Considerazioni sui Puntatori restituiti dalle Funzioni

Una funzione può restituire un puntatore ad un qualche tipo di dato, ma la correttezza nell'uso di questo puntatore si può fare, dipende da come lo spazio in memoria è allocato. Cioè:

esempio corretto: p è allocato dinamicamente quindi, anche se p è una variabile locale, lo spazio allocato sopravvive alla terminazione della funzione

```
int *alloca_vettore_1(  int size )
{
    int *p;
    p = malloc(size*sizeof(int));
    return p ;
}
```

```
void main(void)
{
    int *v;
    v = alloca_vettore ( 10 ) ;
    ...  usa v ....
}
```

---

esempio sbagliato: p è una variabile locale, lo spazio allocato non sopravvive alla terminazione della funzione

```
int *alloca_vettore_2(  int size )
{
    int p[1000];
    return p ;
}
```

```
void main(void)
{
    int *v;
    v = alloca_vettore ( 10 ) ;
    ... usa v .....
}
```

---

esempio corretto: vet_globale è una variabile globale, quindi sopravvive alla terminazione della funzione, ma cosìsia chiaro

```
int vet_globale[10000];

int *spazio_pre_allocato( void )
{
    return vet_globale ;
}
```

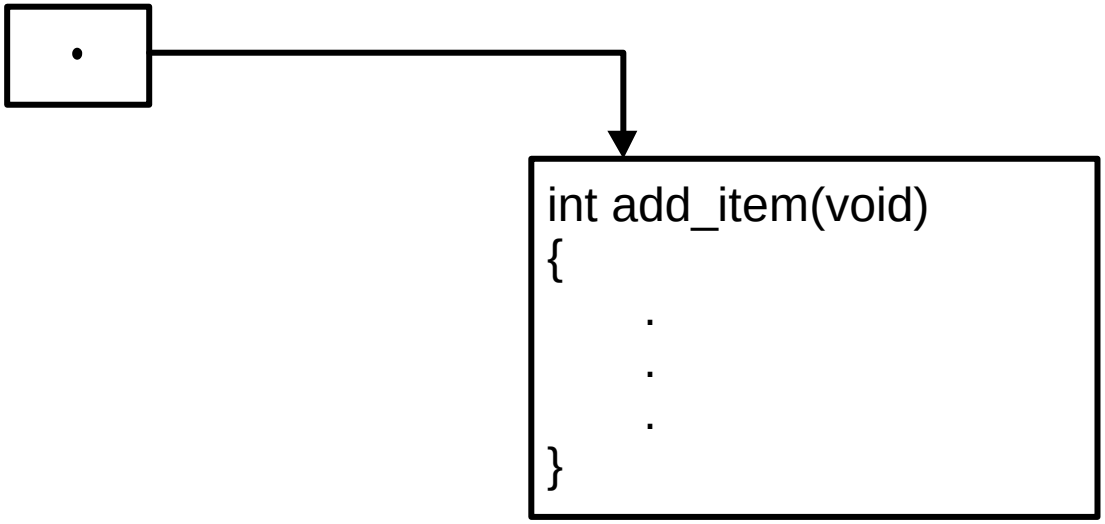
```
void main(void)
{
    int *v;
    v = spazio_pre_allocato();
    ...  usa v ....
}
```

Pointers to Functions

Function names are pointer constants

- not a variable
- can regard as address of function code
- like array name is a pointer constant

So we can have pointers to functions



Puntatori a funzione.

In C è possibile utilizzare dei puntatori a funzioni, ovvero delle variabili a cui possono essere assegnati gli indirizzi in cui risiedono le funzioni, e tramite questi puntatori a funzione, le funzioni puntate possono essere chiamate all'esecuzione.

Confrontiamo la **dichiarazione di una funzione**:

tipo_restituito **nome_funzione** (paramdef1, paramdef2, ...)

con la **dichiarazione di un puntatore a funzione**:

tipo_restituito **(* nome_ptr_a_funz)** (paramdef1, paramdef2, ...)

dove:

- paramdef1, paramdef2, ecc. sono la definizione degli argomenti da passare alla funzione all'atto della chiamata (ad es.: int i).
- tipo_restituito è il tipo del dato che viene restituito come risultato dalla funzione.

ad es, la seguente dichiarazione definisce un puntatore a funzione che punta a funzioni le quali prendono come argomenti due double, e restituiscono un double .

```
double (*ptrf) ( double g, double f);
```

Il C tratta i nomi delle funzioni come se fossero dei puntatori alle funzioni stesse.

