



Puntatori a funzioni

Ver. 3

Indirizzo di una funzione

- Il nome di una funzione (anche contenuta in librerie) se non è seguito dalle parentesi equivale all'indirizzo di memoria della prima istruzione di quella funzione
- Si può invocare una funzione anche utilizzando il suo indirizzo di memoria dopo averlo memorizzato in una variabile
- L'indirizzo di memoria è di tipo puntatore e come tale può essere assegnato a una variabile, memorizzato in un vettore, passato a una funzione, restituito da una funzione

Definizione di variabili

- La sintassi:

tipo_restituito (**nome*) (*tipo_parametri*) ;
definisce la *variabile nome* come *puntatore a una funzione* che restituisce un valore del *tipo_restituito* indicato e che richiede argomenti del tipo *tipo_parametri* indicati

- Le parentesi intorno al nome sono necessarie, infatti senza:

*tipo_restituito *nome* (*tipo_parametri*) ;
si ha il *prototipo di una funzione* che restituisce un puntatore del *tipo_restituito* indicato e che richiede argomenti di tipo *tipo_parametri*

Definizione di variabili

■ Esempi

- `double (*fp)(double, double);`
definisce `fp` come variabile puntatore a una funzione che ha come parametri due `double` e restituisce un `int`
- `double (*ffp)();`
definisce `ffp` come variabile puntatore a una funzione che restituisce un `double`, non avendo indicato nulla per i suoi parametri, questi non vengono controllati
- `int (*fpv[3])(int);`
definisce `fpv` come *vettore* di 3 elementi, ciascuno dei quali è un puntatore a una funzione che ha come parametri un `int` e restituisce un `int`

Assegnazione di variabili

- Essendo normali variabili scalari, possono essere assegnate con '='
- Esempi
Considerando le seguenti funzioni (i prototipi):
`double log(double);`
`double pow(double, double);`
`int abs(int), isalpha(int);`
`fp = pow;`
`fpv[2] = abs;`
- I tipi della variabile e della funzione devono essere identici
`fpv[2] = pow; → Errore`

Chiamata della funzione

- Una variabile **fp** che contiene un puntatore a funzione permette di richiamare la funzione assegnata in due modi equivalenti:
 - $(* \mathbf{fp}) (\textit{argomenti})$
 - $\mathbf{fp} (\textit{argomenti})$
- Il primo metodo rende più evidente che si tratta di un puntatore a funzione e non del nome di una funzione, il secondo è più leggibile

```
c = (* fp) (a, b) ;
```

```
c = fp (a, b) ;
```

```
i = (* fpv [2]) (x) ;
```

```
i = fpv [2] (x) ;
```

Inizializzazione di variabili

- Le definizioni esterne inizializzano a `NULL` le variabili di tipo *puntatore-a-funzione*
- Esempi di inizializzazione
Considerando le funzioni `log`, `pow`, `abs` e `isalpha` già viste, si possono avere le seguenti inizializzazioni:
 - `double (*fp)(double, double) = NULL;`
inizializza `fp` a `NULL`, quindi non si può usare
 - `double (*fp)(double, double) = pow;`
inizializza `fp` con `pow`, avendo gli stessi prototipi si può scrivere `x=fp(1.4)` al posto di `x=pow(1.4)`

Inizializzazione di variabili

- `int (*fpv[3])(int)={isalpha,abs};`
inizializza `fpv[0]` con `isalpha`, `fpv[1]` con `abs` e `fpv[2]` con `NULL`, quindi si può usare `fpv[0]('a')` al posto di `isalpha('a')`, `fpv[1](x)` al posto di `abs(x)`, non si può usare `fpv[1]`. Notare che le `fpv[i]`, `isalpha` e `abs` hanno gli stessi prototipi
- `int (*fpv[3])(int)={NULL};`
inizializza le 3 `fpv[i]` a `NULL`, quindi non si possono usare
- `int (*fpv[3])(int)={log};`
vorrebbe inizializzare `fpv[0]` con `log`, ma i prototipi di `fpv[0]` e `pow` sono diversi per cui si ha un errore del compilatore

Confronto tra variabili

- È un normale confronto tra puntatori, permette ad esempio di determinare se il puntatore si riferisce a una certa funzione o no, oppure se è NULL

```
if (fp == pow)
    printf("fp punta a pow\n");
```

