VARIABILI GLOBALI

- Obiettivi:
  - Introdurre la nozione di variabile globale
  - Mostrarne l'utilizzo, derogando da uno stile di programmazione funzionale puro
  - Costruire un componente con stato come esercizio

AMBIENTE LOCALE E GLOBALE

- In C, ogni funzione ha il suo *ambiente locale* che comprende i parametri formali e le variabili locali alla funzione

- Esiste però anche un *ambiente globale:* quello in cui sono definite tutte le funzioni. Qui sono definite anche le *variabili globali*

- La denominazione *"globale"* deriva dal fatto che l'environment di definizione di queste variabili *è esterno (fuori) da ciascuna funzione* (compreso il main)

VARIABILI GLOBALI

- **In C una *variabile* è detta *globale* se è definita *fuori da qualunque funzione* ("a livello globale")**

```
int trentadue = 32;

float  fahrToCelsius( float F ){
float temp = 5.0 / 9;
     return temp * ( F - trentadue );
}
```

- **tempo di vita = *intero programma***
- **scope = *il file in cui è dichiarata* dal punto in cui è scritta in avanti**

ESEMPIO

```
int trentadue = 32;
float fahrToCelsius(float);

void main(void) {
        printf("%f", trentadue);
 float c = fahrToCelsius(86);
}

float fahrToCelsius(float f) {
  return 5.0/9 * (f-trentadue);
}
```

Definizione (e inizializzazione) **della variabile globale**

Uso

Uso **della variabile globale**

DICHIARAZIONI e DEFINIZIONI

Anche per le variabili globali, come per le funzioni, si distingue fra ***dichiarazione*** e ***definizione***

- **al solito, la dichiarazione esprime proprietà associate al simbolo,** *ma non genera un solo byte di codice o di memoria allocata (es. prototipi)*
- **la *definizione* invece implica *anche* *allocazione di memoria,*** e funge contemporaneamente da dichiarazione

ESEMPIO

```
int trentadue = 32;
float fahrToCelsius(float);

void main(void) {
  float c = fahrToCelsius(86);
}

float fahrToCelsius(float f) {
  return 5.0/9 * (f-trentadue);
}
```

Definizione (e inizializzazione) **della variabile globale**

Uso **della variabile globale**

DICHIARAZIONI e DEFINIZIONI

Come distinguere la dichiarazione di una variabile globale dalla sua definizione?

- Per le funzioni si usa il *prototipo*
- ma qui? non c'è l'analogo …..

ESEMPIO (caso particolare con un solo file sorgente)

```
extern int trentadue;

float fahrToCelsius(float f) {
  return 5.0/9 * (f-trentadue);
}

void main(void) {
  float c = fahrToCelsius(86);
}

int trentadue = 32;
```

Dichiarazione **variabile globale**

Uso della var globale

Definizione **della variabile globale**

DICHIARAZIONI e DEFINIZIONI

Come distinguere la dichiarazione di una variabile globale dalla sua definizione?

- Per le funzioni si usa il *prototipo*
- ma qui? non c'è l'analogo …..

si usa l'apposita parola chiave `extern`

- `int trentadue = 32;`

  è una definizione (con inizializzazione)

- `extern int trentadue;`

  è una dichiarazione (la variabile può essere anche definita in un altro file sorgente appartenente al progetto)

ESEMPIO (un solo file sorgente)

```
int trentadue = 32;

float fahrToCelsius(float f) {
  return 5.0/9 * (f-trentadue);
}

void main(void) {
  float c = fahrToCelsius(86);
}
```

Definizione **della variabile globale**

Uso della var globale

ESEMPIO (un solo file sorgente)

```
extern int trentadue;

float fahrToCelsius(float f) {
  return 5.0/9 * (f-trentadue);
}

void main(void) {
  float c = fahrToCelsius(86);
}

int trentadue = 32;
```

