



UNIVERSITÀ DEGLI STUDI DI SALERNO  
**DIPARTIMENTO DI INFORMATICA**  
**DIPARTIMENTO DI ECCELLENZA**

# Università di degli Studi di Salerno

# Dipartimento di Informatica

Programmazione ad Oggetti

a.a. 2023-2024

Introduzione alla OOP

Docente: Prof. Massimo Ficco

E-mail: *mficco@unisa.it*

# Ciclo di Vita del Software (CVS)

Un modello del ciclo di vita del software (CVS) è una caratterizzazione descrittiva o prescrittiva di come un sistema software viene o dovrebbe essere sviluppato

(W. Scacchi - *Encyclopedia of Software Engineering* Vol. II pag. 860)



# Fasi di un CVS: una vista di alto livello

**Definizione:** si occupa del **cosa**.

Determinazione dei requisiti, informazioni da elaborare, funzioni e prestazioni attese, comportamento del sistema, interfacce, vincoli progettuali, criteri di validazione.

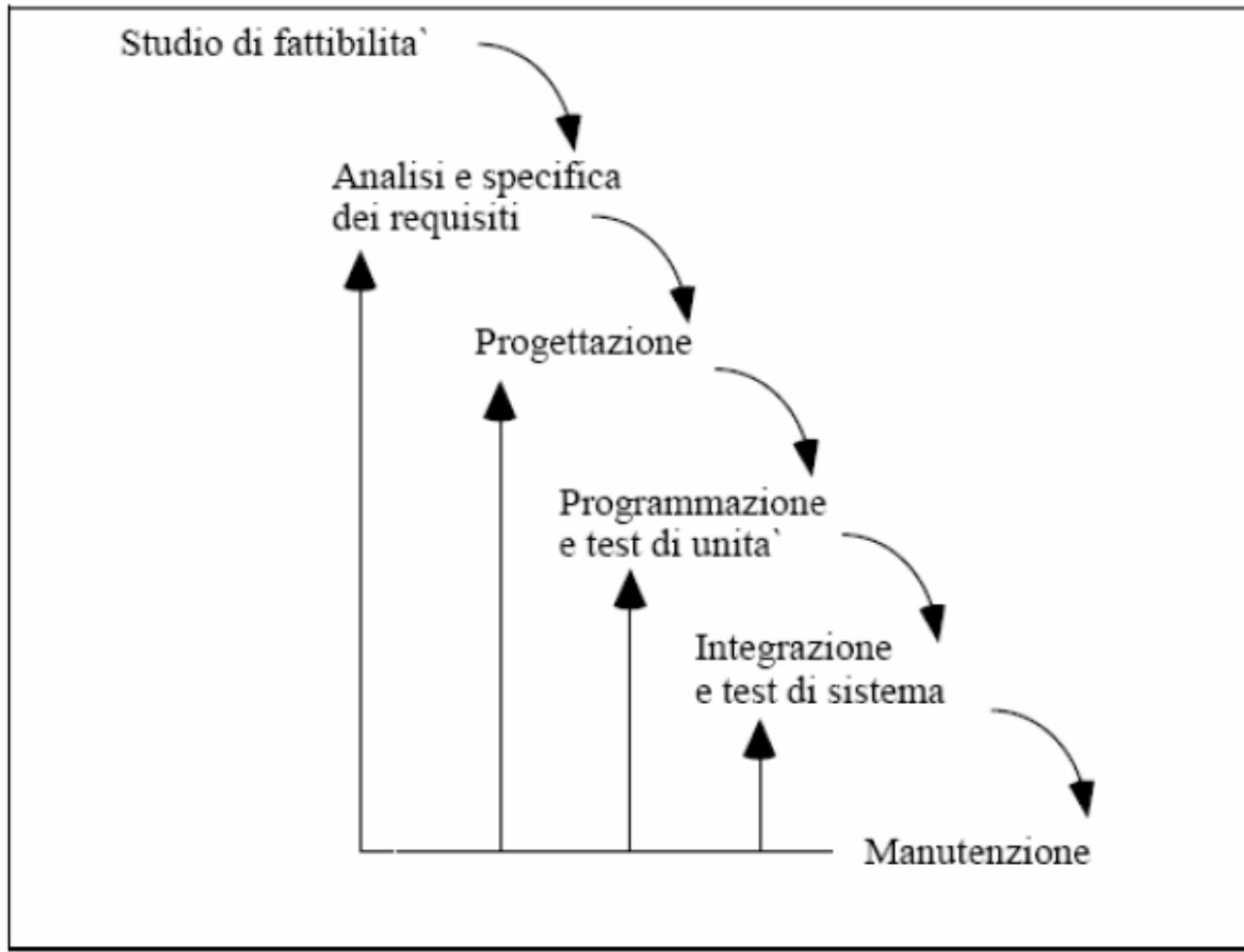
**Sviluppo:** si occupa del **come**

Definizione del progetto, dell'architettura software, della strutturazione dei dati e delle interfacce e dei dettagli procedurali; traduzione del progetto nel linguaggio di programmazione; collaudi.

