Università degli studi di Salerno

**Dipartimento dell'Ingegneria dell'Informazione ed Elettronica e Matematica applicata**

Corso di Laurea in *Ingegneria Informatica* - L-8

# Fondamenti di Programmazione

Appunti delle lezioni di *Gennaro Francesco Landi*

I Sessione di Laurea *A.A. 2022-2023*

*Offrimi un caffè al bar.*
– ***Gennaro Landi***

# Indice

# Capitolo 1

# Concetti di base dell'informatica

## 1.1 Premessa

**CURIOSITA'**
Il termine informatica è stato introdotto da Philippe Dreyfus nel 1962; l'etimologia deriva dal termine francese informatique, come fusione di inform(ation electronique ou autom)atique. Il termine voleva rappresentare di conseguenza il trattamento dell'informazione mediante uno strumento automatico (il calcolatore). Tutti coloro che ritengono l'informatica come una scienza che parla inglese rimarranno stupiti dal fatto che in tale lingua non esiste l'equivalente di informatica; viene invece usato un termine molto più generico, 'computer science', che letteralmente si traduce in 'scienza degli elaboratori'.
Come il termine stesso suggerisce l'informatica è la convergenza di due aspetti distinti: l'informazione e la sua elaborazione automatica, ovvero mediante strumenti automatici.

## 1.2 Algoritmo, programma, esecutore

**Che cos'è un algoritmo?** Una possibile definizione è:

*Un **procedimento sistematico**, costituito da una **sequenza finita** di **operazioni**, ognuna delle quali è **precisa** ed **eseguibile**, da applicare ai **dati in ingresso** perché possa fornire dei **dati in uscita**.*

Per **procedimento sistematico** si indica un'attività elaborativa che progredisce seguendo fedelmente dei criteri prefissati. In particolare, in ogni momento dell'esecuzione è chiaro quale sia la prossima operazione da eseguire, così come è perfettamente definita la decisione circa il momento in cui il procedimento si debba ritenere concluso. Il procedimento è descritto tramite una **sequenza finita**, cioè una lista limitata e ordinata di operazioni, da eseguire nell'ordine in cui queste sono presenti, a meno che l'operazione corrente non specifichi diversamente. Per **operazione precisa** si intende che ognuna delle operazioni è definita in maniera chiara e certa, tale che non sia ambigua l'individuazione delle azioni da compiere. Inoltre, ogni operazione è **eseguibile**, cioè tale da poter essere portata a termine dall'esecutore dell'algoritmo. Al momento dell'esecuzione dell'algoritmo viene fornito un insieme di **dati in ingresso** sui quali sono applicate le operazioni specificate, l'esecuzione delle quali produrrà come risultato i **dati in uscita**. Al variare dei dati in ingresso, l'esecuzione dell'algoritmo potrà richiedere l'esecuzione di un numero di operazioni diverse per arrivare al termine e potrà produrre dati in uscita diversi. Ovviamente, se due esecuzioni di un algoritmo hanno lo stesso insieme di dati in ingresso, i risultati delle due esecuzioni saranno identici.

### 1.2.1 Algoritmo ed esecutore

Quando si definisce un algoritmo, si assume implicitamente l'esistenza di un *esecutore*, cioè un agente (umano o artificiale) che sia in grado di eseguire l'algoritmo. Possiamo individuare tre proprietà di cui deve godere l'esecutore per poter portare a termine il suo compito:

1. Anzitutto, l'esecutore deve essere in grado di gestire correttamente la sequenza di esecuzione delle operazioni così come è presente nell'algoritmo. (interpretare correttamente la sequenza di programmi)

2. L'esecutore deve essere inoltre capace di eseguire ognuna delle operazioni presenti nell'algoritmo.

3. Infine, l'esecutore deve essere in grado di memorizzare i dati che elabora su opportuni supporti di memorizzazione che gli permettano di accedere ai dati memorizzati e, eventualmente, di modificarli.

È interessante notare come, tra le proprietà richieste, non ci sia la **consapevolezza da parte dell'esecutore** di quanto stia facendo. In effetti, per l'esecuzione dell'algoritmo è *sufficiente* che l'esecutore sappia realizzare il meccanismo di interpretazione ed esecuzione delle operazioni dell'algoritmo anche senza sapere qual è lo scopo delle sue attività.

*«L'algoritmo deve adattarsi all'esecutore».*

### 1.2.2 Programma e linguaggio

- La rappresentazione dell'algoritmo comprensibile ed eseguibile dall'esecutore automatico costituisce un **programma**.

- L'elaborazione delle azioni è richiesta all'elaboratore tramite comandi elementari chiamati **istruzioni** espresse attraverso un opportuno formalismo: il ***linguaggio di programmazione***.

- Un **programma** è la *formulazione* di un *testo* di un ***algoritmo***, scritto in accordo al lessico, alla sintassi e alla semantica di un linguaggio di programmazione, che *risolve* un dato *problema*.

- Un algoritmo può essere implementato in linguaggi diversi; ognuno dei programmi ottenuti è **un'implementazione dell'algoritmo originale** ed è quindi **equivalente agli altri programmi**.

### 1.2.3 Risorse e processo

- Una volta ottenuto il **programma**, questo deve essere eseguito su un sistema di elaborazione che realizzi il ruolo di **esecutore**.

- A questo scopo è necessario che il sistema metta a disposizione del programma le opportune **risorse** che rendano possibile l'esecuzione. (Es: memoria per ospitare i dati del programma, dispositivi per inserire i dati in ingresso e fornire i dati in uscita)

- Se le risorse sono disponibili, è possibile avviare l'esecuzione del programma. Da questo momento in poi si parla di **processo**.

- *Differenza sostanziale*: mentre il **programma** è la descrizione di un procedimento risolutivo, il **processo** è invece l'attuazione di tale procedimento.
- Il **processo** *richiede* un **esecutore** che fornisca le risorse necessarie, effettua le azioni previste, riceva i dati di ingresso e produca i dati in uscita.

Possiamo descrivere le azioni che compie l'esecutore per eseguire un programma con l'*aiuto* di un ***algoritmo***:

1. Fase di inizializzazione:
    a. verifica che tutte le risorse necessarie siano disponibili
    b. inizializza un segnaposto alla prima istruzione del programma
2. Esegui ripetutamente i seguenti passi, fino al raggiungimento della fine del programma:
    a. leggi l'istruzione del programma associata al segnaposto, e sposta in avanti il segnaposto
    b. se l'istruzione ha bisogno di dati, acquisiscili
    c. esegui l'istruzione

# Capitolo 2

# Concetti di base della programmazione

*La programmazione, oggi, è una gara tra gli ingegneri informatici,*
*che si sforzano di creare programmi migliori a prova d'idiota,*
*e l'Universo, che cerca di produrre idioti migliori.*
*Per ora, sta vincendo l'Universo.*
– ***Rick Cook***

## 2.1 Rappresentazione degli algoritmi

Per realizzare un procedimento risolutivo, un algoritmo esegue delle **operazioni** su un **insieme di informazioni**, e quindi necessario specificare in maniera formale (precisa e non ambigua):

- le **informazioni** su cui l'algoritmo lavora
- le **operazioni** che l'algoritmo compie

All'interno di un algoritmo un'informazione può essere organizzata in vari modi:

- ***variabile***
- ***costante***
- ***espressione***

#### Variabile

La *variabile* è un ente, appartenente ad un certo **tipo**, che *può assumere uno qualunque dei valori appartenenti al* ***tipo***.

Una *variabile* è identificata da un nome, che riflette il ruolo che questa assume all'interno dell'algoritmo.
Il valore di una *variabile* può essere sia **utilizzato** (*lettura*) che **modificato** (*scrittura*).

#### Costante

La *costante* è un oggetto, appartenente ad un certo **tipo**, *il cui valore rimane immodificato durante l'esecuzione dell'algoritmo*, a cui può essere attribuito un nome.

### Espressione

L'*espressione* è una **sequenza** di operandi, operatori e parentesi, dove gli **operandi** possono essere *variabili* o *costanti*. Il **tipo** dell'espressione complessiva dipende dai tipi degli operandi coinvolti nell'espressione.

**Definizione** Un *operatore* viene detto unitario se coinvolge un solo operando, binario se ne coinvolge due e ternario se invece prevede tre *operandi*.

Gli operatori sono raggruppati, per tipologia, nelle seguenti categorie:

- aritmetici: oltre alle quattro operazioni fondamentali si aggiungono altri operatori, di molto uso comune, come quello di modulo o elevazione a potenza;
- relazionali: che hanno lo scopo di confrontare i valori degli operandi e forniscono un *risultato booleano*. ($>$, $\geq$, $<$, $\leq$, $=$, $\neq$);
- logici: che si applicano ad operandi booleani, fornendo risultati anch'essi booleani (**and**, **or**, **not**);
- bitwise: si applicano a variabili che sono espresse in termini di sequenza di bit.

Ogni operatore ha delle specifiche proprietà di precedenza: l'operatore logico **NOT** ha la precedenza maggiore degli operatori aritmetici, relazionali e dei rimanenti operatori logici; gli **operatori aritmetici** hanno la precedenza sugli operatori logici (tranne *NOT*) e di quelli relazionali; gli **operatori logici** di *AND* e *OR* non hanno precedenza sugli operatori relazionali, l'operatore **AND** ha la precedenza su **OR**.

