

# I linguaggi di alto livello: il concetto di astrazione

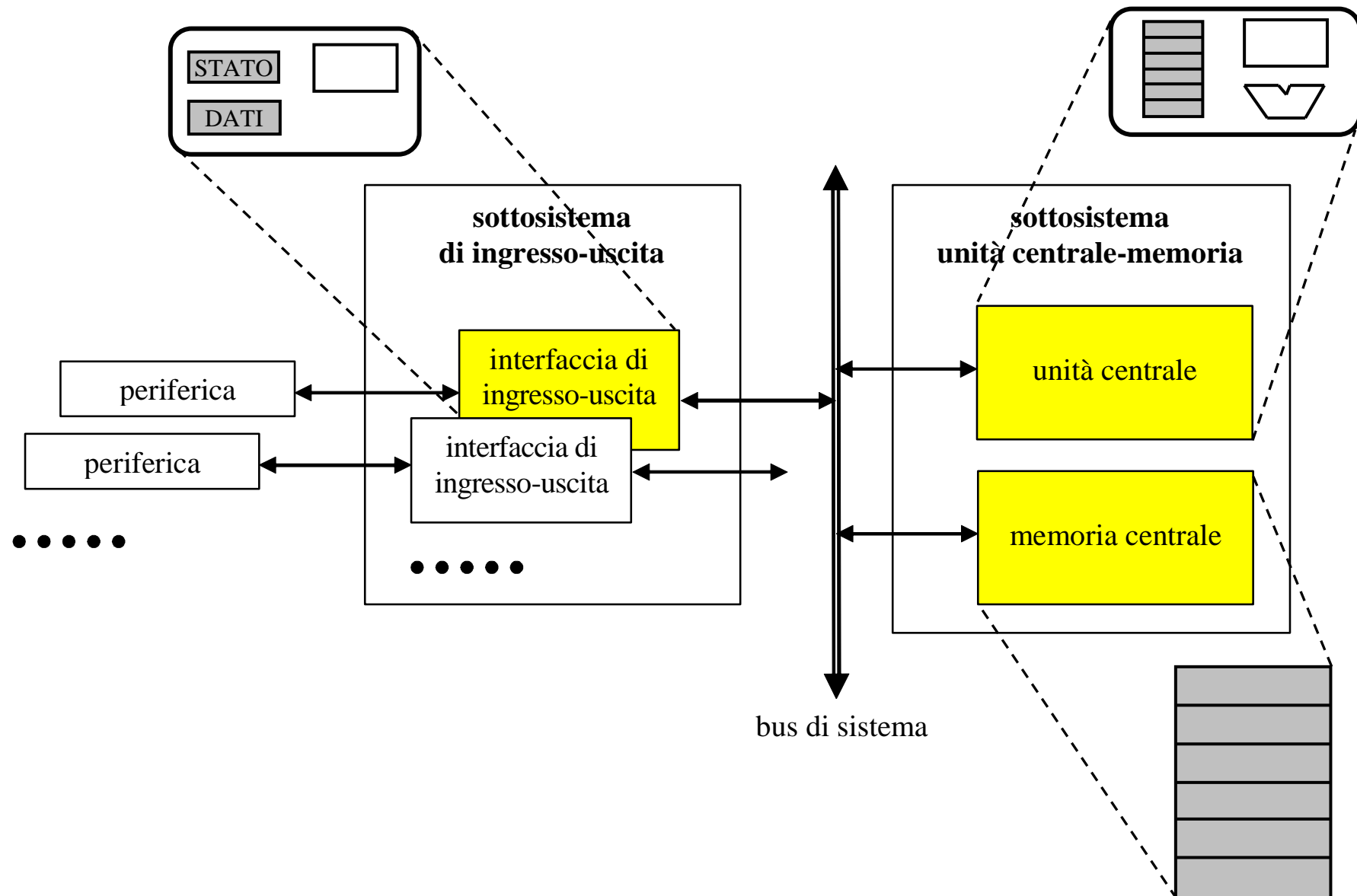
## Fondamenti di Informatica A

*Percorso di Preparazione agli Studi di Ingegneria*

Università degli Studi di Brescia

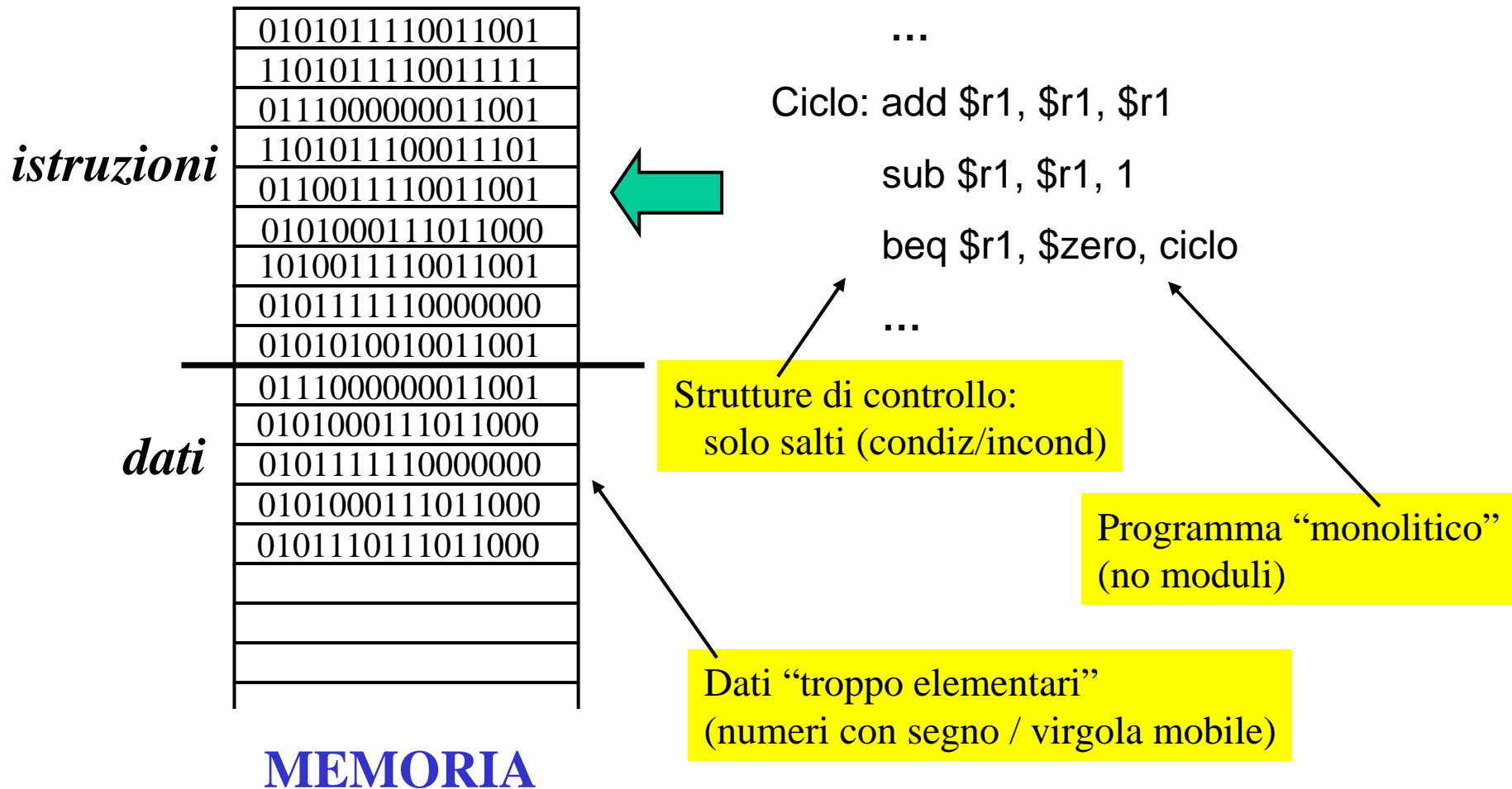
*Docente: Massimiliano Giacomini*

# Il punto della situazione...



# Limiti del linguaggio assembler

**Esempio**: gestione e ordinamento di una rubrica telefonica:  
sarebbe poco praticabile ricorrere al linguaggio assembler



# Linguaggi di programmaz. di alto livello

- Permettono di specificare gli algoritmi con un *insieme ridotto di termini linguistici* (di significato *preciso e univoco*)
- Costrutti specifici per realizzare meccanismi di astrazione di più alto livello rispetto ad assembler. Ciascuna *istruzione* corrisponde a molte istruzioni macchina. Vantaggi:
  - funzionalità (riduzione errori commessi)
  - manutenibilità
  - efficienza (ottimizzazione nella traduzione)
- Sviluppo storico: da specializzazione a orientamento alla metodologia
- Ruolo e dimensioni dell'astrazione:
  - Astrazione sui dati (mediante definizione di tipi di dati)
  - Astrazione sul controllo (mediante strutture di controllo)
  - Astrazione procedurale (mediante definizione di moduli)

# Paradigmi di programmazione

- L'insieme delle caratteristiche che determina il modo di affrontare il progetto e la codifica di un programma, lo “stile di programmazione”

	-	Ad oggetti
Imperativo	<i>Pascal, Cobol, C...</i>	<i>C++, Java, VisualBasic,...</i>
Funzionale	<i>Lisp, APL, Haskell</i>	<i>CLOS</i>
Logico	<i>Prolog</i>	<i>Prolog++</i>

# IL PARADIGMA IMPERATIVO

- Per illustrarne le caratteristiche principali, faremo riferimento alla sintassi del linguaggio C

## Programma

- Il *corpo del programma* è composto da una sequenza di istruzioni
  - Si distinguono 3 tipi di istruzioni:
    - Istruzioni di ingresso/uscita
    - Istruzioni aritmetico-logiche
    - Istruzioni di controllo
- } NB: le stesse del linguaggio macchina!