**Manutenzione:** si occupa delle **modifiche** correzioni, adattamenti, miglioramenti, prevenzione.



# Modello a cascata



# Modello a cascata

**Studio di fattibilità:** Valutazione **Costi/Benefici**

Risorse finanziarie e umane

Soluzioni alternative

Tempi e modalità di sviluppo

**Analisi e specifica dei requisiti:** Valutazione **Requisiti Funzionali**

Produzione di un Documento di Specifica dei Requisiti (DSR)

Piano di Test di Sistema (PTS)

**Progettazione:** **Architettura generale** (hardware e software)

Definizione modulare del software e delle funzionalità associate

Produzione Documento di Progetto (DSP)



# Modello a cascata

## Fase di Test

Test di Unità

Test di Integrazione

Test di Sistema

- *Alfa test*
- *Beta test*

## Manutenzione

Correttiva

Adattativa

Perfettiva



## Fase di Sviluppo



**Programmazione Orientata agli Oggetti  
(OOP)**



# Punto di Partenza

I principi ispiratori di un buon progetto software sono:

- *Astrazione*
- *Information Hiding*
- *Riuso del codice*
- *Modularità*



# L'Astrazione

**L'astrazione** è il processo che porta ad estrarre le proprietà rilevanti di un'entità, ignorando i dettagli inessenziali

Le proprietà estratte definiscono una **vista** dell'entità  
Una stessa entità può dar luogo a viste diverse

Esempio: un'automobile

- vista dal venditore:  
prezzo, durata della garanzia, colore, ...
- vista dal meccanico:  
tipo di motore, cilindrata, tipo di olio, ...



# Meccanismi di astrazione (1/2)

Nella progettazione di un sistema software è opportuno adoperare delle tecniche di astrazione per dominare la complessità del sistema da realizzare.

I meccanismi di astrazione più diffusi sono:

- **ASTRAZIONE SUL CONTROLLO** (o funzionale)
- **ASTRAZIONE SUI DATI**



# Meccanismi di astrazione (2/2)

## ASTRAZIONE SUL CONTROLLO (o funzionale)

- Consiste nell'astrarre una data **funzionalità** dai dettagli della sua implementazione;
- E' ben supportata dai linguaggi di programmazione tradizionali tramite il concetto di **sottoprogramma**.

## ASTRAZIONE SUI DATI

- Consiste nell'astrarre le **entità** (oggetti) costituenti il sistema, descritte in termini di una struttura dati e delle operazioni possibili su di essa;
- Può essere realizzata con un uso opportuno delle tecniche di **programmazione modulare** nei linguaggi tradizionali;
- E' supportata da appositi costrutti nei linguaggi di **programmazione ad oggetti**.



# Incapsulamento e *information hiding*

L'**incapsulamento** consiste nel nascondere e proteggere alcune informazioni di un'entità

- Vantaggi:
  - Parti del programma che non devono essere modificate sono inaccessibili



# Riuso del Codice

- **Riuso del Codice** – Pratica di richiamare o invocare parti di codice precedentemente già scritte ogni qualvolta risulta necessario, senza doverle riscrivere daccapo.
- **Soluzioni in C:** uso di funzioni e librerie di funzioni



# Programmazione Modulare

La modularità è l'organizzazione in parti (per moduli) di un sistema, in modo che esso risulti più semplice da comprendere e manipolare

Gran parte dei sistemi complessi sono modulari

Esempio: Un'automobile è suddivisa in più sottosistemi:

- Motore
- Trasmissione
- ...



# Il concetto di modulo

Un modulo di un sistema software è un componente che:

- ▶ Realizza una astrazione
- ▶ È dotato di una chiara separazione tra:
  - **Interfaccia**
  - **Corpo**

L'**interfaccia** specifica “**cosa**” fa il modulo (l'astrazione realizzata) e “**come**” si utilizza.

