



Alma Mater Studiorum-Università di Bologna Scuola di Ingegneria

Componenti software dal C agli oggetti

Corso di Laurea in Ingegneria Informatica
Anno accademico 2025/2026

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)



COMPONENTI SOFTWARE

Che componenti software avete conosciuto finora?

- **componenti statici *senza stato* (librerie)**
 - non hanno stato: solo collezioni di funzioni *prive di effetti collaterali*
 - ne basta una sola copia in ogni progetto: tutti possono usarla
- **componenti statici *con stato* (moduli come *singleton*)**
 - c'è uno stato *unico e globale* (variabili globali static)
 - possono essere visti come *oggetti singleton*: ne esiste una e una sola copia nel progetto ed è giusto che sia così
- **tipi di dato astratti (ADT)**
 - c'è una *definizione di tipo unica e condivisa* (typedef) + funzioni che operano su quel tipo di dato (header + implementazione)
 - il cliente crea tante variabili (oggetti) di quel tipo quante ne occorrono



COMPONENTI SOFTWARE

Che componenti software avete conosciuto finora?

- **componenti statici senza stato (*librerie*)**

- non hanno stato: solo colle
- ne basta una sola copia in c

Esempi: libreria matematica, libreria stringhe, libreria matrici, etc.

- **componenti statici con stato (*moduli come singleton*)**

- c'è uno stato *unico e globale*
- possono essere visti come una sola copia nel progetto ed è giusto che sia così

Esempi: risorse singleton (centro di stampa, contatore centralizzato, ...)

- **tipi di dato astratti (ADT)**

- c'è una *definizione di tipo* su cui si operano su quel tipo di dato
- il cliente crea tante variabili (oggetti) di quel tipo quante ne occorrono

Esempi: tutti i tipi da voi definiti!
Persone, Frazioni, Liste, ...



COMPONENTI SOFTWARE in C

Che componenti software avete conosciuto finora?

- componenti statici *senza stato* (*librerie*)

Header file con le *dichiarazioni* delle funzioni

zio
ogget

IMPLEMENTAZIONE: file con le *definizioni* delle funzioni

- componenti statici *con stato* (*moduli come singleton*)

Header file con le *dichiarazioni* delle funzioni

so
sing
sia

IMPLEMENTAZIONE: file con le *definizioni* delle funzioni e *variabili statiche* per lo stato

- tipi di dato astratti (ADT)

Header file con le *dichiarazioni* del *tipo* e delle funzioni

cond
der +

IMPLEMENTAZIONE: file con le *definizioni* delle funzioni



COMPONENTI SOFTWARE in C

Che componenti software avete conosciuto finora?

- componenti statici *senza stato* (*librerie*)

Header file con le *dichiarazioni* delle funzioni

USO: il cliente deve solo importare l'header, poi può chiamare le funzioni

- componenti statici *con stato* (*moduli come singleton*)

Header file con le *dichiarazioni* delle funzioni

USO: il cliente deve importare l'header, poi può chiamare le funzioni *nel giusto ordine*

- tipi di dato astratti (ADT)

Header file con le *dichiarazioni* del *tipo* e delle funzioni

USO: il cliente deve sia importare l'header, sia *creare le variabili* (oggetti) da usare con le varie funzioni

LIBRERIE: ESEMPIO in C

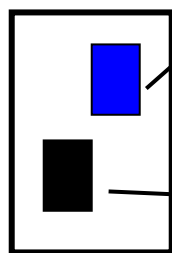
```
#include <stdio.h>
#include "mylib.h"
```

Dichiara la funzione
`int myfun(int x);`

```
int main() {
    int x;
    scanf("%d", &x);
    printf("Esito: %d", myfun(x));
    return 0;
}
```

OSSERVA: *non occorrono operazioni preliminari* per poter usare le funzioni.

Architettura progetto:



mylib.c

main.c

Implementa la funzione
`int myfun(int x);`

MODULI SINGLETON: ESEMPIO in C

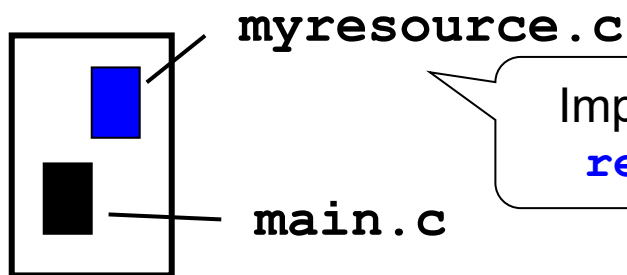
```
#include <stdio.h>
#include "myresource.h"

int main() {
    reset();
    printf("Stato risorsa: %s", getState());
    return 0;
}
```

Dichiara le funzioni **reset** e **getState** che costituiscono l'interfaccia dell'entità «resource»

OSSERVA: non sono necessarie operazioni preliminari, ma *c'è un ordine da rispettare*: occorre *inizializzare la risorsa* prima di usarla.

