Memoria dinamica

© Mauro Fiorentini, 2019

Gestione della memoria dinamica

- Totalmente sotto il controllo del programmatore.
- Basata su due funzioni, dichiarate in stdlib.h, una per allocare memoria, una per liberarla.
  - La **responsabilità** del corretto utilizzo e del rispetto dei vincoli è **del programmatore**.
  - **Non esistono controlli** di alcun genere su dimensioni, tipi e accessi.
  - E’ facile sbagliarsi.

Allocazione della memoria dinamica

- Si effettua con la funzione:

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

- Alloca `size` byte di memoria, allineati in modo tale da poter contenere qualsiasi variabile di quella dimensione; produce il puntatore alla memoria, o `NULL` se l'allocazione fallisce.
  - Il calcolo della dimensione della memoria da allocare si effettua di solito con l'operatore `sizeof`.
  - Dopo l'allocazione il puntatore va convertito al tipo richiesto, con l'operatore cast, per poterlo utilizzare.

Liberazione della memoria dinamica

- Si effettua con la funzione:

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

- Libera l'area puntata da `p`, che deve essere stata precedentemente allocata con una `malloc`.
  - Dopo la liberazione non è consentito accedere all'area.

Esempio di uso della memoria dinamica

```
struct s { … };
struct s* p;

p = (struct s*)malloc(sizeof(struct s));

// utilizzo di p

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

Nota: dopo la `free p` resta inalterato, ma non va utilizzato.

Allocazione di vettori

- Si effettua come l'allocazione di singole variabili, moltiplicando la dimensione per il numero di elementi desiderati.
  - Esempi:

```
struct s     { … };
struct s*    p;

p = (struct s*)malloc(n * sizeof(struct s));
        // alloca un vettore di n elementi.
```

Allocazione di stringhe

- Attenzione alle stringhe: bisogna ricordarsi di allocare spazio per il terminatore!

Esempi:

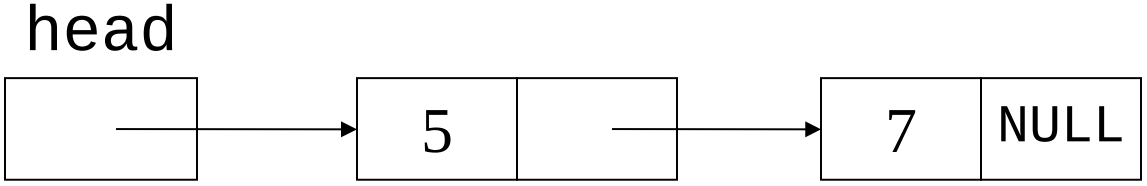
```
char*   p;
char*   s;
p = (char*)malloc((strlen(s) + 1) *
   sizeof(char));
strcpy(p, s);// copia la stringa
```

oppure, dato che `sizeof(char)` è 1:

```
p = (char*)malloc(strlen(s) + 1);
```

Esempio di memoria dinamica: uno stack (1)

- Implementazione di uno stack di interi con una lista.




```
struct node
   {
   int            value;
   struct node*   next;
   };
struct node* head = (struct node*)NULL;
```

Da notare che si può utilizzare il **puntatore** a `struct node` nella dichiarazione di `struct node` stesso.

Esempio di memoria dinamica: uno stack (2)

```
void   push(int i)
  {
  struct node*   p;

  if ((p = (struct node*)
       malloc(sizeof(struct node))) ==
       (struct node*)NULL)
       return; // Bisogna gestire l'errore!
  p->val = i;
  p->next = head;
  head = p;
  }
```

Esempio di memoria dinamica: uno stack (3)

```
int    pop(void)
  {
  int  i;
  struct node*   p;

  if ((p = head) == (struct node*)NULL)
       return 0; // Bisogna gestire l'errore!
  i = p->val;
  head = head->next;
  free(p);
  return i;
  }
```

Esempio di memoria dinamica: uno stack (4)

```
void   clear(void)
  {
  while (head != (struct node*)NULL)
        {
        free(head);
        head = head->next;
        }
  }
```

Dov' è l'errore?

Esempio di memoria dinamica: uno stack (5)

Non è consentito accedere ad aree liberate!

Versione corretta:

```
void clear(void)
  {
  register     struct node*   p;

  while ((p = head) != (struct node*)NULL)
     {
     head = head->next;
     free(p);
     }
  }
```

Pericoli nell'uso della memoria dinamica

- Gli errori più frequenti, usando memoria dinamica sono:
  - aree "morte", aree allocate, non liberate, ma delle quali si è perso il puntatore;
  - liberazione di aree non allocate;
  - accesso ad aree dopo averle liberate;
  - accessi oltre i limiti dell'area allocata.
- Le conseguenze possono essere gravi, ma non esistono controlli.
  - Tranne che nel primo caso, chiamate successive alle funzioni di libreria possono avere **effetti catastrofici**.

Esempio di errori nell'uso di memoria dinamica (1)

- Aree inutilizzate e non liberate.

```
char* p;
char* q;

p = (char *)malloc(100);
…
p = q;
```

L'area precedentemente puntata da `p` **non è più accessibile**, ma resta allocata.

Esempio di errori nell'uso di memoria dinamica (2)

- Liberazione di aree non allocate.

