Allocazione dinamica della memoria

Violetta Lonati

Università degli studi di Milano
Dipartimento di Informatica

Laboratorio di algoritmi e strutture dati
Corso di laurea in Informatica

Argomenti

Automatic vs static storage duration

Attraverso le variabili, il C gestisce la memoria staticamente o automaticamente:

- le variabili con storage duration static sono allocate in memoria principale all'inizio dell'esecuzione del programma e persistono per tutta l'esecuzione del programma
  - es: variabili globali
- le variabili con storage duration automatic sono allocate sullo stack, all'interno dei record di attivazione delle chiamate di funzione; queste variabili *vanno e vengono*, cioè perdono il loro valore quando termina l'esecuzione del blocco in cui sono dichiarate e non persistono tra una chiamata e l'altra.
  - es: variabili locali definite all'interno di un blocco

NB: per blocco si intende:

- il corpo di una funzione, oppure
- una sequenza di istruzioni e dichiarazioni raccolte tra graffe

Secondo lo standard ANSI le dichiarazioni vanno all'inizio del blocco.

Allocazione dinamica della memoria

In C la memoria può essere anche gestita in modo dinamico, attraverso l'allocazione esplicita di blocchi di memoria di data dimensione:

- tali blocchi sono allocati tipicamente in una parte della memoria chiamata heap;
- è possibile accedere a tali blocchi di memoria attraverso l'uso di puntatori;
- lo spazio allocato dinamicamente non viene liberato all'uscita delle funzioni;
- sempre con l'uso di puntatori la memoria che non serve più va deallocata in modo da renderla nuovamente disponibile.

Allocazione dinamica della memoria

A cosa serve?

- Per allocare di vettori e/o stringhe con lunghezza non nota in fase di compilazione, ma calcolata durante l'esecuzione. ($\rightarrow$ C99)
- Per gestire strutture dati che crescono e si restringono durante l'esecuzione del programma (es: liste).
- Per avere maggiore flessibilità nel gestire la durata delle variabili.

Quattro funzioni fondamentali:

```
void *malloc( size_t size );
void *calloc( size_t nmemb, size_t size );
void *realloc( void *p, size_t size );
void free( void *p);
```

I prototipi sono contenuti nel file di intestazione stdlib.h.

Puntatore `NULL`

Quando viene chiamata una funzione per l'allocazione dinamica della memoria , c'è sempre la possibilità che non ci sia spazio sufficiente per soddisfare la richiesta. In questo caso, la funzione restituisce un puntatore nullo, ovvero un puntatore che "non punta a nulla".

Nota bene

Avere un puntatore nullo è diverso da avere un puntatore non inizializzato, o un puntatore di cui non si conosce il valore!!

- Il puntatore nullo è rappresentato da una macro chiamata `NULL`, di valore 0, dichiarata in stdlib.h, stdio.h, string.h e altri.
- I puntatori possono essere usati nei testi: `NULL` ha valore falso (vale 0!), mentre ogni puntatore non nullo ha valore vero (è diverso da `NULL`, cioè da 0!)

```
if ( p == NULL) ...
if( !p ) ...
```

Malloc

La funzione

```
void *malloc( size_t size );
```

alloca un blocco di memoria di `size` byte e restituisce un puntatore all'inizio di tale blocco.

- `size_t` è un tipo definito nella libreria standard (di solito corrisponde ad `unsigned int`);
- il blocco di memoria allocato può contenere valori di tipo diverso, il puntatore di tipo generico `void *` permette di gestire tutti i casi;
- in caso di assegnamento il puntatore restituito dalla malloc viene convertito implicitamente (alcuni esplicitano il cast);
- sul blocco di memoria allocato è possibile usare i puntatori con l'usuale aritmetica.

Malloc - esempi

```
p = malloc( 10000 );
if ( p == NULL ) {
  /* allocazione fallita;
     provvedimenti opportuni */
  ...
}
```

Stringhe allocate dinamicamente

```
/* alloca lo spazio per una stringa di n caratteri
un char occupa sempre un byte! */
  char *p;
  int n;
  ...
  p = malloc( n + 1 );
```

Esempio - restituire un puntatore ad una "nuova" stringa

Il seguente programma concatena le due stringhe `s1` e `s2` in una nuova stringa di cui restituisce l'indirizzo (ovvero un puntatore che punta ad essa).

```
char *concat( const char *s1, const char *s2)  {
  char *result;

  result = malloc( strlen(s1) + strlen(s2) + 1 );
  if ( result == NULL ) {
    print( "malloc␣failure\n" );
    exit(EXIT_FAILURE);
  }

  strcpy( result, s1 );
  strcat( result, s2 );
  return result;
}
```

```
p = concat( "abc", "def");
```