Architettura progetto:



Implementa le funzioni **reset** e **getState**

ADT: ESEMPIO in C

```
#include <stdlib.h>
#include <stdio.h>
#include "fraction.h"
```

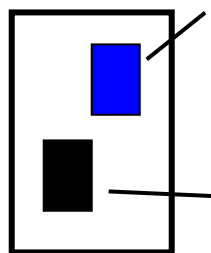
```
int main() {
    Fraction f1;
    Fraction *pf2;
    pf2 = (Fraction*) malloc(sizeof(Fraction));
    initFraction(&f1, 3, 4); initFraction(pf2, 6, 7);
    return 0;
}
```

Dichiara il **tipo** e le *funzioni* che caratterizzano l'entità «frazione»

Occorre preliminarmente *creare le variabili* (o puntatori) che rappresentano i singoli oggetti.
Se puntatori, vanno anche allocati.

MA ATTENZIONE:
la **typedef** svela la
struttura interna
del tipo!

Architettura progetto:



fraction.c

main.c

Implementa le funzioni
initFraction, etc.

Peculiarità: funzioni in C senza argomenti



DICHIARAZIONI VS. DEFINIZIONI DI FUNZIONI IN C

- In C, una funzione dev'essere dichiarata testualmente *prima* di poterla chiamare
- Quindi:
 - o la *definizione* della funzione precede l'uso (ed è nello stesso file)
 - oppure si fa precedere l'uso dalla *dichiarazione* della sua *signature*

```
#include <stdio.h>

int fact(int n){
    return (n==0) ? 1 : n*fact(n-1);
}

int main() {
    printf("fact(%d) = %d", 3, fact(3));
    return 0;
}
```

```
#include <stdio.h>

int fact(int);

int main() {
    printf("fact(%d) = %d", 3, fact(3));
    return 0;
}

int fact(int n){
    return (n==0) ? 1 : n*fact(n-1);
}
```



DICHIARAZIONI VS. DEFINIZIONI DI FUNZIONI IN C: UNA CRITICITÀ

- Però, c'è una *criticità* nel caso di *funzione* dichiarata *senza argomenti* (cioè, con lista di argomenti vuota)
- In tal caso infatti l'interpretazione del C è «*lista di argomenti sconosciuta*», *abolendo di fatto il controllo di tipo!*
 - a riprova di ciò, il programma a lato compila perfettamente, nonostante la signature di **fact** in fase di *dichiarazione* stia omettendo l'argomento **int**!

```
#include <stdio.h>

int fact();

int main() {
    printf("fact(%d) = %d", 3, fact(3));
    return 0;
}

int fact(int n){
    return (n==0) ? 1 : n*fact(n-1);
}
```



DICHIARAZIONI VS. DEFINIZIONI DI FUNZIONI IN C: UNA CRITICITÀ

- Questa criticità diventa **particolarmente problematica** se la **funzione da dichiarare è effettivamente senza argomenti**
- In tal caso, infatti, l'interpretazione del C consente di **chiamarla con qualsiasi lista di argomenti (orrore!)**
 - nel programma a lato, che compila perfettamente, `getValue` era intesa come funzione *senza argomenti*
 - MA in realtà si può chiamarla con qualunque lista di argomenti perché il controllo è stato di fatto *disabilitato*!

```
#include <stdio.h>

int getValue(){
    return 3;
}

int main() {
    printf("valore = %d\n", getValue());
    printf("valore = %d\n", getValue(8, "ciao"));
    return 0;
}
```



DICHIARAZIONI VS. DEFINIZIONI DI FUNZIONI IN C: UNA CRITICITÀ

- Per esplicitare che *non ci devono essere argomenti*, in C la funzione va dichiarata con *lista di argomenti void*
 - il programma a lato ora dà, giustamente, errore!
 - In C++, invece, una lista di argomenti vuota indica effettivamente *assenza di argomenti*, come è logico
 - **Visual Studio** non supporta direttamente il C, perciò lo avete usato in modalità C++

```
#include <stdio.h>

int getValue(void){
    return 3;
}


int main() {
    printf("valore = %d\n", getValue());
    printf("valore = %d\n", getValue(8,"ciao"));
    return 0;
}
```

