## Espressioni

Una espressione $E$ del linguaggio C può essere definita formalmente come segue (definizione induttiva):

- $E$ è una espressione semplice.
- Sia $Op_n$ un operatore del linguaggio di arità $n$ ($n = 1, 2, 3$) e siano $E_1, \ldots, E_n$ delle espressioni. Allora

$$Op_n(E_1, \ldots, E_n)$$

è una espressione.



## Valutazione di una espressione

Ogni espressione $E$ ha:

- un tipo $T$ (`int`, `double`,...), che può essere determinato a compile time (dipende dalle definizioni nel programma).
- un valore (di tipo $T$), che dipende dalla configurazione di memoria nel momento in cui il valore di $E$ è calcolato.

Valutare $E$ significa determinare il valore di $E$. La valutazione di una espressioni può produrre come *side effect* (effetto collaterale) la modifica dello stato del sistema (esempio, espressioni che usano gli operatori ++ e --).

La sintassi delle espressioni è molto complessa (vedere i manuali). Vediamo solo qualche esempio.



## Espressioni semplici: costanti

Una costante è una espressione semplice, il cui tipo e valore sono quelli denotati dalla costante (vedere i manuali).

Il valore di una costante non dipende dall'istante in cui è valutata.

Esempio

*La costante* `-12` *ha tipo* `int` *e valore -12.*
*La costante* `2.4354` *ha tipo* `double` *e valore 2.4354.*
*La costante* `13` *ha tipo* `int` *e valore 13.*
*La costante* `13.0` *ha tipo* `double` *e valore 13..*
*La costante* `'a'` *ha tipo* `int` *e valore 97.*



## Espressioni semplici: variabili

Una variabile è una espressione semplice, il cui tipo è il tipo con cui la variabile è stata dichiarata e il valore è il valore della variabile nel momento in cui avviene la valutazione.

Esempio
Dopo le istruzioni

```
int x = 10, a, b;
a = x - 4; // (A)
x = -3;
b = x - 10;  // (B)
```

In (A), $x$ è una espressione di tipo `int` e valore 10, quindi $x - 4$ è una espressione di tipo `int` e valore 6.
In (B), l'espressione $x$ vale -3, quindi $x - 10$ vale -13.

## Esempi di espressioni composte

- Espressioni arimetiche
  Sono costruite con gli operatori aritmetici binari (arietà 2) + (somma), - (differenza), * (prodotto), / (divisione). Il valore di una espressione aritmetica si ottiene applicando la corrispondente operazione aritmetica, il tipo dell'espressione dipende dal tipo delle sottoespressioni (vedere i dettagli sui manuali).

  Esempio

```
int x=10, y=20;
```

| Espressione | Tipo | Valore |
|---|---|---|
| x | int | 10 |
| y | int | 20 |
| 2*(x+y) | int | 60 |

- Chiamata di funzione
  Una chiamata di funzione è una espressione il cui tipo è il tipo dalla funzione e il valore è quello restituito dalla funzione.



## Puntatori

Sia `x` una variabile di di tipo `T` (`int`, `double`, ...) e sia $L$ la locazione di memoria denotata da `x` (in altri termini, `x` è il *nome* della locazione $L$).

L'espressione

```
&x
```

costruita con l'operatore unario '`&`' di indirizzamento

- ha tipo `T*` (*puntatore* al tipo $T$);
- ha come valore l'*indirizzo* della locazione di memoria $L$.



Più in generale:

- Sia `E` un'espressione di tipo `T` che denota una locazione di memoria $L$ di tipo `T` (ossia, `E` può contenere valori di tipo `T`). L'espressione

```
&E
```

  - ha tipo `T*` (*puntatore* al tipo `T`)
  - ha come valore l'*indirizzo* della locazione di memoria $L$.

La definizione precedente è una caso particolare della seconda in quanto una variabile è un'espressione semplice.



Esempio
Supponiamo di definire le variabili:

```
int a;
int *q;
int **r;
```

Allora:

- La variabile `a` ha tipo `int` e denota una locazione di memoria di tipo `int`.
- La variabile `q` ha tipo `int*` (puntatore al tipo `int`) e denota una locazione di memoria di tipo `int*` (`q` può contenere indirizzi a locazioni di memoria di tipo `int`).
- La variabile `r` ha tipo `int**`, da leggersi come `(int*)*` (puntatore a un puntatore a `int`) e denota una locazione di memoria di tipo `int**` (`r` può contenere valori indirizzi a locazioni di memoria di tipo `int*`).