### Il sistema dei tipi

Il **tipo**, in informatica, sarebbe il sinonimo di *insieme predefinito*.

I tipi semplici che un linguaggio mette a disposizione sono ovviamente dipendenti da quest'ultimo; comunque quelli abbastanza standardizzati sono:

- tipo **intero** che consente di trattare informazioni numeriche del tipo intero, positivo o negativo;
- tipo **reale**, per l'elaborazione di informazioni numeriche decimali con rappresentazione in virgola mobile;
- tipo **reale con doppia precisione** doppia precisione rispetto al precedente;
- tipo **logico**, per trattare informazioni di tipo *booleano* (con due valori possibili: vero o falso);
- tipo **carattere**, usato per trattare un'informazione di tipo carattere alfanumerico;
- tipo **stringa**, con lo scopo di voler trattare un'informazione costituita da una sequenza di caratteri.

Ognuno dei tipi ha un nome, detto ***identificatore*** *di tipo*, che viene utilizzato per associare ad ogni **variabile** il *tipo* che le compete, attraverso una ***dichiarazione di tipo***.

## 2.2 Rappresentazione delle operazioni

Esistono due possibili strumenti per *definire* e *rappresentare* in maniera non ambigua le istruzioni che costituiscono l'algoritmo:

- **Pseudo codice**
- **Diagrammi di flusso (*Flow Chart*)**

Nella risoluzione di un problema, il primo passo verso la modellazione di un algoritmo è cercare di descrivere le azioni da fare in linguaggio naturale.

### Pseduo codice

Lo *pseudo codice* permette la rappresentazione dell'algoritmo in forma testuale dove le particolari operazioni possono essere descritte in modo *informale* e *sintetico*.
Costrutti di controllo spesso descritti con forme e parole chiave corrispondenti o vicine a quelle dei linguaggi di programmazione.

### Diagrammi di flusso

I *diagrammi di flusso* consentono la modellazione grafica di un algoritmo in maniera immediata: consente di descrivere un algoritmo concentrandosi principalmente sulla sequenza delle operazioni di cui si compone.

"*Alternativa allo pseudo codice* per algoritmi non eccessivamente complessi."

Ogni istruzione dell'algoritmo viene rappresentata all'intero di un blocco elementare, la cui forma grafica è determinata dal tipo di istruzione; i blocchi sono collegati tra loro da **linee di flusso**, munite di *frecce*, che indicano il susseguirsi di azioni elementari.

- **Rettangolo**: indica una azione elaborativa.
- **Parallelogramma**: indica un'operazione di input/output.
- **Rombo**: indica una selezione.
- **Ovale**: indica l'inizio o la fine di un programma, o di una sezione di codice (in questo caso si usano anche *simboli cerchietti*, detti simboli di connessione).

### Costrutti di calcolo e assegnazione

Le operazioni di **calcolo e assegnazione** e le operazioni di **Input/Output** sono le prime semplici operazioni che coinvolgono variabili, costanti ed espressioni.
L'effetto è di *aggiornare il valore* di una **variabile** di un certo tipo con il valore ottenuto dalla valutazione di un'*espressione* dello stesso tipo. Il formato è:

$$variabile \;=\; espressione$$

**Il segno** "=" sta ad *indicare* **l'*azione*** di assegnare il valore a destra alla variabile il cui nome è a sinistra.

Infatti a volte si usa il formato:

$$variabile \leftarrow espressione$$

# Capitolo 3

# Controllo di flusso

*Non puoi avere il controllo su ogni evento della tua vita ma puoi decidere di non farti tenere al guinzaglio dalle tue emozioni più negative.*
***- Stephen Littleword***

## 3.1 Le istruzioni semplici

Un algoritmo si presenta come un insieme di istruzioni opportunamente connesse tra loro in accordo a schemi precostituiti.

Le istruzioni si classificano in:

- Istruzioni **dichiarative**: servono fondamentalmente per descrivere i dati che sono utilizzati in un algoritmo, e ne specificano le caratteristiche e la struttura.
- Istruzioni **operative**: corrispondono ad azioni semplici, direttamente eseguibili dall'esecutore. Possono essere ulteriormente suddivise in:
  - Istruzioni di *assegnazione*: hanno il compito di calcolare il valore di un'espressione e di associarlo ad una variabile;
  - istruzioni di *I/O* (ingresso/uscita o lettura/scrittura): sono impiegate per leggere e scrivere i valori dei dati dai supporti di ingresso e uscita del calcolatore, come tastiera e monitor.

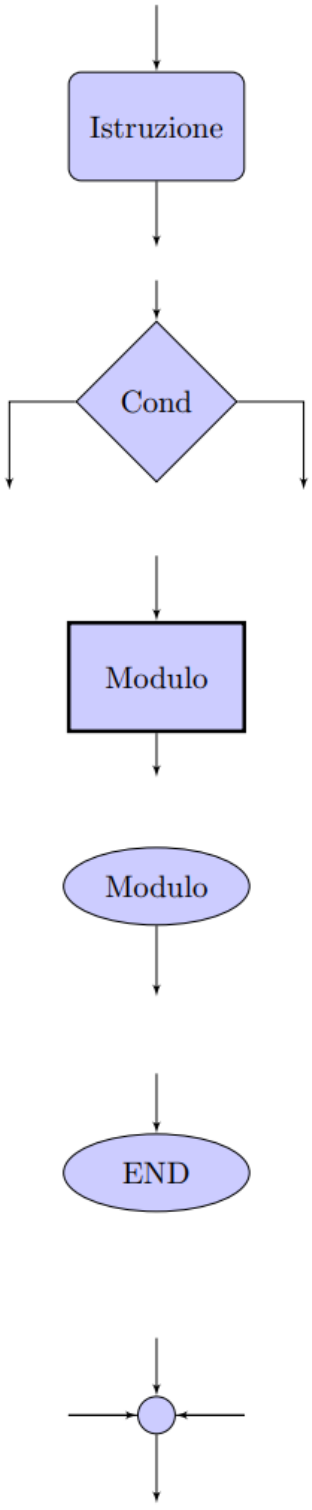
### Operazioni di Input/Output

Le istruzioni di ingresso e di uscita permettono lo scambio di dati tra l'utente e l'esecutore, attraverso i dispositivi fisici, ovvero la tastiera e il monitor.

Con le operazioni di **Input**, il *valore* di una **variabile** viene *modificato* con il valore ottenuto grazie ad un'***operazione di lettura*** dell'unità di ingresso (tastiera).

Con le operazioni di **Output**, un'**espressione** viene **valutata** ed il *valore* ottenuto viene *presentato* sull'unità di uscita (schermo).

### Costrutti di programmazione




Blocco: Rappresenta l'esecuzione di un'istruzione

Selezione: Rappresenta un blocco selettivo che in funzione del valore di Cond, instrada il flusso in due strade distinte

Chiamata di un modulo: Rappresenta l'astrazione di un diagramma di flusso che viene identificato mediante il nome Modulo

Dichiarazione di un modulo: Identifica l'inizio del diagramma di flusso che viene identificato mediante il nome Modulo

Fine: Identifica la fine di un diagramma di flusso

Connettore: riunisce più linee di flusso in ingresso in un unico flusso di uscita

Nella maggior parte dei linguaggi di programmazione, le strutture di controllo (costrutti) sono impiegate in un programma per controllare il **flusso di esecuzione**.

Il *flusso di esecuzione* serve a specificare: **se**, **quando**, **in quale ordine** e **quante volte** devono essere eseguite le istruzioni che compongono il programma stesso. I costrutti sono strumenti per costruire istruzioni composte a partire dalle *istruzioni semplici* (assegnazione e lettura/scrittura).

Sebbene ogni linguaggio di programmazione abbia il proprio insieme di costrutti che può differire da quello impiegato in un altro linguaggio, si evincono alcuni *requisiti* ***comuni*** della **programmazione strutturata**:

- le strutture sono del tipo **one-in one-out** (singolo ingresso, singola uscita); sono considerate come una singola macro-istruzione, con un unico e ben identificato punto di ingresso e un unico punto di uscita;
- le strutture di controllo possono contenere al loro interno istruzioni a loro volta composte, senza limiti a tale procedimento di **nesting** (annidamento);
- l'insieme delle strutture di controllo è funzionalmente completo:
  sono cioè sufficienti a scrivere un qualsiasi algoritmo → ***teorema di Bohm e Jacopini***.

**Teorema di Böhm-Jacopini**

Il teorema di Böhm-Jacopini (1966) afferma che:

*"**Qualunque algoritmo** può essere implementato utilizzando tre sole strutture: la **sequenza**, la **selezione** e il **ciclo** (o iterazione); da applicare ricorsivamente alla composizione di istruzioni elementari".*

Tre tipi di costrutti di controllo:[1]

- *Costrutto* **Sequenza**: La sequenza è la struttura di controllo fondamentale di qualsiasi linguaggio imperativo, inclusi i linguaggi macchina, ed è definita attraverso la semplice successione delle istruzioni di cui si compone.
- *Costrutti* **selettivi**: consentono di operare delle scelte.
- *Costrutti* **iterativi**: consentono di ripetere alcune operazioni un certo numero di volte.

