Puntatori

Nicola Bicocchi

DIEF - UNIMORE

Puntatori a void

- La parola chiave **void** può essere usata per dichiarare dei puntatori che non puntano a nessun tipo di dato in particolare
- E' sempre consentito l'assegnamento di un puntatore a void a qualunque altro tipo di puntatore. Lo è pure l'assegnamento di qualunque puntatore ad un puntatore a void
- L'assegnamento tra puntatori di tipi diversi da void causa invece la generazione di un messaggi di warning

```
1 void *ptr;
2 int *i;
3 float *f;
4
5 i = ptr;
6 ptr = i;
7
8 /* Incompatible pointer types assigning to 'int *' from 'float *' */
9 i = f;
```

Puntatori a void

```c
void stampa_bit(void *ptr) {
    int i;
    printf("%p ", ptr);
    for (i = 31; i >= 0; i--) {
        printf("%d", (*ptr) >> i) & 0x01);
    }
    printf("\n");
}

int main(void) {
    int a = 100;
    float b = 100.0F;
    stampa_bit((void *)&a);
    stampa_bit((void *)&b);
}
```

```
0x7ffee58cc8dc 00000000000000000000000001100100
0x7ffee58cc8d8 01000010110010000000000000000000
```

Puntatori e somma di numeri interi

- Ai puntatori possono essere sommati e sottratti numeri interi. Il risultato della somma di un puntatore e di un numero intero è l'indirizzo dell'elemento n-esimo del vettore
- *Il numero intero non rappresenta il numero di byte da aggiungere nell'indirizzo, ma il numero di elementi*. Il *fattore di scala* appropriato viene applicato dal compilatore in base al tipo cui punta il puntatore

```c
int main(void) {
    int i, v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, *p = v;

    for (i = 0; i < 10; i++) {
        printf("[%d] %d %d %d\n", i, v[i], *(v+i), *(p+i));
    }
}
```

Puntatori e somma di numeri interi

- Se si incrementa/decrementa di 1 un puntatore *p*, il suo valore numerico (indirizzo in memoria espresso in byte) viene incrementato/decrementato di un elemento, che equivale a *sizeof(*p)*, ossia la dimensione dell'oggetto puntato
- Nell'esempio sotto, l'indirizzo contenuto in *p* è incrementato di 5 * sizeof(int), quindi di 20 byte se int ha dimensione pari a 4 byte

```c
int main(void) {
    int v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *p, *q;

    p = &v[0];  /* p = v */
    q = p + 5;

    printf("%d\n", *q);
}
```

Differenza fra puntatori

- E' possibile fare la differenza (ma non la somma!) tra puntatori dello stesso tipo
- Il risultato della differenza fra puntatori è un numero intero che rappresenta il numero di elementi tra i due puntatori
- La dimensione di un singolo elemento è quella definita dal tipo di dato puntato

```c
int main(void) {
    int *p, *q, v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    p = &v[0];  /* p = v */
    q = p + 5;
    printf("%d\n", q - p);
}
```

Operatori

Gli operatori fondamentali per usare i puntatori sono:

- * (da leggere *il valore puntato da*)
- & (da leggere *l'indirizzo di*)
- [] accedo ad un elemento particolare di un vettore
- Si noti che *p == p[0], *(p+i) == p[i]

```
1  int i, v[10], *p;
2
3  p = v;      /* p punta ad indice 0 di v */
4  p = &v[0];  /* p punta ad indice 0 di v */
5  p = &v[4];  /* p punta ad indice 4 di v */
6  p = v + 4;  /* p punta ad indice 4 di v */
7  p++;        /* p punta ad indice 5 di v */
8  i = p - v;  /* i == 5 */
```

Esempi di utilizzo

- Il puntatore *p* è utilizzato per scorrere il vettore, essendo inizializzato all'indirizzo del primo elemento del vettore
- Il ciclo termina quando il valore puntato *p, è nullo (il valore 0 equivale alla condizione logica *falso*)
- Deve esistere almeno un elemento di *v* che vale zero, altrimenti il puntatore assumerà valori non validi andando ad accedere oltre la fine del vettore

```
1  int *p, v[] = {1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
2  int sum = 0;
3
4  for (p = v; *p; p++) {
5    sum += *p;
6  }
```