Il **corpo** descrive il “**come**” l'astrazione è realizzata

**Modulo**

**Interfaccia**

(Visibile dall'esterno)

**Corpo**

(Nascosto all'esterno  
e protetto)



# Modularità: Vantaggi

- **Modularità** - Tecnica di suddividere un progetto software per meglio gestirne la complessità
  - **Modulo:** unità di programma che mette a disposizione risorse e servizi computazionali (dati, funzioni, ...)
  - **Elevata coesione:** le varie funzionalità messe a disposizione da un singolo modulo sono strettamente correlate tra loro
  - **Indipendenza:** sviluppabili separatamente dal resto del programma, con compilazione e testing separati
- Vantaggi:
  - Correzione degli errori facilitata: un errore presente in un modulo può essere corretto modificando soltanto quel modulo



# Modularità e Incapsulamento in C

In C non esiste un apposito costrutto per realizzare un modulo; di solito un modulo coincide con un file con estensione .c

Per esportare le risorse definite in un file (modulo), il C fornisce un particolare tipo di file, chiamato header file (estensione .h), che ne contiene l'interfaccia



# Modularità e Incapsulamento in c

```
void input_array(int a[], int n);  
void output_array(int a[], int n);  
void ordina_array(int a[], int n);  
int ricerca_array(int a[], int n, int elem);  
int minimo_array(int a[], int n);  
...
```

vettore.h

## Modulo vettore

```
#include <stdio.h>  
#include "utils.h" // contiene funzione scambia  
int minimo_i(int a[], int i, int n); // dichiarazione locale
```

vettore.c

```
void input_array(int a[], int n) { ... }  
void output_array(int a[], int n) { ... }  
void ordina_array(int a[], int n) { ... }  
int ricerca_array(int a[], int n, int elem) { ... }  
int minimo_array(int a[], int n) { ... }  
int minimo_i(int a[], int i, int n) { ... } // usata da ordina_array  
...
```

Cliente

*Cliente: può usare le risorse e i servizi esportati dal modulo*



# Modularità e Incapsulamento in C



*Quale problema presenta un programma di questo tipo?*

```
void input_array(int a[], int n);  
void output_array(int a[], int n);  
void ordina_array(int a[], int n);  
int ricerca_array(int a[], int n, int elem);  
int minimo_array(int a[], int n);  
...  
#include <stdio.h>
```

vettore.h

## Modulo vettore

```
#include “utils.h” // contiene funzione scambia  
int minimo_i(int a[], int i, int n); // dichiarazione locale  
  
void input_array(int a[], int n) { ... }  
void output_array(int a[], int n) { ... }  
void ordina_array(int a[], int n) { ... }  
int ricerca_array(int a[], int n, int elem) { ... }  
int minimo_array(int a[], int n) { ... }  
int minimo_i(int a[], int i, int n) { ... } // usata da ordina_array  
...
```

vettore.c

// file main.c

```
# include <stdio.h>  
# include “vettore.h”  
# define MAXELEM 100  
  
int main()  
{  
    ...  
}
```



# Modularità e Incapsulamento in c

```
void input_array(int a[], int n);  
void output_array(int a[], int n);  
void ordina_array(...);  
int ricerca_array(...);  
int minimo_array(...);  
...  
#include <cs50.h>  
#include "vettore.h"
```

vettore.h



I dati di lavoro, ovvero la sequenza di elementi del vettore, sono accessibili anche in altri punti del programma e non solo alle funzioni implementate in vettore.c e in caso di errori è difficile capirne la causa

```
int minimo_i(int a[], int i, int n); // dichiarazione locale  
  
void input_array(int a[], int n) { ... }  
void output_array(int a[], int n) { ... }  
void ordina_array(int a[], int n) { ... }  
int ricerca_array(int a[], int n, int elem) { ... }  
int minimo_array(int a[], int n) { ... }  
int minimo_i(int a[], int i, int n) { ... } // usata da ordina_array  
...
```

```
# define MAXELEM 100  
  
int main()  
{  
    ...  
}
```



# Modularità

La soluzione è encapsulare dati e funzioni, così che solo determinate funzioni possono lavorare su specifici dati.

```
void input_array(int a[], int n);
void output_array(int a[], int n);
void ordina_array(int a[], int n);
int ricerca_array(int a[], int n, int elem);
int minimo_array(int a[], int n);
```

vettore.h

```
#include <stdio.h>
#include "utils.h" // contiene funzione scambia
int minimo_i(int a[], ...);
void input_array(int a[], ...);
void output_array(int a[], int n) { ... }
void ordina_array(int a[], int n) { ... }
int ricerca_array(int a[], int n, int elem) { ... }
int minimo_array(int a[], int n) { ... }
int minimo_i(int a[], int i, int n) { ... } // usata da ordina_array
...
```

