Algoritmi e Strutture di Dati

Esercitazioni in linguaggio C

Array e puntatori

m.patrignani

Nota di copyright

- queste slides sono protette dalle leggi sul copyright
- il titolo ed il copyright relativi alle slides (inclusi, ma non limitatamente, immagini, foto, animazioni, video, audio, musica e testo) sono di proprietà degli autori indicati sulla prima pagina
- le slides possono essere riprodotte ed utilizzate liberamente, non a fini di lucro, da università e scuole pubbliche e da istituti pubblici di ricerca
- ogni altro uso o riproduzione è vietata, se non esplicitamente autorizzata per iscritto, a priori, da parte degli autori
- gli autori non si assumono nessuna responsabilità per il contenuto delle slides, che sono comunque soggette a cambiamento
- questa nota di copyright non deve essere mai rimossa e deve essere riportata anche in casi di uso parziale

Richiami di linguaggio C

- Puntatori in linguaggio C
- Gli operatori `*` e `&`
- Algebra dei puntatori
- Array
- Allocazione statica e dinamica di array

Puntatori in linguaggio C

- Molti linguaggi supportano i riferimenti
  - un riferimento è una variabile che viene utilizzata per identificare un'altra variabile (o un oggetto)

- Nel linguaggio C i riferimenti sono i puntatori
  - un puntatore è una variabile (cioè una cella di memoria con un tipo e un nome) che contiene l'indirizzo di un'altra cella di memoria
    - per esempio l'indirizzo di un'altra variabile

L'operatore *

- Nel linguaggio C il tipo di una variabile puntatore si ottiene dal tipo della variabile puntata seguito dal simbolo * (leggi "star")
- Esempi
  - un puntatore ad intero ha tipo `int*`
    - leggi "int star"
    - è indifferente la presenza di spazi o meno tra la stringa "`int`" e il simbolo "`*`"
  - un puntatore a carattere ha tipo `char*`
    - leggi "char star"
  - un puntatore a puntatore ad intero ha tipo `int**`
    - leggi "int star star"

Esempi di dichiarazioni di puntatori

- La variabile `a` è un puntatore ad intero

```
int* a;
```

- La variabile `b` è un puntatore a puntatore ad intero

```
int** b;
```

- La variabile `c` è un puntatore ad un float

```
float* c;
```

- La variabile `d` è un puntatore ad una struttura

```
struct {
     int minimo;
     int massimo;
}* d;
```


La costante NULL

- Il valore costante `NULL` è un valore convenzionale che si suppone assegnato ad un puntatore che non contiene alcun indirizzo significativo
  - nella rappresentazione interna `NULL` corrisponde al valore `0` (zero), che non è legittimo per un indirizzo di memoria a disposizione dell'utente

La costante NULL

- Quando viene dichiarato un puntatore senza inizializzarlo, si suppone che il suo valore sia `NULL`

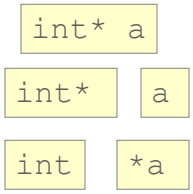
```
int* a;  /* a è ancora uguale a NULL */
```

- Questo può essere anche esplicitato da un'assegnazione

```
int* a = NULL;
```


Due diverse interpretazioni dell'operatore *

- L'espressione "`int* a`" può essere interpretata in due modi equivalenti
  1. la variabile "`a`" è di tipo "`int*`"
  2. "`*a`", cioè "l'oggetto puntato da `a`" è di tipo "`int`"



- Infatti, l'operatore `*` anteposto ad un indirizzo di memoria identifica il contenuto della memoria stessa
  - "`*(a+b)`" è il contenuto della cella di memoria che si trova in posizione `a+b`

L'operatore &

- L'operatore `&` (e commerciale) anteposto ad una variabile ne estrae l'indirizzo
- Esempio
  - l'indirizzo della variabile intera "`a`" è "`&a`"
    - "`&a`" è un'espressione di tipo "`int*`"
- Non ha senso utilizzare l'operatore `&` con espressioni generiche
  - la scrittura "`&(a+b)`" non ha senso perché "`a+b`" non è una variabile, ma solo un valore
    - tecnicamente: non ha left-value ma solo right-value

