Puntatori

Operazioni sui puntatori
Allocazione dinamica

Università
degli Studi
di Pavia

Puntatori a void

- la parola chiave void può essere usata per dichiarare dei puntatori che non puntano a nessun tipo di dato in particolare

```
void *ptr;
```

- è 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 a tipi diversi causa la generazione di un messaggio di warning o di errore da parte del compilatore

Operazioni sui puntatori

- Ai puntatori possono essere sommati e sottratti numeri interi
- Il risultato della somma di un puntatore e di un numero intero n 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

Operazioni sui puntatori

```
int *p, *q;
q = p + 5;
```

- l'indirizzo contenuto in p è incrementato di 5 * sizeof(int), quindi ad esempio di 20 byte se l'int ha dimensione pari a 4 byte
- 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

Operazioni sui puntatori

- è possibile fare la differenza (ma non la somma) tra puntatori dello stesso tipo

```
int v = q - p;
```

- il risultato in questo secondo caso è un numero intero che rappresenta il numero di elementi tra i due puntatori
- la dimensione di un elemento è quella del tipo di dato puntato dal puntatore

Operatori

- gli operatori fondamentali per usare i puntatori sono
  - * (si legge "il puntato da")
  - & (da leggere "l'indirizzo di")
  - [] accedo ad un elemento particolare di un vettore
  - Si noti che *p == p[0]
  - In generale *(p+i) == p[i]

Esempi di utilizzo

```
int i, v[10], *p;

p = v;

p = &v[0];

p = &v[4];

p = v + 4;

p++;

i = p - v;
```

assegna a p il valore di v, ossia l'indirizzo del suo primo elemento

come al punto precedente

assegna a p l'indirizzo del quinto elemento di v

come al punto precedente

incrementa di 1 l'indirizzo p
se p vale v+4, l'istruzione equivale ad assegnare a p l'indirizzo v+5 (equivalente a &v[5])

assegna il valore ad i (in conseguenza delle due righe precedenti)

Esempi di utilizzo

```
sum = 0;
for (p = v; *p; p++) {
  sum += *p;
}
```

- 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 da p, cioè *p, è nullo
  - il valore 0 equivale alla condizione logica "falso"
- il ciclo calcola la somma di tutti i valori considerati
- deve esserci almeno un elemento di v che vale zero, altrimenti il puntatore assumerà valori non validi andando ad accedere oltre la fine del vettore

Assegnamenti tra puntatori

- fare assegnamenti tra puntatori di tipo diverso è una operazione che viene segnalata con un messaggio di avvertimento, a meno di non effettuare un cast esplicito
- è possibile convertire un puntatore da un tipo ad un altro, ma anche un puntatore in numero intero e viceversa
- le conversioni tra puntatori a tipi diversi o tra puntatori e interi non provocano la generazione di codice macchina
  - nel processore i puntatori sono rappresentati da numeri interi
- le conversioni sono necessarie per la pulizia semantica del codice sorgente

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 viene effettuata dal compilatore

L'allocazione dinamica della memoria

- le funzioni per l'allocazione e la gestione della memoria possono essere utilizzate includendo il file di intestazione `<stdlib.h>`

```
void * malloc(size_t n);
void * calloc(size_t n, size_t size);
void * realloc(void *pt, size_t n);
free (void *pt);
```

L'allocazione dinamica della memoria

```
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, oppure NULL se non è stato possibile allocare la memoria
- lo spazio allocato in memoria è contiguo

L'allocazione dinamica della memoria

```
int *pt;
pt = malloc(10 * sizeof(*pt));
if (!pt) {
   /* gestione dell'errore */
}
/* codice che utilizza pt */
free(pt);
```

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

L'allocazione dinamica della memoria

- Per azzerare tutti gli elementi interi memorizzati:

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

- Ma si può anche fare:

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

L'allocazione dinamica della memoria

```
free (void *pt);
```

- libera il blocco di memoria di indirizzo pt 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 che potrebbe essere necessario allocare in seguito e non essere disponibile

L'allocazione dinamica della memoria