### 3.1.1 Costrutti selettivi

I costrutti selettivi (dette anche strutture di controllo) consentono di instradare il flusso di controllo dell'algoritmo su più strade in funzione del valore assunto da una condizione specificata nel rombo.
Esistono diverse tipologie di costrutti selettivi che si differenziano in base al numero di strade differenti previste e alla natura della condizione. I più semplici sono: **if-then** e **if-then-else**.

**Costrutto selettivo: if-then**

Il costrutto if-then è la tipologia più semplice di selezione e viene detta selezione unaria (o selezione ad una via);
• *Solo su una «via» si eseguono le istruzioni!*
Controlla una **condizione** (**Cond**) che, se verificata, abilita l'esecuzione di un **blocco sequenza B** (o blocco istruzioni, con una o più istruzioni).
• La **Cond** è un'espressione logica che può assumere uno dei due valori:

- 1, 0
- Vero, Falso
- Si, No

[1] Mediante l'uso dei flow-chart, è possibile combinare i blocchi per realizzare i costrutti selettivi e iterativi

**Costrutto selettivo: if-then-else**

Il costrutto **if-then-else** è detto selezione binaria (o a due vie);
Prevede l'esecuzione di due blocchi alternativi di istruzioni, sulla base del risultato della condizione **Cond**.
Se **Cond** assume il valore *vero*, viene eseguito il blocco di istruzioni B1, altrimenti (cioè, *se Cond è falsa*) viene eseguito il blocco B2.

Come per la *selezione unaria*, la selezione a due vie prevede che la condizione **Cond** sia di *tipo binario*, ovvero tale da assumere uno tra i due valori possibili, *vero o falso*.

**Costrutto selettivo: case**

*Il **case** è una generalizzazione del costrutto selettivo in cui siano presenti più di due strade distinte.*

La sua logica di funzionamento è molto semplice.

- Si hanno n strade (B1, ... Bn), ognuna delle quali associata a uno dei possibili valori assunti dall'espressione **Espr** (detta espressione di selezione);
- C'è un'ulteriore strada (**Bd**), detta di *default*, che viene percorsa allorquando il valore assunto dall'espressione di selezione non coincide con nessuno dei valori associati alle n strade.
- **Espr NON è una condizione logica ma un'espressione**

### 3.1.2 Costrutti iterativi

Le strutture di controllo **iterative**, dette *cicli*, consentono di *eseguire ripetutamente* una data istruzione (o un blocco di istruzioni).

Ogni costrutto iterativo deve consentire di specificare sotto quali condizioni l'**iterazione** (ripetizione) di tali istruzioni debba terminare → **condizione di permanenza nel ciclo**.

Elementi di un costrutto iterativo:

- ***Inizializzazione***: le variabili usate, e soprattutto quelle nella condizione del ciclo, devono avere un valore iniziale.
- ***Condizione***: deve essere valutata, per determinare la sua ripetizione o la terminazione del ciclo.
- ***Modifica***: almeno una delle variabili della condizione deve essere modificata all'interno del ciclo, in modo che prima o poi la condizione di ripetizione diventi falsa, causando la **terminazione del ciclo**.

Una prima classificazione di *costrutti iterativi* si può ottenere in base al ***numero di iterazioni***:

- **Predeterminato**: noto prima di entrare nel ciclo stesso
- **Non Predeterminato**: il numero delle iterazioni può cambiare durante l'esecuzione del ciclo.

I *cicli iterativi* sono: **for** *(P)*, **while** *(NP)*, **do-while** *(NP)* e il **repeat-until** *(simile a **while**)*.

### Ciclo while

Il ciclo **while** (*Detto anche ciclo a condizione iniziale*) è un costrutto iterativo in cui la ripetizione è espressa in termini di una condizione logica che indica «fin quando si continua la ripetizione» (permanenza nel ciclo):
se **Cond** è vera il ciclo continua a ripetere le istruzioni (blocco sequenza B) in esso contenuto.

Le forme tradizionali di tale ciclo possono così essere parafrasate: **finchè *Cond* è verificata, ripeti *B***.

### Ciclo do-while

Se si inverte la condizione del *repeat-until*:
*«Se **Cond** è vera il ciclo continua, altrimenti termina»*,
si ottiene il ciclo **do-while**:

1. L'iterazione inizia con l'esecuzione del blocco B.
2. Poi viene verificata la condizione **Cond**:
3. se questa è vera si continua con la successiva iterazione,
4. Altrimenti si esce dal ciclo;
5. **Cond** rappresenta quindi la *condizione di continuazione del ciclo.*

### Ciclo for

Il ciclo **for** si usa in genere quando la *condizione di permanenza* in un ciclo è **predeterminata**. Tale ciclo consente di specificare quante volte debbano essere ripetuti l'istruzione o il blocco controllati dal ciclo.

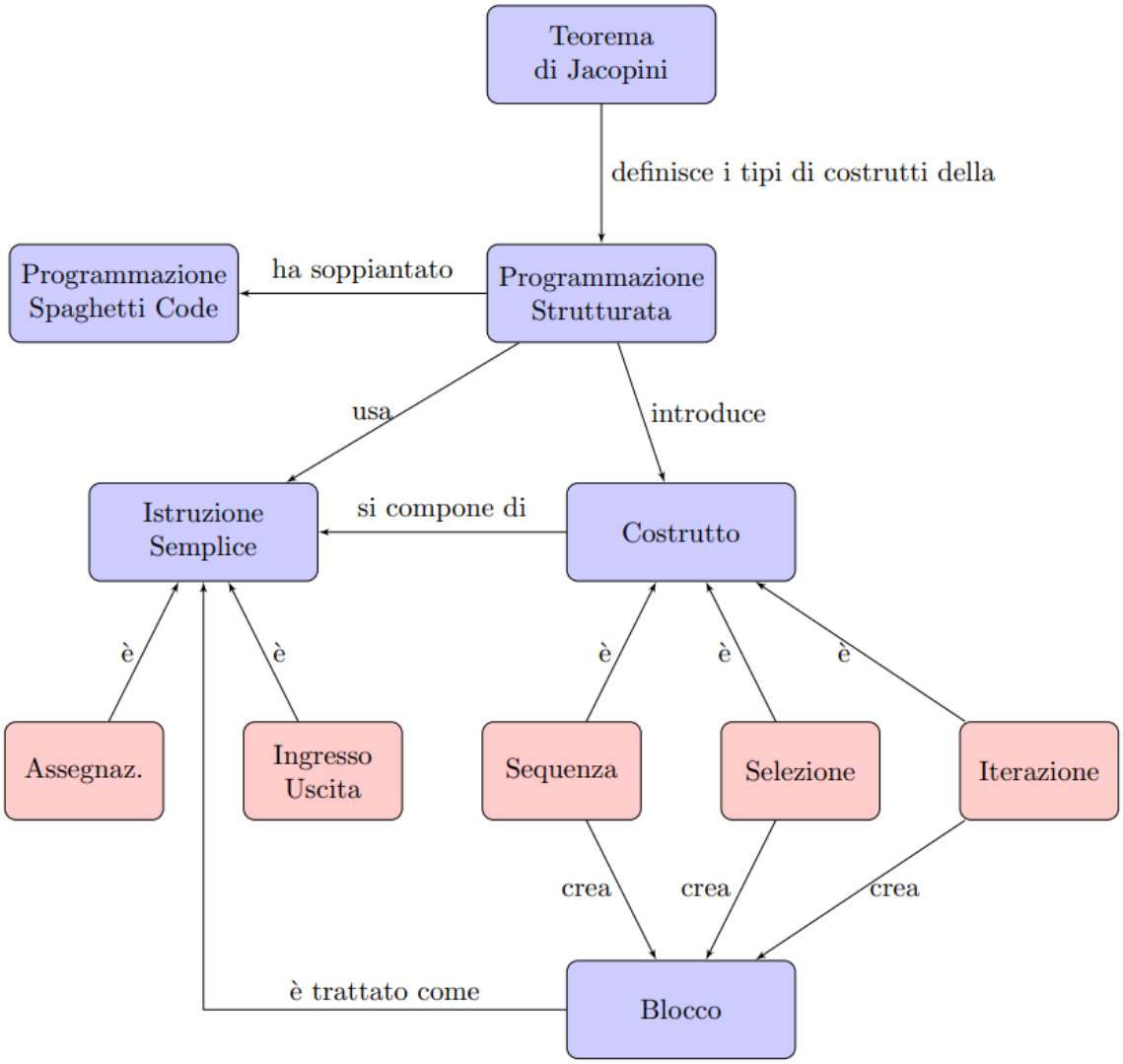
Si impiega una **variabile contatore** (di tipo intero) che conta il numero di iterazioni, o ripetizioni del ciclo.

Il ciclo **for** può esser parafrasato con *"ripeti il blocco di istruzioni B per i valori del contatore **cont** che vanno da un certo valore iniziale **vi** a un certo valore finale **vf** con un certo **passo** (solitamente 1)"*.

### Ciclo repeat-until

Il ciclo **repeat-until** (ripeti finché) differisce dal **while** per *due aspetti*:

- Innanzitutto viene espressa la *condizione di terminazione* del ciclo *anziché* quella di **permanenza** nel ciclo; ciò ha un impatto concettuale marginale, dal momento che le due condizioni sono *l'una la **negazione** dell'altra.*
- La seconda e più rilevante differenza consiste nel fatto che il blocco di ***istruzioni*** *precede la **verifica*** della **condizione**, a differenza del **while** che invece verifica dapprima la condizione da cui dipende l'iterazione.