## Istruzioni di ingresso/uscita

Consentono acquisizione (ingresso) e presentazione (uscita) di dati

In linguaggio C: scanf e printf

## Istruzioni aritmetico-logiche (calcolo + assegnamento)

- Consentono la manipolazione dei dati e la generazione di risultati
- Ruolo centrale: istruzione di **assegnamento**
  - permette di assegnare un **valore** ad una **variabile**
- **Variabile**: permette di “specificare” un’area di memoria



- Assegnamento:  
$$\text{nome\_variabile} = \langle \text{valore} \rangle$$
- Esempio:  
$$\begin{aligned} a &= 5 \\ b &= 6 \\ a &= a + b \end{aligned}$$

# Astrazione sui dati: variabili e tipi

- Ogni variabile è caratterizzata da un *tipo* (oltre che da *nome* e *valore*):
  - il tipo identifica i valori che la variabile può assumere e le **operazioni** che su di essa possono essere compiute
- Ogni variabile viene **dichiarata prima** del suo utilizzo:
  - dichiarazione del nome
  - **dichiarazione del tipo** della variabile
- Esempio (in C):  
*int a;*

La creazione della variabile di tipo '*intero*' riserva un'area di memoria per contenere un valore intero positivo



La **dichiarazione** associa il nome '*a*' a tale area di memoria





## Tipi di dati predefiniti

- **Tipi predefiniti:** numeri (interi, reali, ...), caratteri, booleani → normalmente previsti in tutti i linguaggi di programmazione

- Esempi:

- tipo **int** (*intero*) in C permette di dichiarare variabili di tipo intero

- Su variabili di tipo intero saranno applicabili *operatori aritmetici* (+, \*, -, /, ...)

- Il tipo **bool** (*booleano*) permette di dichiarare variabili che possono assumere solo i valori ‘vero’ e ‘falso’ (true e false)

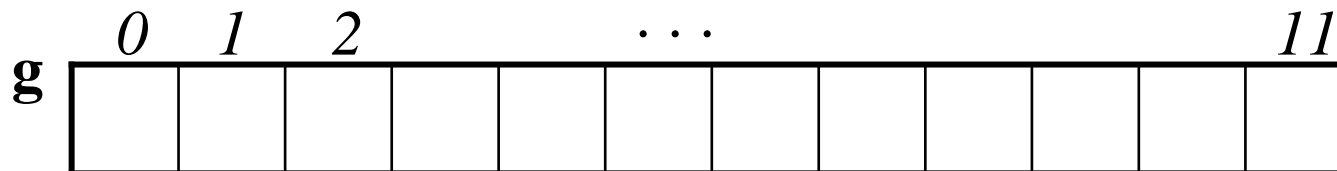
- Su variabili di tipo booleano possono essere applicate *operazioni logiche*, ad esempio AND (&& in C) e OR (|| in C)

## Tipi di dati definiti dal programmatore

- A partire dai dati predefiniti l’utente può definire altri tipi
- Vengono definiti mediante **costruttori** (permettono di aggregare insiemi di tipi già disponibili in modo articolato)
- Esempi: **array** e **record**

## VETTORI (array)

- Le variabili dichiarate di un tipo predefinito sono contenitori di singoli valori:  
*int x* ➔ significa che la variabile *x* è un contenitore di un valore intero
- E se volessimo memorizzare i guadagni di 12 mesi in 12 variabili? Possiamo...
  - a) ... dichiarare dodici variabili di tipo intero: *g1, g2, ..., g12*
  - b) ... oppure dichiarare un **VETTORE (ARRAY)** *g* di 12 posizioni
- Ogni vettore ha un *nome* e un *tipo* e può contenere un **numero stabilito *n* di elementi dello stesso tipo**, ogni elemento è identificato da un *indice* che varia fra 0 e *n*



## In C: somma dei guadagni di 12 mesi

```
main()    /* C */
{
    int g[12];
    int z;
    g[0]=900;
    g[1]=2300;
    ...

    z = g[0]+g[1]+g[2]+g[3]+ g[4]+g[5]+g[6]+g[7]+g[8]+g[9]+g[10]+g[11];
    printf ("%d", z);
}
```

indicizzazione

NB: il costruttore array non definisce alcuna operazione specifica sul tipo di dato, ad eccezione dell'indicizzazione – le operazioni possibili dipendono dal tipo di dato su cui è costruito l'array

# RECORD

- Nei vettori: elementi devono tutti dello stesso tipo
- Si supponga che per ogni elemento si vogliano però memorizzare dati di tipo diverso

Ad esempio: per ogni **studente** si vuole memorizzare *numero di matricola, nome, cognome, numero degli esami sostenuti*

- **Record**: struttura definita da:
  - insieme di insieme finito di elementi, detti *campi*
  - ogni campo:
    - > ha un *tipo specifico*
    - > è identificato da un *nome*
- L'unica operazione definita dal costruttore record è la *selezione*:  
identificazione di un elemento nella forma  
*nome-record.nome-campo*