## Modulo vettore

vettore.c

```
// file m...
```

```
# include <stdio.h>
# include "vettore.h"
```

Come è possibile realizzare questa tecnica  
di information hiding?

```
{ ...
}
```



# Tipi di dati astratti

Il concetto di **tipo** di dato in un linguaggio di programmazione tradizionale è quello di insieme dei valori che può assumere un dato (una variabile).

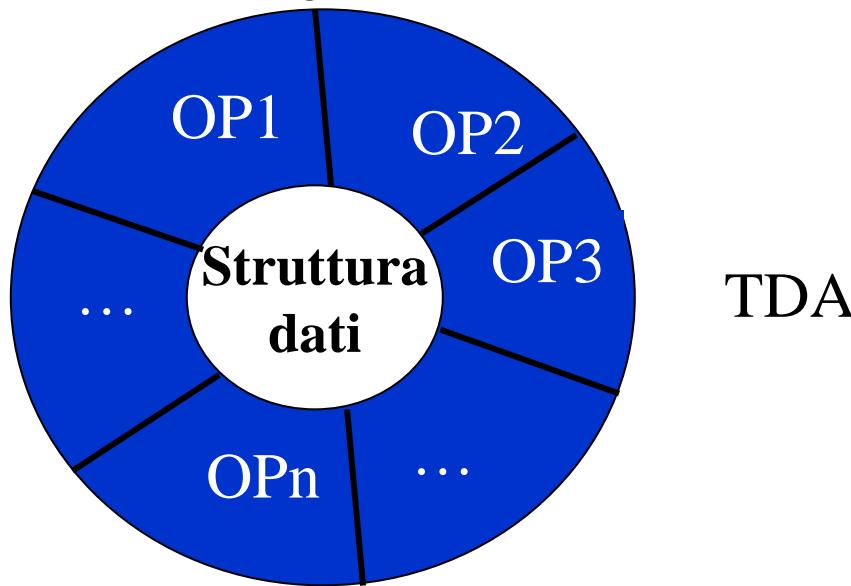
- ▶ Nel linguaggio C
  - ▶ Tipi di dati primitivi: forniti direttamente dal linguaggio: int, char, float, double
  - ▶ Dati aggregati: array, strutture, enumerazioni, unioni
  - ▶ Puntatori



# Tipi di dati astratti

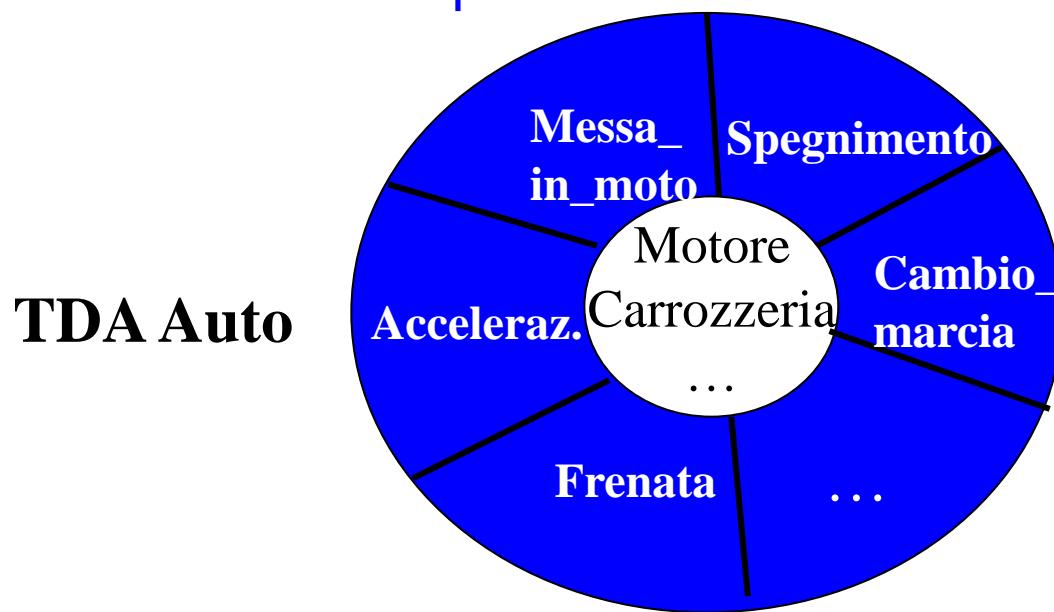
Il **tipo di dati astratto** (TDA) estende questa definizione, includendo anche l'insieme di **tutte e sole le operazioni** possibili su dati di quel tipo. La struttura dati “concreta” è **incapsulata** nelle operazioni su di essa definite.