Teorema
di Jacopini
definisce i tipi di costrutti della
Programmazione
Spaghetti Code
ha soppiantato
Programmazione
Strutturata
usa
introduce
Istruzione
Semplice
si compone di
Costrutto
è
è
è
è
è
Assegnaz.
Ingresso
Uscita
Sequenza
Selezione
Iterazione
crea
crea
crea
è trattato come
Blocco

# Capitolo 4

# L'informazione e rappresentazione

Il concetto di **informazione** è legato indissolubilmente al concetto di **scelta**. Se non esiste scelta non c'è informazione, e *solo dove esiste una scelta parliamo di informazione*; possiamo ancora dire che, quanto più ampia è la *scelta*, maggiore è l'*informazione*.

Da un punto di vista formale un'**informazione** è una tripla $I = \{Attributo, Tipo, Valore\}$ dove:

- l'**Attributo** è il nome associato all'informazione, che ne chiarisce anche il significato
- il **Tipo**[1] è l'insieme (*finito*) di tutti i possibili valori che l'Informazione può assumere
- il **Valore** è la scelta di uno specifico elemento all'interno del Tipo

### 4.0.1 Rappresentazione binaria dell'informazione

Strettamente legato ad un'informazione è il concetto di *rappresentazione*.

La **Rappresentazione** è il risultato di un ***procedimento*** che adotta delle *regole* per trovare una **codifica** non ambigua dell'***informazione*** in un particolare **linguaggio**.

**CURIOSITA'**
L'**informazione** si misura in **bit**[a], o meglio attraverso il numero di *bit minimo* per rappresentare il tipo **T** (ovvero tutte le possibili scelte associate all'informazione stessa); quindi se la cardinalità di **T** è $n$, la quantità $q$ dell'informazione relativa è pari a: $q = \lceil \log_2 n \rceil$, dove la notazione $\lceil x \rceil$ (detta ceil) indica il numero intero immediatamente più grande di $x$. Dunque esiste una *relazione di tipo logaritmica* tra la *cardinalità* di **T** e la *quantità di bit minima* per rappresentare l'**informazione relativa**. Con $q$ bit possiamo creare $2^q$ stringhe diverse, ognuna delle quali può essere associata ad un valore dell'informazione da codificare. Deve quindi essere $2^q \geq n$, cioè $q \geq \log_2 n$, da qui la precedente $q = \lceil \log_2 n \rceil$.

[a] Binary + digIT

Il metodo più importante per trattare le informazioni è la **rappresentazione binaria**. Si basa sulla rappresentazione delle due cifre binarie (0 e 1), dette `digit` da cui il termine `rappresentazione digitale`.

[1] Bisogna prestare attenzione alla cardinalità del Tipo, ovvero da quanti elementi esso si compone: un tipo deve essere finito, altrimenti non può essere adeguatamente rappresentato in un calcolatore.

La rappresentazione digitale di un'informazione ha un'importanza fondamentale perché: l'*informazione* può essere rappresentata come **sequenza di cifre binarie**, può essere **memorizzata**, **trasferita** o **elaborata** da un computer. Dunque è stato scelto il **sistema binario** perché l'*affidabilità* di *2 stati possibili* permette la realizzazione più **sicura**.

Dicendo che il sistema binario è `fisicamente conveniente`, si intende dire che è:

- *realizzabile*
- *affidabile*
- facilmente *leggibile* e *scrivibile*

### Che cos'è l'informazione

L'`informatica` rappresenta la convergenza di due aspetti distinti: l'**informazione** e la sua **elaborazione automatica**. La parola *informazione* è qualcosa che se fornita dà **conoscenza**, *dissipa i dubbi* ed aumenta la **certezza**.

Quando si riceve un'informazione, ogni simbolo ricevuto permette di "*eliminare alternative*", escludendo altri simboli possibili ed il loro significato associato.

- "Se non esiste scelta non c'è *informazione*"
- "*solo* dove esiste una scelta si ha informazione"
- "quanto più ampia è la scelta, maggiore è l'informazione che si riceve"
- L'informazione riduce l'**incertezza** di una ***scelta***.

Bisogna prestare *attenzione* alla **cardinalità del tipo** (di quanti elementi esso si compone).

Un Tipo ***deve essere finito***, altrimenti non potrà essere adeguamene rappresentato in un calcolatore elettronico o su un qualunque altro mezzo o supporto fisico.

### Codifica

La stessa **informazione** potrebbe essere *rappresentata* in diversi modi; la stessa **rappresentazione/codifica** può essere usata per *informazioni* differenti.

L'informazione deve essere "*comprensibile*" affinché chi riceve tale informazione sia capace di interpretare quello che il mittente ha inviato; è importante *conoscere* il `sistema di codifica` (o codice).

La **codifica** rappresenta il processo che porta ad assegnare una rappresentazione all'informazione.

$$\text{Informazione} \rightarrow \text{Codifica} \rightarrow \text{Rappresentazione}$$

Un **sistema di codifica** usa un insieme di simboli (*alfabeto*) e la combinazione di questi simboli (*configurazioni, stati, stringhe*).

**Definizione**: Sia data un'informazione di tipo $T$ di cardinalità $n$ e un alfabeto di $k$ simboli $A = \{S_1, S_2, \cdot, S_k\}$; $S$ è l'insieme di tutte le stringhe (o configurazioni) composte da $m$ simboli di $A$.

La **codifica di *T*** è una *funzione C(T)* che ad ogni valore $v \in T$ possibile dell'informazione associa una stringa $\sigma \in S$, ovvero:

$$C : \forall v \in T,\ v \ \rightarrow \ \sigma \in S \tag{4.1}$$

Particolarmente importanti sono le codifiche che a **valori diversi** *associano* **stringhe** `codificate` **diverse**:

$$C : \forall v_1, v_2 \in T,\ v_1 \ \rightarrow \ \sigma_1 \in S,\ v_2 \ \rightarrow \ \sigma_2 \in S,\ \ v_1 \neq v_2 \ \rightarrow \ \sigma_1 \neq \sigma_2 \tag{4.2}$$

Ogni informazione è rappresentata da una **combinazione** di questi simboli e ha un significato associato.

#### Cardinalità di un tipo

Da quali elementi è composto, quanti valori può assumere la variabile. Un **tipo** deve essere formato da un insieme di valori *finiti*, altrimenti non potrà essere inizializzato/rappresentato dall'esecutore.

#### Rappresentazione

Risultato di un procedimento eseguito secondo le regole di un determinato linguaggio di rappresentazione.

### 4.0.2 Rappresentazione dei numeri naturali

Un sistema di numerazione si dice **posizionale** se le cifre *(o più in generale i simboli)* usate per scrivere i numeri assumono valori diversi a seconda della posizione che occupano. Si dice **pesato** perché ad ogni *posizione* è associato un peso espresso come potenza della base.

**- Fondamentale la presenza dello 0**.

**LSB** (Least Significant Bit) = Bit meno significativo
**MSB** (Most Significant Bit) = Bit più significativo

# Capitolo 5

# Linguaggi di programmazione

Le informazioni si possono codificare come sequenze di bit di dimensione predefinita.
Una simile rappresentazione può essere assunta anche per le **istruzioni**, cioè le singole azioni elementari che l'unità centrale può eseguire.
Nello specificare un'istruzione, bisogna precisare l'**operazione** da compiere e i **dati** coinvolti nell'operazione.
L'insieme delle diverse operazioni che l'unità centrale è in grado di eseguire è **finito** e quindi è possibile codificarlo con un certo numero di bit (**codice operativo**).

Un'istruzione sarà quindi rappresentabile da una sequenza di bit divisa in due parti:

- un **codice operativo**
- uno o più **operandi**

In questo modo, un esecutore automatico può permettere la memorizzazione e l'esecuzione di un **programma**, cioè di una sequenza di istruzioni che realizzano un particolare algoritmo e che sono descritte nel linguaggio interpretabile dal calcolatore.

**Caratteristiche del linguaggio** - *linguaggio macchina*

- è codificato tramite sequenze di bit
- ogni istruzione può compiere solo azioni molto semplici. In particolare, non sono disponibili come istruzioni di base i costrutti visti precedentemente
- non gestisce direttamente i tipi di dati di interesse
- è strettamente legato alla particolare macchina su cui è definito

Per *implementare* un dato *algoritmo* scrivendo un **programma** in `linguaggio macchina` sarebbe quindi necessario:

- conoscere dettagliatamente tutti i codici operativi e la loro codifica
- decidere in quali porzioni di memoria vadano memorizzati i dati
- definire un'opportuna tecnica di codifica per ogni tipo di dati considerato
- determinare, per ogni singola operazione richiesta dall'algoritmo, la sequenza di istruzioni in linguaggio macchina che la realizzano
- limitarsi a utilizzare solo i calcolatori per cui esista una tale competenza, tenendo comunque presente che il programma scritto per un certo calcolatore non è eseguibile su altre macchine.