Utilizzo con typedef

- Con `typedef` si può definire un nuovo tipo, equivalente a un tipo puntatore a funzione avente un ben determinato tipo restituito e con un ben determinato numero di parametri ciascuno di un ben determinato tipo

- `typedef double (*fpType) (double, double);`
dichiara `fpType` come tipo "puntatore a funzione che restituisce un `double` con due argomenti `double`"

```
fpType fp = pow;
```

```
fpType fpv[5] = {pow, NULL, atan2};
```

Nota: `fpv[3]` e `fpv[4]` sono inizializz. a `NULL`

Passaggio a funzione

- Un puntatore a funzione fp può essere passato a una funzione $funz$ come argomento (ovviamente senza indicare le parentesi di fp)
- Questo permette di passare alla funzione $funz$ una funzione diversa a ogni chiamata, così da ottenere che la stessa funzione $funz$ si comporti in modo diverso
- Nella chiamata alla funzione $funz$ l'argomento deve essere il nome di una funzione (come già detto non è necessario anteporre '&')

Passaggio a funzione

- I parametri formali di una qualsiasi funzione hanno sempre la forma di una definizione di variabile, ossia viene indicato il nome e il tipo
- Quando il parametro formale è un function pointer, il suo tipo deve specificare i dettagli delle funzioni che gli verranno passate come argomento, quindi richiede siano indicati il tipo del valore restituito e il tipo degli argomenti
- Il parametro puntatore a funzione può essere quindi espresso in due modi equivalenti:
 - $\text{tipo_restituito} \ (* \text{fp}) \ (\text{tipo_parametri})$
 - $\text{tipo_restituito} \ \text{fp} \ (\text{tipo_parametri})$

Passaggio a funzione

- Nel corpo della funzione a cui viene passato il function pointer, questo viene ovviamente utilizzato con il nome del parametro formale (nell'esempio precedente era f_p):
 - $(*f_p)$ (*argomenti*)
 - f_p (*argomenti*)

Passaggio a funzione

- Esempio

Si vuole una funzione `calc` che, a seconda della funzione passata, calcoli la somma o la differenza dei valori passati

Si supponga di avere le due seguenti funzioni:

- `int piu(int a, int b) → somma 2 numeri int`

- `int meno(int a, int b) → sottrae 2 numeri int`

Le chiamate a `calc` saranno del tipo:

```
m = calc(12, 23, piu);
```

per calcolare la somma dei due valori

```
n = calc(12, 23, meno);
```

per calcolare la differenza dei due valori

Passaggio a funzione

```
int piu(int a, int b) {return a+b;}
int meno(int a, int b) {return a-b;}
int calc(int x, int y, int (*funz) (int, int))
{
    return funz(x, y);
}
int main()
{
    int m, n;
    m = calc(12, 23, piu); → somma
    n = calc(12, 23, meno); → differenza
    ...
}
```

prototipo funzioni chiamabili, **funz** viene dichiarato qui...

...e usato qui

Funzione restituita da funzione

- Una funzione può restituire un puntatore a funzione, due sono i metodi possibili

- **Primo metodo**

Si definisce con l'istruzione `typedef` il tipo del function pointer restituito dalla funzione chiamata:

```
typedef int (*TipoFunz) (int, int);
```

TipoFunz è un nuovo tipo: *puntatore-a-funzione-che-ha-2-argomenti-int-e-restituisce-un-int*

E lo si usa nel solito modo:

```
TipoFunz operaz (int x);
```


Funzione restituita da funzione

- ```
typedef int (*TipoFunz) (int, int);
int piu(int a, int b) {return a+b;}
int meno(int a, int b) {return a-b;}
TipoFunz operaz(int f)
{
 TipoFunz fpv[2]={piu, meno};
 return fpv[f];
}
main()
{
 int y;
 TipoFunz op; /* var. funct point */
 op = operaz(1); /* scelta operaz. */
 y = (*op)(12, 23);
}
```

# Funzione restituita da funzione

- **Secondo metodo** (meno chiaro)

Si scrive ogni cast esplicitamente

```
int (* operaz (int f)) (int, int)
```

```
{
```

```
 int (*fpv[2]) (int, int) = {piu, meno};
```

```
 return fpv[f];
```

```
}
```

- Definisce la funzione **operaz** che come parametro formale ha `f` di tipo `int` e restituisce un puntatore a funzione che restituisce un `int` e ha due parametri formali di tipo `int`
- la parte NON sottolineata è il tipo della funzione restituita da **operaz** (dove **(int, int)** sono i parametri della funzione restituita)

