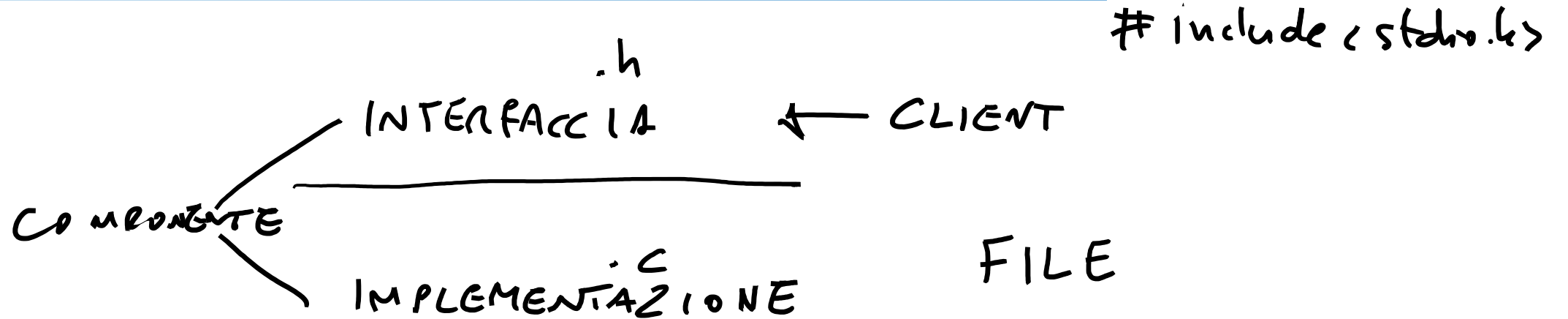


ADT



Obiettivi

- Introdurre le tipologie di componente software e la nozione di Abstract Data Type
- Mostrare come realizzare un ADT in linguaggio C

Componenti software

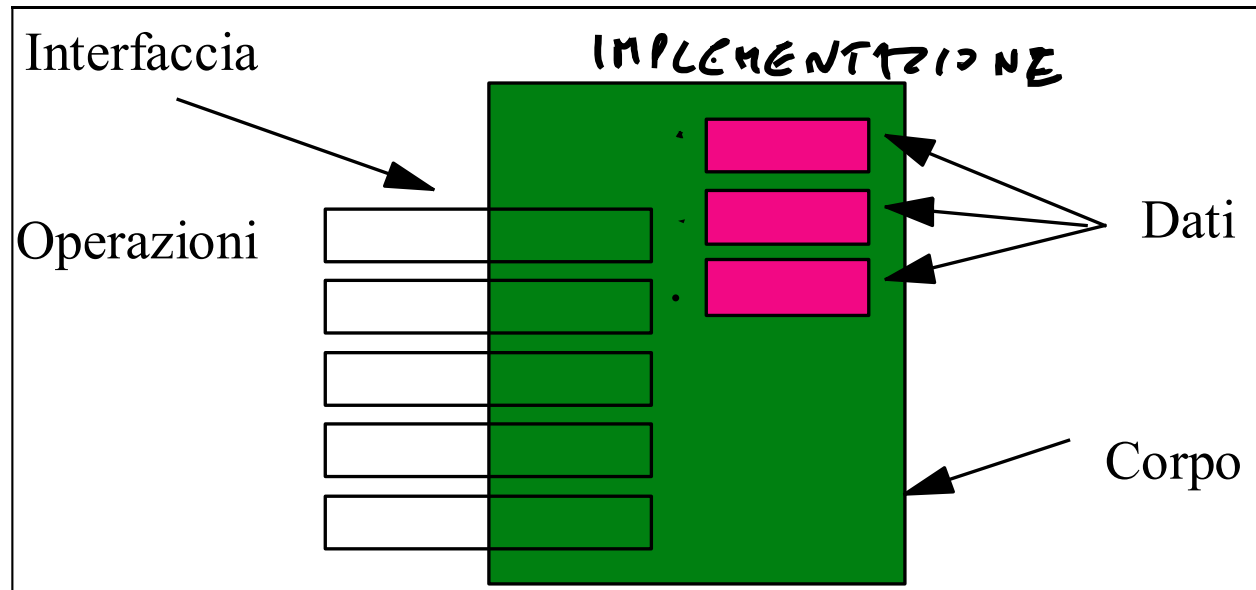
- *Librerie*
- *Astrazioni di dato*
- *Tipo di Dato Astratto (ADT)*