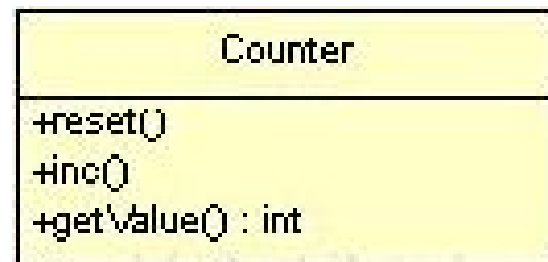
Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:17:29: error: too many arguments to function 'getValue'
   17 |     printf("valore = %d\n", getValue(8,"ciao"));
      |                               ^~~~~~
main.c:11:5: note: declared here
   11 | int getValue(void){
```

Un caso concreto: il contatore

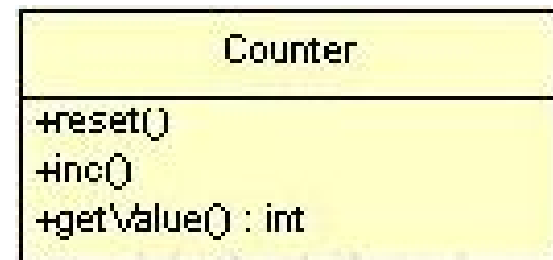
UN ESEMPIO CONCRETO: IL “CONTATORE”

- OBIETTIVO: progettare un *contatore*
 - già, ma... cos'è un «contatore»? 
 - non si può andare «a sentimento»: serve una specifica precisa
- Come specificare un «*contatore*» ?
 - NON riferendosi a una tecnologia, o (peggio!) a codice: serve un modo *tecnologicamente neutro* per esprimere attese e requisiti
 - occorre riferirsi al *comportamento osservabile*: che servizi ci aspettiamo che abbia? Come ci aspettiamo di poterlo usare?
- È utile usare un *diagramma UML*
 - linguaggio grafico, adatto a specificare struttura e funzionamento di «entità»
 - Il diagramma di struttura a lato indica i *servizi attesi* da un «contatore»



UN ESEMPIO CONCRETO: IL “CONTATORE”

- **Possibile definizione (*comportamento osservabile*):**
un «contatore» è un'entità *caratterizzata da un valore*, intero e variabile nel tempo, e manipolabile tramite *tre operazioni pubbliche*:
 - **reset** per impostare il contatore a un valore iniziale noto (ad es. 0)
 - **inc** per incrementare il valore attuale del contatore
 - **getValue** per recuperare il valore attuale del contatore sotto forma di numero intero



Non importa come sarà realizzato dentro: **importa solo il suo *comportamento osservabile***



USARE (& COLLAUDARE) IL “CONTATORE”

- Per *usare* questo componente serve solo conoscere il suo *comportamento osservabile: non come è fatto dentro!*
 - anzi: conoscere i dettagli interni sarebbe *controproducente*, perché potrebbe portarci a inserire dipendenze inopportune.
- Fra tali usi c'è anche il collaudo: per progettargli *non si deve sapere come è fatto dentro, anzi!*
 - il collaudo va progettato "a priori", perché il suo scopo è proprio *validare il componente* indipendentemente da come sarà realizzato
- Dunque, come potremmo collaudare un contatore?
 - come minimo, test di ogni operazione in diverse situazioni, *con special cura per i casi limite e i casi particolari.*



CONTATORE: PIANO DI COLLAUDO

- Poiché il contatore espone tre operazioni, occorre sicuramente collaudarle tutte, singolarmente
- Come esprimere il "risultato atteso" del collaudo?
 - *non certo* indicando cosa stamperà a video!
 - occorre *dire cosa ci aspettiamo che avvenga* in risposta a una certa operazione (o sequenza di operazioni)
- Possibile piano di collaudo:
 - mi aspetto che eseguendo nell'ordine **reset** e **inc**, il valore del contatore sia 1
 - mi aspetto che eseguendo **reset** e 4 **inc**, il valore del contatore sia 4
 - ...eccetera...

| Counter |
|--|
| <code>+reset()</code> <code>+inc()</code> <code>+getValue() : int</code> |



PIANO DI COLLAUDO: PERCHÉ?

- A che serve questo?
 - ora possiamo assegnare la realizzazione ad altri
 - *quando ci porteranno il componente realizzato, avremo modo di controllare la bontà e la qualità della altrui realizzazione*
 - se non ci convince, *non lo paghiamo!*
- Il piano di collaudo diventa *parte integrante delle specifiche* del "capitolato contrattuale"
 - *chi accetta la nostra commessa di realizzazione del contatore SA che il suo lavoro sarà CONTROLLATO E VALIDATO da noi SECONDO QUESTE SPECIFICHE*
 - patti chiari, amicizia lunga



PIANO DI COLLAUDO: COME ?

MA.. come esprimiamo tutto questo in pratica?

- solo a parole..?
 - possibile, ma adatto solo a piccoli progetti
 - insostenibile per progetti complessi, con piani di collaudo ampi
 - occorre uscire dall'artigianato per approdare all'ingegneria, che richiede *specifiche formalizzate e strumenti di supporto*
- collaudo formalizzato e strumentato
 - tramite il concetto di **asserzione** si specificano affermazioni che devono essere vere in specifici punti del programma
 - disponibile anche in C, tramite la libreria **<assert.h>**
 - idea usata a tappeto in Java, C#, Scala, Kotlin per strumentare il *collaudo sistematico e automatico*
 - MORALE: il progettista progetta i test e lo strumento li esegue 😊

ASSERZIONI

- Un'asserzione esprime un fatto vero in quel punto
 - può essere una funzione, una macro, o una keyword specifica
 - l'argomento è un'espressione booleana che *ci si aspetta sia vera*
 - Esempio in C:
`reset(&c) ; inc(&c) ; assert(getValue(c)==1) ;`
- Dalla singola asserzione alla *suite di test*
 - MA riempire il codice con decine di asserzioni finisce per inquinarlo: molto meglio pensare a una *suite di test* separata e modularizzata
 - conviene inoltre predisporre un *tool* per eseguire i test *in automatico*
- In definitiva
 - si progettano *decine di test case* e si eseguono *automaticamente*
 - idea sfruttata estensivamente in Java dallo strumento *JUnit*



CONTATORE: UNA REALIZZAZIONE IN C

- Una prima possibile scelta: **contatore = modulo con stato**
 - Stato del contatore *inaccessibile dall'esterno*,
ossia fuori dal file di definizione → keyword **static**
 - Accesso allo stato solo attraverso *funzioni (pubbliche)*:

```
void reset(void) ;  
void inc(void) ;  
int  getValue(void) ;
```

mcounter.h

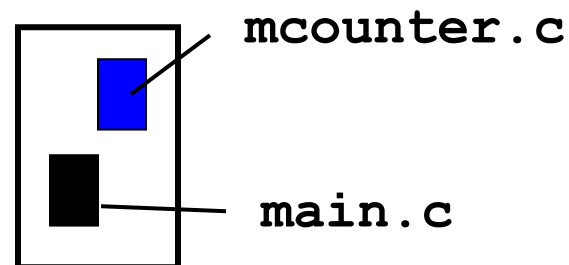
- Per USARE il contatore, si deve:
 - aggiungere al progetto il sorgente del contatore (es. **mcounter.c**)
 - includere nel file cliente (es. **mymain.c**) l'opportuno file *header* (es. **mcounter.h**) che dichiari le tre operazioni sopra citate

UN POSSIBILE CLIENTE

```
/* mymain.c */  
#include <stdio.h>  
#include "mcounter.h"  
  
void main() {  
    reset(); inc(); int v = getValue();  
    printf("Esito: %s", v==1 ? "ok" : "failed" );  
}
```

```
void reset(void);  
void inc(void);  
int getValue(void);
```

OSSERVA: *questo main sta applicando "artigianalmente" il Piano di Collaudo...!*



NOTE:

- Non occorre creare esplicitamente il contatore, perché esso coincide col modulo `mcounter.c` che fa parte del progetto
- Le operazioni non hanno più l'argomento contatore perché è implicito su quale contatore agiscano: l'unico presente!

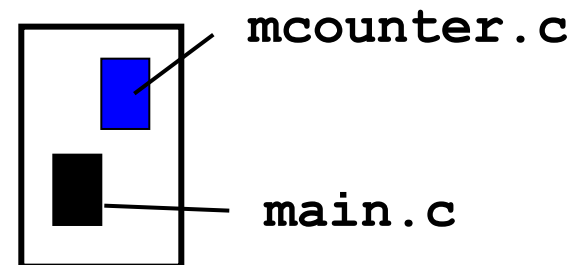
UN POSSIBILE CLIENTE (BIS)