## In C: esempio di definizione ed uso di un record

```
struct studente
{
    int    matricola;
    char   nome[30];
    char   cognome[30];
    int    num_esami;
}
```

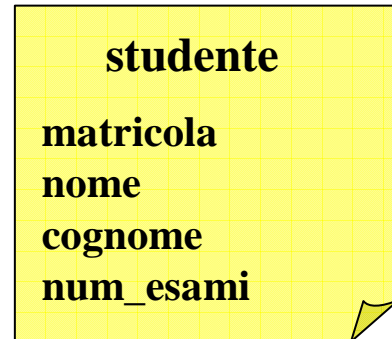
```
struct studente stud1;
struct studente stud2;
```

```
stud1.matricola = 25891;
```

```
...
```

```
stud1.num_esami = stud2.num_esami + 1;
```

*Definizione  
del tipo  
'studente'*



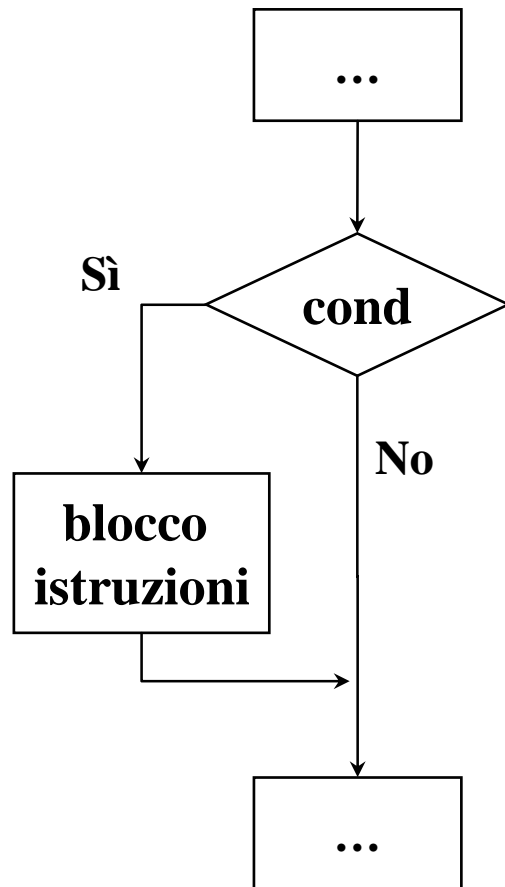
**Dichiarazione delle  
variabili stud1 e stud2**

**Assegnamenti...**

# Astrazione sul controllo

- Le istruzioni di controllo consentono di **modificare il flusso di esecuzione** delle istruzioni all'interno di un programma
- L'**esecuzione sequenziale** viene alterata attraverso le istruzioni di controllo che consentono di realizzare due tipi di strutture di controllo:
  - **condizionali** (*semplici o a due vie*)
  - **cicli** (*a condizione iniziale e a condizione finale*)

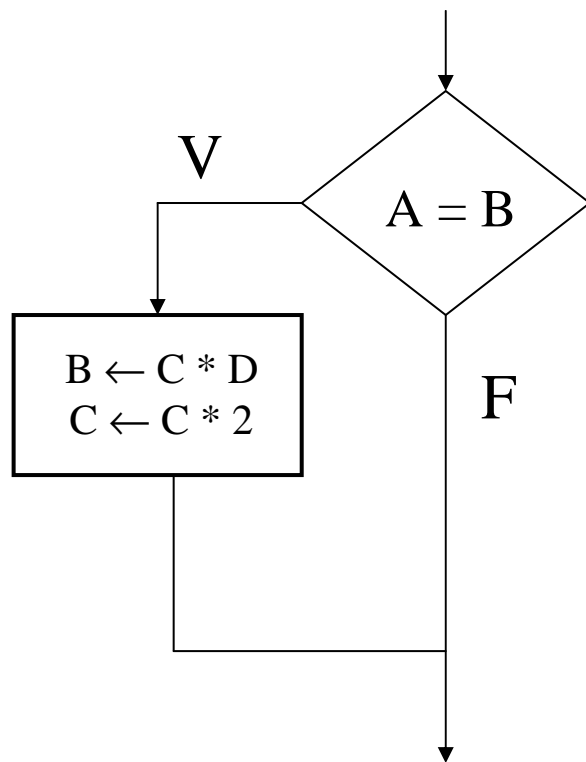
# CONDIZIONALE SEMPLICE



**main()**

```
{ ...  
    /* condizionale semplice */  
    if (cond)  
    { ...  
        /*blocco istruzioni */  
        /* eseguito solo se */  
        /* cond è true */  
        ...  
    }  
    ...  
}
```

## ESEMPIO: C e ASSEMBLER



```
main() /* C */  
{  
    int A, B, C, D;  
    ...  
    if (A == B)  
    {  
        B = C * D;  
        C = C * 2;  
    }  
    ...  
}
```

// A, B, C, D costanti...