```
char* p;
char  s [100];

p = s;
…
free(p);
```

L'area puntata da `p` **non era stata allocata** con una `malloc`.

Esempio di errori nell'uso di memoria dinamica (3)

- Liberazione di aree non allocate.

```
char* p;

p = (char *)malloc(100);
++p;
…
free(p);
```

p non punta **all'inizio** dell'area allocata.

Esempio di errori nell'uso di memoria dinamica (4)

- Accesso ad aree liberate.

```
char*   p;
char*   q;

p = (char *)malloc(100);
q = p;
…
free(p);
```

Alla fine `q` punta a un'area alla quale **è vietato accedere**.

Esempio di errori nell'uso di memoria dinamica (5)

- Accesso oltre i limiti dell'area allocata.

```
int*  p;

p = (int *)malloc(100 *
   sizeof(int));
p [100] = 0;
```

Il massimo indice utilizzabile è 99.

Approfondimenti sui tipi

© Mauro Fiorentini, 2019

Definizione di nomi di tipi

- Il costrutto `typedef` permette di dare nuovi nomi ai tipi.
  - Nota bene: **non crea nuovi tipi**.
  - Si utilizza per semplicità (evita la ripetizione delle parole chiave `struct`, `union` e `enum`) e chiarezza.
  - **Non migliora** il controllo dei tipi.

- Sintassi:

  `typedef` <*tipo*> <*nome*> `;`

Esempi di definizione di nomi di tipi (1)

Dichiarando:

```
struct point
       {
       int   x, y;
       };
typedef      struct point      point_type;
```

si può scrivere:

```
point_type*  p;
p = malloc(sizeof(point_type));
```

invece di:

```
struct point * p;
p = malloc(sizeof(struct point));
```

Esempi di definizione di nomi di tipi (2)

Anche semplicemente:

```
typedef struct              // struct anonima
      {
      int   x, y;
      } point_type;
```

Per rendere piu chiaro il tipo:

```
typedef     unsigned char     small_int;
typedef     double            weight_type;

small_int   age;
weight_type w;
```

Esempi di definizione di nomi di tipi (3)

Nota bene:

```
typedef     double      weight_type;
typedef     double      length_type;

double compute(weight_type w, length_type l);

weight_type w;
length_type l;

x = compute(l, w);
```
invece di `x = compute(w, l);`

Scambiando `weight_type` e `length_type` nella chiamata, non si ha alcuna segnalazione di errore (sono comunque `double`).

Uguaglianza di tipi

- E’ nominale, non strutturale:
  - due tipi sono uguali se hanno lo stesso **nome**;
  - due vettori sono dello stesso tipo se sono uguali i tipi base, il numero di dimensioni e i limiti di ogni dimensione.
- L'uguaglianza di tipi si conserva attraverso catene di `typedef`.

Esempi di uguaglianza di tipi

```
struct point        { int     x, y; };
struct newpoint     { int     x, y; };
typedef   struct point   point_2;

struct point        p, p1;
struct newpoint     np;
point_2             p2;
```

`p = p1;` OK: `p` e `p1` sono dello stesso tipo

`p = np;` Errore: `p` e `p2` non sono dello stesso tipo

`p = p2;` OK: `p` e `p2` sono dello stesso tipo

Qualificatori di tipo (1)

- Sono dichiarati tramite le parole riservate `const`, `volatile` e `restrict` in una dichiarazione di tipo.
  - Sono cumulabili, ma non ripetibili in una dichiarazione.
- Sono **parte integrante** del tipo: `int` e `const int` **non sono lo stesso tipo**.
- `const` e `volatile` **non sono facoltativi**; vanno utilizzati ogni volta che appropriati.

Qualificatori di tipo (2)

- Si riferiscono all'entità immediatamente seguente:

  ```
  const char*  p;
  ```

- È l'**oggetto puntato** (`char`) a essere costante, non il puntatore, mentre:

  ```
  char* const p;
  ```

  descrive un puntatore **costante**, a un oggetto **variabile**.

Il qualificatore const

- Indica che l'oggetto cui si riferisce non verrà modificato **dal programma**.
- Si utilizza per descrivere variabili che non vanno modificate.
  - Può permettere alcune ottimizzazioni.
  - Il compilatore effettua tutti i controlli possibili, dando maggior sicurezza.
- Le variabili dichiarate `const` possono venir allocate separatamente, in memoria **a sola lettura**.

Usi del qualificatore const

- Dichiarare `const` una variabile può sembrare incoerente, ma esistono due casi molto frequenti:
  - strutture dati inizializzate, da non modificare;
  - argomenti di funzioni.

Esempio di strutture dati da non modificare

```
const unsigned int primes [] =
  {
  2, 3, 5, 7, 11, 13, 17, 19
  };
```

- Garantisce che il vettore non sarà modificato per errore.

Esempio di argomenti di funzioni

- La dichiarazione:

```
int  printf(const char *format,
            ...);
```

  garantisce che la funzione `printf` non modificherà la stringa primo argomento.
  - È meglio di un commento, perché il compilatore **verifica** che la stringa non sia modificata, neppure in funzioni di livello inferiore.

Il qualificatore volatile

- Indica che l'oggetto cui si riferisce **può essere modificato** da enti esterni al programma o **ha effetti** su enti esterni.
  - Si utilizza per descrivere variabili di comunicazione con **hardware esterno** o in **memoria condivisa** o modificate in modo **asincrono**
  - Impedisce ottimizzazioni indesiderate.
- Il compilatore **non elimina** alcun accesso né in lettura né in scrittura alle variabili `volatile`, né ne **modifica l'ordine**.

Il qualificatore restrict

- Si usa per dichiarare puntatori.
- Indica che se l'oggetto cui si riferisce sarà modificato:
  - gli **accessi** avverranno **esclusivamente** tramite quel puntatore, per la durata della vita del puntatore stesso;
  - il puntatore non punterà **ad altra variabile**, pur restando libero di spostarsi nel vettore.
- Serve a permettere alcune ottimizzazioni fornendo al compilatore **garanzia dell'assenza di aliasing**.

Implicazioni del qualificatore restrict (1)

- Dichiarando `restrict` un puntatore globale, si indica che punterà a variabili alle quali si accede esclusivamente tramite quel puntatore per tutta l'esecuzione del programma.
  - La variabile non è necessariamente unica, perché può essere allocata e deallocata più volte.
- Dichiarando `restrict` un puntatore locale o un argomento, si indica che all'interno del blocco o funzione non si modifica l'area di memoria puntata o vi si accede **esclusivamente** tramite quel puntatore.

Implicazioni del qualificatore restrict (2)

- La responsabilità del rispetto della garanzia ricade sul programmatore: se sono rispettati i vincoli, il comportamento è **indefinito**.
- I vincoli si estendono a blocchi interni e **funzioni chiamate** durante la vita del puntatore.

Usi del qualificatore restrict (1)

```
void   f(register unsigned int n,
       register int * restrict p,
       register const int * restrict q)
       {
       while (n-- > 0)
            *p++ = *q++;
       }
```

`restrict` garantisce che i due vettori passati come argomenti **non sono sovrapposti in alcun modo**, dato che l'area puntata da `p` viene modificata.

La funzione potrebbe essere compilata come una `memcpy`, migliorandone l'efficienza.

Usi del qualificatore restrict (2)

```
double    a [100], b [100], c [100];

void      sum(register unsigned int n,
  register double * restrict result,
  register const double * restrict v1,
  register const double * restrict v2)
  {
  register     unsigned int   i;

  for (i = 0; i < n; ++i)
    result [i] = vl [i] + v2 [i];
  }
```

Usi del qualificatore restrict (3)

- La funzione `sum` può essere tradotta mediante istruzioni vettoriali, perché è garantita la non sovrapposizione degli argomenti.
- Chiamando `sum(100, a, b, c)`, si sommano i vettori `b` e `c`, lasciando il risultato in `a`.
- Chiamando `sum(100, a, b, b)`, si somma il vettore `b` a se stesso, lasciando il risultato in `a`.
  - Va bene, perché il vettore `b` non viene modificato.
- Chiamando `sum(100, a, a, b)`, si ottiene un comportamento indefinito.

Usi del qualificatore restrict (3)

```
extern    int  array [100];

void function(int * restrict p)
  {
  array [0] = 0;
  ++*p;
  …
  array [1] += array [0];
  …
  if (array [n] > 0)
  …
```

Usi del qualificatore restrict (4)

- Il compilatore può supporre che `array[0]` resti invariato, perché `p` non può puntare ad `array`, quindi può ottimizzare il codice.
- L'ottimizzazione sarebbe **illegale** in assenza del qualificatore `restrict`.

Compatibilità di puntatori qualificati

- Nelle operazioni (sottrazioni e confronti) i puntatori qualificati sono compatibili con i puntatori **corrispondenti** non qualificati.
- Negli assegnamenti e passaggi come argomenti è consentito **aggiungere** qualificatori, ma non **rimuoverne**.

Esempi di compatibilità di puntatori qualificati

```
const      char*          cp;
           char*          p;
```

Assegnamento non valido, perché permetterebbe poi di modificare l'oggetto puntato tramite `p`:

```
p = cp;
```

Assegnamento valido, perché **aggiunge** un qualificatore:

```
cp = p;
```

Qualificatori e puntatori a più livelli (1)

- A un puntatore a più livelli è consentito **aggiungere** qualificatori a un livello intermedio solo se **tutti i livelli superiori sono qualificati** `const`. Es.:

```
int**                 p;
const int**           cp;
const int* const*     ccp
cp = p;    // Errore
ccp = p;   // OK
ccp = cp;  // OK
cp = ccp;  // OK
```

Qualificatori e puntatori a più livelli (2)

- La regola serve a evitare un sottile errore:

```
const  char     c = 'a';
       char*    pc;
const  char**   cpp;

cpp = &pc;      // Vietato
*cpp = &c;      // OK
*pc = 'x';      // OK
```

E avremmo modificato una variabile `const`.