```
#include <assert.h>
#include <stdio.h>
#include "mcounter.h"
```

```
void reset(void);
void inc(void);
int getValue(void);
```

```
main() {
    reset(); inc(); int v = getValue();
    assert(v==1);
}
```

Alternativa migliore: l'asserzione



OSSERVA:

- Non occorre creare esplicitamente il contatore, perché esso coincide col modulo `mcounter.c` che fa parte del progetto
- Le operazioni non hanno argomenti perché è implicito su quale contatore agiscano: l'unico presente, che coincide col modulo stesso.



UNA POSSIBILE REALIZZAZIONE IN C

mcounter.h

```
void reset(void) ;  
void inc(void) ;  
int  getValue(void) ;
```

Questa realizzazione è accettata come corretta *solo se supera il collaudo* svolto come indicato nel Piano di Collaudo.

mcounter.c

```
static int stato;  
void reset(){ stato=0; }  
void inc(){ stato++; }  
int getValue(){ return stato; }
```

Stato *inaccessibile dall'esterno del file*

UNO SCENARIO PROBLEMATICO (1/3)

- Si supponga di voler *contare le persone in entrata* e le *persone in uscita* da un museo o da un grande magazzino
- A livello di principio, il problema è facile: *basta avere due contatori*, attivati ciascuno da un sensore sulla porta

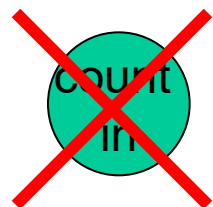
count
in



count
out

UNO SCENARIO PROBLEMATICO (2/3)

- Dunque, dovrebbe essere altrettanto semplice costruire il sistema software...
- ... MA il nostro contatore è un singleton: un pezzo unico!
- ***NON possiamo averne due, perché non possiamo includere due volte lo stesso componente nel progetto!***
 - il C non consente che un modulo (file) sia incluso due volte
 - neanche in Java, C#, Scala, Kotlin ammettono che la stessa classe (o lo stesso object) possa comparire due volte nell'applicazione

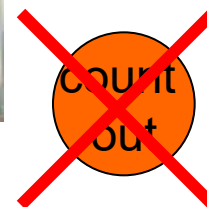
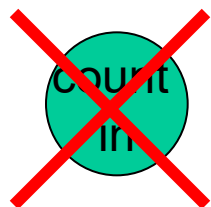


Occorre cambiare approccio.

UNO SCENARIO PROBLEMATICO (3/3)

- La soluzione *non* è il copia & incolla!
 - copiare Contatore in Contatore2 duplicherebbe tutto,
 - renderebbe *un incubo* la successiva manutenzione,
 - e comunque *non è un approccio scalabile!*Se ne serve un altro che si fa, Contatore3.. ? E poi..?

*Serve un approccio alternativo,
intrinsecamente scalabile.*





TIPI DI DATO ASTRATTO (ADT)

- Un **tipo di dato astratto (ADT)** definisce una categoria concettuale con le sue proprietà:
 - *definizione di tipo* su un dominio D
 - *insieme di operazioni ammissibili* su tale dominio.
- In C, gli ADT si definiscono tramite il costrutto **typedef**
 - si possono creare tante entità di quel tipo quante ne servono
 - tali entità sono espresse tramite *variabili* di quel tipo
- Purtroppo, però, **typedef non supporta l'incapsulamento**
 - la **struttura interna dell'ADT**, pur lasciata concettualmente sullo sfondo, è in realtà **perfettamente visibile** e **nota a tutti**
perché i file header vanno inclusi ovunque l'ADT venga usato
 - non vi è alcuna reale possibilità di *impedire usi errati* degli oggetti, perché l'accesso è *sostanzialmente libero!*



IL SOLITO ESEMPIO: CONTATORE COME ADT

- Per prima cosa si deve definire il tipo di dato astratto “*contatore*” tramite un'opportuna **typedef**:
typedef contatore;
- In questa fase *non importa come il contatore sia fatto*, poiché ciò è irrilevante per i clienti che lo useranno
 - però, poi, la **typedef** dovrà essere *inclusa da tutti i clienti* e quindi in realtà tutti sapranno come è fatto ☹
- Quindi si devono specificare le operazioni ammesse, con la relativa signature:

```
typedef ... contatore;  
void reset(contatore*);  
void inc(contatore*);  
int  getValue(contatore);
```

contatore.h



IL CONTATORE COME ADT: USO

- Queste informazioni sono sufficienti per *usare e collaudare* il nuovo tipo **contatore**
 - per compilare il cliente basta includere l'header, che contiene la definizione del tipo (typedef) e le dichiarazioni di funzioni.
 - Il cliente può **istanziare tanti contatori quanti gliene occorrono**:

```
#include "contatore.h"
```

```
void main() {  
    int v1, v2;  
    contatore c1, c2;  
    reset(&c1); reset(&c2);  
    inc(&c1); inc(&c1); inc(&c2);  
    v1=getValue(c1);  
    v2=getValue(c2);  
}
```

```
typedef ..... contatore;  
void reset(contatore*);  
void inc(contatore*);  
int getValue(contatore);
```

PIANO DI COLLAUDO:
dire a priori cosa ci si
aspetta come risultato
per v1 e v2....



UNA PRIMA IMPLEMENTAZIONE

- La struttura interna del contatore diventa *rilevante* quando giunge il momento di *realizzarlo*.
 - Il piano di collaudo, *stabilito a priori*, servirà per validarlo
- Se ora scegliamo di rappresentare lo stato con un intero, avremo:

`typedef int contatore;`

- Sotto questa ipotesi, la definizione delle operazioni precedentemente dichiarate assume la forma:

contatore1.c

```
#include "contatore.h"
```

```
void reset(contatore* pc) { *pc = 0; }  
void inc(contatore* pc)   { (*pc)++; }  
int  getValue(contatore c) { return c; }
```

Supera il collaudo?

include
typedef

UNA SECONDA IMPLEMENTAZIONE

- **VARIANTE:** *rappresentare lo stato con una struttura* che racchiuda un intero:

```
typedef struct {int value;} contatore;
```

- Perché? Perché molti tipi di uso corrente sono tipi strutturati e dunque seguono questo schema.
- Sotto questa ipotesi, la definizione delle operazioni precedentemente dichiarate assume la forma:

contatore2.c

```
#include "contatore.h"
```

```
void reset(contatore* pc) { pc -> value =0;  
void inc(contatore* pc)    { (pc ->value)++;  
int getValue(contatore c) { return c.value;
```

Supera il collaudo?

include
typedef



UNA TERZA IMPLEMENTAZIONE

- Come ulteriore alternativa, potremmo rappresentare lo stato con una **stringa di 'I'** (*notazione del detenuto*):

```
typedef char contatore[21] ;
```

Qui "" indica 0, "I" indica 1, "II" indica 2, etc.

OK solo
fino a 20

- Sotto questa ipotesi, la definizione delle operazioni precedentemente dichiarate assume la forma:

contatore3.c

```
#include "contatore.h"
```

```
void reset(contatore* pc) { (*pc)[0] = '\0'; }
```

```
void inc(contatore* pc) {
```

```
    int x = strlen(*pc); (*pc)[x]='I'; (*pc)[x+1]='\0'; }
```

```
int getValue(contatore c) { return strlen(c); }
```

Supera il
collaudo?



BILANCIO

Definire gli ADT in C tramite **typedef** è possibile, ma:

- non c'è unitarietà fra **parte-dati** (espressa da **typedef**) e **parte-operazioni** (scritte successivamente e altrove)
- non c'è **protezione** dall'uso improprio, perché tutti vedono **typedef** e dunque *tutti possono aggirarla*
- le **signature delle operazioni fanno trasparire dettagli** (puntatori) che non si dovrebbero vedere a questo livello

Conclusione:

*livello di espressività inadeguato
a scalare con l'aumento della complessità*



L'ADT CONTATORE (rieccolo)

- Riconsideriamo una delle realizzazioni del contatore come ADT in C – in particolare, la seconda:

contatore.h

```
typedef struct {int value;} contatore;  
void reset(contatore *pc);  
void inc(contatore *pc);  
int  getValue(contatore c);
```

contatore.c

```
#include "contatore.h"  
void reset(contatore *pc){ pc -> value=0; }  
void inc(contatore *pc) { (pc -> value)++;}  
int  getValue(contatore c){return c.value;}
```

include
typedef



ESEMPIO D'USO (già visto)

- Questo main istanziava vari "contatori", ma lo faceva *senza far uso della memoria dinamica*:

```
#include "contatore.h"
main() {
    int v1, v2;
    contatore c1, c2;
    reset(&c1); reset(&c2);
    inc(&c1); inc(&c1); inc(&c2);
    v1=getValue(c1); v2=getValue(c2);
}
```

normali variabili C



VARIANTE: ESEMPIO D'USO CON MEMORIA DINAMICA

- Questa nuova variante invece istanzia "contatori" *facendo uso della memoria dinamica*:

```
#include "contatore.h"
```

```
main() {
```

```
    int v1, v2;
```

```
    contatore *c1, *c2;
```

puntatori C

allocazione in
memoria dinamica

```
    c1 = (contatore*)malloc(sizeof(contatore)) ,
```

```
    c2 = (contatore*)malloc(sizeof(contatore)) ;
```

```
    reset(c1) ; reset(c2) ;
```

```
    inc(c1) ; inc(c1) ; inc(c2) ;
```

```
    v1=getValue(*c1) ; v2=getValue(*c2) ;
```

```
    free(c1) ; free(c2) ;
```

deallocazione esplicita

```
}
```

Un nuovo esempio di ADT: le frazioni



FRAZIONI: ANALISI (1/2)

- In matematica, una frazione è caratterizzata da
 - una coppia di interi (n,d) , $d \neq 0$, solitamente scritta « n/d »
 - una serie di operazioni ammissibili
- Quali operazioni?
 - "costruzione" di una frazione a partire da una coppia di interi
DUBBIO: "costruzione" significa inizializzazione..?
 - accesso a numeratore e denominatore
 - *riduzione ai minimi termini*
DUBBIO: altera la frazione data o ne produce una nuova..?
 - *test di uguaglianza con un'altra frazione*
DUBBIO: «uguale» significa numeratore e denominatore identici, o rispetto della condizione di equivalenza ?



LE GRANDI QUESTIONI

- Inizializzare, Creare, o Costruire?
 - **Inizializzare** = *dare valore iniziale* a un oggetto che *già esiste*
 - **Creare** = *allocare memoria* per un *nuovo oggetto*
 - **Costruire** = creare l'oggetto + *inizializzarlo*
- Modificare oggetti esistenti o sintetizzare nuovi oggetti?
 - **Modificare** = alterare l'oggetto ricevuto
 - **Sintetizzare** = costruire un nuovo oggetto modificato
- Attribuzione delle responsabilità
 - **Chi fa cosa?**
 - Cosa è responsabilità del *chiamante* e cosa del *chiamato*?



FRAZIONI: ANALISI (2/2)

Alcune possibili scelte:

- costruzione di una frazione da due interi
 - opzione 1: il cliente crea e inizializza anche
 - opzione 2: il cliente *delega la creazione*, inizializza soltanto
 - opzione 3: il cliente crea, ma *delega l'inizializzazione*
- riduzione ai minimi termini
 - è una variante del caso precedente: la frazione ridotta è, di fatto, una nuova frazione (num e den diversi)
 - opzione 1: il cliente passa la frazione per riferimento e la funzione la altera irrevocabilmente, "riducendola" ai minimi termini
 - opzione 2: il cliente passa la frazione per valore, e la funzione costruisce e restituisce una *nuova frazione-risultato*, senza alterare quella ricevuta



FRAZIONI: PROGETTO

- Dalla definizione dell'ADT:
 - una frazione è caratterizzata da **due proprietà, numeratore e denominatore**, impostate inizialmente e poi mai più modificate
 - numeratore e denominatore devono essere recuperabili singolarmente → **due funzioni, getNum e getDen**
- Dalle scelte precedenti:
 - se il cliente crea, ma delega l'inizializzazione, l'ADT serve una **funzione di inizializzazione, init**, che, dati due interi, inizializzi con tali valori la frazione → passaggio per riferimento (&)
 - la riduzione ai minimi termini, **minTerm**, restituisce una nuova frazione-risultato, senza alterare la frazione corrente
 - il test di uguaglianza, **equals**, si basa sulla condizione di equivalenza, non sulla mera uguaglianza di numeratore e denominatore; l'argomento è l'altra frazione, con cui confrontare quella attuale



ADT FRAZIONE in C

frazioni.h