Vettori allocati dinamicamente

Si può usare `malloc` anche per allocare spazio per un vettore (come per le stringhe). La differenza è che gli elementi dell'array possono occupare più di un byte (a differenza dei `char`).

```
int *a, i, n;

/* alloca lo spazio per un array di n interi */
a = malloc( n * sizeof(int) );

/* inizializza l'array a 0 */
for ( i = 0; i < n; i++ )
  a[i] = 0;
```

Calloc

```
void *calloc( size_t nmemb, size_t size );
```

alloca spazio per un array di `nmemb` elementi, ciascuno di dimensione `size`, li inizializza a 0 e restituisce il puntatore al primo elemento (oppure `NULL`).

Esempio

A volte può essere comodo usare calloc con primo argomento pari a 1, in questo modo è possibile allocare e inizializzare anche oggetti diversi da un array.

```
struct point{ float x, y } *p;
p = calloc( 1, sizeof( struct point) );
```

Alla fine dell'esecuzione di queste istruzioni, `p` punterà ad una struttura di tipo `point` i cui membri `x` e `y` sono inizializzati a 0.

Realloc

```
void *realloc( void *p, size_t size );
```

ridimensiona lo spazio puntato da `p` alla nuova dimensione `size` e restituisce il puntatore al primo elemento (oppure `NULL`):

- il puntatore `p` deve puntare ad un blocco di memoria già allocato dinamicamente, altrimenti il comportamento è indefinito;
- tendenzialmente `realloc` cerca di ridimensionare il vettore in loco, ma se non ha spazio può allocare nuovo spazio altrove, copiare il contenuto del vecchio blocco nel nuovo e restituire l'indirizzo del nuovo blocco;
- attenzione ad aggiornare eventuali altri puntatori dopo la chiamata di `realloc` perchè il blocco potrebbe essere stato spostato!

Free

Quando un blocco di memoria allocato dinamicamente non serve più, è importante deallocarlo e renderlo nuovamente disponibile usando la funzione

```
void free( void *p)
```

L'argomento di free deve essere stato allocato dinamicamente, altrimenti il comportamento è indefinito.

Errori tipici: fallimento nell'allocazione

E' importante sempre verificare che l'allocazione abbia avuto successo e il puntatore restituito non sia `NULL`.
In caso contrario si rischia di usare il puntatore `NULL` come se puntasse a memoria allocata, e si provocherebbero errori.

```
char *ptr;
ptr = malloc(10);
*ptr = 'a';
/* RISCHIOSO: se malloc restituisce NULL... */
```

Errori tipici: dangling pointer

Dopo la chiamata `free(p)`, il blocco di memoria puntato da `p` viene deallocato, ma il valore del puntatore `p` non cambia; eventuali usi successivi di `p` possono causare danni!

```
char *p = malloc(4);
...
free(p);
...
strcpy(p, "abc"); /* SBAGLIATO! */
```

Si dice in questo caso che `p` è un dangling pointer (letteralmente: puntatore ciondolante).

Errori tipici: memory leak - esempio 1

```
p = malloc(...);
q = malloc(...);
p = q;
```

L'oggetto puntato da `p` prima dell'ultimo assegnamento non è più raggiungibile! Quel blocco di memoria resterà allocato ma non utilizzabile. Si parla in questo caso di memory leak.
Prima di effettuare l'assegnamento `p = q;` bisogna deallocare il blocco puntato da `p`:

```
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

Errori tipici: memory leak - esempio 2

E' importante usare sempre un puntatore temporaneo per il valore di ritorno di realloc. In caso contrario può succedere che il puntatore originario venga trasformato in `NULL`. Ad esempio:

```
int *ptr, *tmp, size = N;
ptr = malloc(size);
...
/* vogliamo raddoppiare l'area allocata */
size *= 2;
tmp = realloc(ptr, size);
if ( tmp != NULL )
  ptr = tmp;
```


Errori tipici: memory leak - esempio 3

```
void f(void) {
  void* s = malloc( 50 );
}

int main(void) {
     while (1) f();
}
```

Ad ogni chiamata di `f`, la memoria viene allocata e puntata da `s`. Quando la funzione restituisce il controllo al main, lo spazio rimane allocato, ma `s` viene distrutta quindi la memoria allocata diventa irrangiungibile.
Prima o poi la memoria verrà esaurita!
Il codice va corretto in uno dei seguenti modi:

- aggiungere l'istruzione `free(s)` alla fine di `f`
- far si che `f` restituisca `s` alla funzione chiamante, la quale si dovrà preoccupare di deallocare lo spazio.