Quindi, **quando vogliamo assegnare ad un puntatore a funzione l'indirizzo di una certa funzione dobbiamo effettuare un operazione di assegnamento del nome della funzione al nome del puntatore a funzione**

Se ad es. consideriamo la funzione dell'esempio precedente:

double somma(double a, double b);

allora **potremo assegnare la funzione somma al puntatore ptrf** così:

ptrf = somma;

Analogamente, **l'esecuzione di una funzione mediante un puntatore che la punta,viene effettuata con una chiamata in cui compare il nome del puntatore come se fosse il nome della funzione**, seguito ovviamente dai necessari parametri.

---------- esempio di uso dei puntatori a funzione ----------

per es. riprendiamo l'esempio della somma:

```
double somma( double a, double b) ;          /*  dichiarazione  o prototipo */

void main( void)
{
      double A=10 , B=29,  C;
      double (*ptrf) ( double g, double f);

      ptrf = somma;
      C = ptrf (A,B);                        /* chiamata alla funz. somma */
}

double somma( double a, double b)            /* definizione */
{     return a+b ;     }
```

Osservazione: spesso è complicato definire il tipo di dato puntatori a funzione, ed ancora di più definire funzioni che prendono in input argomenti di tipo puntatori a funzione.
In queste situazioni **è sempre bene ricorrere alla typedef per creare un tipo di dato puntatore a funzione** per funzioni che ci servono, ed utilizzare questo tipo di dato nelle altre definizioni.

---- Usare la typedef per rendere leggibile il codice C ---

Esempio:
esiste in ambiente unix (l'esempio è preso da LINUX Slackware) una funzione detta signal che puo' servire ad associare una funzione al verificarsi di un evento. Ad es. puo' servire a far eseguire una funzione allo scadere di un timer.
Il prototipo della funzione, contenuto in signal.h, e' il seguente:

```
#include <signal.h>
void (*signal(int signum, void (*handler)(int) ) )(int);          /* ????? */
```

NON E' SUBITO CHIARISSIMO COSA SIA 'STA ROBA !!!

Il significato è che
1) la funzione signal vuole come parametri un intero ***signum***, ed un puntatore ***handler*** ad una funzione che restituisce un void e che vuole come parametro un intero.
2) la funzione signal restituisce un puntatore ad una funzione che restituisce un void e che vuole come parametro un intero.

Converrebbe definire un tipo di dato come il puntatore a funzione richiesta:

```
typedef  void (*tipo_funzione) (int);
```

ed utilizzarlo per definire la signal così:

```
tipo_funzione   signal ( int signum,  tipo_funzione handler );
```

Nel man della signal e' presente questo **commento**:

> If you're confused by the prototype at the top of this manpage, it may help to see it separated out thus:
>
> ```
> typedef void (*handler_type)(int);
> handler_type signal(int signum, handler_type handler);
> ```

Pointers to Functions

Declaring pointers to functions

cannot omit parenthesis

int (*p)(); p is ptr to function that returns an int

int *p(); p is function that returns ptr to int

assign address of function as with array

```
int fname(void);
int (*fnptr)(void);
fnptr = fname;
```

fname *è un indirizzo!!!!!*

Pointers to Functions

Usage of pointers to functions

- are variables, just like other pointers
- can be used as arguments to functions to “pass” one function to another
- used to direct a sort
  - to make it data type independent
  - to allow choices of different function calls for ascending vs. descending
- used to implement “dispatch” tables
  - as might be used in a menuing system

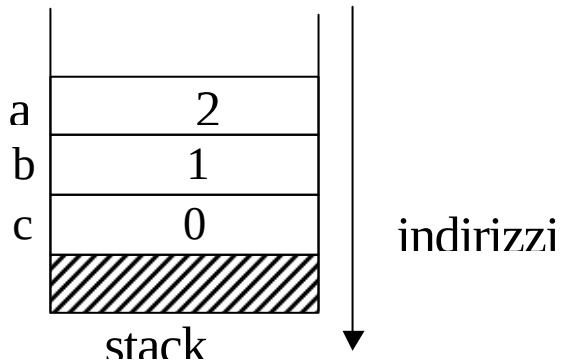
Ordine di Valutazione degli argomenti passati alle funzioni.