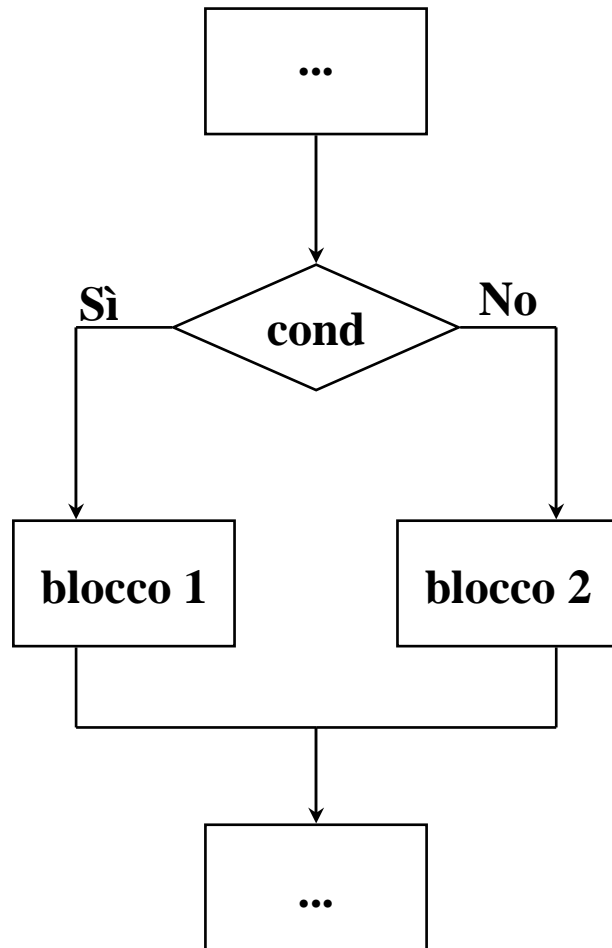
```
lw $r1, $r0, A  
lw $r2, $r0, B  
lw $r3, $r0, C  
lw $r4, $r0, D  
...  
bne $r1, $r2, CONT  
mul $r2, $r3, $r4  
sw $r2, $r0, B  
add $r3, $r3, $r3  
sw $r3, $r0, C
```

CONT:

...



## CONDIZIONALE A DUE VIE



```
main()  /* C */
```

```
{ ...
```

```
/* condizionale a 2 vie */
```

```
if (cond)
```

```
    { ... /* blocco 1 */ }
```

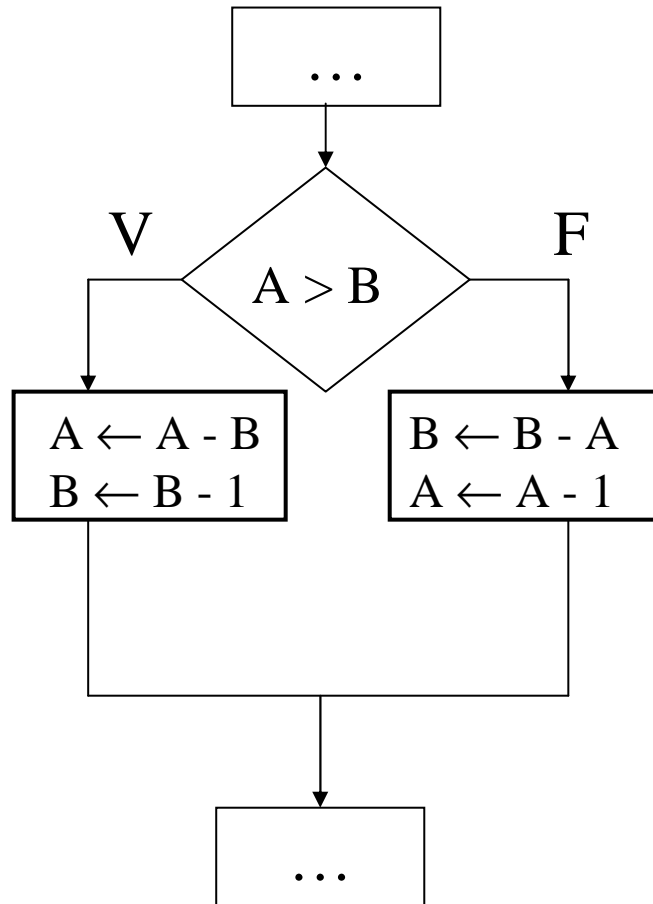
```
else
```

```
    { ... /* blocco 2 */ }
```

```
...
```