**Linguaggio di programmazione**

**Esecutore umano**

- linguaggio formale (*flow chart*), semplice e di uso generale
- istruzioni in grado di implementare direttamente i principali costrutti di programmazione
- gestione completa dei tipi

**Sistema di elaborazione**

- linguaggio rigido, complicato e specifico del particolare esecutore
- istruzioni estremamente semplici
- gestione dei tipi quasi nulla

## 5.1 Linguaggi ad alto livello (HLL)

Per passare in maniera semplice e rigorosa dalla descrizione dell'algoritmo comprensibile all'esecutore umano ad un programma eseguibile dal sistema di elaborazione, è necessario un **linguaggio ad alto livello** che:

- metta a disposizione del programmatore un supporto semplice ed efficace all'impiego dei costrutti di programmazione e dei principali tipi di dati
- permetta una traduzione affidabile del programma espresso in tale linguaggio in un programma espresso in linguaggio macchina eseguibile sul sistema di elaborazione

**Vantaggi**

L'uso di linguaggi ad alto livello permette di:

- realizzare un programma che implementa l'algoritmo in maniera precisa tramite costrutti più vicini al livello di astrazione con il quale un essere umano descrive un algoritmo
- trascurare tutti i dettagli relativi alla rappresentazione dei dati all'interno del sistema di elaborazione
- realizzare un programma che non dipende dal particolare sistema di elaborazione su cui è stato realizzato

Tali caratteristiche *agevolano* significativamente il **compito della programmazione**, rendendola più *semplice* e *veloce*. Soprattutto consentono di scrivere programmi di *qualità* adeguata, facilmente **manutenibili** e **riusabili**.

**Linguaggio = macchina virtuale**

In effetti, il programmatore non deve interagire con la macchina reale e le sue limitazioni, ma "*vede*" una **macchina astratta** che nasconde le particolarità della macchina reale e con la quale è molto più agevole *interagire*.

### 5.1.1 Dal linguaggio di alto livello al linguaggio macchina

I linguaggi di programmazione ad alto livello sono **linguaggi formali**, in cui la forma delle frasi, la **sintassi**, e il loro significato, la **semantica**, sono definiti sulla base di regole *rigide* e *precise*.

In tal modo viene *eliminata l'ambiguità* e le ridondanze tipiche del linguaggio naturale ed è possibile realizzare in modo *automatico* l'analisi di un programma scritto in un **linguaggio ad alto livello** e la sua traduzione in **linguaggio macchina**.

I programmi che svolgono il compito di tradurre un programma in linguaggio ad alto livello in un programma in linguaggio macchina sono detti **traduttori** (*compilatori* o *interpreti*).

#### Compilatori

La *traduzione* eseguita tramite **compilatore** (*compilazione*) si applica all'intero *programma* scritto in `linguaggio ad alto livello` (*linguaggio sorgente*) e produce un file oggetto, contenente la traduzione in `linguaggio macchina` del codice sorgente, ma ***non*** *ancora* ***eseguibile***.

Una successiva operazione di *collegamento* (**linking**) completa il `codice oggetto` con codice messo a disposizione dal **compilatore** (*libreria del compilatore*) e produce il **programma eseguibile** *finale*.

#### Interpreti

Un **interprete** *effettua la traduzione* istruzione per istruzione: **iterativamente** viene letta un'istruzione del programma in linguaggio sorgente, viene tradotta nel corrispondente insieme di istruzioni in linguaggio macchina e queste ultime eseguite.

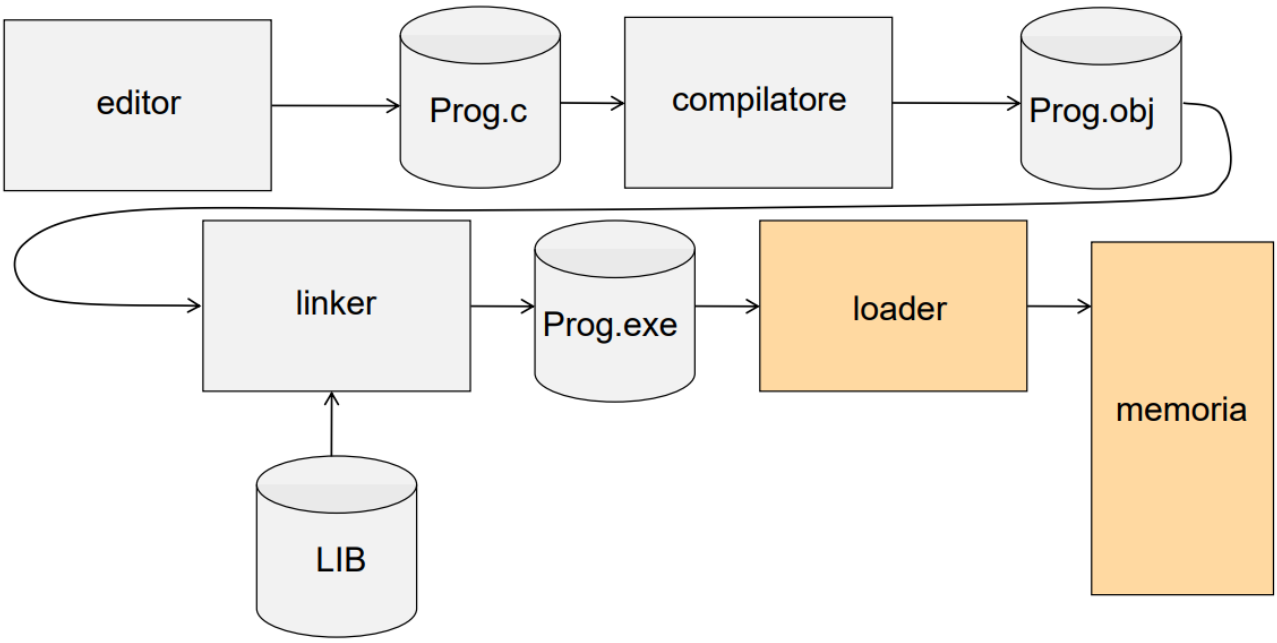
#### Dal produttore all'esecutore

Prima di essere eseguito, un **programma** attraversa le seguenti *fasi*:

- **Compilazione** : Traduzione di tutto il programma in linguaggio sorgente in linguaggio macchina: *codice oggetto*
- **Collegamento** : Il codice oggetto non è già direttamente eseguibile, ma necessita di ulteriori interventi. Un programma è in genere composto da più file oggetto (in genere uno per ogni modulo di programma) e impiega librerie che non sono scritte dal programmatore ma messe a disposizione dal compilatore stesso. Si rende quindi necessaria una fase di *collegamento* che ha l'obiettivo di collegare tutti i *moduli oggetto* (ad opera del **linker**) e le librerie per dar vita al file eseguibile finale
- **Caricamento in memoria** : L'eseguibile viene caricato nella memoria del calcolatore ed eseguito

I prodotti delle varie fasi sono ospitati in files:

- **.c** (*High Level Language*)
- **.obj** (*Low Level Language*)
- **.exe** (*Machine Language*)




Per ognuna delle *fasi* evidenziate sono disponibili strumenti **software appositi**:

- **Compilatore** (*compiler*)
- **Collegatore** (*linker*)
- **Caricatore** (*loader*)

**OSSERVAZIONE**

Una soluzione ibrida tra compilazione ed interpretazione prevede la compilazione del codice sorgente in un linguaggio intermedio, detto **bytecode**. Il bytecode viene in fase di esecuzione interpretato da una *virtual machine* (VM) ossia un interprete di basso livello che *emula* una **CPU**. Presenta innumerevoli **vantaggi**; il più significativo è la grande portabilità del codice, legata al fatto che lo stesso bytecode può essere eseguito su tutte le CPU per cui esiste un'implementazione della *VM*. Oggigiorno ogni produttore di dispositivi, siano essi Personal Computer, Smartphone, Tablet, Smart Tv, Game console, Set top box, etc. vengono fornite con una **macchina virtuale Java** (*JVM*); scrivere un programma in java implica che, una volta tradotto in bytecode, quest'ultimo senza ulteriori compilazioni può essere direttamente eseguito su uno qualsiasi dei suddetti dispositivi.

### 5.1.2 Strumenti di sviluppo

Un ambiente agevole per la progettazione e la realizzazione di programmi è fornito dagli **Integrated Development Environment** (`IDE`).

Un IDE è una collezione di strumenti software integrati, con interfaccia visuale, che aiuta il programmatore nello sviluppo di un programma in tutte le fasi della realizzazione.
Tipicamente un IDE è composto da:

- un **editor** di codice sorgente guidato da sintassi (*syntax highlighting*), è in grado di riconoscere le principali strutture lessicali e sintattiche del linguaggio e agevolare la fase di redazione del programma, evidenziando le parole chiave, le strutture di controllo e operando semplici controlli nel codice sorgente per diminuire la probabilità di errore;

- un **compilatore** e/o un interprete che possono essere avviati all'interno dell'IDE stesso, in maniera *friendly*;
- un tool di **building automatico**, che collega tutte le unità che costituiscono il programma e produce il file eseguibile;
- un **debugger** che offre la possibilità di eseguire il programma controllandone l'esecuzione mediante la visualizzazione dei valori che assumono le sue variabili o prevedendone l'avanzamento istruzione per istruzione.

