

PROCEDURE

Una procedura permette di

- *dare un nome a una istruzione*
- *rendendola parametrica*
- *non denota un valore, quindi non c'è tipo di ritorno → void*

```
void p(int x) {  
    x = x * 2;  
    printf("%d", x);  
}
```

PROCEDURE COME SERVITORI

Una *procedura* è un *componente software* che cattura l'idea di “macro-istruzione”

- molti possibili parametri, che ***possono anche essere modificati*** mentre nelle funzioni normalmente ***non devono*** essere modificati
- nessun “valore di uscita” esplicito

Come una funzione, una procedura è un servitore

- *passivo*
 - che serve *un cliente per volta*
 - che può trasformarsi in cliente *invocando se stessa o altre procedure*
- In C, una procedura ha la stessa struttura di una funzione, salvo il ***tipo di ritorno*** che è **`void`**

PROCEDURE

L'istruzione *return* provoca solo la restituzione del controllo al cliente e non è seguita da una espressione da restituire -> *non è necessaria* se la procedura termina “spontaneamente” a fine blocco

Nel caso di una procedura, non esistendo valore di ritorno, cliente e servitore comunicano solo:

- mediante ***parametri***
- mediante ***aree dati globali***

➡ Occorre il ***passaggio per riferimento*** per fare cambiamenti permanenti ai dati del cliente

PASSAGGIO DEI PARAMETRI

In generale, un parametro può essere trasferito dal cliente al servitore:

- **per valore o copia** (*by value*)
si trasferisce il valore del parametro attuale
- **per riferimento** (*by reference*)
si trasferisce un riferimento al parametro attuale

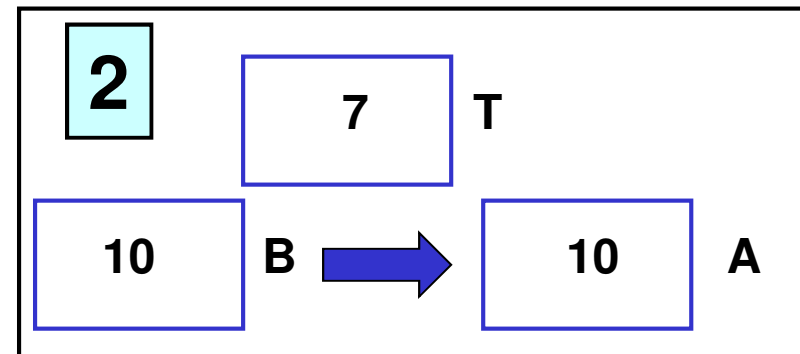
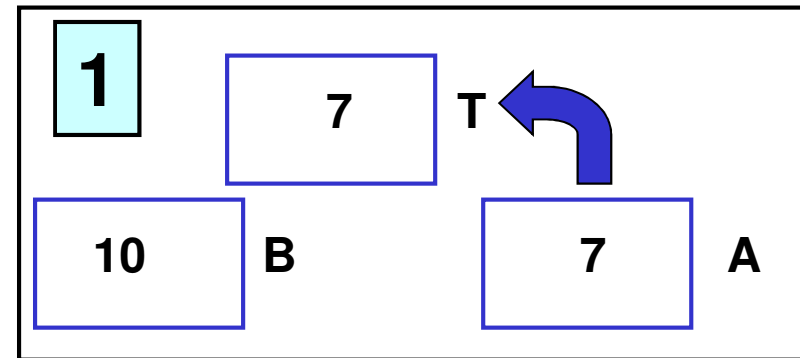
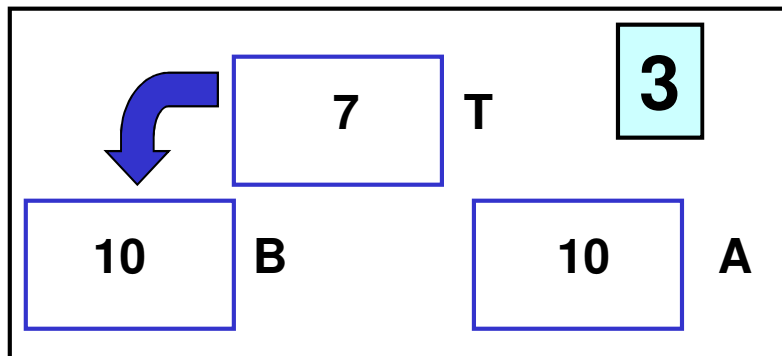
ESEMPIO

Perché il passaggio per valore non basta?

Problema: scrivere una procedura che *scambi i valori di due variabili intere*

Specifica:

Dette A e B le due variabili, ci si può appoggiare a una *variabile ausiliaria T*, e svolgere lo scambio in *tre fasi*



ESEMPIO

Supponendo di utilizzare, senza preoccuparsi, il passaggio per valore usato finora, la codifica potrebbe essere espressa come segue:

```
void scambia(int a, int b) {  
    int t;  
    t = a;    a = b;    b = t;  
    return; /* può essere omessa */  
}
```

ESEMPIO

Il cliente invocherebbe quindi la procedura così:

```
int main() {  
    int y = 5, x = 33;  
    scambia(x, y);  
    /* ora dovrebbe essere  
       x=5, y=33 ...  
       MA NON È VERO  
    */  
}
```

Perché non funziona?