Il compilatore C opera in modo che le espressioni passate come argomenti alle funzioni sono prima valutate, ed il risultato della valutazione viene scritto sullo stack per essere disponibile al codice che implementa la funzione stessa.
Sorgono due problemi:
- in che ordine vengono valutate le espressioni ?
- in che ordine vengono scritti sullo stack i risultati delle valutazioni ?

La risposta alle due domande è la stessa:
vengono prima valutate (e il risultato scritto sullo stack) le espressioni passate come ultimo argomento della funzione (le più a destra), e poi via via quelle più a sinistra.

es:

```
void stampa3( int a, int b, int c){
      printf("a=%d       b=%d       c=%d\n", a, b, c);
      printf("&a=%p     &b=%p    &c=%p\n", &a, &b, &c);
}
void main( void) {
      int x=0;
      stampa3 (x++, x++, x++);
}
```



L'output del programma sarà del tipo:
a=2 b=1 c=0 (c valutato per primo)
&a=9003:0FF8 &b=9003:0FFA &c=9003:0FFC (c primo su stack)
che indica come il terzo argomento (ultimo, cioè più a destra) venga valutato (e scritto sullo stack) per primo, e poi gli argomenti via via più a sinistra.

Funzioni con numero di argomenti variabile.

In C e' possibile definire funzioni aventi un numero di argomenti variabile, cioe' funzioni che in chiamate diverse possono avere un numero di argomenti diverso, **ma ne debbono avere sempre almeno uno**. Ne è un esempio la printf();
Si utilizza a questo scopo una struttura **va_list** definita nel file stdarg.h .
Vediamo un esempio di funzione che riceve in input n argomenti, di cui il primo e' un intero che contiene il numero di argomenti seguenti, e gli altri sono delle stringhe, cioe' dei puntatori a char. La funzione deve solo stampare tutti gli argomenti passati.

```
void print_lista_argomenti(int narg, ...)
{
va_list va;
int i;
char *ptrchar;

/* va_start inizializza va_list all'argomento, tra passati a print_lista_argomenti, che segue l'argomento narg indicato come secondo argomento nella va_start, cioè inizializza la lista va_list al primo degli argomenti variabili  */
va_start(va,narg);
for(i=0;i<narg;i++)
   {
   ptrchar=va_arg(va,char*);  /*ptrchar punta alla stringa passata*/
   printf("%d %s\n",i,ptrchar);
   }
va_end(va);
}
```

la funzione potra' essere chiamata in una di questi modi

```
print_lista_argomenti(0);
print_lista_argomenti(5,"primo","secondo","terzo","quarto","quinto");
```

Vediamo un esempio complicato ma utile di uso della lista di argomenti variabili. Stampa gli argomenti passati, anche se sono di tipo diverso. Usa un primo parametro come formato per sapere cosa viene passato nei successivi argomenti, indicando con s una stringa, con d un intero, con c un carattere.

```
#include <stdio.h>
#include <stdarg.h>
void stampa_argomenti(char *fmt, ...)
{
    va_list ap;
    int d;
    char c, *p, *s;

    va_start(ap, fmt);
    while (*fmt)
        switch(*fmt++) {
        case 's':          /* string */
            s = va_arg(ap, char *);
            printf("string %s\n", s);
            break;
        case 'd':          /* int */
            d = va_arg(ap, int);
            printf("int %d\n", d);
            break;
        case 'c':          /* char */
            c = va_arg(ap, char);
            printf("char %c\n", c);
            break;
        }
    va_end(ap);
}

void main (void) {
      stampa_argomenti ( "sdc" , "stringa" , (int) 10 , (char)'h' );
      stampa_argomenti ("s" , "stringa2");
}
```