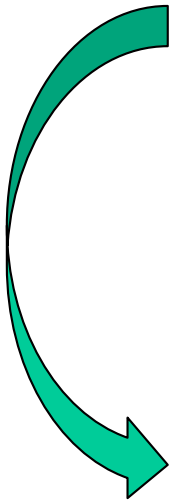
Dichiarazione **variabile globale**

Uso della var globale

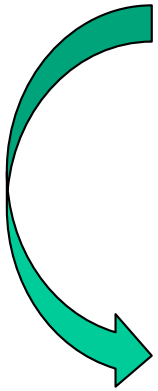
Definizione **della variabile globale**

ESEMPIO su 3 FILE

File `main.c` chiama la funzione **`fahrToCelsius`**



File `f2c.c` definisce la funzione **`fahrToCelsius`** usando la variabile globale **`trentadue`**



File `c32.c` definisce la variabile **`trentadue`**

ESEMPIO su 3 FILE

File `main.c`

```
float fahrToCelsius(float f);
void main(void)
    { float c = fahrToCelsius(86); }
```

File `f2c.c`

```
extern int trentadue;
float fahrToCelsius(float f) {
  return 5.0/9 * (f-trentadue)
}
```

File `c32.c`

```
int trentadue = 32;
```

VARIABILI GLOBALI: USO

- **Il cliente deve *prevedere la dichiarazione* della variabile globale che intende usare:**

```
extern int trentadue;
```

- ***Uno dei file sorgente nel progetto dovrà poi contenere la definizione*** (ed eventualmente l'inizializzazione) **della variabile globale**

```
int trentadue = 32;
```

AREE DI MEMORIA

- **CODE SEGMENT**
  - funzioni
- **DATA SEGMENT**
  - **variabili globali** (extern, globali multi-file; static, globali single-file e locali)
- **STACK**
  - variabili automatiche (var. locali – parametri formali delle funzioni)
- **HEAP**
  - variabili dinamiche allocate (**malloc**) e deallocate (**free**) esplicitamente dall'utente e referenziate tramite puntatori
- Esiste un'ulteriore classe di memorizzazione (**register**) che alloca variabili su registri macchina.

VARIABILI GLOBALI

A che cosa servono le variabili globali?

- **per scambiare informazioni fra cliente e servitore** ***in modo alternativo al passaggio dei parametri***
- **per costruire specifici componenti software** ***dotati di stato***

ESEMPIO: Stile funzionale puro

```
int fattoriale(int n){
      int i, p=1;
      for(i=1; i<=n; i++) p =p*i;
      return p;}


void main(void) {
       printf("%d",fattoriale(4)); }
```

- La comunicazione chiamante-chiamato avviene solo attraverso i parametri formali e il valore restituito
- Stile di programmazione funzionale puro (è **preferibile**)

ESEMPIO: Side Effect

```
int N = 1;

void fattoriale(int n){
          int i;
          for(i=1; i<=n; i++) N=N*i;   }

void main(void) {
        fattoriale(4);
        printf("%d",N);
        fattoriale(4);  //quanto vale N??  }
```

- N è cambiata al termine dell'invocazione
- Stile di programmazione non funzionale puro (***è da evitare***, perché è più difficile trovare errori e modificare il software)

VARIABILI GLOBALI

Nell'esempio con side effect, le variabili globali:

- **sono un mezzo *bidirezionale*:** la funzione può sfruttarle per memorizzare una informazione *destinata a sopravviverle (effetto collaterale* o *side effect)*
- **ma *introducono un accoppiamento* fra cliente e servitore che *limita la riusabilità*** rendendo la funzione stessa *dipendente dall'ambiente esterno*
  - Deroga dallo stile di programmazione funzionale puro, il codice diventa **più difficile da leggere, correggere e modificare → SCONSIGLIATO!!!**

VARIABILI GLOBALI

A che cosa servono le variabili globali?