**Tipi di Dati Astratti (ADT)** - Definiti distinguendo **Specificità** e **implementazione** (nascosta al programmatore, seguendo il principio dell'**Incapsulamento** - information hiding)



# TDA (2/2)

Non è possibile accedere alla struttura dati encapsulata (né in lettura né in scrittura) se non attraverso le operazioni definite su di essa  
Esempio:



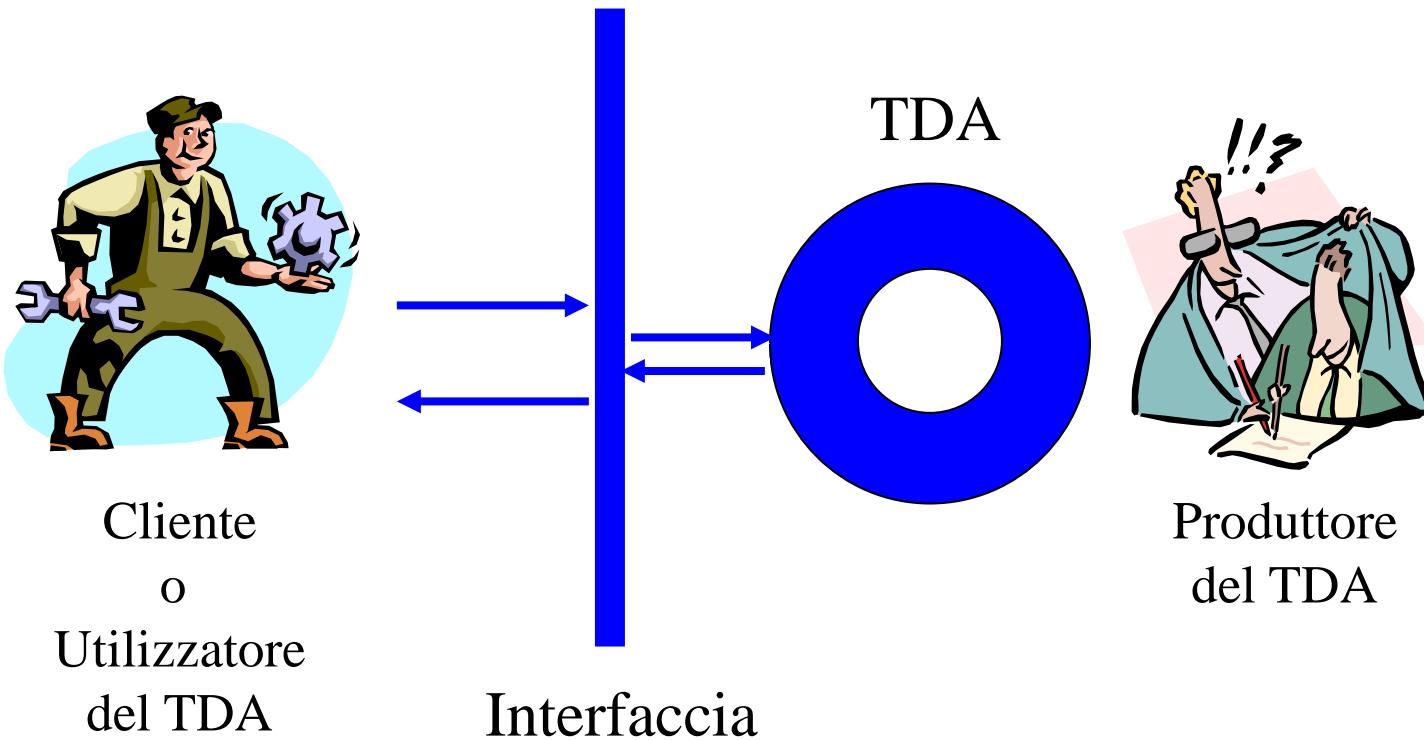
*Un vantaggio:* la struttura dati interna non può venire alterata da operazioni scorrette da parte dell'utente, in quanto ad essa si accede solo tramite le operazioni previste e realizzate dal produttore