Puntatori e stringhe

```
1  char *nome_ptr = "paolo";
2  char nome_vet[] = "paolo";
```

- Vettori e puntatori sono concetti affini, ma esistono sottili differenze
- I puntatori possono contenere indirizzi variabili nel corso dell'esecuzione, mentre i vettori rappresentano *indirizzi costanti*
- Le stringhe memorizzate in un vettore possono essere modificate in ogni momento o accedendo ai singoli elementi oppure tramite apposite funzioni (e.g., *strcpy*). Le stringhe memorizzate attraverso puntatore sono stringhe *senza nome* che possono essere memorizzate in aree di memoria in *sola lettura*
- Operatore *sizeof* si comporta in modo diverso. In un caso ritorna la dimensione del vettore, nell'altro la dimensione del puntatore

Puntatori e stringhe

```c
char s1[] = "prova";
char s2[] = {'p', 'r', 'o', 'v', 'a', '\0'};
char c, *t;

c = *s1;          /* c = p */
t = s1 + 2;       /* t contine indirizzo del carattere 'o' */

s1[0] = *t;       /* s1 == orova */

t++;              /* t contine indirizzo del carattere 'v' */

/* errore */
s1++;
```

Puntatori e stringhe

- L'esempio seguente mostra un possibile utilizzo dell'aritmetica dei puntatori al fine di calcolare la lunghezza di una stringa *zero-terminata*

```
1  unsigned str_len(char *ptr) {
2      unsigned size = 0;
3
4      for (; *ptr; ptr++) size++;
5      return size;
6  }
7
8  int main(void) {
9      char s[] = "prova";
10     printf("%d\n", str_len(s));
11 }
```

Vettori di stringhe

L’allocazione dinamica della memoria

- Il linguaggio C permette di effettuare l’allocazione di memoria anche durante l’esecuzione del programma, sulla base della necessità e di opportune condizioni che possono verificarsi durante l’esecuzione
- Questo tipo di allocazione di memoria è detta dinamica, proprio perché avviene dinamicamente durante l’esecuzione. L’allocazione cosiddetta statica è quella che invece viene effettuata dal compilatore a seguito della dichiarazione delle variabili
- Il tempo di vita di porzioni di memoria allocate dinamicamente *non dipende* da quello dello funzione in cui l’allocazione è avvenuta

```c
#include <stdlib.h>
void *malloc(size_t n);
void *calloc(size_t n, size_t size);
void *realloc(void *pt, size_t n);
free(void *p);
```

L’allocazione dinamica della memoria (malloc)

```c
void *malloc(size_t n);
```

- **malloc** (Memory ALLOCation) richiede come argomento il numero di byte da allocare in memoria
- Restituisce l’indirizzo al quale la memoria è stata allocata
- Restituisce NULL se non è stato possibile allocare la memoria
- *Lo spazio allocato in memoria è contiguo*

```c
int *p;
/* Dipendente dal tipo di dato */
p = malloc(10 * sizeof(int));

/* Indipendente dal tipo di dato, da preferire */
p = malloc(10 * sizeof(*p));
```

L’allocazione dinamica della memoria (malloc)

```c
int *p;
p = malloc(10 * sizeof(*p));

if (!pt) {
    /* gestione dell'errore */
}

for (int i = 0; i < 10; i++) {
    p[i] = i;   /* oppure *(p + i) = i; */
}

free(p);
```

- Viene allocato lo spazio necessario per memorizzare 10 valori interi contigui, uno spazio di memoria che può quindi essere acceduto come fosse un vettore
- E’ poi possibile utilizzare il puntatore indicizzandolo opportunamente per accedere alla memoria allocata

L’allocazione dinamica della memoria (malloc)

- Per azzerare tutti gli elementi interi memorizzati:

```
1 for (i = 0; i < 10; i++)
2     pt[i] = 0;
```