- **per scambiare informazioni fra cliente e servitore *in modo alternativo al passaggio dei parametri***
- **per costruire specifici componenti software *dotati di stato.***

Componenti con stato: ESEMPIO

Si vuole costruire un componente software *numeriDispari* che fornisca una funzione

```
int prossimoDispari(void)
```

che *restituisca via via il "successivo" dispari*

- Per fare questo, tale componente deve ***tenere memoria*** al suo interno ***dell'ultimo valore fornito***
- Dunque, *non è una funzione in senso matematico,* perché, **interrogata più volte, *dà ogni volta una risposta diversa***

ESEMPIO

- un file **`dispari.c`** che definisca la funzione ***e una variabile globale*** **che ricordi lo stato**
- un file **`dispari.h`** che dichiari la funzione

dispari.c

```
int ultimoValore = 0;

int prossimoDispari(void){
  return 1 + 2 * ultimoValore++ ; }
```

(sfrutta il fatto che i dispari hanno la forma 2k+1)

dispari.h

```
int prossimoDispari(void);
```

GENERATORE DISPARI (stato)

dispari.c

```
int ultimoValore = 0;

int prossimoDispari(void){

        return 1 + 2 * ultimoValore++;}
```

dispari.h

```
int prossimoDispari(void);
```

main.c

```
#include "dispari.h"

#include <stdio.h>

void main(void){
   int i;
   for(i=1;i<10; i++)
   printf("%d",prossimoDispari()); }
```

Osservazioni:

- La variabile globale **`ultimoValore`** definita nel file **`dispari.c`** ha tempo di vita pari alla durata del programma …
- Potenzialmente è visibile anche in altri file …

GENERATORE DISPARI (stato accessibile)

main.c

```
#include "dispari.h"

#include <stdio.h>

extern int ultimoValore; //DICHIARAZIONE

void main(void){
  int i;
  for(i=1;i<10; i++)
     { printf("%d",prossimoDispari());
       ultimoValore=0; }
}
```

- La variabile che realizza lo stato è acceduta e modificata anche dal main.

AMBIENTE GLOBALE e PROTEZIONE

Il fatto che le *variabili globali* in C siano potenzialmente visibili *in tutti i file* dell'applicazione pone dei ***problemi di protezione:***

- ***Che cosa succede se un componente dell'applicazione altera una variabile globale?***
- Nel nostro esempio: cosa succede se il main altera la variabile globale `ultimoValore`?

AMBIENTE GLOBALE e PROTEZIONE

Potrebbe essere utile avere variabili

- ***globali* nel senso di *permanenti* come tempo di vita** (per poter costruire componenti dotati di stato)...
- **… ma anche *protette*, nel senso che *non tutti* possano accedervi**

VARIABILI STATICHE

VARIABILI static

In C, una *variabile* può essere dichiarata `static`:

- **è *permanente* come tempo di vita**
- **ma è *protetta*, in quanto è *visibile solo entro il suo scope di definizione***

Nel caso di una variabile globale **`static`**, ogni tentativo di accedervi da altri file, tramite dichiarazioni **`extern`**, sarà *impedito* dal compilatore

GENERATORE DISPARI (stato)

dispari.c

```
int ultimoValore = 0;

int prossimoDispari(void){

        return 1 + 2 * ultimoValore++;}
```

dispari.h

```
int prossimoDispari(void);
```

main.c

```
#include "dispari.h"

#include <stdio.h>

void main(void){
   int i;
   for(i=1;i<10; i++)
   printf("%d",prossimoDispari()); }
```

GENERATORE DISPARI (stato protetto)

dispari.c

```
static int ultimoValore = 0;

int prossimoDispari(void){

        return 1 + 2 * ultimoValore++;}
```

dispari.h

```
int prossimoDispari(void);
```

main.c

```
#include "dispari.h"

#include <stdio.h>

void main(void){
   int i;
   for(i=1;i<10; i++)
   printf("%d",prossimoDispari()); }
```

GENERATORE DISPARI (stato inaccessibile)

main.c

```
#include "dispari.h"
#include <stdio.h>
extern int ultimoValore;          //ERRORE!
void main(void){
  int i;
  for(i=1;i<10; i++)
     { printf("%d",prossimoDispari());
       ultimoValore=1; }
}
```

- Il linker non trova la variabile **ultimoValore** dichiarata static.