```
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

```
void * realloc(void *pt, size_t n);
```

- ridimensiona un blocco di memoria già allocato e puntato da pt preservando il contenuto della memoria già allocata
- la nuova dimensione è n

L'allocazione dinamica della memoria

- ci sono funzioni di libreria che utilizzano malloc per espletare i loro compiti

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

- (STRing DUPlicate) dichiarata in string.h
- ritorna un puntatore a una nuova stringa che è un duplicato di s
- alloca memoria per la nuova stringa con malloc
- la memoria deve essere esplicitamente liberata con free

Uso di malloc

```
int *vett, n;
```

- Quando si alloca dinamicamente la memoria mediante malloc per memorizzare n elementi di tipo int nel vettore vett, le due istruzioni seguenti sono equivalenti:

```
vett = malloc(n * sizeof(int));
vett = malloc(n * sizeof(*vett));
```

- la prima è più immediata da comprendere
- la seconda è una notazione che non cambia al cambiare del tipo associato a vett

Allocazione dinamica di una matrice

`alloca-matrice.c`

Allocazione di una matrice triangolare

```
for (i = 0; i < rig; i++){
  mat[i] = malloc((i + 1) * sizeof(**mat));
```

- Anche la scansione della matrice dovrà essere fatta in modo da far variare l'indice di colonna nel range dei valori corrispondenti agli elementi effettivamente allocati
- Esempio di ciclo che accede ai valori memorizzati nella matrice triangolare allocata:

```
for (i = 0; i < rig; i++){
  for(j = 0; j < i + 1; j++){
    /* accesso all'elemento mat[i][j] */
  }
}
```

Puntatori dangling

```
char *p = malloc(DIM);
/* ... */
free(p);
```

- p continua a puntare a una locazione di memoria, che però ora non è più utilizzabile
  - si parla di puntatori dangling

```
p = NULL;
```

- ora p non è più dangling
  - è sempre bene annullare il puntatore
  - dopo l'assegnamento, un tentativo di utilizzare il puntatore p genera un errore di accesso alla memoria (segmentation fault), diviene così più semplice scoprire eventuali accessi errati

Funzioni che restituiscono un indirizzo

```
tipo dato;
/* inizializzo dato */
```

- Versione sbagliata

```
return &dato;
```

- dato è una variabile locale, al termine dell'esecuzione della funzione la sua memoria verrà riutilizzata!

- Versione corretta

```
tipo *pt = malloc(sizeof(tipo));
*pt = dato;
return pt;
```

Funzioni che restituiscono un indirizzo

```
#include <stdlib.h>

int * f1(int v)
{
    int v1 = v;
    return &v1;
}
int * f2(int v)
{
   int *pt = malloc(sizeof(int));
   *pt = v;
   return pt;
}
$ gcc -Wall -c funz.c
funz.c: In function `f1':
funz.c:6: warning: function returns address of local variable
```

Allocazione dinamica di un vettore

history.c

user@ubuntu: ~

```
$ gcc -Wall -DVERSIONE=1 history.c
$ ./history if=hh --debug
input: '!gcc': eseguo: gcc -Wall rettangolo.c
input: '!zz': comando non trovato
$ ./history if=hh max=4 --debug
riallocato 8!
input: '!gcc': eseguo: gcc -Wall rettangolo.c
riallocato 16!
input: '!zz': comando non trovato
riallocato 32!
riallocato 64!
$
```

Funzioni per le stringhe (string.h)

```
strcmp (s1, s2); confronta due stringhe
strcasecmp (s1, s2); versione case insensitive
strcpy(dest, src); copia src in dest
strcat(dest, src); appende src a dest
strdup(s1); duplica s1
strlen(s1); lunghezza di s1

come sopra ma considerando al più n caratteri
strncmp (s1, s2, n);
strncasecmp (s1, s2, n);
strncpy(dest, src, n);
strncat(dest, src, n);
```