# Funzione restituita da funzione

## Secondo metodo (Continuazione)

```
main()
{
 int y;
 int (*op)(int, int); /* puntatore */
 op = operaz(1); /* scelta operaz. */
 y = (*op)(12, 23);
 return 0;
}
```

# Cast puntatore-a-funzione

- Si ricordi che il tipo di una funzione è definito dal tipo e numero dei parametri e dal tipo del valore restituito
- Un cast di tipo *puntatore-a-funzione* può essere premesso al nome di una funzione o a una variabile di tipo puntatore-a-funzione
- Detto cast permette di rendere la funzione sottoposta a cast compatibile con un'altra, ad esempio per passarla come argomento a un'altra funzione (vedere l'esempio della `qsort` seguente)

# Cast puntatore-a-funzione

- La compatibilità prodotta dal cast è solo sintattica, ma potrebbe mascherare una reale non-compatibilità

```
double (*fp)(int);
```

```
fp = (double (*)(int))strlen;
```

- Quanto sopra permette di assegnare (con un Warning) `strlen` a `fp` (che è una funzione di ben altro tipo), ma in realtà la `fp` così assegnata non può ricevere un `int` come parametro né restituisce un `double` (al runtime il comportamento è indefinito)

# Cast puntatore-a-funzione

- Il cast viene applicato *solo* all'identificatore della funzione, non ai suoi argomenti né al valore restituito

- ```
int (*intsqrt) (int);  
intsqrt = (int (*) (int)) sqrt;
```

Il cast indicato permette di passare `sqrt` a una funzione che richiede come parametro una funzione con un parametro `int` e valore restituito `int`, ma né l'argomento di `intsqrt` né il risultato vengono convertiti in `int`

- Argomenti e valore restituito devono essere compatibili per non avere errori al run-time

Passaggio a funzione con cast ²³

Esempi di utilizzo

- Esempi tipici di utilizzo del passaggio di un puntatore a funzione con cast sono l'utilizzo delle funzioni:
 - `bsearch` in `<stdlib.h>` che cerca un elemento in un vettore ordinato con il metodo di bisezione
 - `qsort` in `<stdlib.h>` che ordina un vettore con il metodo Quicksort
- Nel seguito si tratta in dettaglio l'uso della `qsort`, mentre la `bsearch` è nella soluzione di uno degli esercizi proposti

Passaggio a funzione con cast ²⁴

qsort

- La funzione `qsort` in `<stdlib.h>` ordina un vettore, deve essere chiamata indicando il nome della funzione da usare per il confronto degli elementi, il suo prototipo è:

```
void qsort(  
    void *base, → puntatore al vett da riordinare  
    size_t n, → dim del vett (numero di elementi)  
    size_t size, → dim di ogni elemento (in byte)  
    int (*cmp) (const void*, const void*)  
)
```

cmp è un puntatore-a-funzione a cui passare il puntatore alla funzione da utilizzare per confrontare due elementi di quel vettore

Passaggio a funzione con cast ²⁵

qsort

- Per la funzione passata per *cmp* si richiede che:
 - abbia come argomenti due puntatori agli elementi del vettore da confrontare (`qsort` passerà a *cmp* gli indirizzi dei due elementi mediante `&base[i]`)
 - restituisca un valore `int`:
 - `<0` se il 1° elemento è minore del 2°
 - `=0` se sono uguali
 - `>0` se il 1° elemento è maggiore del 2°
- Il tipo `void*` permette di passare puntatori di qualsiasi tipo: *sarà la funzione passata a utilizzarli dopo averli convertiti al tipo necessario* (il cast non si applica agli argomenti, questi devono essere compatibili)

Passaggio a funzione con cast ²⁶

qsort su vettore di scalari

- Ordinare un vettore di elementi scalari richiede che questi vengano scambiati di posto in base alla *relazione di precedenza* tra di essi: determinare quale “preceda” e quale “segua” (quale dei due sia il “minore”)
- Per valori scalari questa relazione è ovvia: un numero piccolo precede (è minore di) un numero grande
- La funzione `qsort` richiede il passaggio di una funzione di confronto, quindi è necessario definirne una che utilizzi i normali operatori relazionali applicate agli elementi stessi