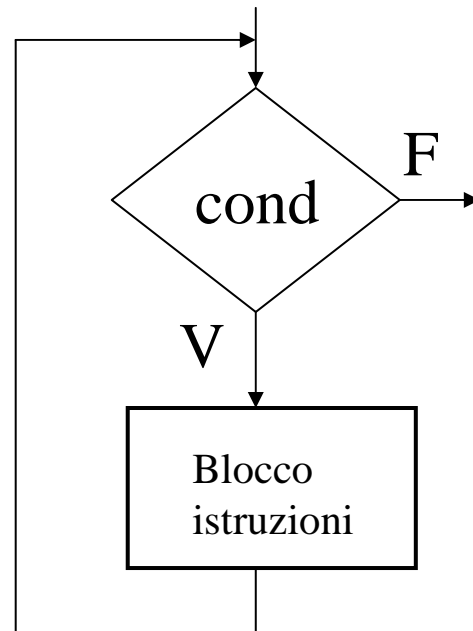
```
}
```

# ESEMPIO



```
main() /* C */  
{  
  int A, B;  
  ...  
  if (A > B)  
  {  
    A = A - B;  
    B = B - 1;  
  }  
  else  
  {  
    B = B - A;  
    A = A - 1;  
  }  
  ...  
}
```

# CICLO A CONDIZIONE INIZIALE



```
main() /* C */  
{ ...  
  /* ciclo a condizione  
    iniziale */  
  
  while (cond)  
  {  
    /* blocco istruzioni  
      eseguito quando cond  
      è vero */  
  }  
  ... /* eseguito quando  
    cond è falso */  
}
```

## ESEMPIO PRECEDENTE: SOMMA DEI GUADAGNI DI UN ANNO

### Dati

g[12] vettore di interi

w, z interi positivi

### Risoluzione

w  $\leftarrow$  1

z  $\leftarrow$  0

finchè (w  $\leq$  12) ripeti

    z  $\leftarrow$  z + g[w]

    w  $\leftarrow$  w + 1

fine ciclo

scrivi z

```
main()  /* C */
```

```
{
```

```
    int g[12];
```

```
    int w, z;
```

```
    w = 0;
```

```
    z = 0;
```

```
    while (w < 12)
```

```
    { z = z + g[w];
```

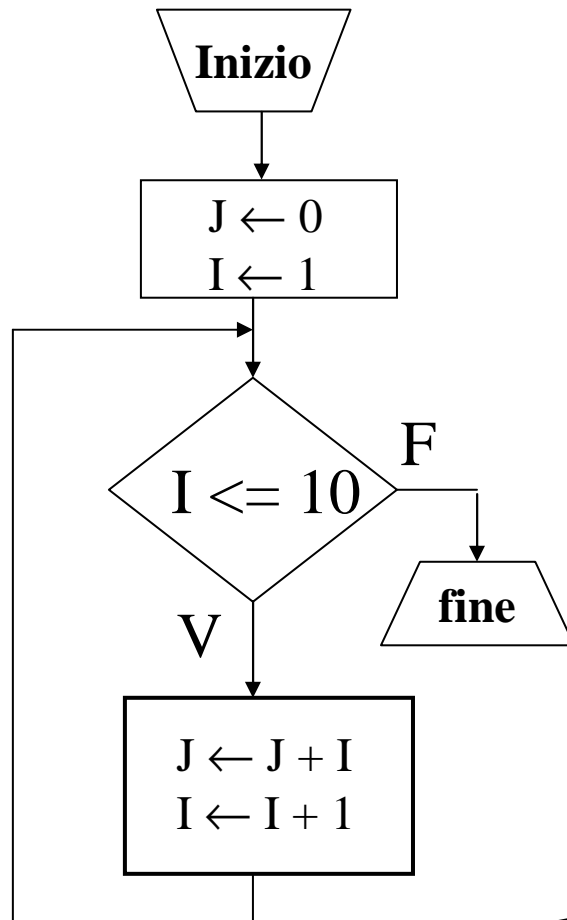
```
      w = w + 1;
```

```
    }
```

```
    printf ("%d", z);
```

```
}
```

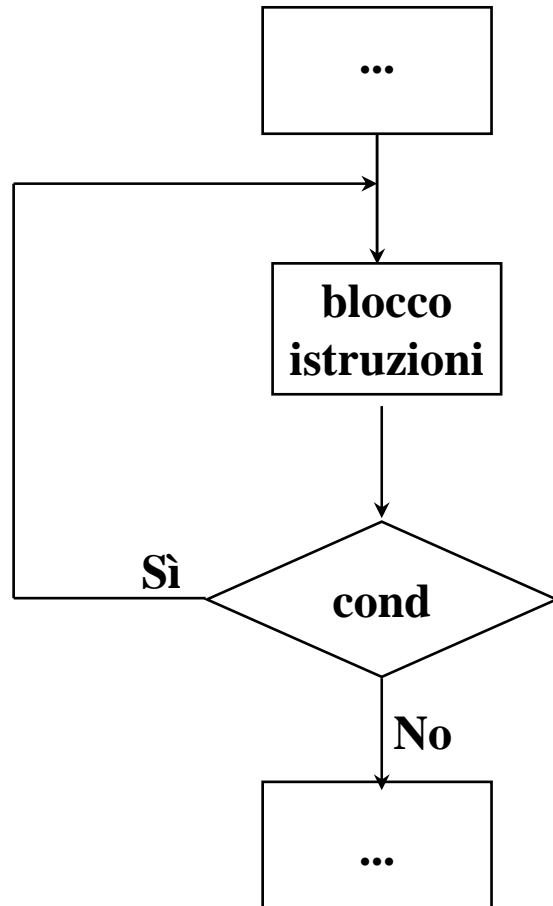
# ESEMPIO



```
main() /* C */
{
  int I, J;
  J = 0;
  I = 1;
  while (I <= 10)
  {
    J = J + I;
    I = I + 1;
  }
}
```

