

# 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 messaggio 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

```
1 void stampa_bit(void *ptr) {
2     int i;
3     printf("%p ", ptr);
4     for (i = 31; i >= 0; i--) {
5         printf("%d", (*ptr) >> i) & 0x01);
6     }
7     printf("\n");
8 }
9
10 int main(void) {
11     int a = 100;
12     float b = 100.0F;
13     stampa_bit((void *)&a);
14     stampa_bit((void *)&b);
15 }
```

[illegible]

# Aritmetica dei puntatori

- 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 valore numerico del puntatore (indirizzo in memoria espresso in byte) viene incrementato/decrementato della dimensione di un oggetto puntato (`sizeof(*p)`)

```
1  int main(void) {  
2      int i, v[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
3      int *p = v;  
4  
5      for (i = 0; i < 10; i++) {  
6          printf("[%d] %d %d %d\n", i, v[i], *(v + i), *(p + i));  
7      }  
8  
9      for (i = 0; i < 10; i++, p++) {  
10         printf("[%d] %d %d %d\n", i, v[i], *(v + i), *p);  
11     }  
12 }
```

# Aritmetica dei puntatori

- 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 *false*)
- L'unico valore di  $v$  con valore zero deve essere ultimo, altrimenti il puntatore assumerà valori non validi andando ad accedere oltre la fine del vettore oppure il ciclo terminerà in modo prematuro

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

# Aritmetica dei 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

```
1  int main(void) {  
2      int *p, *q, v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
3  
4      p = &v[0];  /* p = v */  
5  
6      q = p + 2;  
7      printf("%d\n", q - p);  /* Output: 2 */  
8  
9      q = p + 4;  
10     printf("%d\n", q - p);  /* Output: 4 */  
11 }
```

# 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 */
```

# Puntatori e stringhe

- 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* (non è possibile modificare l'indirizzo ad un vettore!)
  - 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

```
1  int main(void) {
2      char s1[] = "nicola";
3      char *s2 = "nicola";
4
5      s1 = s2;          /* Errore in compilazione */
6      s2[0] = 'a';      /* Errore a runtime */
7      printf("%lu %lu\n", sizeof(s1), sizeof(s2));    /* Output: 7 8 */
8  }
```



# Puntatori e stringhe

```
1 char s1[] = "prova";
2 char s2[] = {'p', 'r', 'o', 'v', 'a', '\0'};
3 char c, *t;
4
5 c = *s1;          /* c == 'p' */
6 t = s1 + 2;       /* t contiene indirizzo del carattere 'o' */
7
8 s1[0] = *t;        /* s1 == "orova" */
9
10 t++;              /* t contiene indirizzo del carattere 'v' */
11
12 s1++;              /* errore in compilazione */
```

# Puntatori e stringhe

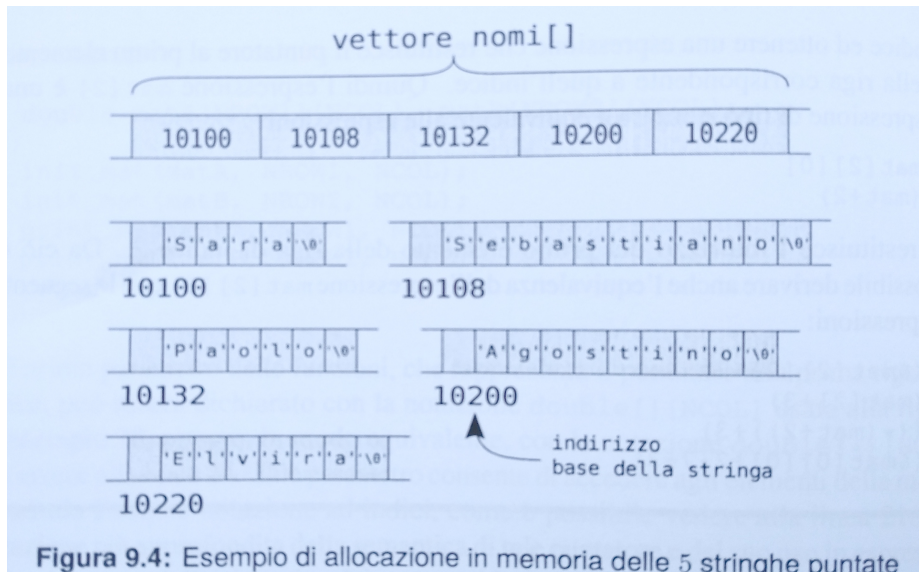
- L'esempio seguente mostra un possibile utilizzo dell'aritmetica dei puntatori al fine di calcolare la lunghezza di una stringa *zero-terminata*