ESEMPIO

- La procedura ha *effettivamente scambiato* i valori di A e B al suo interno (in C nel suo record di attivazione)
- ma questa modifica non si è propagata al cliente, perché sono state scambiate *le copie locali alla procedura, non gli originali*
- al termine della procedura, le sue variabili locali sono state distrutte → nulla è rimasto del lavoro svolto dalla procedura

X

33

Y

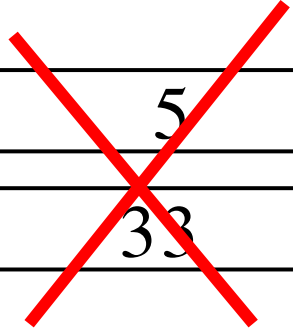
5

A

5

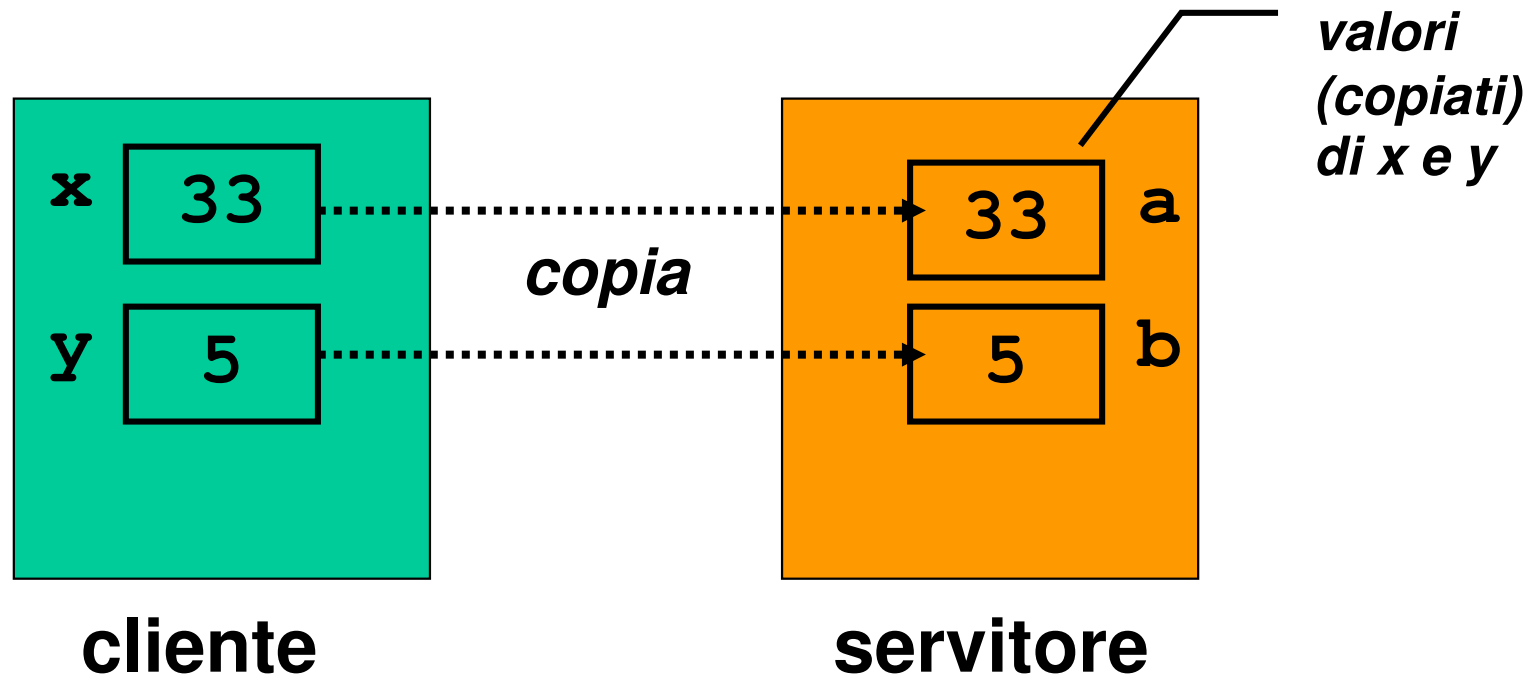
B

33



PASSAGGIO PER VALORE

Ogni azione fatta su *a* e *b* è strettamente locale al servitore. Quindi *a* e *b* vengono scambiati ma quando il servitore termina, tutto scompare



PASSAGGIO DEI PARAMETRI IN C

Il C adotta **sempre il passaggio per valore**

- le variabili del cliente e del servitore sono ***disaccoppiate***
- ma *non consente di scrivere componenti software il cui scopo sia diverso dal calcolo di una espressione*
- per superare questo limite occorre il ***passaggio per riferimento (by reference)***

PASSAGGIO PER RIFERIMENTO

Il passaggio per riferimento (*by reference*)

- NON trasferisce ***una copia del valore*** del parametro attuale
- ***ma un riferimento al parametro***, in modo da dare al servitore accesso diretto al parametro in possesso del cliente
- il servitore, quindi, ***accede direttamente*** al dato del cliente e ***può modificarlo***