## Attenzione agli asterischi

Nelle definizioni precedenti, l'asterisco va associato al tipo, non fa parte del nome della variabile.

Le definizioni

```
int*  q;       int *  q;       int *q;
```

sono equivalenti: tutte definiscono una variabile di *nome* `q` e di *tipo* `int*` (puntatore a `int`).

Le definizioni

```
int a;
int *q;
int **r;
```

possono anche essere scritte usando un'unica definizione:

```
int a, *q, **r; /* definizione di tre variabili */
```



## Definizione con inizializzazione

Anche le variabili di tipo puntatore possono essere inizializzate quando vengono definite.

Esempio

Con le definizioni

```
int a = 5, *q = &a , **r = &q;
```

- La variabile `a` (tipo `int`) ha come valore iniziale 5.
- La variabile `q` (tipo `int*`) ha come valore iniziale l'indirizzo di `a` (si dice che "`q` *punta ad* `a`").
- La variabile `r` (tipo `int**`) ha come valore iniziale l'indirizzo di `q` (`r` punta a `q`).

Nota

Verificare che l'espressione a destra di = ha lo stesso tipo della variabile a sinistra di =.



## Dimensione del tipo puntatore

L'occupazione di memoria di un puntatore dipende dal sistema, ma *non* dal tipo del puntatore. Le variabili di tipo `T*`, dove `T` è un tipo qualunque, occupano tutte lo *stesso* spazio di memoria (spazio necessario per rappresentare un indirizzo della memoria).

Per determinare l'occupazione in byte di una variabile o di un tipo si può usare l'operatore unario `sizeof`.

Esempio

Le linee di codice

```
double d;     double *q;
printf("%d ", sizeof(d)); /* stampa il valore di sizeof(d) */
printf("%d ", sizeof(q));
printf("%d ", sizeof(double**));
```

stampano

```
8 4 4
```

che sono rispettivamente la dimensione di `d` (che è quella del tipo `double`) e la dimensione dei tipi `double*` e `double**`.

## Stampa di un valore di tipo puntatore

È possibile stampare il *valore* di un puntatore mediante `printf()` usando il carattere di conversione `%p` (stampa il valore in esadecimale) oppure facendo un cast a un tipo intero.

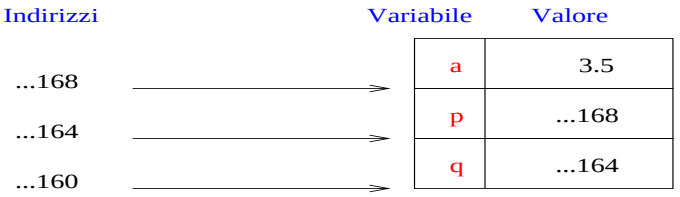
- Poiché gli indirizzi sono numeri molto grandi, è opportuno usare il tipo intero `unsigned int`.
- Per stampare con `printf()` un valore di tipo `unsigned int` usare il carattere di conversione `%u`.

```
int main(){
  double  a = 3.5, * p = &a , ** q = &p;

 printf("sizeof(a) = %d ", sizeof(a));
 printf("sizeof(p) = %d ", sizeof(p));
 printf("sizeof(q) = %d\n", sizeof(q));
 printf("indirizzo a:  %p(esad.) = %u\n", &a,  (unsigned int) &a);
 printf("indirizzo p:  %p(esad.) = %u\n", &p,  (unsigned int) &p);
 printf("indirizzo q:  %p(esad.) = %u\n", &q,  (unsigned int) &q);
 return 0;
}
```

Output:

```
sizeof(a) = 8 sizeof(p) = 4 sizeof(q) = 4
indirizzo a:  0xbfffe760(esad.) = 3221219168
indirizzo p:  0xbfffe75c(esad.) = 3221219164
indirizzo q:  0xbfffe758(esad.) = 3221219160
```




## Uguaglianza fra puntatori

Due puntatori sono uguali se e solo se hanno lo stesso valore, cioè puntano alla stessa locazione di memoria.

