

AMBIENTE LOCALE E GLOBALE

In C, ogni funzione ha il suo *ambiente locale* che comprende i parametri e le variabili definite localmente alla funzione

Esiste però anche un *ambiente globale*:
quello dove tutte le funzioni sono definite.
Qui si possono anche definire variabili, dette *variabili globali*

La denominazione "*globale*" deriva dal fatto che l'*environment di definizione* di queste variabili *non coincide con quello di nessuna funzione* (neppure con quello del main)

VARIABILI GLOBALI

- Una *variabile globale* è dunque definita *fuori da qualunque funzione* (“a livello esterno”)
- tempo di vita = *intero programma*
- scope = *il file in cui è dichiarata dal punto in cui è scritta in avanti*

```
int trentadue = 32;

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

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*
- la **definizione** invece implica anche **allocazione di memoria**, e funge contemporaneamente da dichiarazione

ESEMPIO

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

Definizione (e inizializzazione) della variabile globale

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

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

Uso della variabile globale

DICHIARAZIONI e DEFINIZIONI

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

- nelle funzioni è facile perché la dichiarazione ha un ";" al posto del corpo { }
- ma qui non c'è l'analogo

si usa l'apposita parola chiave extern

- **int trentadue = 10;**
è una definizione (con inizializzazione)
- **extern int trentadue;**
è una dichiarazione (la variabile sarà definita in un altro file sorgente appartenente al progetto)

ESEMPIO (caso particolare con un solo file sorgente)

```
extern int trentadue;
```

*Dichiarazione
variabile globale*

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

Uso della var globale

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

```
int trentadue = 32;
```

*Definizione della
variabile globale*

VARIABILI GLOBALI: USO

- Il cliente deve incorporare 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 = 10;

ESEMPIO su 3 FILE

File main.c

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

File f2c.c

```
extern int trentadue;

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

File 32.c

```
int trentadue = 32;
```

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**

VARIABILI GLOBALI

Nel primo caso, 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*

- la funzione opera correttamente solo se l'ambiente globale definisce tali variabili *con quel preciso nome, tipo e significato*

Secondo Caso: 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 mate-matico*, 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);
```

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 qualcuno altera **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

ESEMPIO rivisitato

Realizzazione alternativa del componente:

dispari.c

```
static int ultimoValore = 0;

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

(dispari.h non cambia)

ESEMPIO rivisitato

In che senso la variabile static è "protetta"?

- 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”

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*, in quanto il suo tempo di vita diventa quello dell'intero programma

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

ESEMPIO rivisitato (2)

Realizzazione alternativa del componente:

dispari.c

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

(dispari.h non cambia)

VARIABILI STATICHE

Quindi, la parola chiave **static**

- **ha sempre e comunque due effetti**
 - rende l'oggetto **permanente**
 - rende l'oggetto **protetto**
(invisibile fuori dal suo scope di definizione)
- **ma se ne vede sempre uno solo 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**

DIRETTIVE AL PREPROCESSORE

Per ora abbiamo visto due tipi di direttive al preprocessore

#include

#define

Ne esistono tante altre. Queste sono quelle che vedremo:

#if

#else

#elif

#endif

#ifdef

#ifndef

DIRETTIVE CONDIZIONALI DI COMPILAZIONE

Esistono direttive condizionali di compilazione ossia direttive che permettono di evitare la compilazione di alcune parti di codice

#if
#else
#elif
#endif

Se l'espressione che segue #if è vera allora la porzione di codice tra #if e #endif verrà compilata. Altrimenti sarà ignorata dal compilatore.

Tali direttive possono essere inserite in qualunque punto del programma

DIRETTIVE CONDIZIONALI DI COMPILAZIONE

```
#include <stdio.h>
#define EU 1
#define UK 2
#define US 3

#define AREA EU

int main(){
#if AREA == EU
    char moneta[] = "euro";
#elif AREA == US
char moneta[] = "dollaro";
#else
char moneta[] = "sterlina";
#endif

printf("Moneta corrente: %s", moneta);
}
```

DIRETTIVE CONDIZIONALI DI COMPILAZIONE

Esistono direttive condizionali di compilazione che si basano sul fatto che una certa macro sia definita o meno

```
#define DEBUG
int main() {
    int i = 100;
    #ifdef DEBUG
    printf("la var i vale %d\n",i);
    #endif
    ...
}
```

DIRETTIVE CONDIZIONALI DI COMPILAZIONE

Serve anche se per esempio abbiamo definito la stessa struttura in due file header diversi e li vogliamo includere entrambi nello stesso file sorgente:

Esempio: file azioni.h

```
typedef struct {
    char nome[64];
    float valutazione} azione;
```

....

Esempio: file trova.h

```
typedef struct {
    char nome[64];
    float valutazione} azione;
```

....

DIRETTIVE CONDIZIONALI DI COMPILAZIONE

Allora si può creare la seguente parte di codice. La prima volta che il file viene compilato si entra nell'if, si definisce AZIONE e la typedef, le successive no

```
#ifndef AZIONE
#define AZIONE
typedef struct {
    char nome[64];
    float valutazione} azione;
#endif
```