PASSAGGIO DEI PARAMETRI IN C

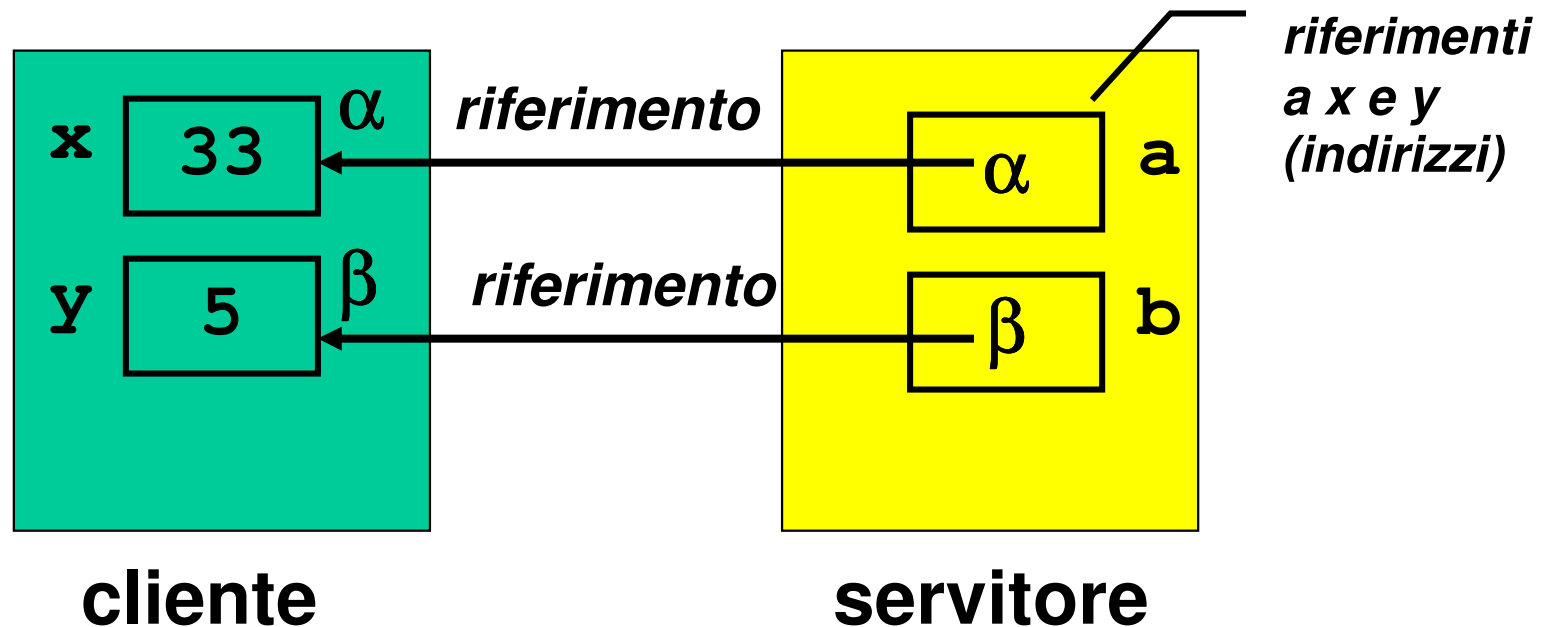
Il linguaggio C **NON** supporta *direttamente* il **passaggio per riferimento**

- è una grave mancanza
- viene fornito indirettamente solo per alcuni tipi di dato
- occorre quindi ***costruirlo quando serve***

PASSAGGIO PER RIFERIMENTO

Si trasferisce un riferimento ai parametri attuali (cioè i loro indirizzi)

Ogni azione fatta su **a** e **b**
*in realtà è fatta su **x** e **y***
nell'environment del cliente



REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

Il C *non* fornisce *direttamente* un modo per attivare il passaggio per riferimento -> a volte occorre *costruirselo*

È possibile costruirlo? Come?

- Poiché passare un parametro per riferimento comporta la capacità di manipolare ***indirizzi di variabili...***
- ... gestire il passaggio per riferimento implica la capacità di *accedere, direttamente o indirettamente, agli indirizzi delle variabili*

REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

In particolare occorre essere capaci di:

- ***ricavare l'indirizzo*** di una variabile
- ***dereferenziare un indirizzo*** di variabile, ossia “recuperare” il valore dato l'indirizzo della variabile

Nei linguaggi che offrono direttamente il passaggio per riferimento, *questi passi sono effettuati* in modo trasparente all'utente

In C il ***programmatore deve conoscere gli indirizzi*** delle variabili e quindi accedere alla macchina sottostante

INDIRIZZAMENTO E DEREFERENCING

Il C offre a tale scopo *due operatori*, che consentono di:

- *ricavare l'indirizzo* di una variabile
operatore estrazione di indirizzo &
- *dereferenziare un indirizzo* di variabile,
denotando la variabile (e il valore contenuto in
quell'indirizzo)
operatore di dereferenziamiento *

INDIRIZZAMENTO E DEREFERENCING

Se x è una variabile,

$\&x$ denota l'*indirizzo in memoria* di tale variabile:

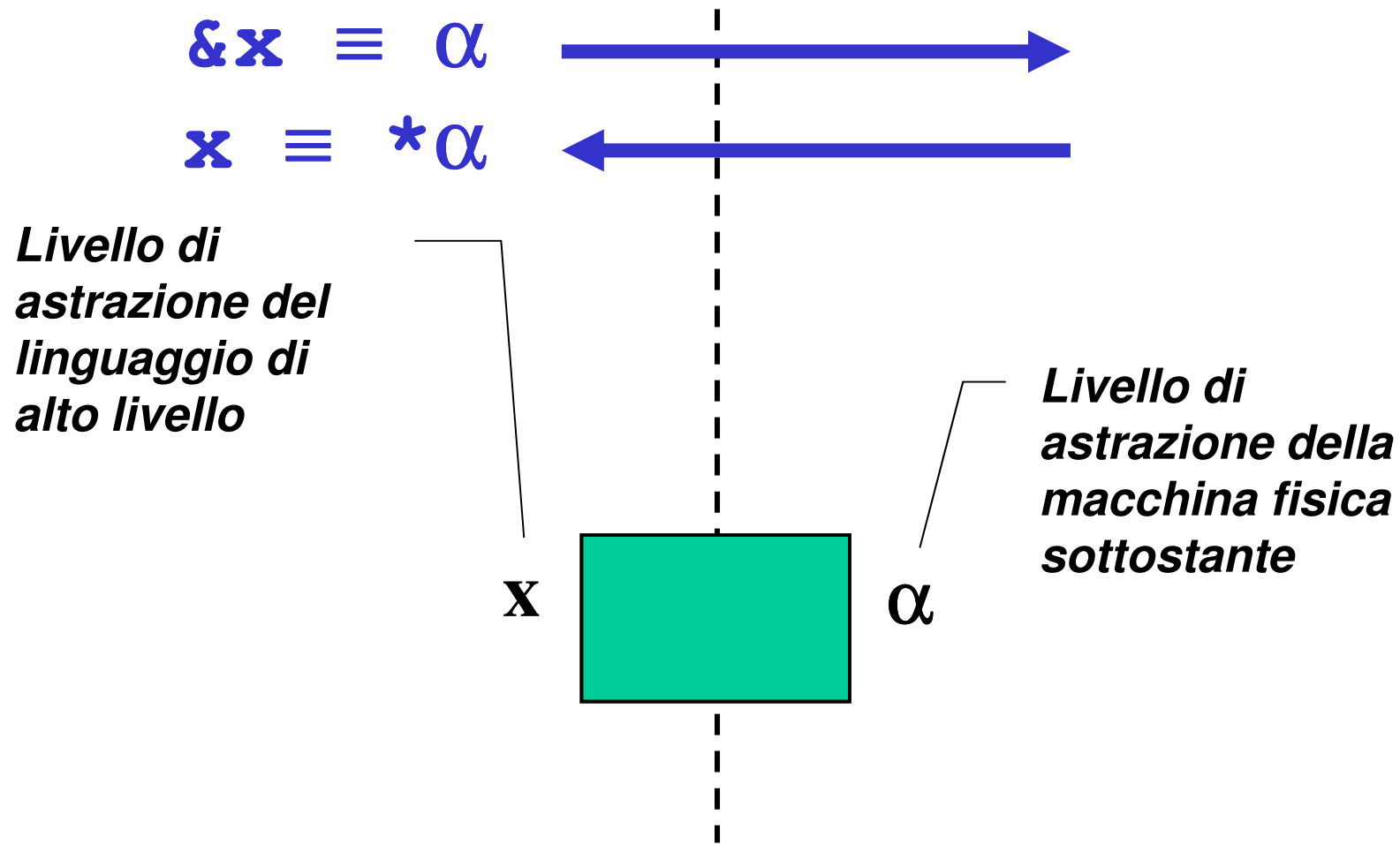
$$\&x \equiv \alpha$$

Se α è l'indirizzo di una variabile,

$*\alpha$ denota *tale variabile*:

$$x \equiv *\alpha$$

INDIRIZZAMENTO E DEREFERENCING



PUNTATORI

Un **puntatore** è il costrutto linguistico introdotto dal C (e da altri linguaggi) come *forma di accesso alla macchina sottostante* e in particolare agli **indirizzi di variabili**

- Un **tipo puntatore a T** è un tipo che denota l'indirizzo di memoria di una variabile di tipo T
- Un **puntatore a T** è una variabile di “*tipo puntatore a T*” che può contenere l'indirizzo di una variabile di tipo T

PUNTATORI

Definizione di una variabile puntatore:

<tipo> * <nomevariabile> ;

Esempi:

`int *p;`

`int* p;`

`int * p;`

Queste tre forme sono equivalenti e definiscono p come “puntatore a intero”

PASSAGGIO PER RIFERIMENTO IN C

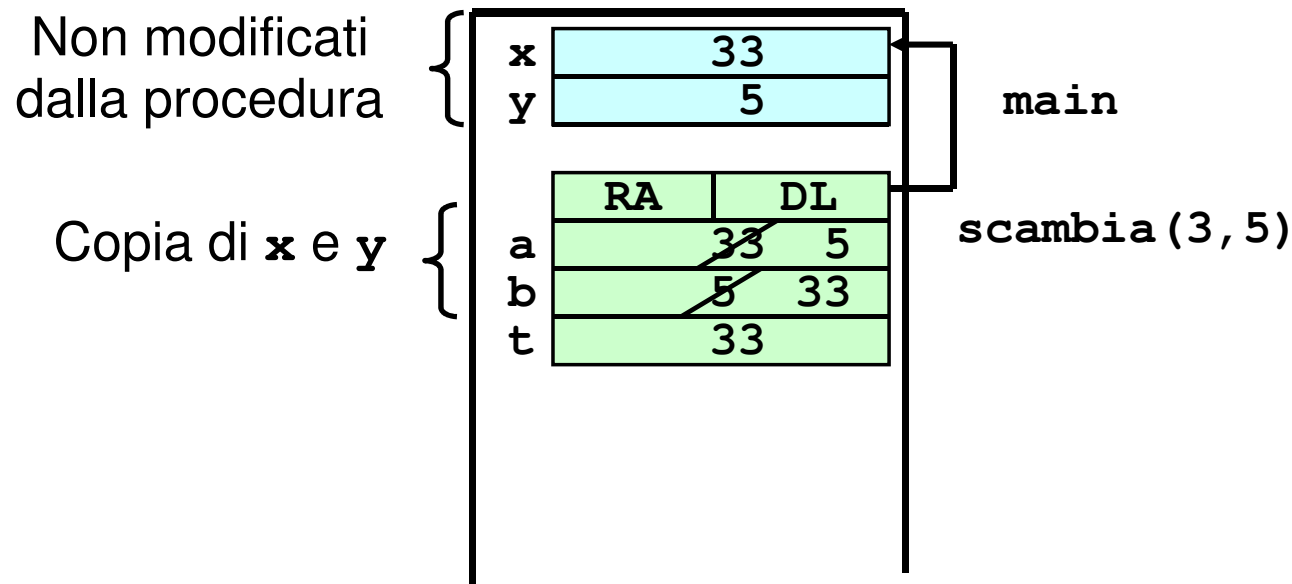
- il cliente deve passare esplicitamente gli indirizzi
- il servitore deve prevedere esplicitamente dei puntatori come parametri formali

```
void scambia(int* a, int* b) {  
    int t;  
    t = *a;  *a = *b;  *b = t;  
}
```

```
int main() {  
    int y=5, x=33;  
    scambia(&x, &y);  
}
```