Esempio

```
int a, b, *p = &a , *q = &b;
if(p == q)
  printf("uguali\n");
else
  printf("diversi\n");
q = &a;
if(p == q)
  printf("uguali\n");
else
  printf("diversi\n");
```

Viene stampato:

```
diversi
uguali
```



## Compatibilità fra puntatori

- Un puntatore può assumere il valore 0. Per denotare l'indirizzo 0 si usa l'identificatore `NULL`, definito in `stdio.h` come:

  `#define NULL 0`

- Nell'ANSI C non sono permessi assegnamenti fra puntatori di tipo diversi, a meno che il tipo di uno sia `void*` (puntatore al tipo `void`) o a destra ci sia la costante 0. Negli altri casi occorre un cast esplicito.

Nota

Il tipo `void*` ha il ruolo di *puntatore generico*, in quanto un valore di tipo `void*` può essere assegnato a una variabile puntatore di qualunque tipo.
Tuttavia, con puntatori generici non si hanno controlli di tipo in compilazione e questo può causare errori nel codice, pertanto il tipo `void*` va usato con molta cautela.



Esempio 1

```
int  *pi;
double *pd;
pi = 1;  // (A)
pd = 100; // (B)
```

L'istruzione (A) è scorretta in quanto a `pi` possono essere assegnate solamente espressioni di tipo `int*`, mentre 1 ha tipo `int`.
Per motivi analoghi, anche (B) è scorretta.

Versione corretta:

```
pi = (int*)1;       // cast del valore 1 al tipo  int*
pd = (double*)100;  // cast del valore 100 al tipo  double*
```

Sono invece corrette le istruzioni

```
pi =  NULL;       //  Equivale a:  pi = 0;
pd =  NULL;       //  Equivale a:  pd = 0;
```

in quanto 0 non richiede conversione a un tipo puntatore.

Esempio 2

```
int  *pi, **q;
double  *pd;
```

I seguenti assegnamenti sono tutti illegali per incompatibilità di tipo

```
pi = pd;  pd = pi;  pi = q;  q = pi;  q = pd;  pd = q;
```

Nota
In presenza di istruzioni scorette del tipo precedente, il compilatore aggiunge automaticamente i cast mancanti segnalando il messaggio di warning

```
warning: assignment from incompatible pointer type
```

L'assegnamento fra puntatori non compatibili è tuttavia pericoloso e può causare errori in esecuzione.



Esempio 3

```
int  i, *pi, **q;
double  d, *pd;
```

Le seguenti istruzioni sono tutte corrette

```
pi = &i;
q = &pi;
pd = &d;
```

A `pi` viene assegnato l'indirizzo di `i`, a `q` l'indirizzo di `pi` e a `pd` l'indirizzo di `d` (verificare che il tipo dell'espressione a destra di = coincide con quello della variabile a sinistra di =)

Anche

```
q = (int**) &pd;
```

è corretta, in quanto `&pd` (che ha tipo `double*`) viene convertito al tipo `int**` della variabile `q`. Tuttavia questo è un modo innaturale e pericoloso e di usare i puntatori.



## Dereferenziazione di un puntatore

Se `p` è una variabile di tipo `T*` (puntatore al tipo `T`) il cui valore è l'indirizzo di una locazione di memoria $L$ (di tipo `T`),

`*p`

è un'espressione costruita con l'operatore unario '`*`' di dereferenziazione tale che:

- Il tipo di `*p` è $T$.
- Il valore di `*p` è il contenuto $L$

L'espressione `*p` *denota* la locazione di memoria $L$.



Più in generale:

- Sia `E` è un'espressione di tipo `T*` il cui valore è l'indirizzo di una locazione di memoria $L$ (di tipo `T`).

  L'espressione

  `*E`

  - ha tipo `T`;
  - ha come valore il valore contenuto in $L$.

  L'espressione `*E` *denota* la locazione di memoria $L$.

Nota
Con i puntatori è possibile dare più nomi a una stessa locazione di memoria (*aliasing*).

Gli operatori * e & sono l'uno l'inverso dell'altro.

Esempio

```
int x = 10;
```

l'espressione *&x equivale a x. Quindi le istruzioni

```
printf("%d", x);          printf("%d", *&x);
```

sono equivalenti ed entrambe stampano 10 (valore di x).

Non è invece corretta l'istruzione

```
printf("%d", &*x);
```

in quanto l'espressione &*x non è sintatticamente corretta (l'operatore * può essere applicato solamente a una espressione di tipo puntatore).



## Esempi

1. La linea di codice

```
int x = 10 ,  *p = &x;
```

definisce due variabili:

- la variabile x di tipo int e valore 10;
- la variabile p di tipo int* (puntatore a int) e valore &x.