ESEMPIO rivisitato

La variabile static è "protetta" nel senso che:

- **La variabile `ultimoValore` è ora *inaccessibile* dall'esterno di questo file: l'unico modo di accedervi è tramite `prossimoDispari()`**
- Se anche qualcuno, fuori, tentasse di accedere tramite una dichiarazione **`extern`, il linker *non troverebbe la variabile***
- Se anche un altro file definisse **un'altra variabile globale di nome `ultimoValore`, *non ci sarebbe comunque collisione fra le due***, perché quella static **"non è visibile esternamente": sono due diverse variabili**

Tutto chiaro fin qui? ...

VARIABILI STATICHE dentro a FUNZIONI

Una *variabile statica* può essere definita *anche dentro a una funzione*. Così:

- **è comunque *protetta* in quanto visibile solo dentro alla funzione** *(come ogni variabile locale)*
- ***ma è anche permanente (allocata nell'area globale o DATA SEGMENT),*** in quanto il suo tempo di vita diventa quello dell'intero programma

Consente di costruire componenti (funzioni) ***dotati di stato, ma indipendenti dall'esterno (ovvero dall'ambiente globale)***

ESEMPIO rivisitato (2)

Realizzazione alternativa del componente:

dispari.c

```
int prossimoDispari(void){
   static int ultimoValore = 0;
 return 1 + 2 * ultimoValore++;
}
```

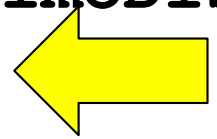
(dispari.h non cambia)

GENERATORE DISPARI (stato inaccessibile)

main.c

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


void main(void){
  int i;
  for(i=1;i<10; i++)
     { printf("%d",prossimoDispari());
       ultimoValore=1; }          //ERRORE!
}
```



- Il linker non trova la variabile ultimoValore, locale e dichiarata static.

AREE DI MEMORIA

- **CODE SEGMENT**
  - funzioni
- **DATA SEGMENT**
  - **variabili globali** (extern, globali multi-file; **static**, globali single-file e locali)
- **STACK**
  - variabili automatiche (var. locali – parametri formali delle funzioni)
- **HEAP**
  - variabili dinamiche allocate (**malloc**) e deallocate (**free**) esplicitamente dall'utente e referenziate tramite puntatori
- Esiste un ulteriore classe di memorizzazione (**register**) che alloca variabili su registri macchina.

VARIABILI STATICHE

Quindi, la parola chiave ***static***

- ***ha sempre e comunque due effetti***
  - rende l'oggetto ***permanente (DATA SEGMENT)***
  - rende l'oggetto ***protetto***
    (invisibile fuori dal suo scope di definizione)
- ***ma agisce sempre un effetto per volta***
  - **una variabile definita in una funzione**, che è comunque protetta, **viene resa *permanente***
  - **una variabile globale**, già di per sé permanente, **viene resa *protetta***

ESEMPIO rivisitato (vers.3) (stato protetto)

dispari.c

```
static int ultimoValore = 0;

int prossimoDispari(void){

        return 1 + 2 * ultimoValore++;}
```

dispari.h

```
int prossimoDispari(void);
```

main.c

```
#include "dispari.h"

#include <stdio.h>

void main(void){
   int i;
   for(i=1;i<10; i++)
   printf("%d",prossimoDispari()); }
```

ESEMPIO rivisitato (vers.2) (stato protetto)

Realizzazione alternativa del componente:

dispari.c

```
int prossimoDispari(void){
   static int ultimoValore = 0;
 return 1 + 2 * ultimoValore++;
}
```

(dispari.h non cambia)

- La variabile locale dichiarata static diventa globale come allocazione, ma protetta come visibilità

ESEMPIO sbagliato

Realizzazione sbagliata del componente:

dispari.c