- Ma si può anche fare:

```
1 for (i = 0; i < 10; i++, p++)
2     *pt = 0;
```

L’allocazione dinamica della memoria (free)

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

- Libera il blocco di memoria di indirizzo *p* precedentemente allocato tramite *malloc*, *calloc* o *realloc*
- La memoria allocata dinamicamente deve essere rilasciata quando non è più necessaria, per evitare di occupare inutilmente memoria
- Con *memory leak* si intende il mancato utilizzo della funzione *free*. Come conseguenza, il sistema perde di continuo memoria disponibile

L’allocazione dinamica della memoria (calloc, realloc)

```
1 void *calloc(size_t n, size_t size);
```

- Alloca un puntatore ad un blocco di memoria in grado di contenere un vettore di *n* elementi ciascuno dei quali ha dimensione *size*
- Il blocco di memoria viene inizializzato a 0 byte per byte

```
1 void *realloc(void *p, size_t n);
```

- Ridimensiona ad *n* un blocco di memoria già allocato e puntato da *p*
- Preservando il contenuto della memoria già allocata e non inizializza il blocco in aggiunta
- In caso *p sia un puntatore non allocato o su cui è già stata chiamata *free* il comportamento è non definito

L'allocazione dinamica della memoria (calloc, realloc)

- In questo esempio viene allocato dinamicamente (e inizializzato a 0) lo spazio necessario a contenere 10 interi. Successivamente, lo spazio allocato viene allargato per contenere 20 interi

```c
int main(void) {
    int i, *p;

    p = calloc(10, sizeof(*p));
    for (i=0; i<10; i++) {
        printf("%d\n", p[i]);
    }

    p = realloc(p, 20 * sizeof(*p));
    for (i=10; i<20; i++) {
        p[i] = 0;
    }
}
```

L'allocazione dinamica della memoria (esempio strdup)

- Esistono funzioni di libreria che utilizzano *malloc* per espletare i loro compiti

```
1  char *strdup(const char *s);
```

- *strdup* (STRing DUPlicate) dichiarata in string.h, ritorna un puntatore a una nuova stringa che è un duplicato della stringa *s* passata come parametro
- La funzione, al suo interno, alloca memoria per la nuova stringa con malloc. Quando la copia generata non viene più utilizzata, la memoria deve essere esplicitamente liberata con *free*

```
1  The strdup() function returns a pointer to a new string which is
2  a duplicate of the string s.  Memory for the new string is
3  obtained with malloc(3), and can be freed with free(3).
```

Problemi con i puntatori (dangling references)

Questa situazione ricorre durante l'accesso tramite puntatore ad un'area di memoria non (piu') allocata.

```
1  int *p;                          /* puntatore a intero (definizione) */
2
3  p=(int *)malloc(sizeof(int));    /* allocazione della memoria */
4
5  *p=57;                           /* impiego dell'area allocata */
6
7  free(p);                         /* deallocazione memoria */
8
9  *p=20;                           /* Errore! Dangling Reference */
10                                  /* L'area di memoria puntata da p non e' piu
                                        ' disponibile !!!! */
11
12 p=NULL;                          /* Non accedo alla memoria puntata da p */
13                                  /* Accedo a p e lo faccio puntare a NULL */
```

Problemi con i puntatori (aree non piu' utilizzabili)

Questo problema avviene quando, per un qualsiasi motivo, viene perso l'indirizzo di un'area di memoria ancora allocata. Chiaramente l'area di memoria interessata non e' piu' referenziabile e nemmeno deallocabile!

```
1  int *p1, *p2;                          /* definizione di 2 puntatori a intero
       */
2
3  p1 = (int *)malloc(sizeof(int));       /* alloco 1^ area di memoria */
4  p2 = (int *)malloc(sizeof(int));       /* alloco 2^ area di memoria */
5
6  /* Errore! p2 punta all'area di p1. Non posso piu' accedere alla memoria
       allocata con la seconda malloc() */
7  p2 = p1;
```