L'espressione *p di tipo int denota la stessa locazione di memoria denotata da x, quindi *p è un *alias* (sinonimo) di x e può essere usata in tutti i contesti in cui lo è x.



Le istruzioni

```
x = x+10;     x = (*p)+10;
```

```
(*p) = x+10;      (*p) = (*p) + 10;
```

hanno lo stesso significato

Anche le istruzioni

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

sono equivalenti



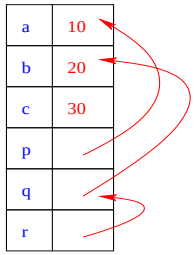
2. Supponiamo che in una funzione siano definite le variabili locali

```
int a = 10,  b = 20, c = 30, *p = &a,  *q,  **r;
```

Dopo le istruzioni

```
q = &b;
r = &q;
```

il record di attivazione della funzione può essere rappresentato come segue (il *valore* di una variabile di tipo puntatore è rappresentato da una freccia):



Nella situazione precedente vale

| Espressione | Tipo | Valore | |
|---|---|---|---|
| `a` | `int` | 10 | |
| `b` | `int` | 20 | |
| `c` | `int` | 30 | |
| `p` | `int*` | `&a` | |
| `*p` | `int` | 10 | (`*p` è sinonimo di `a`) |
| `q` | `int*` | `&b` | |
| `*q` | `int` | 20 | (`*q` è sinonimo di `b`) |
| `r` | `int**` | `&q` | |
| `*r` | `int*` | `&b` | (`*r` è sinonimo di `q`) |
| `**r` | `int` | 20 | (`**r` è sinonimo di `b`) |

Si noti che `**r` equivale a `*(*r)`, quindi a `*q`, quindi a `b`.



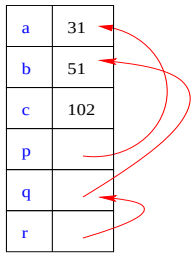
Dopo le istruzioni

```
*p = a+b;   /*  a = a + b = 10 + 20 = 30 */

*q = (++(*p)) + (*q);  /*  b = (++a) + b  =
                                  31 + 20   = 51
                           e ora a vale 31 */

c =  2 * (**r);     /* c = 2*b = 2*51  = 102 */
```

si ha:






La valutazione dell'espressione

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

equivalente a

```
++a
```

avviene come segue:

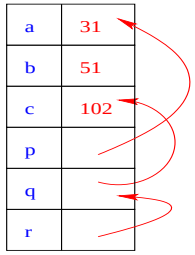
1. Viene incrementato di 1 il valore di a, che passa da 30 a 31.
2. Il valore dell'espressione è 31.



Eseguendo ora

```
*r = &c;      /* q = &c */
```

si ha:



Vale:

| Espressione | Tipo | Valore |
|---|---|---|
| `a` | `int` | 31 |
| `b` | `int` | 51 |
| `c` | `int` | 102 |
| `p` | `int*` | `&a` |
| `*p` | `int` | 31 (`*p` è sinonimo di `a`) |
| `q` | `int*` | `&c` |
| `*q` | `int` | 102 (`*q` è sinonimo di `c`) |
| `r` | `int**` | `&q` |
| `*r` | `int*` | `&c` (`*r` è sinonimo di `q`) |
| `**r` | `int` | 102 (`**r` è sinonimo di `c`) |



## Passaggio dei parametri per valore e per indirizzo

Nei linguaggi di programmazione esistono fondamentalmente due modi per passare i parametri a una funzione:

- Passaggio parametri per valore (by value)
  Il parametro formale della funzione chiamata è inizializzato con il valore passati dal chiamante.
- Passaggio parametri per indirizzo (by reference)
  Il parametro formale della funzione chiamata diventa sinonimo del parametro passato dal chiamante, che deve essere il nome di una locazione di memoria (ad esempio, il nome di una variabile).

Il C permette *solamente* il passaggio di parametri per valore.

Tuttavia è possibile *simulare* il passaggio per indirizzo tramite i puntatori.



## Esempio: la funzione `swap()`

Il problema

Scrivere una funzione `swap()` che scamba i valori di due variabili di tipo `int`.

Osservazione

È un tipico problema in cui si richiede di simulare una chiamata per indirizzo, in quanto la funzione `swap()` deve accedere a due locazioni di memoria (contenenti valori di tipo `int`) esterne.