## 5.2 Il sistema dei tipi

Ogni linguaggio di programmazione mette a disposizione del programmatore un insieme di tipi predefiniti detto sistema dei tipi. **Tipi numerici**:

- `intero`: costituito da un sottoinsieme **limitato** dei numeri interi, rappresentazione in complementi alla base su un numero prefissato $m$ di bit (*l'unico che ha l'operatore "modulo"*)
- `reale`: costituito da un sottoinsieme **limitato e discreto** dei numeri reali, rappresentazione su un numero prefissato $m$ di bit (*floating point*)
- `reale doppia precisione`: costituito da un sottoinsieme **limitato e discreto** dei numeri reali, ma con *range e precisione maggiore* rispetto al tipo reale, rappresentazione su un numero prefissato $m$ di bit (*floating point*) maggiore di quanto fissato per il tipo reale.

**Tipi non numerici**:

- `carattere`: Consiste in un insieme di caratteri, alcuni stampabili (caratteri alfabetici, cifre, caratteri di punteggiatura, ecc.) ed altri non stampabili tramite i quali si gestisce il formato dell'input/output (caratteri di controllo).

  I sottoinsiemi delle lettere e delle cifre sono ordinati e coerenti. Per la rappresentazione interna, viene tipicamente usato il codice ASCII, che mette in corrispondenza ogni carattere con un numero intero compreso tra 0 e 255.
- `logico`: È un tipo costituito dai due soli valori **vero** e **falso**. Il tipo rappresenta le informazioni di tipo logico (es. il risultato di un confronto, il verificarsi di una situazione)
- `stringa`: Consente di trattare un'informazione costituita da una sequenza di caratteri (es. una parola)

### Tipi enumerativi

Un tipo `enumerativo` è un **tipo di dati definito dal programmatore** costituito da un insieme di valori denominati *elementi* o *membri*.
I nomi degli elementi sono generalmente identificatori che si comportano come costanti nel linguaggio. Ad una variabile appartenente ad un tipo enumerativo può essere assegnato uno qualsiasi degli elementi come valore.

Gli elementi di un tipi enumerativo sono confrontabili e ordinati. La rappresentazione interna è tipicamente gestita direttamente dal compilatore.

## 5.3 Il sistema degli operatori

Un **operatore** è un simbolo che specifica un'operazione da eseguire su uno o due **operandi** *definendo* un'**espressione**.
All'interno di un linguaggio il **sistema degli operatori** è un insieme di regole e convenzioni sintattiche che consentono al programmatore di definire e valutare in maniera corretta espressioni anche complesse che coinvolgono più operatori ed operandi.
Il sistema degli operatori specifica i seguenti aspetti:

- operatori disponibili e loro tipologia
- precedenza degli operatori
- associatività degli operatori

Gli operatori possono essere classificati in base al numero di operandi che accettano, ovvero in base al numero di dati su cui lavorano:

- gli operatori unari lavorano su un singolo operando
- gli operatori binari lavorano su due operandi
- gli operatori ternari lavorano su tre operandi

Un'altra possibile classificazione si basa sulla tipologia di operazione realizzata:

- operatori aritmetici
- operatori relazionali
- operatori logici

#### Associatività

L'*associatività* di un operatore è una proprietà che determina come vengono raggruppati gli operatori della stessa precedenza in assenza di parentesi.
Un operatore è associativo a sinistra se, a parità di priorità, viene applicato da sinistra verso destra.

Per riflettere l'uso normale, gli operatori di addizione, sottrazione, moltiplicazione e divisione sono solitamente associativi a sinistra.

Per lo stesso motivo, gli operatori unari +, - e not sono associativi a destra.

# Capitolo 6

# Tipi strutturati

In alcuni casi, l'informazione che bisogna elaborare consiste di un'aggregazione di valori, piuttosto che di un valore solo.
Questo significa che sarebbe conveniente indicare l'insieme di valori di interesse con una sola variabile piuttosto che con tante variabili quante sono i valori da considerare: una variabile di **tipo strutturato**.
Le informazioni con cui entriamo in contatto nella vita nella maggior parte dei cari è forma aggregata.

Molteplici informazioni semplici sono raggruppate insieme assumendo un significato ben preciso così da rendere naturale il trattamento nella loro globalità, esattamente come se fossero informazioni semplici.

Per introdurre un dato strutturato è necessario definire:

- **il tipo di ogni informazione componente il dato strutturato**;
- **il costruttore di tipo strutturato**, ovvero una forma sintattica che il linguaggio deve mettere a disposizione del programmatore per definire dei tipi aggiuntivi a quelli di base (detti tipi semplici) che consentano l'aggregazione di informazioni;
- **una funzione di accesso agli elementi**; un'informazione strutturata deve infatti poter essere usata, oltre che nella sua interezza, avendo anche accesso alle singole parti; ciò si realizza mediante una funzione di accesso che rappresenta la forma sintattica da usare per accedere alle singole componenti dell'informazione strutturata a partire dal nome collettivo.

Esistono due tipi di variabili aggregate:

- i **vettori** (*arrays*) e
- le **strutture** (*struct* o record).

## 6.1 Array e Strutture

Un array è strutturato come un insieme di variabili, **tutte dello stesso tipo**, identificato da un nome unico. Gli elementi dell'array sono disposti in memoria in **posizioni consecutive**.
Per definire una *variabile array*, è necessario specificare:

- il **nome** della variabile array

- il **tipo** degli elementi
- il **numero** degli elementi presenti (*cardinalità dell'array*)

#### Funzione di accesso agli elementi dell'array

Per accedere ai singoli elementi di un array, è necessario specificare il nome della variabile array e la posizione dell'elemento di interesse tramite un valore intero (*variabile o costante*) che si definisce **indice**.

L'insieme di tutti i valori possibili dell'array si ottiene considerando che ogni suo elemento assume un valore in T e quindi l'insieme di tutti gli elementi è un elemento del suddetto prodotto cartesiano.

#### Array bidimensionali

Un array **monodimensionale** richiede un solo indice per l'*individuazione* di un elemento.

È possibile definire anche array **bidimensionali**, in cui l'organizzazione degli elementi è di **tipo matriciale**. Sono necessari due indici per identificare un elemento nell'array.
Per definire un array bidimensionale è necessario specificare:

- il **nome della variabile array**
- il **tipo** degli elementi
- il **numero** degli elementi presenti nelle **due** dimensioni (*cardinalità di **riga** e cardinalità di **colonna** dell'array*)

#### Funzione di accesso agli elementi dell'array

Per accedere ai singoli elementi di un array *bidimensionale*, è necessario specificare il nome della variabile array e gli indici di riga e di colonna che individuano l'elemento desiderato.

### 6.1.1 Struttura

Una struttura o record è un insieme finito di elementi:

- il **numero** degli elementi è rigidamente fissato a priori
- gli **elementi** possono essere di tipo diverso
- il **tipo** di ciascun elemento componente (*campo*) è prefissato
- l'**accesso** ai singoli elementi avviene tramite un identificatore

Per accedere ad un particolare elemento di una variabile di tipo struttura è quindi necessario **specificare il nome** della **variabile** ed il **nome** con cui l'*elemento* viene *identificato*.

Una struttura è una collezione di dati il cui tipo coincide con il prodotto cartesiano delle sue componenti.
La variabile del tipo *T-struttura* ha un **valore strutturato** costituito dall'insieme dei valori delle singole componenti. Poiché ogni sua componente $c_i$ può assumere uno dei valori nel corrispondente tipo $T_i$, l'insieme dei valori di un'informazione strutturata è pari alle possibili combinazioni di valori, *rappresentato dal suddetto prodotto cartesiano*.

# Capitolo 7

# Sottoprogrammi

Al crescere dell complessità del problema, diventa sempre più difficile progettare in un unico passo l'algoritmo corrispondente.

Uno dei principali modi per affrontare tale complessità è quello di decomporre un grande problema in diversi sottoproblemi più piccoli.

## 7.1 Decomposizione top-down / step-wise refinement

1. Decomporre il problema da risolvere in termini di problemi **indipendenti** più semplici che in prima battuta immaginiamo di saper affrontare e risolvere;
2. Analizziamo ciascuno di questi sottoproblemi e applichiamo ad essi il medesimo procedimento decomponendoli a loro volta in problemi più semplici;
3. Il processo di decomposizione termina quando si giunge a sottoproblemi di modeste complessità, risolvibili quindi con poche istruzioni.

Possibili problemi di ricopiare ogni volta le istruzioni:

- **Minore leggibilità del programma**: Il codice prodotto per ciascun sottoproblema potrebbe essere corposo e quindi avremmo un programma complessivo estremamente lungo e difficile da controllare;
- **Maggiore probabilità di errori**: l'inserimento del codice prodotto per ciascun sottopreblema richiederà un adeguamento al codice già presente (*verificare la coerenza dei nome delle variabili*) e questo può essere ulteriore fonte di errori;
- **Minore manutenibilità del programma**: nel caso dovessimo fare qualche modifica al codice sviluppato (*cambiando le specifiche del problema originale oppure troviamo un algoritmo migliore per un particolare sottoproblema*) l'individuazione della parte di programma da modificare sarebbe difficile e la stessa modifica potrebbe avere degli effetti collaterali negativi sul resto del codice.

L'ideale sarebbe disporre di un meccanismo che permettesse di inserire all'interno del programma una istruzione che eseguisse il codice realizzato per il particolare programma.