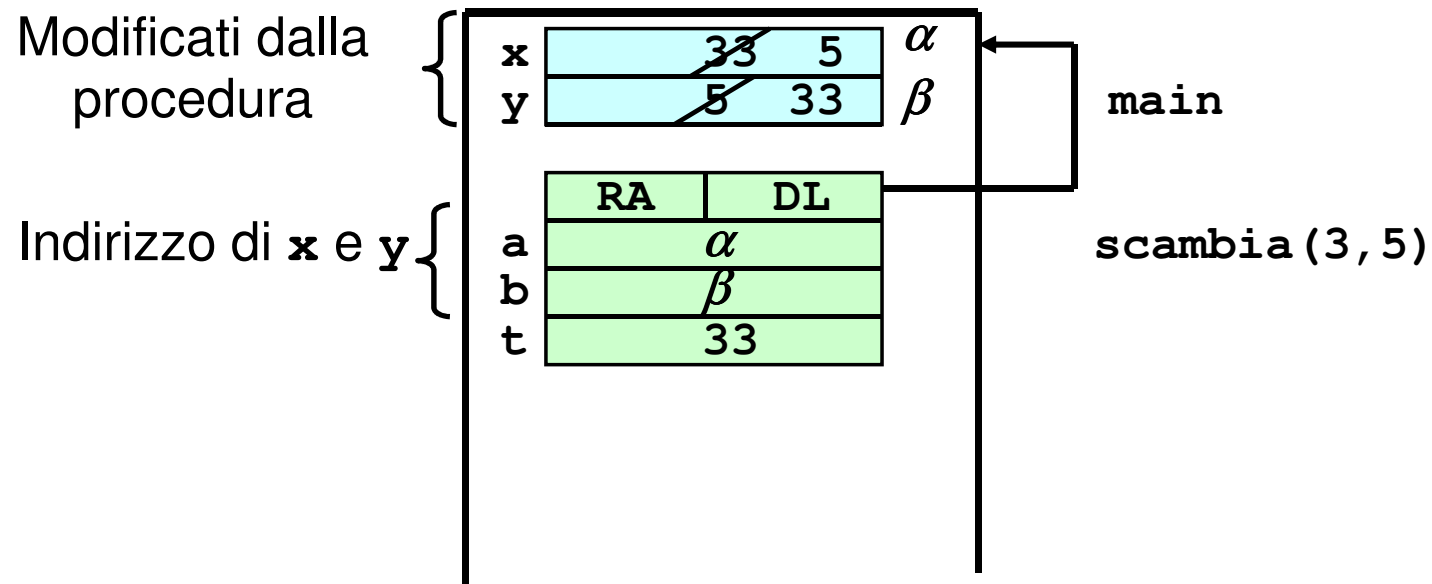
ESEMPIO: RECORD DI ATTIVAZIONE

Caso del *passaggio per valore*:



ESEMPIO: RECORD DI ATTIVAZIONE

Caso del *passaggio per riferimento*:



OSSERVAZIONE

Quando un puntatore è usato per realizzare il passaggio per riferimento, *la funzione **non dovrebbe mai alterare il valore del puntatore***

Quindi, se **a** e **b** sono due puntatori:

***a = *b** **SI**

~~**a = b**~~ **NO**

In generale una funzione PUÒ modificare un puntatore, ma *non è opportuno che lo faccia se esso realizza un passaggio per riferimento*

PUNTATORI

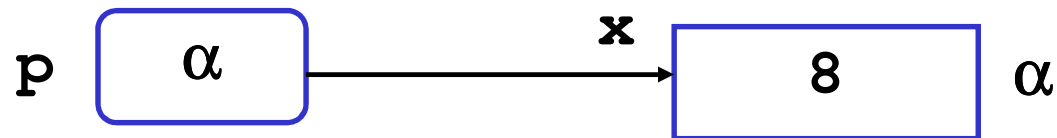
- Un **puntatore** è una variabile *destinata a contenere l'indirizzo di un'altra variabile*
- Vincolo di tipo: un puntatore a T può contenere solo l'indirizzo di variabili di tipo T

Esempio:

```
int  x = 8;
```

```
int* p;
```

```
p = &x;
```



Da questo momento, ****p*** e ***x*** sono ***due modi alternativi per denotare la stessa variabile***

PUNTATORI

`int x = 8;`

x 8 α

`int* p = &x;`

p α $\xrightarrow{\text{b}} \text{b}$ 8 α

`*p = 31;`

p α $\xrightarrow{\text{b}} \text{b}$ ~~8~~ 31 α

`x--;`

p α $\xrightarrow{\text{b}} \text{b}$ ~~31~~ 30 α

PUNTATORI

Un puntatore non è legato per sempre alla stessa variabile; può essere modificato