```
int prossimoDispari(void){
        int ultimoValore = 0;
 return 1 + 2 * ultimoValore++;
}
```

(dispari.h non cambia)

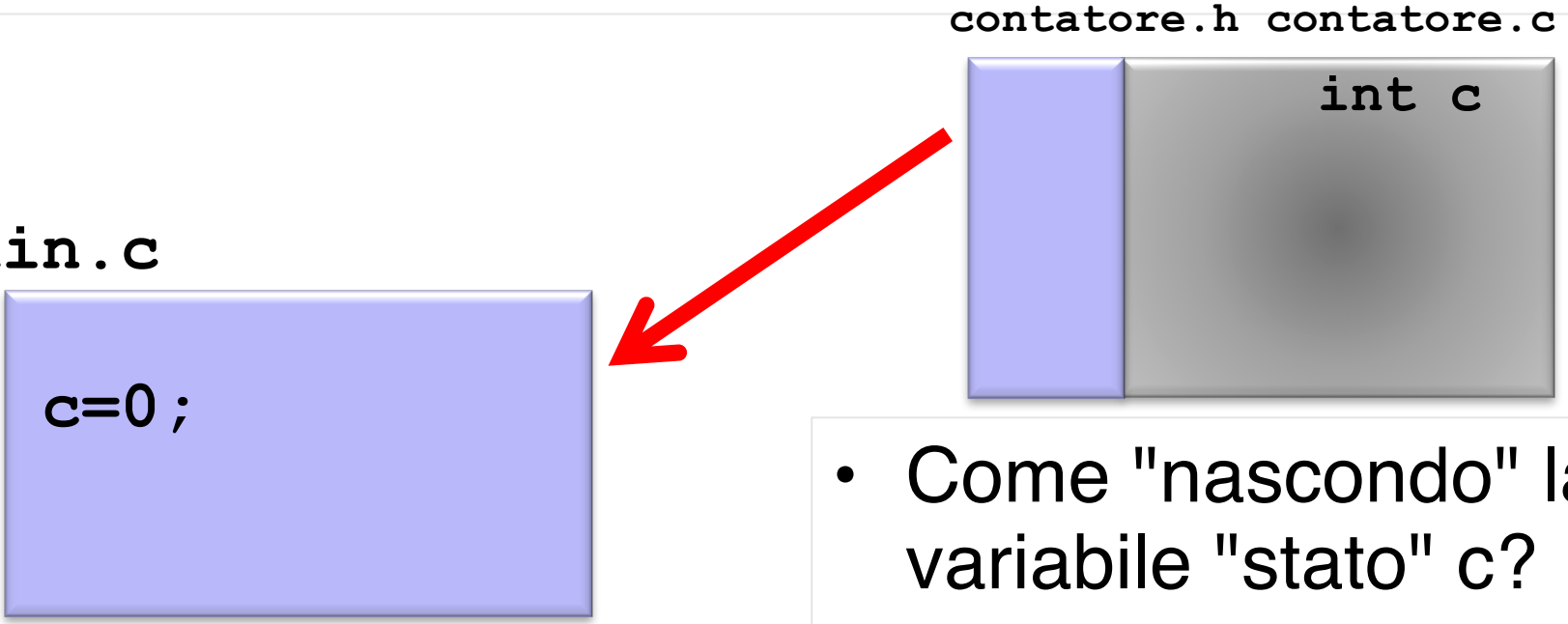
- La variabile locale è protetta come visibilità, ma è una diversa variabile (inizializzata a 0) per ogni nuova attivazione della funzione

Esercizio (in Laboratorio)

- Si realizzi un ***componente dotato di stato*** che rappresenta un ***contatore*** (con valore intero protetto), con operazioni:
  - `void reset (void)`
  - `void inc(void)`
  - `int getValue(void)`
- Il main che utilizza il contatore, lo resetta e acquisisce da input una sequenza di interi positivi (terminata da 0) usati come incremento del contatore; si verifichi che il valore intero del contatore è protetto.

Esempio: il contatore