Occorre passare a `swap()` gli *indirizzi* delle locazioni tramite due parametri formali `a` e `b`. Poiché i valori da scambiare hanno tipo `int`, le variabili `a` e `b` devono avere tipo di tipo `int*`.

Il prototipo della funzione è:

```
void swap(int *a, int *b)
```



Definizione di `swap()`:

```
/* Scambia  *a con *b (valori di tipo int) */

 void swap(int *a, int *b){
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Si noti che all'interno di `swap()`:

- L'espressione `*a` denota la locazione di memoria a cui il parametro `a` si riferisce.
- L'espressione `*b` denota la locazione il cui indirizzo è `b`.

## Esempio di chiamata

Supponiamo che nella funzione `f()` siano definite le variabili locali

```
int a=5, b=2, s = &a, t = &b;
```

Per scambiare i valori di `a` e `b` occorre fare una delle seguenti chiamate:

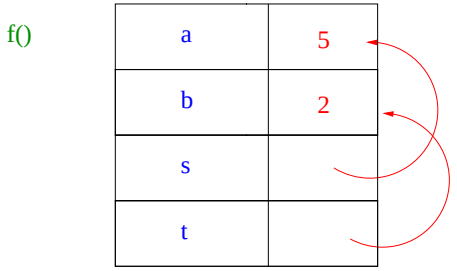
```
swap(&a, &b)      swap(&a, t)

swap(s, &b)       swap(s, t)
```



## Esecuzione di `f()`

(1). Quando `f()` viene chiamata, il suo record di attivazione è posto sullo stack.

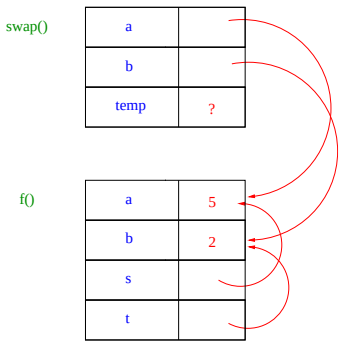




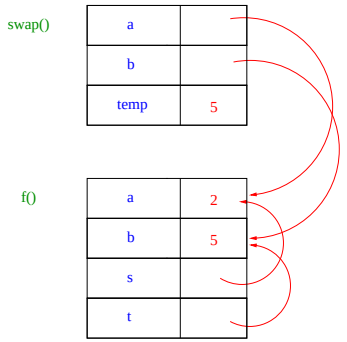

(2). Inizia l'esecuzione di `f()` con la chiamata

```
swap(&a, &b);  /* oppure: swap(&a, t)  ....  */
```

Viene posto sullo stack il record di attivazione di `swap()` e vengono passati come parametri gli indirizzi delle variabili `a` e `b` locali a `f()`.




(3). Inizia l'esecuzione di `swap()`. In `swap()`, l'espressione `*a` denota la locazione di memoria `a` nel record di attivazione di `f()`, `*b` denota `b`. Al termine dell'esecuzione di `swap()` si ha




e i valori di `a` e `b` locali a `f()` risultano scambiati.

## Osservazione

Supponiamo di scrivere

```
swap(a, &b);  /* chiamata errata */
```

- La chiamata è errata in quanto il primo argomento ha tipo `int` anziché `int*`
- Tuttavia il codice viene compilato, in quanto il compilatore traduce la chiamata in

  ```
  swap((int*)a, &b);  /* chiamata errata */
  ```

  forzando un cast a `int*` del primo argomento e segnalando il warning

  ```
  warning: passing arg 1 of ‘swap’ makes pointer
   from integer without a cast
  ```



In esecuzione, il primo parametro passato a `swap()` è 5 (interpretato come indirizzo), quindi il valore di `a` è 5.

L'istruzione

```
*a  =  *b;
```

tenta di modificare il contenuto della locazione di indirizzo 5, e questo provoca in esecuzione errore di *segmentation fault* (violazione di memoria).

Per evitare errori di questo tipo (in genere molto frequenti), controllare attentamente i warning segnalati dal compilatore.



## Esercizio

Si supponga di definire la funzione `swap()` come segue

```
void swap(int a, int b){
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

e di chiamare `swap()` in `f()` nel seguente modo:

```
void f(){
 int a=5, b=2;
 swap(a, b);
 }
```

Quali sono i valori di `a` e `b` locali a `f()` al termine della chiamata a `swap()`? Eseguire il codice passo-passo.