# Interfaccia, uso e realizzazione

**Interfaccia**: specifica del TDA, descrive la parte direttamente accessibile dall'utilizzatore

**Realizzazione**: implementazione del TDA



# Implementazione ADT in C

## Implementazione ADT in C - Strutture:

- Definizione: tipo di dati composito che include un elenco di variabili fisicamente raggruppate in un unico blocco di memoria

```
struct my_vettore_t {
    int arr_size;
    int arr[];
};

typedef struct my_vettore_t *vettore_ptr;
void input_array(vettore_ptr);
void output_array(vettore_ptr);
void ordina_array(vettore_ptr);
int ricerca_array(vettore_ptr, int elem);
int minimo_array(vettore_ptr);
...
```

```
// file main.c

#include <stdio.h>
#include "vettore.h"
#define MAXELEM 100

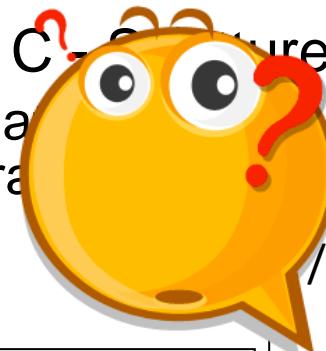
int main()
{
    int vet_size = 10;
    vettore_ptr vet =
        malloc(sizeof(struct my_vettore_t)
        + fam_size));
    ...
}
```



# Implementazione ADT in C

## Implementazione ADT in C - Structure:

- Definizione: tipo di dati che rappresenta variabili fisicamente raggruppate



*Abbiamo raggiunto il nostro obiettivo?*

/ file main.c

```
struct my_vettore_t {
    int arr_size;
    int arr[];
};

typedef struct my_vettore_t *vettore_ptr;
void input_array(vettore_ptr);
void output_array(vettore_ptr);
void ordina_array(vettore_ptr);
int ricerca_array(vettore_ptr, int elem);
int minimo_array(vettore_ptr);
...
```

```
# include <stdio.h>
# include "vettore.h"
# define MAXELEM 100

int main()
{
    int vet_size = 10;
    vettore_ptr vet =
        malloc(sizeof(struct my_vettore_t)
              + fam_size));
    ...
}
```



# Implementazione ADT in C

Implementazione ADT in C - Structure:

- Definizione: tipo di dati che rappresenta variabili fisicamente raggruppate

*Abbiamo raggiunto il nostro obiettivo?*

/ file main.c

```
struct my_vettore_t {  
    int arr_size;  
    int arr[];  
};  
  
typedef struct my_vettore_t vettore_ptr;  
  
void input_array(vettore_ptr);  
void output_array(vettore_ptr);  
void ordina_array(vettore_ptr);  
int ricerca_array(vettore_ptr, int elem);  
int minimo_array(vettore_ptr);  
...  
# include <stdio.h>
```

Se definiamo la funzione all'interno della struct?

```
{  
    int vet_size = 10,  
    vettore_ptr vet =  
        malloc(sizeof(struct  
        + fam_size));  
    ...  
}
```



# Implementazione ADT in C

## Implementazione ADT in C - Strutture:

- Definizione: tipo di dati composito che include un elenco di variabili fisicamente raggruppate in un unico blocco di memoria

```
struct my_vettore_t {  
    int arr_size;  
    int arr[];  
    void input_array(vettore_ptr);  
    void output_array(vettore_ptr);  
    void ordina_array(vettore_ptr);  
    int ricerca_array(vettore_ptr, int elem);  
    int minimo_array(vettore_ptr);  
};  
typedef struct my_vettore_t *vettore_ptr;  
...
```

vettore.h

```
// file main.c  
  
# include <stdio.h>  
# include "vettore.h"  
# define MAXELEM 100  
  
int main()  
{  
    int vet_size = 10;  
    vettore_ptr vet =  
        malloc(sizeof(struct my_vettore_t)  
        + fam_size));  
    ...  
}
```



# Implementazione ADT in C



In

In C non è possibile definire funzioni in una struttura, al massimo possano introdurre dei puntatori a funzione.

aria

```
struct my_vettore_t {
    int arr_size;
    int arr[];
    void input_array(vettore_ptr);
    void output_array(vettore_ptr);
    void ordina_array(vettore_ptr);
    int ricerca_array(vettore_ptr, int elem);
    int minimo_array(vettore_ptr);
};

typedef struct my_vettore_t *vettore_ptr;
```