```
#include "contatore.h"
#include <stdio.h>
int main(){int i;
   reset(); for(i=0;i<6;i++) inc();
   printf("%d",   getvalue());
 }
```



- Come "nascondo" la variabile "stato" c?

Esempio: il contatore

```
#include "contatore.h"
#include <stdio.h>
int main(){int i;
   reset(); for(i=0;i<6;i++) inc();
   printf("%d",   getvalue());
 }
```

contatore.h contatore.c

main.c

```
c=0; ERRORE!!
```

- Come "nascondo" la variabile "stato" c?

Esempio: il contatore

1. *file header* ***contatore.h***

```
void reset(void);
void inc(void);
void stampa(void);
```

Definisce cosa si può fare con il contatore

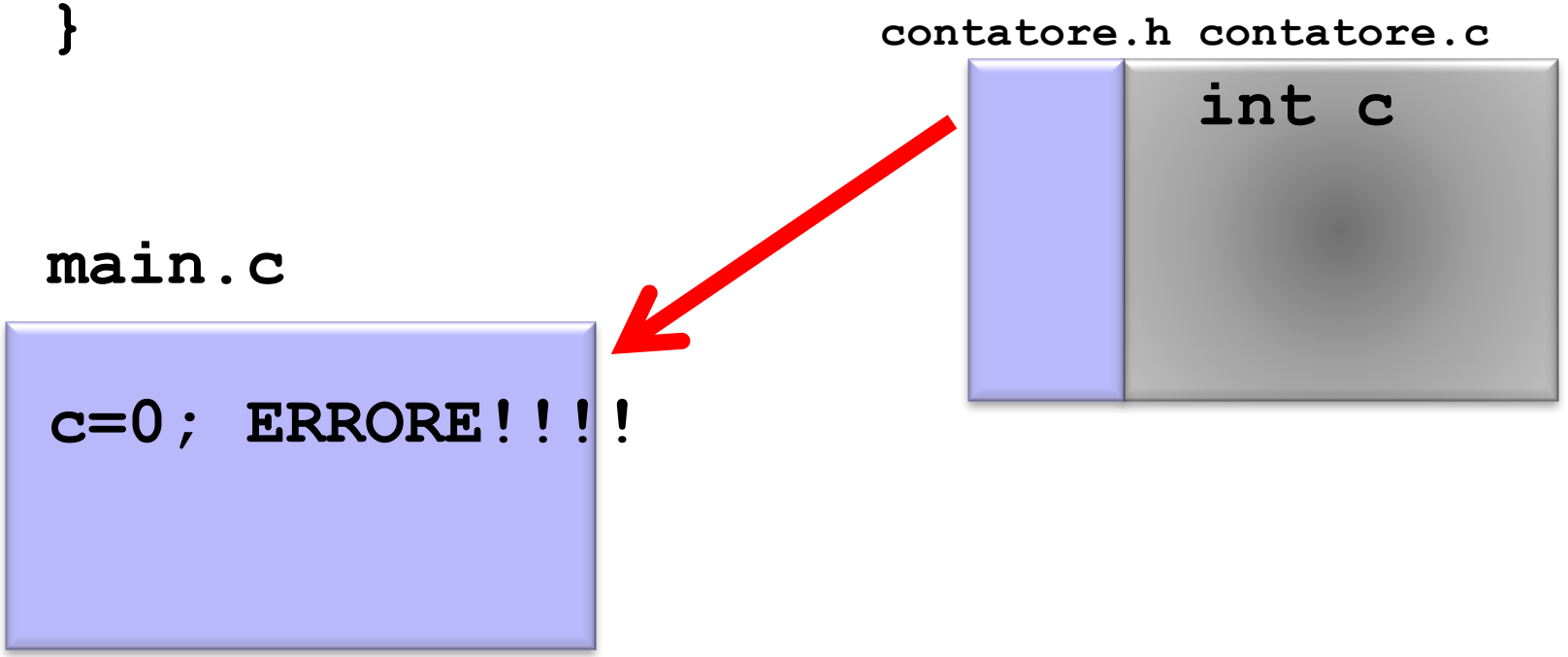
2. *file di implementazione* ***contatore.c***

