Uso avanzato dei puntatori
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

26 ottobre 2016

Argomenti

Puntatori

- Stringhe e puntatori
- Array di stringhe
- Argomenti da linea di comando

Allocazione dinamica della memoria

- Puntatore nullo
- Funzioni per l'allocazione dinamica della memoria
- Deallocare memoria
- Errori tipici
- Esempi

Stringhe e puntatori

- Ogni letterale stringa (es: `"ciao"`) è un array di caratteri, quindi è trattato come puntatore a carattere.
- Ad esempio il primo parametro nel prototipo della funzione printf è un puntatore a carattere:

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

- Se una stringa è argomento di una funzione, alla chiamata essa non viene copiata, ma viene passato l'indirizzo della sua prima lettera.

Stringhe e puntatori - attenzione alle dichiarazioni

Dichiarazione come array

```
char data[] = "13␣maggio";
```

- data è un vettore che contiene i caratteri '1', '3', ...
- i singoli caratteri possono essere modificati (Es: `data[1] = '4'`);

Dichiarazione come puntatore

```
char *data = "13␣maggio";
```

- il letterale costante "13␣maggio" è memorizzato in un array;
- l'inizializzazione fa si che `data` punti al letterale costante;
- il puntatore `data` può essere modificato in modo che punti altrove.

File di intestazione `string.h`

Copia di stringhe

Non si possono usare assegnamenti tipo `str = "abcd"`; usiamo la funzione

```
char *strcpy(char *dest, const char *src);
```

che copia `src` in `dest` e ne restituisce l'indirizzo.
Esempio: `sctrcpy(str, "abcd")` copia `"abcd"` in `str`.

Concatenazione di stringhe

```
char *strcat(char *dest, const char *src);
```

aggiunge il contenuto di `src` alla fine di `dest` e restituisce `dest` (ovvero il puntatore alla stringa risultante).

Confronto tra stringhe

```
int strcmp(const char *s1, const char *s2);
```

restituisce un valore maggiore, uguale o minore di 0 a seconda che `s1` sia maggiore, uguale o minore di `s2`.

Esempio: calcolare la lunghezza di una stringa

```c
/* Prima versione */
int lun_stringa( const char *s ) {
  int n = 0;
  while ( *s != '\0' ) {
    n++; s++;
  }
  return n;
}
```

```c
/* Seconda versione */
int lun_stringa( const char *s ) {
  int n = 0;
  while ( *s++ != '\0' )
    n++;
  return n;
}
```

Esempio: calcolare la lunghezza di una stringa - continua

```c
/* Terza versione */
int lun_stringa( const char *s ) {
  int n = 0;
  while ( *s++ )
    n++;
  return n;
}
```

```c
/* Quarta versione */
int lun_stringa( const char *s ) {
  const char *p = s;
  while ( *s++ )
    ;
  return s - p - 1;
}
```

Array di stringhe

Un vettore bidimensionale di caratteri può essere pensato come un array di stringhe di lunghezza costante: ogni riga contiene una stringa della stessa lunghezza.

```
char colori1[6][10] = {"rosso", "blu", "oltremare"
                       "verde", "nero", "giallo"};
```

- Il numero di colonne è pari almeno alla lunghezza massima delle stringhe (incluso il fine stringa): in questo esempio occupo lo spazio per $6 \times 10 = 60$ caratteri.
- Buona parte di questo spazio è occupata dal carattere nullo `'\0'`.
- NB: se si usa un vettore bidimensionale come parametro di funzione, è necessario specificare la seconda dimensione, ad esempio:

```
void stampa( char c[][10] );
```

Array di stringhe - 2

Il modo più corretto (ed efficiente) di gestire un insieme di stringhe di lunghezza variabile è attraverso l'uso di un array frastagliato, ovvero di un array di puntatori a char:

```
char *colori2[6] = {"rosso", "blu", "oltremare"
                    "verde", "nero", "giallo"};
```

- In questo caso occupo lo spazio di 6 puntatori a char, più lo spazio strettamente necessario a contenere le 6 stringhe, pari a $(5+1)+(3+1)+(9+1)+(5+1)+(4+1)+(6+1) = 38$ caratteri.
- Il nome di un array di stringhe può essere considerato come un puntatore a stringa. Ad esempio, come parametro di funzione si può usare indifferentemente: `char *c[]` oppure `char **c`.
- NB: nei parametri di funzione devo usare dichiarazioni diverse se uso array frastagliati o array bidimensionali:
  `char c[][10]` non equivale a `char **c`!!

Argomenti da linea di comando

Ogni programma C può avere degli argomenti passati da linea di comando. Per poterli usare è necessario che la funzione `main` sia definita con due parametri, chiamati solitamente `argc` e `argv`.

```
int main ( int argc, char *argv[] ) { ... }
```

oppure

```
int main ( int argc, char **argv ) { ... }
```

- `argc` è pari al numero degli argomenti (incluso il nome del comando);
- `argv` è un array frastagliato di stringhe, di lunghezza `argc`+1:
  - `argv[0]` è il nome del comando;
  - `argv[1]` è il primo argomento;
  - `argv[argc-1]` è l'ultimo argomento;
  - `argv[argc]` è il puntatore `NULL` (...);

Argomenti da linea di comando - esempio

```c
/* somma.c - programma che somma i suoi argomenti */
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] ) {
  int i, somma = 0;
  for ( i = 1; i < argc; i++ )
    somma += atoi( argv[i] );

  printf( "Somma = %d\n", somma );
  return 0;
}
```

```
$ gcc -o somma somma.c
$ ./somma 1 3 10
Somma = 14
```

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.

Lettura di una riga con allocazione di memoria

La seguente funzione legge da standard input una sequenza di caratteri terminata da `\n` e la memorizza in una stringa di dimensione opportuna allocata dinamicamente.

- La dimensione della stringa viene incrementata man mano che vengono letti i caratteri: all'inizio viene allocato lo spazio per 2 caratteri; quando lo spazio è tutto occupato, la dimensione viene raddoppiata.
- Uso due variabili intere: `size` rappresenta la dimensione allocata; `n` rappresenta il numero di caratteri letti.
- Se `n >= size`, bisogna allocare nuovo spazio!

```c
char *read_line( void ) {

  char *p, c;
  int n = 0, size = 2;

  p = my_malloc( size );

  while ( ( c = getchar() ) != EOF ) {

    if ( n >= size ) { /* spazio terminato, lo raddoppio */
      size *=2;
      p = my_realloc( p, size );
    }

    if ( c == '\n' ) { /* fine stringa, interrompo */
      p[n] = '\0';
      break;
    }

    p[n++] = c;
  }

  return p;
}
```

Lettura di una parola con allocazione di memoria

La funzione precedente può essere modificata in modo da leggere una sola parola (fino al primo carattere non alfabetico) memorizzandola in una stringa di dimensione opportuna allocata dinamicamente.
Al posto di:

```
if ( c == '\n' )
```

devo scrivere:

```
if ( !isalpha( c  ) )
```

Allocazione dinamica di una matrice bidimensionale I

La seguente funzione alloca lo spazio per una matrice bidimensionale di caratteri e la inizializza con il carattere '.'

```
char **creaMatrice( int n ){
  char **m;
  int r, c;

  m = malloc( n * sizeof( char * ) );
  for ( r = 0; r < n; r++ ) {
    *(m+r) = malloc( n * sizeof( char ) );
  }

  for ( r = 0; r < n; r++ )
    for ( c = 0; c < n; c++ )
      m[r][c] = '.';
  return m;
}
```

Allocazione dinamica di una matrice bidimensionale II