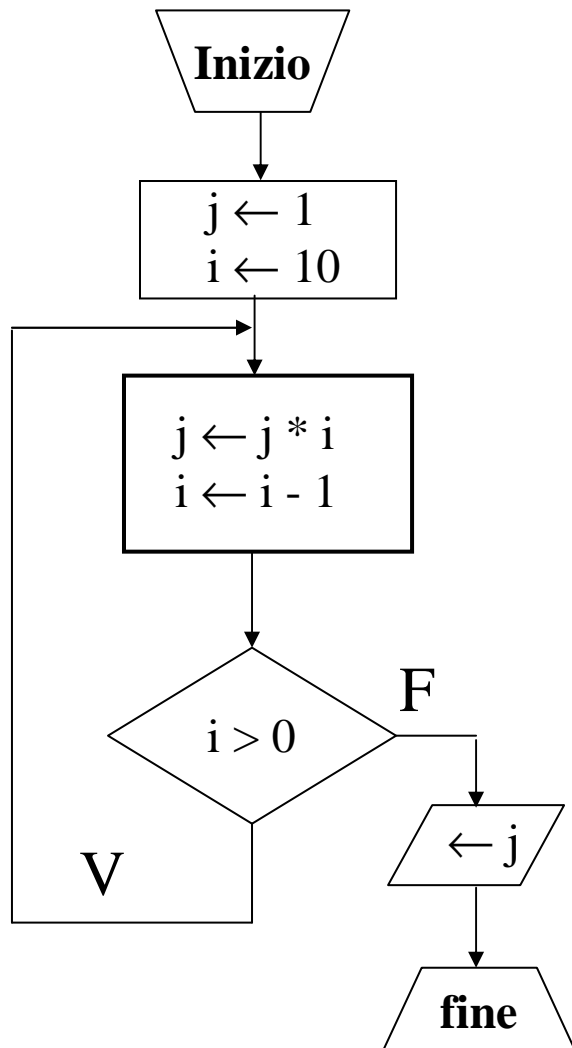
**Esercizio:** cosa fa il programma? Quali sono i valori finali di I e J?

# CICLO A CONDIZIONE FINALE



```
main() /* C */
{...
/* ciclo a condizione */
/* finale */
do
{
/* blocco istruzioni
   eseguito almeno
   una volta e finchè
   cond rimane vera */
} while (cond)
... /* eseguito quando cond
   è falsa */
}
```

# ESEMPIO



```
main() /* C */
{
    int i, j;
    j = 1; i = 10;
    do
    {
        j = j * i;
        i = i - 1;
    } while (i > 0)
    printf("%d", j);
}
```

**Esercizio:** cosa calcola questo programma?

# Astrazione procedurale

- Soluzione di problemi complessi (eventualmente lavoro di più persone in modo coordinato): è utile scomporre un programma in moduli più semplici, detti *procedure*
- Facilitano leggibilità, funzionalità (controllo correttezza), manutenzione del programma
- Definizione di una procedura (*nome + corpo istruzioni*):

*nome\_procedura(<parametri formali>)*  
*{ istruzioni... }*

- Parametri formali: sono gli argomenti della procedura (passati dalla procedura o dal programma chiamante):  
variabili utilizzate dalla procedura al suo interno



- *Chiamata* di una procedura nel programma:  
*nome\_procedura (<parametri attuali>)*
- Parametri attuali: variabili o espressioni il cui valore è attribuito ai parametri formali (**passaggio dei parametri**)
- I tipi dei parametri attuali devono corrispondere ai parametri formali
- Effetto della chiamata:
  - passaggio dei parametri
  - controllo passato alla procedura, le cui istruzioni vengono eseguite (poi il controllo ritorna all'istruzione successiva alla chiamata)

NB: *è come se il linguaggio si arricchisse di nuove istruzioni...*  
ogni nuova istruzione è la chiamata di una procedura e quindi corrisponde all'esecuzione di un sottoprogramma

## ESEMPIO (in C)

```
void stampafattoriale(int w)
```

```
{ int fatt;
```

```
  fatt=1;
```

```
  while (w>0)
```

```
  { fatt = fatt * w;
```

```
    w = w - 1;
```

```
  }
```

```
  printf ("%d", fatt);
```

```
}
```