```
#include "contatore.h"
static int c;
void reset(void){ c=0; }
void inc(void){ c++; }
void stampa(void) { printf("%d", c); }
```

Esempio: il contatore

- Il cliente:

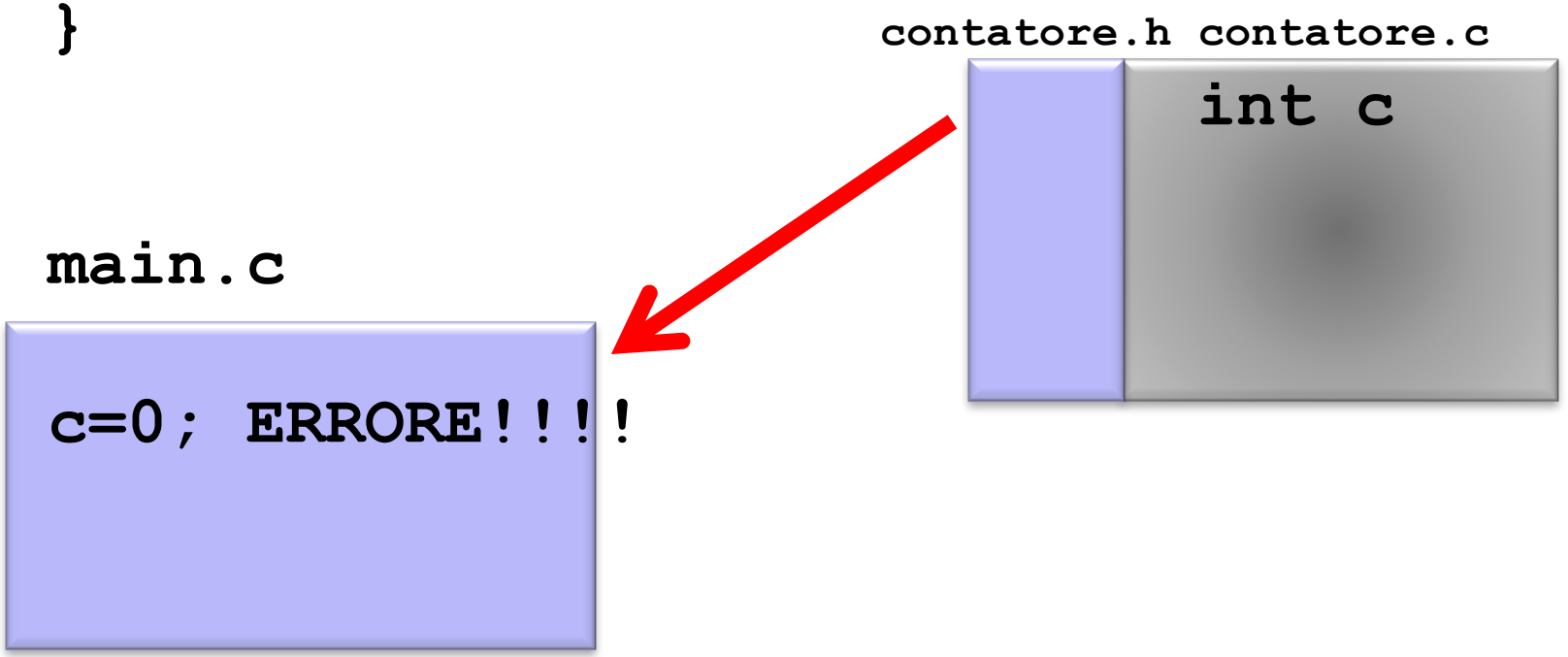
```
#include "contatore.h"
int main(){
  reset(); inc(); stampa();
}
```



Esempio: il contatore

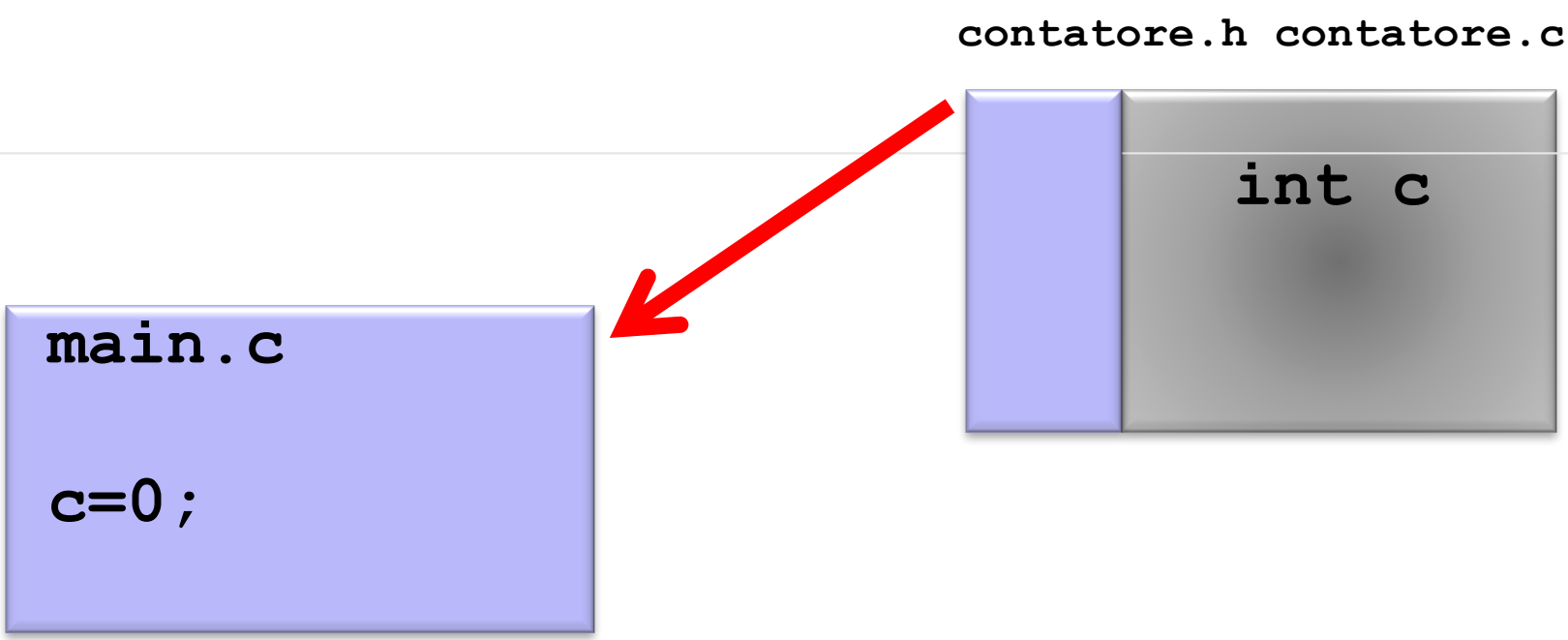
- Il cliente:

```
#include "contatore.h"
int main(){
  reset(); inc(); stampa();
}
```



Esempio: il contatore

```
#include "contatore.h"
   extern int c;
   int main(){
     reset(); inc(); stampa();
   }
```



Esempio: il contatore

1. *file header* ***contatore.h***

```
void reset(void);
void inc(void);
void stampa(void);
```

Definisce cosa si può fare con il contatore

2. *file di implementazione* ***contatore.c***

```
#include "contatore.h"
#include <stdio.h>
static int c;
void reset(void){ c=0; }
void inc(void){ c++; }
void stampa(void) { printf("%d", c); }
```

Incapsula il contatore c (IL dato) e specifica il codice delle funzioni con cui si agisce sul contatore

Esempio: il contatore

- Il cliente:

```
#include "contatore.h"
int main(){
   reset(); inc(); stampa();
}
```