Librerie

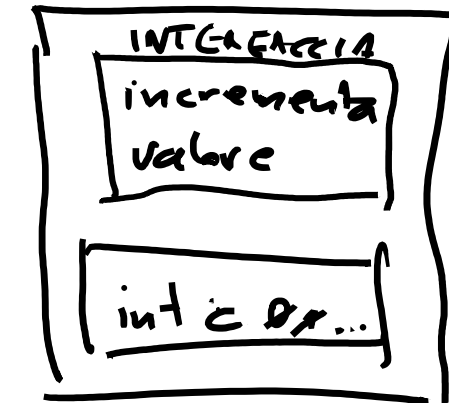
- *Il componente rende visibili procedure e funzioni che non fanno uso di variabili non locali. Il componente è una collezione di **operazioni** (ad esempio, funzioni matematiche) ed eventuali costanti (globali)*
- *Di sistema o “user-defined”, ad esempio:*
 - *math.h*, *stdio.h*, *mylib_ord.h*
- *Per usarle:*
 - `#include <stdio.h>`
 - `#include "mylib_ord.h"`

Astrazioni di dato

- Il componente ha dati locali (nascosti) e rende visibili all'esterno i prototipi delle operazioni invocabili (procedure e funzioni) su questi dati locali, ma non gli identificatori dei dati. Attraverso una di queste operazioni si può assegnare un valore iniziale ai dati locali nascosti.



CONTATTO

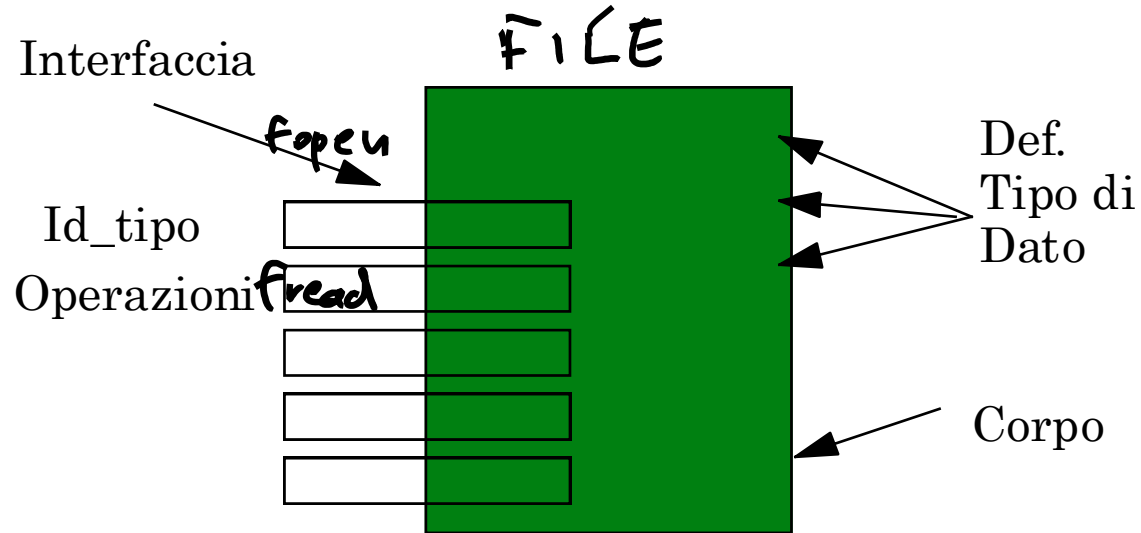


Tipo di dato astratto

(Abstract Data Type – ADT)

- Il componente esporta un **identificatore di tipo T** ed i prototipi delle operazioni eseguibili su dati dichiarati di questo tipo. I “clienti” del componente dichiarano e controllano quindi il tempo di vita delle variabili di tipo T.

#include <stdio.h>



Tipi di dato astratto

Un **tipo di dato astratto (ADT)** definisce una categoria concettuale con le sue proprietà:

- una **definizione di tipo** `typedef struct {`

➤ *implica un dominio, D*

`...
} frazione;`

- un **insieme di operazioni ammissibili su**
stdio.h **oggetti di quel tipo**

`frazione somma (frazione f1, frazione f2);`

`typedef ...
FILE;
FILE* fopen(...);`

OPERAZIONI DI UN ADT

Quali operazioni definire per un ADT?