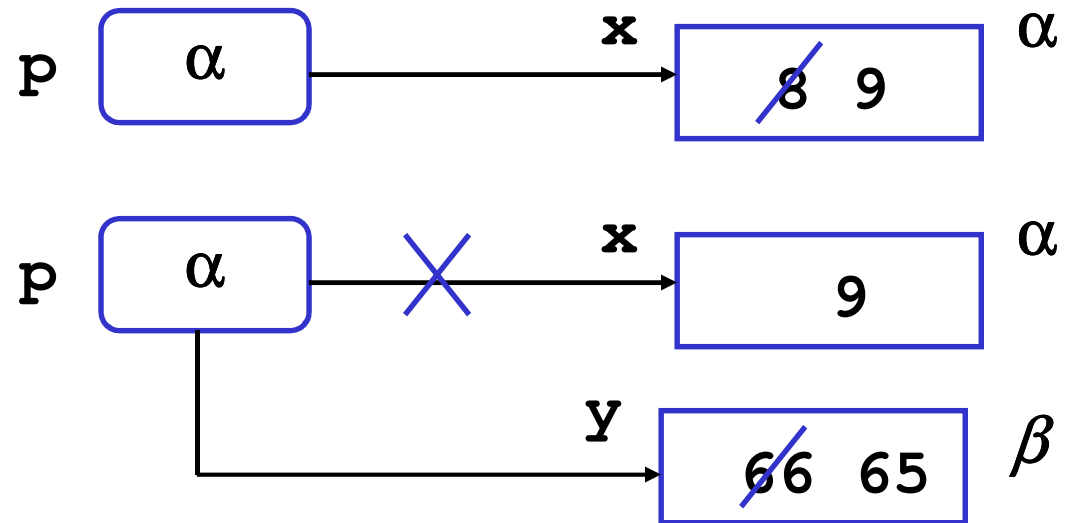
```
int x = 8, y = 66;
```

```
int *p = &x;
```

```
(*p) ++;
```

```
p = &y;
```

```
(*p) --;
```



Le parentesi sono necessarie? Che cosa identifica la scrittura `*p--` ?

PUNTATORI

Un puntatore a T può contenere *solo l'indirizzo di variabili di tipo T*: ***puntatori a tipi diversi sono incompatibili tra loro***

Esempio:

```
int x=8, *p;    float *q;  
p = &x;        /* OK */  
q = p;         /* NO! */
```

MOTIVO: il tipo *del puntatore* serve per *dedurre il tipo dell'oggetto puntato*, che è una ***informazione indispensabile per effettuare il dereferencing***

PUNTATORI

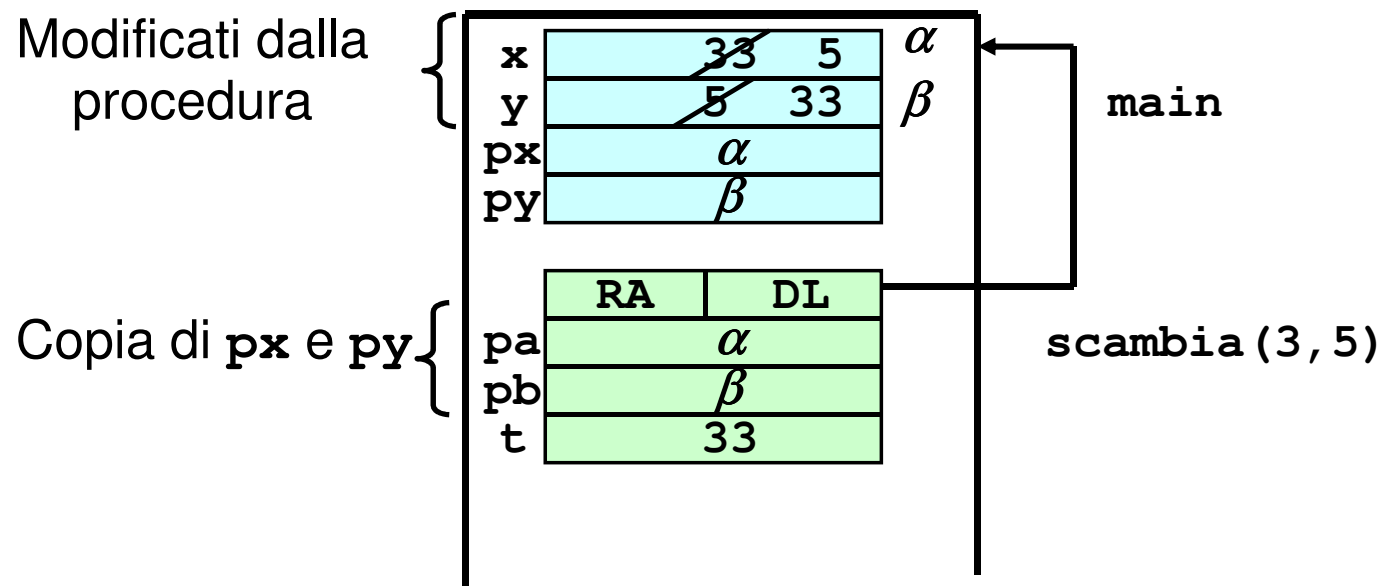
```
void scambia(int* pa, int* pb) {  
    int t;  
    t = *pa;  *pa = *pb;  *pb = t;  
}
```

```
int main() {  
    int y = 5, x = 33;  
    int *py = &y, *px = &x;  
    scambia(px, py);  
}
```

Variazione dall'esempio precedente: i puntatori sono memorizzati in `px` e `py` prima di passarli alla procedura