```
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

```
1  int main(void) {
2      int i;
3      char *strings[] = {
4          "Sara", "Sebastiano",
5          "Paolo", "Agostino",
6          "Elvira", NULL,
7      };
8      char **p = strings;
9
10     for (i = 0; i < 3; i++) {
11         printf("[%d] %s %s %s\n", i, strings[i], *(strings + i), *(p + i));
12     }
13
14     for (p = strings; *p; p++) {
15         printf("[%d] %s\n", i, *p);
16     }
17 }
```

# Vettori di stringhe



# Vettori di stringhe

```
1  int main(void) {
2      char *strings[] = {
3          "Sara", "Sebastiano", "Paolo", "Agostino", "Elvira", NULL,
4      };
5      char **p = strings;
6
7      /* Indirizzo del primo puntatore a carattere */
8      printf("%p %p %p\n", strings, &strings[0], p);
9
10     /* Indirizzo del secondo puntatore a carattere */
11     printf("%p %p\n", &strings[1], p + 1);
12
13     /* Indirizzo del primo carattere della prima stringa */
14     printf("%p %p\n", *p, &strings[0][0]);
15
16     /* Valore del primo carattere della prima stringa */
17     printf("%c %c\n", **p, strings[0][0]);
18 }
```

# L'allocazione dinamica della memoria

- Il linguaggio C permette di effettuare l'allocazione di memoria anche durante l'esecuzione del programma, sulla base di opportune condizioni che possono verificarsi durante l'esecuzione
- Questo tipo di allocazione di memoria è detta dinamica (*heap*), proprio perché avviene dinamicamente durante l'esecuzione. L'allocazione cosiddetta statica (*stack*) 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 della funzione in cui l'allocazione è avvenuta

```
1 #include <stdlib.h>
2 void *malloc(size_t n);
3 void *calloc(size_t n, size_t size);
4 void *realloc(void *pt, size_t n);
5 free(void *p);
```

# L'allocazione dinamica della memoria (malloc)

```
1 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*

```
1 int *p;  
2 /* Dipendente dal tipo di dato */  
3 p = malloc(10 * sizeof(int));  
4  
5 /* Indipendente dal tipo di dato, da preferire */  
6 p = malloc(10 * sizeof(*p));
```

# L'allocazione dinamica della memoria (malloc)

```
1  int *p;  
2  p = malloc(10 * sizeof(*p));  
3  
4  if (!p) {  
5      /* gestione dell'errore */  
6  }  
7  
8  for (int i = 0; i < 10; i++) {  
9      p[i] = i;    /* oppure *(p + i) = i; */  
10 }  
11  
12 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' possibile utilizzare il puntatore tramite indici (notazione []) per accedere alla memoria allocata



# L'allocazione dinamica della memoria (malloc)

- Tecniche per azzerare gli elementi di un vettore (allocato dinamicamente e non)

```
1  /* calloc: alloca un vettore e lo inizializza a 0 */
2  p = calloc(10, sizeof(*p));
3
4  /* for e puntatori */
5  for(i = 0; i < 10; i++)
6      p[i] = 0;
7
8  for (i = 0; i < 10; i++, p++)
9      *p = 0;
10
11 /* memset */
12 memset(p, 0, 10 * sizeof(*p));
13
14 /* bzero */
15 bzero(p, 10 * sizeof(*p));
```

# 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

```
1  int main(void) {  
2      int i, *p;  
3  
4      p = calloc(10, sizeof(*p));  
5      for (i=0; i<10; i++) {  
6          printf("%d\n", p[i]);  
7      }  
8  
9      p = realloc(p, 20 * sizeof(*p));  
10     for (i=10; i<20; i++) {  
11         p[i] = 0;  
12     }  
13 }
```

# L'allocazione dinamica della memoria (esempio strdup)

- Esistono funzioni di libreria che utilizzano *malloc* per svolgere 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)

- Un puntatore *dangling* è puntatore che punta ad un'area di memoria non valida

```
1  int *p;                /* puntatore a intero (definizione) */
2
3  p = 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
11                          ' disponibile !!!! */
12
13  p = NULL;              /* Non accedo alla memoria puntata da p */
                          /* Accedo a p e lo faccio puntare a NULL */
```

# Problemi con i puntatori (dangling references)

- In questo caso viene ritornato l'indirizzo di una variabile memorizzata nella porzione di stack riservata alla funzione *func*. Sfortunatamente, quella porzione di memoria viene resa disponibile non appena la funzione *func* termina. Questa circostanza genera un warning in compilazione.

```
1  int *func(void) {  
2      int n = 13;  
3      return &n;  
4  }  
5  
6  int main(void) {  
7      int *p;  
8  
9      p = func();  
10     printf("%d\n", *p);  
11 }
```