// file main.c

```
# include <stdio.h>
# include "vettore.h"
# define MAXELEM 10

int main()
{
    int vet_size = 10;
    vettore_ptr vet =
        malloc(sizeof(struct my_vettore_t)
        + fam_size));
    ...
}
```



# Implementazione ADT in C

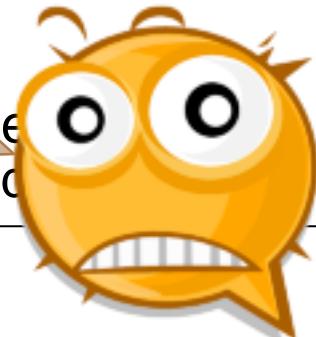
## Implementazione ADT in C

- Definizione di variabili Questa soluzione implica una sintassi molto complessa, ma non risolve il problema (i dati sono ancora accessibili).

```
struct my_vettore_t {  
    int arr_size;  
    int arr[];  
    void input_array(vettore_ptr);  
    void output_array(vettore_ptr);  
    void ordina_array(vettore_ptr);  
    int ricerca_array(vettore_ptr, int elem);  
    int minimo_array(vettore_ptr);  
};  
typedef struct my_vettore_t *vettore_ptr;  
...
```

vettore.h

```
# include <stdio.h>  
# include "vettore.h"  
# define MAXELEM 100  
  
int main()  
{  
    int vet_size = 10;  
    vettore_ptr vet =  
        malloc(sizeof(struct my_vettore_t)  
        + fam_size));  
    ...  
}
```



# Programmazione ad Oggetti

La programmazione ad oggetti rappresenta un ulteriore sviluppo rispetto alla programmazione modulare.

Nella OOP esiste un nuovo tipo di dato, la **classe**, che rappresenta un implementazione di una astrazione sui dati.

Questo tipo di dato serve a modellare un insieme di oggetti dello stesso tipo.

Un **oggetto** è caratterizzato da un insieme di attributi e un insieme di funzionalità (metodi) che operano sugli attributi dell'oggetto stesso.



# Programmazione ad Oggetti

- ▶ La programmazione orientata agli oggetti è un paradigma di programmazione che permette di definire oggetti software, istanze di classi, in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi.



NOME CLASSE
(Caratteristiche della classe e variabili usate)
METODI
(Operazioni che la classe deve eseguire, di solito funzioni o procedure)

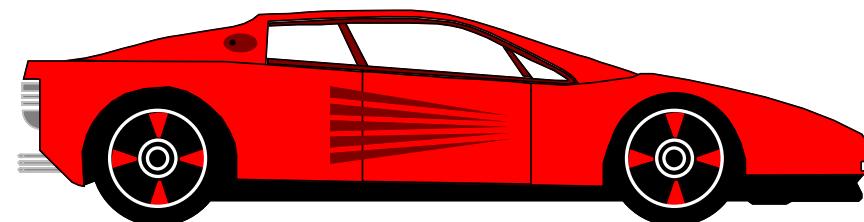
→	LIBRO	OggettoLibro
→	Titolo, Autore, Editore, Codice, Prezzo, Genere	"Harry Potter", "J. K. Rowling", "Salani Ed.", ...
→	Catalogazione(), Prestiti(), Restituzione()	Catalogazione(), Prestiti(), Restituzione()



# Un esempio di oggetto

## Funzioni      Dati

- |              |                     |
|--------------|---------------------|
| - Avviati    | - Targa             |
| - Fermati    | - Colore            |
| - Accelerata | - Cilindrata motore |
| - ...        | - ...               |



Il conducente interagisce con una interfaccia per effettuare le operazioni consentite sull'automobile:

- Pedale del freno
- Pedale dell'acceleratore
- Leva del cambio
- Sistema di accensione
- ...