Passaggio a funzione con cast²⁷

qsort su vettore di scalari

```
#include <stdlib.h>
#include <stdio.h>
int cfr(const int *a, const int *b)
{
    if (*a < *b) return -1;
    if (*a > *b) return +1;
    return 0;
}
int main()
{
    int vett[10] = {3,2,5,4,7,9,0,1,6,8};
    qsort(vett, 10, sizeof(int),
        (int (*)(const void*,const void*))cfr );
    return 0;
}
```

Passaggio a funzione con cast ²⁸

qsort su vettore di scalari

- La *compatibilità sintattica* di `cfr` con la *cmp* del prototipo di `qsort` è assicurata dal cast:
`(int (*)(const void*, const void*)) cfr`
- La *compatibilità effettiva* di `cfr` con la *cmp* deriva dall'aver prototipo compatibile:
 - il tipo restituito è `int`
 - gli argomenti sono puntatori-a-`int`, questi sono compatibili con (assegnabili a) puntatori-a-`void` anche senza cast (il cast sulla funzione non fa il cast degli argomenti e qui non serve)
 - gli argomenti sono `const` come nel prototipo (solo un valore `const` può essere assegnato a una variabile `const` senza Warning)

Passaggio a funzione con cast²⁹

qsort su vettore di scalari

- Implementazioni alternative della `cfr`

```
int cfr(const void *p, const void *q)
{
    const int *a = p;
    const int *b = q;
    if (*a < *b) return -1;
    if (*a > *b) return +1;
    return 0;
}
```

- Quando viene chiamata la `cfr`, sono passati i `&vett[i]` che sono puntatori-a-int, questi sono convertiti in puntatori-costanti-a-void dal prototipo e nella funzione sono poi assegnati a puntatori-costanti-a-int

Passaggio a funzione con cast³⁰

qsort su vettore di scalari

- Implementazioni alternative della `cfr`

```
int cfr(const void *p, const void *q)
{
    return *(int *)p - *(int *)q;
}
```

- Qui cambia il metodo di calcolo del risultato, ma importa notare che i `&vett[i]` passati, che sono puntatori-a-int, sono convertiti in puntatori-costanti-a-void dal prototipo e nella funzione convertiti in puntatori-costanti-a-int
- Poiché i cast non producono un valore assegnabile (sono r-value), è implicita la conservazione di `const` negli `int *`



Passaggio a funzione con cast ³¹

qsort su vettore di scalari

- Negli ultimi due esempi, la **cfr** ha lo stesso identico prototipo della *cmp* quindi la chiamata non richiede alcun cast:

```
qsort(vett, 10, sizeof(int), cfr);
```

Passaggio a funzione con cast ³²

qsort su vettore di strutture

- Ordinare un vettore di strutture richiede che gli elementi vengano scambiati di posto in base alla *relazione di precedenza* tra di essi: bisogna stabilire quale "precede" e quale "segue" (quale dei due sia il "minore")
- La relazione tra due strutture dipende dal valore dei suoi membri e quindi può essere diversa a seconda del membro considerato
- La funzione di confronto dovrà pertanto basarsi sul valore di un membro della struttura
- La relazione di precedenza può essere più complessa utilizzando più membri

Passaggio a funzione con cast ³³

qsort su vettore di strutture

- Negli esempi che seguono si useranno:

- una `struct` definita come:

```
struct intstr {  
    int x;  
    char nome[N];  
};
```

- funzioni di confronto con prototipo:

```
int (*) (const void*, const void*)
```

- quindi le chiamate saranno senza cast sulla funzione di confronto:

```
qsort(vett, n, sizeof(struct intstr),  
      funzione_di_confronto);
```

Passaggio a funzione con cast ³⁴

qsort su vettore di strutture

- La funzione di confronto che si basa sul membro `x` di tipo `int` sarà:

```
int strcmp_x(const void *p,  
             const void *q)  
{  
    const struct intstr *a = p;  
    const struct intstr *b = q;  
  
    if (a->x > b->x) return +1;  
    if (a->x < b->x) return -1;  
    return 0;  
}
```

Passaggio a funzione con cast ³⁵

qsort su vettore di strutture

- Forma alternativa per il confronto su `x`:

```
int  structmp_x(const void *p,
                const void *q)
{
    if ( ((struct intstr *)p)->x >
         ((struct intstr *)q)->x)
        return +1;
    if ( ((struct intstr *)p)->x <
         ((struct intstr *)q)->x)
        return -1;
    return 0;
}
```