# 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 = malloc(sizeof(int)); /* alloco 1^ area di memoria */
4  p2 = 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;
```



# Allocazione dinamica e matrici

- E' possibile allocare dinamicamente matrici (array bi-dimensionali), utilizzando sia un singolo puntatore (type `*matrix`) che un puntatore a puntatore (type `**matrix`)
- Il primo (singolo puntatore) è semplice, immediato, ma impedisce uso di indicizzazione esplicita (`matrix[i][j]`) e necessita di calcolo manuale dell'offset (`offset = i * cols + j`)
- Il secondo (puntatore a puntatore) richiede un meccanismo più complesso per allocare e disallocare la memoria, ma consente l'uso di indicizzazione esplicita (`matrix[i][j]`)
- Nel caso di oggetti relativamente complessi come le matrici, in cui i dati veri e propri (il contenuto della matrice) è sempre abbinato ad informazioni aggiuntive come al esempio il numero di righe e di colonne, l'utilizzo di strutture (aggregazione del codice) è consigliabile!

# Allocazione dinamica e matrici (singolo puntatore)

```
1  int *allocate_matrix(int rows, int cols) {  
2      int *m;  
3  
4      m = malloc((unsigned long)(rows * cols) * sizeof(*m));  
5      if (m == NULL) return NULL;  
6      return m;  
7  }
```

```
1  int main(void) {  
2      int *m;  
3  
4      m = allocate_matrix(10, 10);  
5      /* code here */  
6      free_matrix(m);  
7  }
```

# Allocazione dinamica e matrici (singolo puntatore)

```
1 void show_matrix(int rows, int cols, int *m) {
2     int i, j, offset;
3
4     for (i = 0; i < rows; i++) {
5         for (j = 0; j < cols; j++) {
6             offset = i * cols + j;
7             printf("%4d", m[offset]);
8         }
9         printf("\n");
10    }
11 }
```

```
1 void free_matrix(int *m) {
2     free(m);
3 }
```

# Allocazione dinamica e matrici (doppio puntatore)

```
1  int **allocate_matrix(int rows, int cols) {  
2      int i, **m;  
3  
4      m = malloc((unsigned long)(rows) * sizeof(*m));  
5      if (m == NULL) return NULL;  
6      for (i = 0; i < rows; i++) {  
7          m[i] = malloc((unsigned long)(cols) * sizeof(**m));  
8          if (m[i] == NULL) return NULL;  
9      }  
10     return m;  
11 }
```

```
1  int main(void) {  
2      int **m;  
3      m = allocate_matrix(3, 2);  
4      /* code here */  
5      free_matrix(3, m);  
6  }
```

# Allocazione dinamica e matrici (doppio puntatore)

```
1 void show_matrix(int rows, int cols, int **m) {
2     int i, j;
3     for (i = 0; i < rows; i++) {
4         for (j = 0; j < cols; j++) {
5             printf("%4d", m[i][j]);
6         }
7         printf("\n");
8     }
9 }
```

```
1 void free_matrix(int rows, int cols, int **m) {
2     int i;
3     for (i = 0; i < rows; i++) {
4         free(m[i]);
5     }
6     free(m);
7 }
```

# Allocazione dinamica e strutture

```
1 struct matrix {  
2     size_t rows, cols;  
3     double **data;  
4 };
```

```
1 int main(void) {  
2     struct matrix *m, *m_trans;  
3  
4     m = allocate_matrix(3, 2);  
5  
6     fill_matrix(m);  
7  
8     m_trans = matrix_transpose(m);  
9  
10    free_matrix(m);  
11    free_matrix(m_trans);  
12 }
```

# Allocazione dinamica e strutture

```
1  struct matrix *allocate_matrix(size_t rows, size_t cols) {
2      struct matrix *m;
3      int i;
4
5      m = malloc(sizeof(*m));
6      m->rows = rows;
7      m->cols = cols;
8      m->data = malloc((unsigned long)(rows) * sizeof(*m));
9      if (m == NULL) return NULL;
10
11     for (i = 0; i < rows; i++) {
12         m->data[i] = malloc((unsigned long)(cols) * sizeof(**(m->data)));
13         if (m->data[i] == NULL) return NULL;
14     }
15
16     return m;
17 }
```

# Allocazione dinamica e strutture

```
1 void show_matrix(const struct matrix *m) {  
2     size_t i, j;  
3     for (i = 0; i < m->rows; i++) {  
4         for (j = 0; j < m->cols; j++) {  
5             printf("%8.2lf", m->data[i][j]);  
6         }  
7         printf("\n");  
8     }  
9 }
```

```
1 void free_matrix(struct matrix *m) {  
2     int i;  
3     for (i = 0; i < m->rows; i++) {  
4         free(m->data[i]);  
5     }  
6     free(m->data);  
7     free(m);  
8 }
```