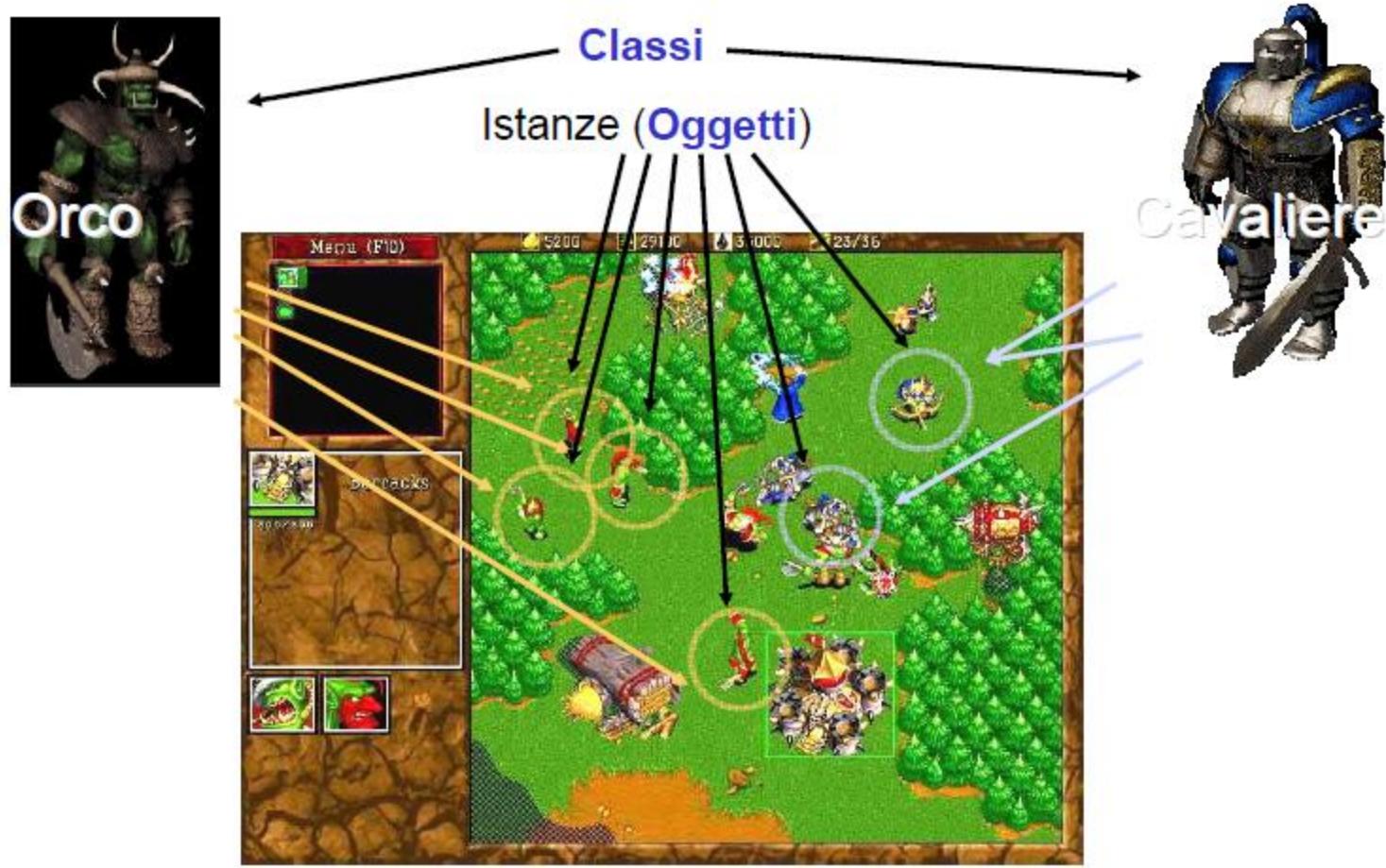


# Programmazione orientata agli oggetti

La programmazione orientata agli oggetti (Object Oriented Programming – OOP) è un paradigma di programmazione, in cui un programma viene visto come un insieme di oggetti che interagiscono tra loro.



# Classi e Oggetti



# Vantaggi della programmazione OO

Rispetto alla programmazione tradizionale, la programmazione orientata agli oggetti (OOP) offre vantaggi in termini di:

- **Astrazione e modularità**: le classi sono i moduli del sistema software;
- **information hiding**: sia le strutture dati che gli algoritmi possono essere nascosti alla visibilità dall'esterno di un oggetto;
- **coesione dei moduli**: una classe è un componente software ben coeso in quanto rappresentazione di una unica entità;
- **disaccoppiamento dei moduli**: gli oggetti hanno un alto grado di disaccoppiamento in quanto i metodi operano sulla struttura dati interna ad un oggetto;
- **riuso**: l'ereditarietà consente di riusare la definizione di una classe nel definire nuove (sotto)classi; inoltre è possibile costruire librerie di classi raggruppate per tipologia di applicazioni;
- **estensibilità**: il polimorfismo agevola l'aggiunta di nuove funzionalità, minimizzando le modifiche necessarie al sistema esistente quando si vuole estenderlo.