```
typedef struct {int num, den;} frazione;  
void init(frazione *f, int n, int d);  
int getNum(frazione f);  
int getDen(frazione f);  
frazione minTerm(frazione f);  
int equals(frazione f1, frazione f2);
```

Questo int è un boolean sotto falso nome

frazione.c

```
#include "frazioni.h"  
void init(frazione* pf, int n, int d) {  
    pf -> num = n; pf -> den = d;  
}  
...
```

include
typedef



ADT: ESEMPIO D'USO

- Questo main illustra un possibile scenario d'uso:

```
#include "frazioni.h"
```

```
void main() {
```

```
    frazione *f1, *f2, f3;
```

```
    f1 = (...)malloc(...); init(f1, 3, 4);
```

```
    f2 = (...)malloc(...); init(f2, 6, 8);
```

```
    f3 = minTerm(*f2);
```

```
    printf("%d/%d\n", getNum(*f1), getDen(*f1));
```

```
    printf("%d/%d\n", getNum(*f2), getDen(*f2));
```

```
    printf("%s",
```

```
           equals(*f1, *f2) ? "uguali" : "diverse");
```

```
    free(f1); free(f2);
```

```
}
```

Puntatori

Inizializzazione
delegata

Alternanza di casi
con * e senza *

Necessaria deallocazione esplicita



COSTRUZIONE: UN TIPICO SCHEMA

- Osserviamo che, per *costruire* una frazione *compiutamente configurata*, sono necessari due passi:
 - prima, l'allocazione della memoria
 - poi, l'inizializzazione di tale memoria con l'opportuno contenuto

```
f1 = (frazione*)malloc(sizeof(frazione));  
init(f1, 3, 4);
```

- Questo scenario riflette in effetti un *tipico pattern*, MA espone al *serio rischio di «lasciare a metà» il lavoro*
 - se ci si dimentica l'allocazione, BOOM al primo uso
 - se ci si dimentica l'inizializzazione, valori casuali nel sistema!
- Per avere software robusto, servono *programming practice* per *prevenire* queste situazioni → **unificare quei due passi**



UNA DIVERSA FRAZIONE IN C

- Possiamo *unificare* introducendo una funzione **make**, che:
 - allochi memoria
 - inizializzi l'entità con gli opportuni valori, ricevuti come argomento
 - restituisca (l'indirizzo del-) l'entità così costruita
- Una tale funzione svolge il ruolo di *costruttore*
 - ritroveremo lo stesso principio nei linguaggi a oggetti
 - ovviamente, con ben altro tipo di supporto 😊

```
typedef struct {int num, den;} frazione;
```

```
frazioni.h
```

```
frazione* make(int n, int d);
```

```
int getNum(frazione f);
```

```
int getDen(frazione f);
```

```
frazione minTerm(frazione f);
```

```
int equals(frazione f1, frazione f2);
```



ESEMPIO D'USO con make

- Il main riformulato facendo uso di **make**:

```
#include "frazioni.h"
```

```
void main() {
```

```
    frazione *f1, *f2, f3;
```

```
    f1 = make(3,4);
```

```
    f2 = make(6,8);
```

```
    f3 = minTerm(*f2);
```

```
    printf("%d/%d\n", getNum(*f1), getDen(*f1));
```

```
    printf("%d/%d\n", getNum(*f2), getDen(*f2));
```

```
    printf("%s",
```

```
           equals(*f1,*f2) ? "uguali":"diverse");
```

```
    free(f1); free(f2);
```

```
}
```

Costruzione

Purtroppo, permane
però la sgradevole
alternanza di casi
con * e senza * ☹

SUMMING UP...

- Tramite opportune *programming practice* possiamo cercare di *prevenire* possibili criticità, ma **solo fino a un certo punto**
 - tutto è ancora *troppo artigianale*, troppo lasciato al «fai da te»
 - basta *dimenticare un passaggio... e la robustezza se ne va!*
- La sintassi del C, così vicina ai meccanismi, non aiuta
 - la totale visibilità dei meccanismi low-level (es. puntatori, indirizzi) è utile per lavorare «vicino» al sistema operativo o per software «molto vicino all'hardware», *ma mal si adatta alle applicazioni «robuste»*
 - moltissimi bug, causa di ingenti costi e perditempo, sono nati così
 - la leggibilità del codice all'aumentare della complessità è *scarsa*
- Serve un **cambio di paradigma**
 - nuovo approccio, nuovi costrutti adatti a scalare con la complessità