout << “Numeri tra 0 e “;
cin >> n_max;

for (int n = 0; n <= n_max; n++) {
    cout << n << “ “;
}
```

n_max è una variabile il cui valore è inserito inserito dall’utente

Output:

```
Numeri tra 0 e 7
0 1 2 3 4 5 6 7
```

Osservazioni sul costrutto for

- Per eseguire un blocco esattamente N volte i costrutti tipici sono:

  for (int **i = 0; i < N**; i++) { … }

  for (int **i = 1; i <= N**; i++) { … }

Prestare attenzioni ai limiti!
Errori **"off-by-one"**
sono tipici!

- In C++ il blocco un blocco for è sempre equivalente ad un blocco while scritto nel seguente modo:

```
for (X; Y; Z) {
        S;
}
```

```
{
    X;
    while (Y) {
        { S; }
        { Z; }
    }
}
```

Blocco do-while

- Il blocco do while eseguie il ciclo almeno una volta, poi valuta il predicato per decidere se ripetere l'esecuzione del blocco:

C++

```
char risposta;
do {
    cout << "Vuoi proseguire? (y/n): ";
    cin >> risposta;
} while (risposta != 'y' && risposta != 'n');
cout << "Risposta = " << risposta << endl;
```

Output:

```
Vuoi proseguire? (y/n): q
Vuoi proseguire? (y/n): k
Vuoi proseguire? (y/n): z
Vuoi proseguire? (y/n): n
Risposta = n
```

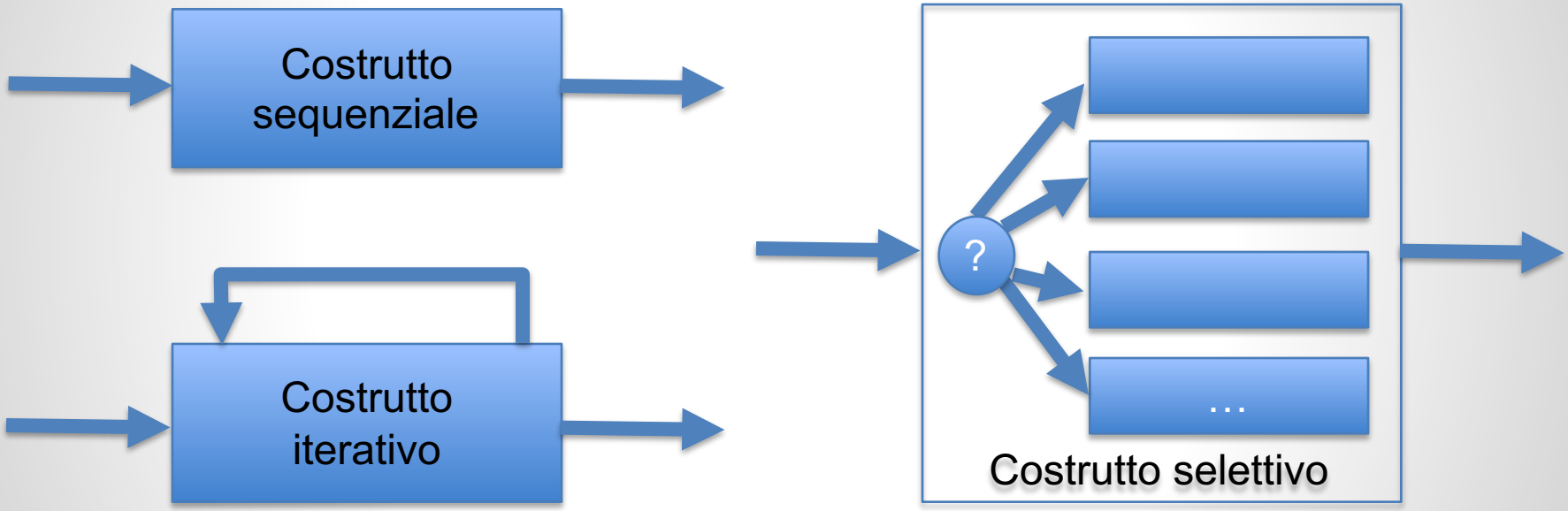
ASPETTI AVANZATI

Istruzioni non strutturate

- Esistono **keyword** del linguaggio che permettono di introdurre **eccezioni** nello svolgimento di un flusso di controllo stutturato:
  - **goto** <label>
    - permette di spostare l'esecuzione arbitrariamente al punto del codice sorgente che è stato "etichettato" con <label>.
  - **break**
    - interrompe arbitrariamente l'esecuzione del costrutto iterativo nel quale è inserito.
  - **continue**
    - salta arbitariamente le seguenti istruzioni nel costrutto iterativo nel quale è inserito, senza però interrompere il ciclo.

La programmazione strutturata

- I costrutti di controllo permettono di realizzare programmi comprensibili, che gestiscono in maniera disciplinata il flusso trasformazionale dei dati.
  - Ogni costrutto segue una logica di controllo one-in/one-out.



- È sempre possibile scrivere un algoritmo che rispetti i principi della programmazione strutturata!
  - Le istruzioni goto, break e continue diminuiscono la qualità del software, perciò si limita ad usarli per casi eccezionali.

Switch-case

```
switch(<espressione intera>)
{
    case (<valore costante 1>):
        <sequenza di istruzioni 1>
        break;

    case (<valore costante 2>)
        <sequenza di istruzioni 2>
        break;

    ...

    case (<valore costante N>)
        <sequenza di istruzioni N>
        break;

    default:
        <sequenza di istruzioni N+1>
}
```

- Il **costrutto switch-case** serve a eseguire codice diverso a fronte di valori diversi assunti da una espressione.
  - Ad esempio risulta utile quando dobbiamo impostare il comportamento del programma in seguito alla scelta di un'opzione da parte di un utente.
- Il costrutto valuta la condizione intera passata a ***switch*** e rimanda lo svolgimento del programma al blocco in cui il parametro di ***case*** ha lo stesso valore di quello dell'istruzione ***switch***
- Se il blocco termina con ***break;*** allora il programma esce dallo switch, altrimenti vengono eseguiti anche i blocchi seguenti fino ad un break; o fino alla fine
- Se nessun blocco corrisponde ad un valore uguale a quello dell'istruzione switch viene eseguito il blocco in ***default:***