---

```
main()    /* C */
```

```
{ int numero;
```

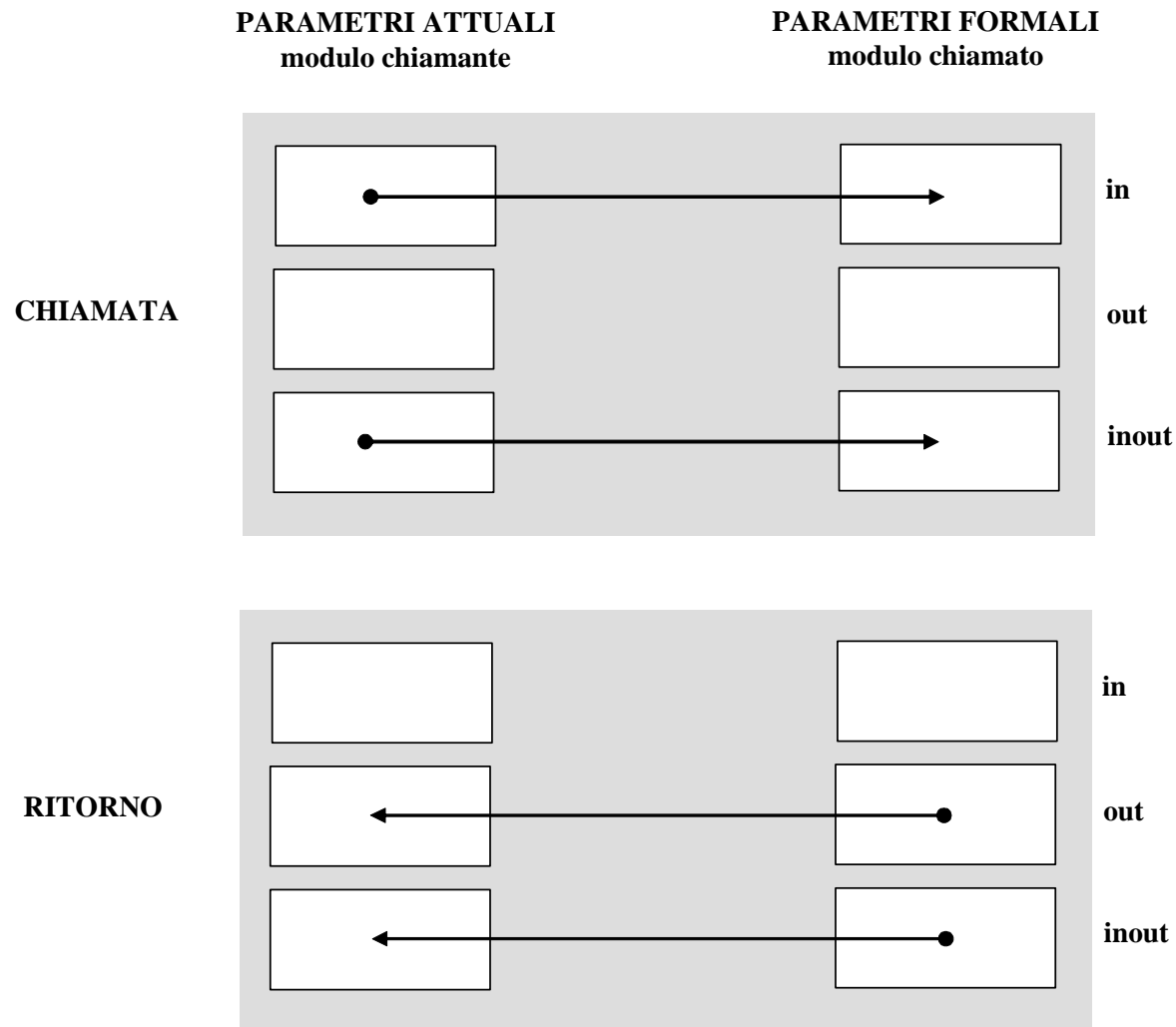
```
  ....
```

```
  numero = 5;
```

```
  stampafattoriale(numero)
```

```
}
```

# IL PASSAGGIO DEI PARAMETRI (non in C!)



NB: *in*, *out* e *inout* sono definite per ciascun parametro formale

- Alcuni linguaggi prevedono invece queste due modalità:
  - **per valore**: il corrispondente parametro attuale non è modificato
  - **per indirizzo**: la zona di memoria del parametro formale non contiene il valore, ma l'*indirizzo* del parametro attuale; tutte le operazioni sul parametro formale sono in realtà effettuate sul parametro attuale

### Esempio (in C++)

```
void fattoriale(int& fatt, int w)
{ fatt=1;
  while (w>0)
  { fatt = fatt * w;
    w = w - 1;
  }
}
int fatt1;
fattoriale (fatt1, 5);
```

C'è differenza?

```
void incrementacoppia(int& n, int& m)
```

```
{ n = n+1;  
  m = m+1;  
}
```

VS

```
void incrementacoppia(int inout n, int inout m)
```

```
{ n = n+1;  
  m = m+1;  
}
```

Si consideri la chiamata *incrementacoppia(x, x)*...

NB

- in C le cose sono leggermente diverse: non si parla di “procedure” ma di “funzioni” che ritornano un valore
- il passaggio è solo per valore (non è consentito il passaggio per indirizzo, anche se è possibile realizzarlo utilizzando i puntatori)

BUON LAVORO... IN LABORATORIO!...