Senza le parentesi blu il cast sarebbe su `p->x`

Poiché i cast non producono un valore assegnabile (sono r-value), è implicita la conservazione di `const`

Passaggio a funzione con cast ³⁶

qsort su vettore di strutture

- Altra forma alternativa per il confronto su x , sottraendo i valori:

```
int strucmp_x(const void *p,  
              const void *q)  
{  
    return ((struct intstr *)p) ->x -  
           ((struct intstr *)q) ->x;  
}
```

Poiché i cast non producono un valore assegnabile (sono r-value), è implicita la conservazione di `const`

Passaggio a funzione con cast ³⁷

qsort su vettore di strutture

- Le funzioni di confronto che si basano sul membro `nome` di tipo `char[N]` saranno simili alle precedenti, ma usando la `strcmp` per il confronto dei membri `nome`

- Modificando ad esempio l'ultima vista si ha:

```
int strucmp_nome (const void *p,  
                  const void *q)  
{  
    return strcmp(  
        ((struct intstr *) p) ->nome,  
        ((struct intstr *) q) ->nome);  
}
```

Passaggio a funzione con cast ³⁸

qsort su vettore di stringhe

- Ordinare un vettore di stringhe con `qsort` è possibile solo per *vettori di puntatori a char* come `char *vs[N]`: in questo modo possono essere scambiate le variabili puntatore `vs[i]`
- Al contrario, la `qsort` non funziona su vettori di stringhe costituiti da matrici di caratteri come `char vs[N][M]` perché i `vs[i]` non sono variabili puntatori a stringhe, ma indirizzi di memoria costanti, quindi non possono essere scambiati (si può definire un vettore di puntatori inizializzato con i `vs[i]` e si ordina quello)

Passaggio a funzione con cast ³⁹

qsort su vettore di stringhe

- Per determinare la reciproca relazione tra due stringhe, ossia quale tra due “preceda” l’altra, ma non si può semplicemente passare la funzione `strcmp`
- Gli elementi che `qsort()` passa alla funzione di confronto sono `&vs[i]`, cioè gli *indirizzi dei puntatori* `vs[i]` (quindi dei puntatori-a-puntatori-a-char), mentre `strcmp()` richiede puntatori-a-char, cioè proprio i puntatori `vs[i]`
- È necessaria una funzione che riceva i puntatori ai `vs[i]` e chiami `strcmp` passandole i puntatori `vs[i]` stessi

Passaggio a funzione con cast ⁴⁰ qsort su vettore di stringhe

- Però i puntatori a `vs[i]` (che sono puntatori-a-puntatori-a-char), quando viene chiamata la `qsort` sono convertiti in semplici puntatori-costanti-a-void dal prototipo, quindi nella funzione devono essere assegnati a puntatori-a-puntatori-costanti-a-char (costanti perché è l'oggetto puntato da `const void` che è costante)
- La deferenziamento di questi valori puntatori-a-puntatori-costanti-a-char produce dei puntatori-costanti-a-char (`vs[i]`) come la `strcmp` richiede (che siano costanti qui non è più importante)

Passaggio a funzione con cast ⁴¹

qsort su vettore di stringhe

- La funzione ausiliaria sarà la seguente:

```
int scp(const void *p, const void *q)
{
    char * const *a=p;
    char * const *b=q;
    return strcmp(*a, *b);
}
```

Passaggio a funzione con cast ⁴²

qsort su vettore di stringhe

- Implementazione alternativa della **scp** :

```
int scp(const void *p, const void *q)
{
    return strcmp(*(char **)p,
                  *(char **)q);
}
```

- Poiché i cast non producono un valore assegnabile (sono r-value), è implicita la conservazione di `const` negli `int *`

Esercizi

1. Scrivere un programma che riordini con `qsort` un vettore di stringhe
2. Scrivere un programma che
 1. definisca un vettore di 10 `struct` contenenti due valori interi `x` e `y`
 2. inizializzi il vettore
 3. ordini il vettore con `qsort` in base alla somma dei valori (se due somme sono uguali, il confronto è tra i primi membri, es. $\{4, 6\} > \{3, 7\}$)
 4. cerchi un elemento, dati i due valori in input, con la `bsearch` di `<stdlib.h>` e ne fornisca l'indice