I **sottoprogrammi** danno al programmatore la possibilità di:

- specificare le istruzioni che codificano l'algoritmo risolutivo del sottoproblema;
- specificare i dati coinvolti;

- specificare il nome con cui identificare il codice da eseguire.

Un sottoprogramma è una **particolare unità di codice che non può essere eseguita autonomamente**, ma soltanto su richiesta del programma principale o di un altro sottoprogramma.

Un sottoprogramma viene realizzato per svolgere un compito specifico (*leggere o stampare gli elementi di un array, calcolare il valore di una particolare funzione matematica*) per il quale implementa un opportuno algoritmo.

Il **modulo** (programma principale o sottoprogramma) che richiede l'esecuzione del sottoprogramma viene detto **chiamante**.

Per eseguire il sottoprogramma, il *chiamante* dovrà trasferirgli i dati necessari per l'elaborazione, eseguire un comando di avviamento dell'elaborazione (**chiamata**), aspettare che l'elaborazione termini e gestire il trasferimento dei risultati prodotti dal sottoprogramma.

Ogni sottoprogramma è identificato da un **nome** che viene utilizzato nella chiamata.

Il trasferimento dei dati in ingresso ed in uscita dal sottoprogramma avviene sulla base di regole condivise tra chiamante e sottoprogramma.

Per portare a termine il suo compito, il sottoprogramma utilizza variabili proprie, distinte da quelle del chiamante.

Un sottoprogramma può essere attivato più volte in uno stesso programma o anche utilizzato da un programma diverso da quello per cui era stato inizialmente progettato.

#### Vantaggi dei sottoprogrammi

L'uso dei sottoprogrammi permette di organizzare in modo particolarmente efficace la progettazione di un programma articolando il programma complessivo in più sottoprogrammi, ognuno di quali realizza un compito preciso e limitato.

Infatti, con l'uso dei sottoprogrammi è possibile:

- rendere più semplice la progettazione e comprensione del programma;
- progettare, codificare e verificare ad uno ad uno i singoli sottoprogrammi;
- riutilizzare in un programma diverso un sottoprogramma già codificato e verificato;
- limitare al minimo gli errori dovuti ad interazioni non previste tra parti diverse del programma (*effetti collaterali*).

## 7.2 Progettazione dei sottoprogrammi

Nel progettare un sottoprogramma è necessario precisare **compito** e **flusso di dati**.

- Quale **compito** il sottoprogramma realizza
- Qual è il **flusso di dati** tra il sottoprogramma ed il codice che lo ha attivato. In particolare:
  - Quali sono i *dati forniti* dal *chiamante* in ingresso al sottoprogramma
  - Quali sono i *dati in uscita* dal sottoprogramma verso il *chiamante*

**Flusso dei dati**

Il passaggio dei dati in ingresso dal programma chiamante alla funzione avviene attraverso un insieme di variabili del sottoprogramma, dette **argomenti** o **parametri formali**. Esse sono destinate ad ospitare i dati in ingresso al sottoprogramma.

Con la istruzione di chiamata, il chiamante fornisce al sottoprogramma una lista di **parametri effettivi**, costituti dai valori su cui la funzione deve *effettivamente* operare.

**La corrispondenza tra parametri effettivi e formali è fissata per ordine.**
Nel definire un sottoprogramma che svolge un particolare compito, il programmatore deve quindi specificare:

- Il nome del sottoprogramma
- L'insieme dei dati in ingresso
- L'insieme dei dati in uscita

Queste sono le uniche informazioni necessarie al chiamante per attivare il sottoprogramma. Costituiscono l'**interfaccia** del sottoprogramma.

### 7.2.1 Astrazione

Il ***chiamante*** *non ha la necessità di conoscere la struttura interna del* ***sottoprogramma***. Di fatto, il sottoprogramma implementa delle funzionalità elaborative e mette a disposizione un'interfaccia che specifica come può essere impiegato, senza che ci sia bisogno di alcuna conoscenza della struttura interna.

Il meccanismo dei sottoprogrammi realizza un importante livello di **astrazione**: l'uso del sottoprogramma da parte del chiamante è possibile se è noto che cosa il sottoprogramma è in grado di fare e quali sono i **dati** che sono *scambiati*, ma ignorando totalmente *come* sarà effettuata l'elaborazione.

## 7.3 Funzioni e Procedure

I sottoprogrammi si dividono in due tipologie: **funzioni** e **procedure**.

### 7.3.1 Funzioni

- Una **funzione** è un sottoprogramma che, sulla base dei valori dei suoi parametri di ingresso calcola un unico risultato (*valore di ritorno*).
- Una **procedura** è un sottoprogramma che effettua una elaborazione che comunica con il chiamante mediante un inseme di parametri di ingresso e di uscita (di solito non prevede il valore di ritorno).

Il valore di ritorno, restituito dal sottoprogramma, viene attribuito al nome stesso della funzione e non è assegnato ad uno dei parametri. *I parametri sono solo di input.*

Nella istruzione che invoca il sottoprogramma il nome della funzione compare all'interno di una espressione da valutare. La valutazione dell'espressione avvia l'esecuzione del sottoprogramma.

**Chiamata di una funzione**

- La chiamata di una funzione avviene all'interno di un'espressione in cui compare il nome della funzione seguito dai parametri effettivi tra parentesi tonde.
- Nell'espressione la funzione partecipa fornendo un valore del tipo restituito.
- La valutazione dell'espressione avvia l'esecuzione della funzione.
- Come parametri effettivi possono essere presenti anche espressioni (ovviamente, del tipo assegnato al parametro formale corrispondente).

**Esecuzione di una funzione**

1. Nel programma chiamante, la valutazione di un'espressione attiva la chiamata della funzione;
2. All'atto della chiamata, i parametri effettivi vengono valutati ed assegnati ai rispettivi parametri formali;
3. L'esecuzione del programma chiamante viene sospesa e il controllo viene ceduto alla funzione;
4. Inizia l'esecuzione della funzione: i parametri formali sono inizializzati con i valori dei parametri effettivi;
5. Le istruzioni della funzione sono eseguite;
6. L'ultima istruzione della funzione è un'istruzione speciale (detta di **return**) che fa terminare l'esecuzione della funzione, ridare il controllo al programma chiamante e restituire il valore calcolato (che viene inoltrato al chiamate tramite il nome della funzione);
7. Continua la valutazione dell'espressione nel programma chiamante sostituendo al nome della funzione il valore restituito.

### 7.3.2 Procedure

- Nei sottoprogrammi di tipo **procedura** il flusso di dati con il chiamante si realizza solo tramite i parametri.
- Non ci sono quindi restrizioni al numero di parametri di ingresso e di uscita.
- L'istruzione che invoca il sottoprogramma contiene il nome della procedura seguito dalla lista dei parametri effettivi.
- L'esecuzione dell'istruzione avvia l'esecuzione della procedura.

**Esecuzione di una procedura**

1. Nel programma chiamante, l'esecuzione dell'istruzione attiva la chiamata della procedura;
2. All'atto della chiamata, i parametri effettivi vengono valutati ed assegnati ai rispettivi parametri formali;

3. L'esecuzione del programma chiamante viene sospesa e il controllo viene ceduto alla procedura;
4. Inizia l'esecuzione della procedura: i parametri formali sono inizializzati con i valori dei parametri effettivi;
5. Le istruzioni della procedura sono eseguite;
6. L'ultima istruzione della procedura è un'istruzione speciale (detta di **return**) che fa terminare l'esecuzione della funzione e ridare il controllo al programma chiamante ( *"Quali differenze rispetto alla funzione?"*;
7. L'istruzione nel chiamante che ha attivato la procedura termina e si procede ad eseguire l'istruzione successiva.

## 7.4 Ambiente dei sottoprogrammi

L'insieme delle variabili definite in un sottoprogramma può dividersi in due gruppi:

- **Parametri formali**: utilizzati per gestire il *flusso di dati* con il chiamante
- Altre variabili utilizzate per l'implementazione (*es. indici, variabili di appoggio, ecc.*)

L'insieme di queste variabili viene definito **ambiente della funzione**.

Analogamente, l'insieme delle variabili *definite nel chiamante* costituisce l'**ambiente del chiamante**.

## 7.5 Ambienti e visibilità

- *"Quale relazione esiste tra ambiente del chiamante e ambiente della funzione?"*
- *"Il sottoprogramma ha la **visibilità** (cioè, può fare uso) delle variabili del chiamante e viceversa?"*

Sono **due ambienti distinti** per cui:

- Le variabili del chiamante non sono visibili dal sottoprogramma e viceversa.
- Nei due ambienti possono quini esistere variabili con lo stesso nome, ma sono **due variabili distinte e separate**.
- L'unico canale per scambiarsi dati è quindi fornito dallo scambio di parametri.

Si dice che le variabili sono **locali** al modulo: sono cioè utilizzabili solo all'interno del modulo (programma principale o sottoprogramma) in cui sono definite.

**Variabili globali**

Variabili visibili da tutti i moduli, definite al di fuori del programma principale e di ogni sottoprogramma.
Non si usano perché: tramite le **variabili globali**, funzioni differenti possono realizzare uno scambio di dati che *non è visibile e chiaramente definito* come invece accade tramite il passaggio di parametri. ***Conseguenze***:

- Eventuali errori legati all'uso delle variabili globali (*errata inizializzazione o aggiornamento da parte di qualche funzione*) sono difficilmente individuabili.
- Le funzioni non sarebbero facilmente riutilizzabili in altri programmi.