ESEMPIO: RECORD DI ATTIVAZIONE

Il record di attivazione si modifica come segue



COMUNICAZIONE TRAMITE ENVIRONMENT GLOBALE

Una procedura può anche comunicare con il cliente
mediante aree dati globali: ad esempio, *variabili globali*

Le *variabili globali* in C:

- sono allocate ***nell'area dati globale*** (fuori da ogni funzione)
- esistono ***prima*** della chiamata del ***main***
- sono visibili, previa dichiarazione ***extern***, in tutti i file dell'applicazione
- sono ***inizializzate automaticamente a 0*** salvo diversa indicazione
- possono essere *nascoste* in una funzione da una variabile locale omonima

ESEMPIO

Esempio: Divisione intera x/y con calcolo di quoziente e resto. Occorre calcolare *due* valori che supponiamo di mettere in due variabili globali

```
int quoziente, int resto;
```

variabili globali **quoziente** e **resto** visibili in tutti i blocchi

```
void dividi(int x, int y) {  
    resto = x % y; quoziente = x/y;  
}
```

```
int main() {  
    dividi(33, 6);  
    printf("%d%d", quoziente, resto);  
}
```

Il risultato è disponibile per il cliente nelle variabili globali **quoziente** e **resto**

SOLUZIONE ALTERNATIVA

Esempio: Con il passaggio dei parametri per indirizzo avremmo il seguente codice

```
void dividi(int x, int y, int* quoziente,  
            int* resto) {  
    *resto = x%y;    *quoziente = x/y;  
}  
  
int main() {  
    int k = 33, h = 6, quoz, rest;  
    int *pq = &quoz, *pr = &rest;  
    dividi(33, 6, pq, pr);  
    printf("%d%d", quoz, rest);  
}
```

Una Parentesi: PROGETTI SU PIÙ FILE SORGENTE

- Sono considerate “applicazioni di piccola dimensione”, applicazioni con qualche migliaio di linee di codice
- Un’applicazione anche “di piccola dimensione” non può essere sviluppata in un unico file → ***modularità, riutilizzo, leggibilità***
- Deve necessariamente essere strutturata ***su più file sorgente***
 - ***Compilabili separatamente***
 - Da collegare insieme successivamente per costruire l’applicazione

DICHIARAZIONE di FUNZIONE

La **dichiarazione** di una funzione è costituita dalla sola interfaccia, **senza corpo** (sostituito da un ;)

<dichiarazione-di-funzione> ::=
<tipoValore> <nome> (<parametri>) ;

- Per usare una funzione **non occorre conoscere tutta la definizione**
- È sufficiente conoscerne la **dichiarazione** ovvero la specifica del *contratto di servizio*

DICHIARAZIONE di FUNZIONE

- La ***dichiarazione*** specifica:
 - ***nome della funzione***
 - ***numero e tipo dei parametri*** (non necessariamente *il nome*)
 - ***tipo del risultato***

Nota: il nome dei parametri non è necessario, se c'è viene ignorato...

→ Avrebbe significato solo nell'ambiente di esecuzione (vedi record di attivazione) della funzione, che però al momento non esiste (non essendoci la definizione)

DICHIARAZIONE vs. DEFINIZIONE

- **Definizione: dice come è fatto il componente**
 - costituisce *l'effettiva realizzazione* del componente
 - ***NON può essere DUPLICATA***
 - compilazione di una definizione genera ***codice oggetto*** corrispondente alla funzione
- La ***dichiarazione*** di una funzione costituisce solo una ***specifica delle proprietà*** del componente
 - Può essere duplicata senza problemi

FUNZIONI e FILE

- `main()` può essere scritto dove si vuole nel file
 - viene invocato dal sistema operativo, che lo ***identifica sulla base del nome***
- Una funzione deve rispettare una regola fondamentale di visibilità
 - Prima che qualcuno possa invocarla, la funzione deve essere stata ***dichiarata*** (va bene anche definizione – contiene una dichiarazione)
 - ...altrimenti → **errore di compilazione!**

ESEMPIO: File Singolo

```
int fact(int);
```

*Dichiarazione (prototipo):
deve precedere l'uso*

```
int main()
```

```
{
```

```
    int y = fact(3);
```

```
    printf("%d", y);
```

```
    return 0;
```

```
}
```

Uso (invocazione)

```
int fact(int n)
```

```
{
```

```
    return (n<=1) ? 1 : n * fact(n-1);
```

```
}
```

Definizione

PROGETTI su PIÙ FILE

- Per strutturare un'applicazione su più file sorgente, occorre che ogni file ***possa essere compilato separatamente*** dagli altri
 - Successivamente avverrà il collegamento
- Affinché un file possa essere compilato, tutte le funzioni usate devono essere ***almeno dichiarate*** prima dell'uso
 - Non necessariamente definite

```
int fact(int);

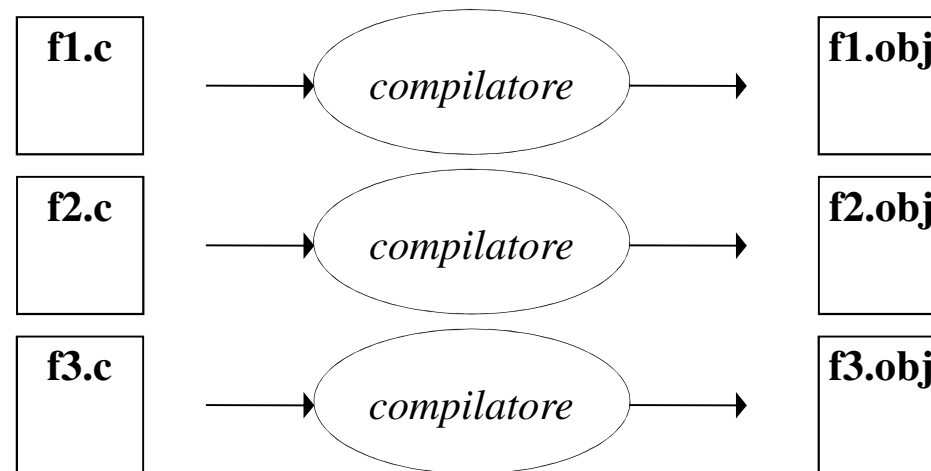
int main()
{
    int y = fact(3);
    printf("%d", y);
    return 0;
}
```