Frazione nuova Frazione (int num,
int den);

- **costruttori** (costruiscono un oggetto di questo tipo, a partire dai suoi “costituenti elementari”)
- **selettori** (restituiscono uno dei “mattoni elementari” che compongono l’oggetto) int numeratore (Frazione f);
- **predicati** (verificano la presenza di una proprietà sull’oggetto, restituendo vero o falso); int valida (Frazione f);
- **funzioni** (restituiscono un valore calcolato sull’oggetto) Frazione inversa (Frazione f);
- **trasformatori** (cambiano lo stato dell’oggetto) void accrescere (Frazione f);

ADT – Abstract Data Type

- ADT è rappresentato da un'interfaccia, che nasconde l'implementazione corrispondente
- Gli utenti di un ADT utilizzano solo l'interfaccia, ma non accedono (almeno non dovrebbero) direttamente all'implementazione poiché questa in futuro può cambiare
- Tutto ciò è basato sul principio di **information hiding**, ovvero proteggere i "clienti" da decisioni di design che in futuro possono cambiare
- La potenza del concetto di ADT sta nel fatto che l'implementazione è **nascosta** al cliente: viene resa pubblica solo l'interfaccia
- Questo significa che un ADT può essere implementato in vari modi ma, **finché non viene cambiata l'interfaccia, i programmi che lo utilizzano non devono essere alterati**

cliente

```
#include "f.h"
Nome v;
f1(v);
```

INTERFACCIA

f.h

```
typedef
    "Nome";
f1(...);
f2(...);
```

f.c

IMPLEMENTAZIONE

```
#include "f.h"
f1(...) {
}
f2(...) {
}
```


TIPI DI DATO ASTRATTO IN C

In C, un **ADT** si costruisce definendo:

- il nuovo tipo con **typedef**
- una funzione per ogni operazione

\mathbb{Z}

Esempio: ADT contatore

una entità caratterizzata da un valore intero

typedef int **counter**;

con operazioni per

counter c,

reset (&c);

inc (&c);

val (c) → 1

ctr ➤ inizializ. contatore a zero

reset (counter*);

trsf ➤ incrementare il contatore

inc (counter*);

funz valore

int val (counter);

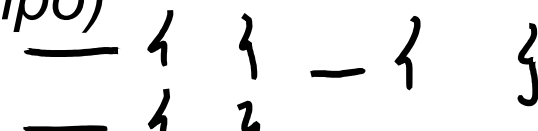
ORGANIZZAZIONE DI ADT IN C

- *Un ADT in C può essere realizzato separandone l'interfaccia e l'implementazione in due file:*

1. **un file header (nomefile.h)**, contenente

- `typedef int counter;`
- **dichiarazione** delle funzioni 

2. **un file di implementazione (nomefile.c)**, contenente

- direttiva `#include` per includere il proprio header (per importare la definizione di tipo)
- **definizione** delle funzioni 

Esempio: ADT counter

1. file header counter.h

```
typedef int counter;
void reset(counter*);
void inc(counter*);
int val(counter c);
```

Definisce in astratto che cos'è un counter e che cosa si può fare con esso

2. file di implementazione counter.c

```
#include "counter.h"
```

```
void reset(counter *c) { *c=0; }
void inc(counter* c) { (*c)++; }
```

+ altre funzioni

Specifica come funziona (quale è l'implementazione) di counter

```
int val(counter c) { return c; }
```

ADT counter: un cliente

Per usare un counter occorre:

- includere il relativo file header
- **definire una o più variabili di tipo counter**
- **operare su tali “istanze” mediante le sole operazioni (funzioni) previste**

main.c

```
#include "counter.h"  INTERFACCIA ADT

int main() {
    counter c1, c2;
    reset(&c1); reset(&c2);
    inc(&c1); inc(&c2); inc(&c2);
    printf("%d", val(c2));
}
```

ADT IN C: LIMITI

- *L'ADT così realizzato funziona, ma molto dipende dall'autodisciplina del programmatore*
- *Non esiste alcuna protezione contro un uso scorretto dell'ADT*

*l'organizzazione suggerisce di operare sull'oggetto solo tramite le funzioni previste, ma **NON riesce a impedire** di aggirarle a chi lo volesse
(ad esempio: `counter c1; c1++;`)*

*typedef struct {
...
} counter;*

- *La struttura interna dell'oggetto è visibile a tutti (nella typedef)*

ADT counter: un cliente “birichino”

```
#include "counter.h"

int main() {
    counter c1, c2;
    reset(&c1); reset(&c2); inc(&c1); c1++;
    inc(&c2); inc(&c2);
}
```

- *Il cliente ha la visibilità dell'organizzazione fisica (un `int`) del dato*
- *Il C non garantisce un livello adeguato di protezione (**information hiding**) e includendo l'header il cliente conosce la struttura (`typedef`)*

ADT IN C: LIMITI

Superare questi limiti è uno degli obiettivi cruciali della programmazione orientata agli oggetti