## 7.6 Associazione dei parametri

**L'associazione avviene per ordine**:

1. Il numero dei parametri formali ed il numero dei parametri effettivi devono essere uguali;
2. Il parametro effettivo i-mo è associato all'i-mo parametro formale;
3. Il tipo del i-mo parametro effettivo deve essere lo stesso del i-mo parametro formale.

### 7.6.1 Tecniche di associazione

Esistono due tecniche principali di associazione tra i parametri:

- **Associazione per valore**:
  - Detta anche **passaggio per valore** (o *by value*): il valore del parametro effettivo viene copiato del parametro formale.
  - Il parametro formale costituisce quindi una **copia locale** del parametro effettivo.
  - Ogni modifica fatta sul parametro formale non si riflette sul parametro effettivo.
- **Associazione per riferimento**:
  - Detta anche **passaggio per riferimento** (o *by reference*): al parametro viene assegnato il riferimento del parametro effettivo.
  - In questo modo, al sottoprogramma è possibile accedere al supporto che ospita il parametro effettivo e fare delle modifiche che saranno poi visibili al programma chiamante.
  - In altre parole, qualunque modifica effettuata sul parametro formale avrà effetto sul parametro effettivo corrispondete.

## 7.7 Struttura di una funzione in C

In C esiste un unico tipo di sottoprogrammi: la **funzione**. Le **procedure** sono rese come *particolari funzioni*.

Sono riconoscibili due parti:

- l'**interfaccia** (o *intestazione*): riporta le informazioni principali relative alla funzione: *nome, tipo, restituito, parametri di ingresso*.
- il **blocco**: costituito da una **parte dichiarativa** (*variabili locali*) e una **parte esecutiva** (*istruzioni*).

Il **nome della funzione** identifica univocamente la funzione e ha gli stessi vincoli dei nomi delle variabili. Il nome deve cominciare con una lettera che può essere seguita da una combinazione di lettere, cifre, underscore.

#### Parte esecutiva

- La **parte esecutiva** contiene l'*insieme di istruzioni* che implementa l'operazione che la funzione deve realizzare.
- Le **istruzioni** *lavorano sull'insieme* formato dai parametri di ingresso e dalle variabili definite all'interno.
- Le **istruzioni** possono essere *costrutti di qualunque tipo* (calcolo e assegnazione, I/O, selezioni, cicli, commenti, linee vuote, chiamate di altre funzioni).
- Al termine c'è un'istruzione di **return** il cui scopo è di:
  - **terminare** l'esecuzione della funzione
  - **restituire** il valore tra parentesi come valore della funzione.

Anche il blocco identificato da **main** è una funzione a tutti gli effetti. *Main* viene chiamata dal Sistema Operativo all'atto dell'esecuzione del programma; il flusso di dati avviene con il S.O., sia per i *parametri effettivi in ingresso*, sia per il valore restituito da **return**.

### 7.7.1 Prototipo di una funzione

- Come per le variabili, anche le funzioni devono essere *definite prima di essere usate.*
- Nel caso ci siano più funzioni, il **main** andrebbe in fondo al file sorgente, rendendo meno leggibile il codice.
- In effetti, per poterle gestire correttamente, il compilatore ha bisogno solo delle informazioni presenti nell'interfaccia della funzione.
- Quindi è possibile anticipare al **main** solo le *interfacce* delle funzioni (**prototipi**) e inserire dopo il **main** le *definizioni* delle funzioni.
- Il **prototipo** è formato dall'*intestazione* della funzione terminato con *';'*.

#### Funzioni che restituiscono void

Il C una **procedura** viene *definita* come una funzione che non restituisce valori. Questo si realizza tramite il tipo **void**. Può essere presente l'istruzione **return** che in questo caso ha **solo** la *funzione di terminare l'esecuzione* della funzione.

### Passaggio per riferimento

Un **parametro formale** che debba essere *passato per* **riferimento** viene dichiarato nella lista dei parametri formali anteponendo un '*' al nome del parametro.

All'interno del sottoprogramma, quando si *utilizza uno dei parametri formali passati per riferimento*, il nome continua ad essere preceduto da un '*'.

**Nella chiamata**, *al parametro effettivo* corrispondente ad un parametro formale passato per riferimento è anteposto un '&' al nome.

# Capitolo 8

# I pattern di base

## 8.1 Design pattern

Un design pattern può essere definito una soluzione progettuale generale a un problema ricorrente, non è una libreria di software ma un **modello** da applicare per risolvere un problema che può presentarsi in diverse situazioni durante la progettazione e lo sviluppo del software.

## 8.2 Progettare un pattern

Progettare un pattern significa definire un algoritmo sufficientemente generale da poter essere utilizzato per tutti i problemi proposti, a meno di piccole modifiche per adattarli alla specifica soluzione.

#### L'accumulatore

La variabile accumulatore ha l'obiettivo di conservare il risultato parziale (basata solo sugli elementi già analizzati) del calcolo della proprietà richiesta.

## 8.3 Algoritmi di base su sequenza di dati

### 8.3.1 Sequenze senza memorizzazione

E' la classe di problemi informatici che impiegano sequenze di dati, e che non necessitano di memorizzare la sequenza.

- **Sequenze con terminatore**: La terminazione della sequenza si indica mediante l'inserimento in coda di un elemento fittizio, detto tappo (o elemento di fine sequenza), che attesta che la sequenza S termina con l'elemento immediatamente precedente.

  Il terminatore C è scelto in modo da essere immediatamente identificabile, come un elemento **non appartenente** alla sequenza stessa.

- **Sequenze con lunghezza nota**: Questo tipo di sequenza prevede che la lunghezza sia esplicitamente codificata.

### 8.3.2 Generazione di sequenze

La generazione di una sequenza è un problema informatico molto più frequente e importante da trattare.

- La generazione di una sequenza può essere trattata introducendo una opportuna funzione generatrice g che ha l'obiettivo di creare per passi successivi tutti gli elementi della sequenza stessa.
- In particolare, da un valore iniziale x1 detto seme, la funzione generatrice ottiene il valore del secondo elemento x2 = g (x1), e, analogamente, applicazioni successive della funzione generatrice, forniscono progressivamente i valori degli elementi successivi.

### 8.3.3 Visita di sequenze

La visita è una operazione che consiste nel raggiungere tutti gli elementi di una sequenza in ordine prefissato, in genere dal primo all'ultimo elemento. La visita è finalizzata a realizzare un'operazione preassegnata:

- Caso banale: stampa del valore corrente
- Proprietà cumulativa da verificare sul valore corrente
- Proprietà caratteristica per effettuare la selezione dell'elemento corrente.

**Tipologie di visita**

- Visita di una sequenza **con tappo**
- Visita di una sequenza di **lunghezza nota**
- Accumulazione su una sequenza **con tappo**
- Accumulazione su una sequenza di **lunghezza nota**
- Selezione di un elemento in una sequenza **con tappo**
- Selezione di un elemento in una sequenza di **lunghezza nota**

### 8.3.4 Accumulazione su una sequenza

Come detto, la visita è solitamente associata alla necessità di calcolare delle proprietà cumulative sulla sequenza in esame.

Tali proprietà possono essere molto diverse a seconda delle specifiche esigenze di elaborazione

Una tipica situazione si verifica quando bisogna fare delle totalizzazioni (dette accumulazioni) sugli elementi che compongono la sequenza.

### 8.3.5 Selezione di un elemento in una sequenza

La selezione di uno specifico elemento è un problema tipico dell'informatica.

L'elemento da selezionare deve soddisfare particolari condizioni, una specifica proprietà

La progettazione di un pattern per la selezione si sviluppa naturalmente dal pattern per la visita della sequenza.

Ad ogni iterazione si deve controllare se è verificata una data condizione **ConSel**, specifica per la risoluzione del particolare problema, al fine di selezionare o meno l'elemento corrente.

## 8.4 Relazione tra vettori e sequenze

Come visto, la **sequenza** è una successione ordinata di elementi (in cui l'ordine è importante, a differenza degli insiemi), tra i quali sia possibile distinguere un primo, un secondo, un terzo...

Un **array** è un'aggregazione di elementi del medesimo tipo che costituisce una collezione indicizzata:

- La collezione si dice indicizzata perché ad ogni elemento è associato un valore, detto indice, che ne permette la sua identificazione.
- L'indice è usato dalla funzione di accesso per accedere ad un particolare elemento della collezione.

### 8.4.1 Pattern su sequenze con memorizzazione

Un vettore ha una cardinalità stabilita all'atto della propria dichiarazione (vettori statici) e può ospitare un numero massimo di elementi che il programmatore deve specificare nel momento in cui realizza il programma.

Spesso, all'atto della esecuzione, il numero effettivo di elementi presenti che nel vettore è **inferiore alla sua cardinalità.**

Nasce pertanto un problema di come termina il vettore, cioè occorre stabilire quale sia l'ultimo elemento del vettore in questione. Due alternative:

- tappo
- informazione esterna, destinata ad accogliere il numero degli elementi rappresentativi del vettore (detta riempimento)