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


COMPILAZIONE di una APPLICAZIONE

2. **Compilare** i singoli file che costituiscono l'applicazione

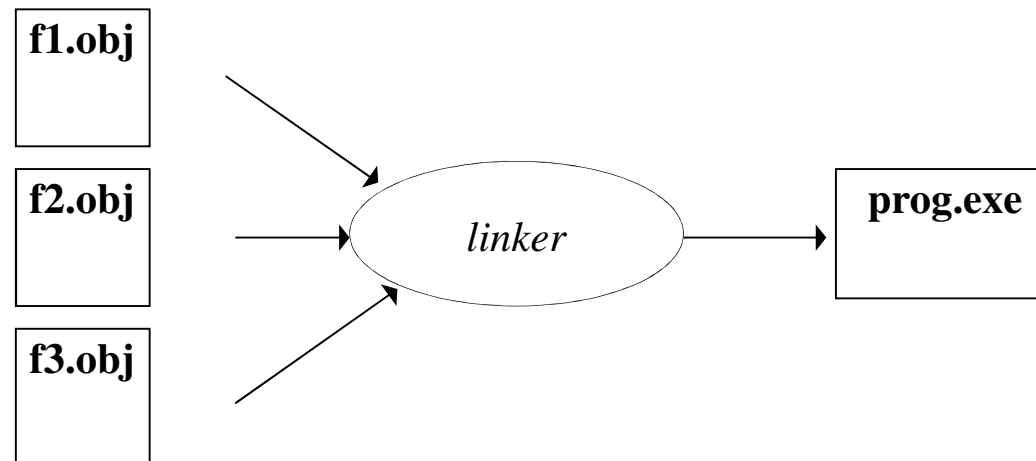
- File *sorgente*: estensione .c
- File *oggetto*: estensione .o oppure .obj



COMPILAZIONE di una APPLICAZIONE

2. Collegare i file oggetto fra loro e con le librerie di sistema

- File *oggetto*: estensione .o oppure .obj
- File *eseguibile*: estensione .exe o nessuna



RIASSUMENDO...

Perché la produzione dell'eseguibile vada a buon fine:

- ogni funzione deve essere definita una e una sola volta in uno e uno solo dei file sorgente
 - ***se la definizione manca, si ha errore di linking***
- ogni cliente che *usi* una funzione deve incorporare la dichiarazione opportuna
 - se la dichiarazione manca, ***si ha errore di compilazione*** nel file del cliente

PROGETTI COMPLESSI...

- Ogni ***chiamante deve contenere le dichiarazioni delle funzioni*** che utilizza

Per ***automatizzare la gestione delle dichiarazioni, si introduce il concetto di header file*** (file di intestazione)

- Scopo: evitare ai clienti di dover trascrivere riga per riga le dichiarazioni necessarie
 - il progettista predispone un *header file* contenente tutte le dichiarazioni relative alle funzioni definite nel suo componente software (o modulo)
 - i clienti potranno semplicemente includere tale header tramite ***direttiva #include***

HEADER FILE

Il *file di intestazione (header)*

- ha *estensione* **.h**
- ha (per convenzione) *nome uguale al file* **.c** di cui fornisce le dichiarazioni

Esempio:

- se la funzione **func1** è definita nel file **file2c.c**
- il corrispondente *header file*, che i clienti potranno includere per usare la funzione **func1**, dovrebbe chiamarsi **file2c.h**

HEADER FILE

Due formati:

1) `#include <libreria.h>`

include normalmente header di **libreria di sistema**
macchina runtime del C sa dove cercare file header (all'interno di un elenco di directory predefinite)

2) `#include "miofile.h"`

- include normalmente un header di "propria produzione"
Attenzione: un **header file dovrebbe contenere solo dichiarazioni** e non definizioni (sia di funzioni che di variabili)
 - Possibile problema di una definizione compilata più volte, generando poi anche errori di linking
 - E se servono **variabili globali utilizzate da più file sorgenti?**
 - Clausola **extern** (vedi lucidi seguenti)

Esempio: CONVERSIONE °F/°C

Suddivisione su due file separati

File main.c

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

File f2c.c

```
float fahrToCelsius(float  
    f) {  
    return 5.0/9 * (f-32);  
}
```

Esempio: CONVERSIONE °F/°C

Si può introdurre un file header per includere automaticamente la dichiarazione

File `main.c`

```
#include "f2c.h"  
int main() { float c = fahrToCelsius(86); }
```



File `f2c.h` (*header*)

```
float fahrToCelsius(float);
```


Esempio: CONVERSIONE °F/°C

Struttura finale dei file dell'applicazione

- Un file ***main.c*** contenente il ***main***
- Un file ***f2c.c*** contenente la funzione di conversione
- Un file ***header f2c.h*** contenente la dichiarazione della funzione di conversione
 - Incluso da main.c

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

Definizione (e inizializzazione) della variabile globale

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

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 chiamante e funzione chiamata *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 sopravvivere (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 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);

```

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

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 conflitto*, 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)

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***