Algebra degli operatori * e &

- Regola mnemonica
  - ogniqualvolta si incontra l'operatore * lo si può sostituire con "il puntato da" oppure "il contenuto di"
  - ogniqualvolta si incontra l'operatore & lo si può sostituire con "l'indirizzo di"
- I due operatori possono essere combinati insieme in modo spesso complesso

```
int a = 3;
int* b = &a;
int** c = &b;
*b = *b+1;          /* ora a vale 4 */
**c = **c+1         /* ora a vale 5 */
```


Gli array

- Un array è una sequenza di variabili omogenee, tutte memorizzate in celle contigue di memoria e indicizzate con un indice intero
- In linguaggio C un array si dichiara premettendo il tipo degli elementi e posponendo il numero degli elementi tra parentesi quadre

```
int a[10];    /* a array di 10 interi */
float b[5];   /* b array di 5 float */
```

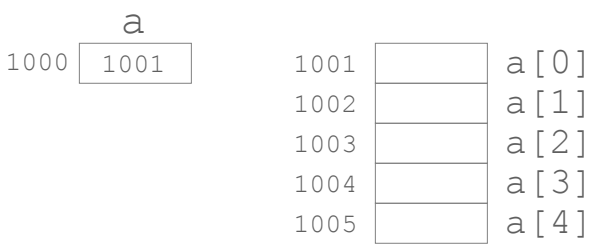
  - in ansi C il numero di elementi deve essere una costante, non può essere una variabile o un'espressione generica

```
int n=10;     /* numero di celle */
int a[n];     /* errore! */
```


Array e allocazione della memoria

- Cosa succede in memoria quando dichiariamo un array?

```
int a[5]; /* a array di 5 interi */
```



- Quante celle di memoria vengono allocate quando dichiaro un array di 5 interi?

Array e puntatori

- Un array è a tutti gli effetti un puntatore
  - è l'indirizzo della prima cella di memoria dell'array
- In linguaggio C l'identità tra array e puntatori è esplicita

```
int a[5];      /* a è di tipo int* */
int* b = a;    /* b ha lo stesso valore
                  di a */
```

- Tuttavia un array è un puntatore costante
  - non è legittimo sovrascrivere o modificare l'indirizzo di memoria associato ad un array

Incremento di un puntatore

- Incrementando un puntatore si salta una porzione di memoria pari alla dimensione dell'oggetto puntato

```
int a[5];     /* a array di 5 interi */
int* b = a;   /* b punta alla prima
                 cella di a */
b = b + 1;    /* b punta alla seconda
                 cella di a */
b++;          /* b punta alla terza
                 cella di a */
```


Interpretazione delle parentesi quadre

- Date due espressioni `x` e `y`, il costrutto `x[y]` è equivalente all'operazione `*(x+y)`
- Quindi queste espressioni sono equivalenti

```
a[0] = 100;   <-->   *(a+0) = 100;
a[1] = 200;   <-->   *(a+1) = 200;
```

- Si può verificare l'equivalenza persino delle seguenti espressioni

```
a[1] = 200;   <-->   1[a] = 200;
```

  - infatti `*(a+1)` è uguale a `*(1+a)`

Allocazione statica degli array

- La dichiarazione di array che abbiamo visto alloca le celle dell'array direttamente sullo stack

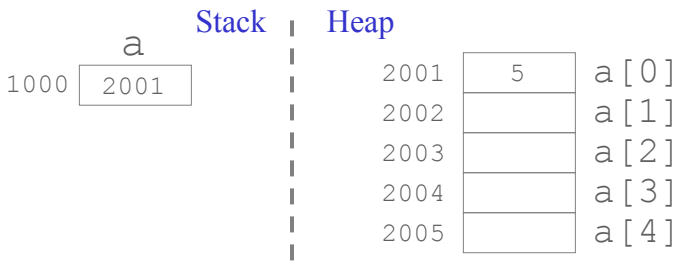
```
int a[5]; /* a è nello stack.
             Anche a[0],a[1],… ,a[4]
             sono nello stack */
```

- Questo comporta due vincoli
  - il numero delle celle non può essere modificato
  - il valore di `a` non può essere modificato

Allocazione dinamica degli array

- Nell'allocazione dinamica l'array viene posto nello heap tramite la funzione `calloc`

```
int* a;
a = (int *)calloc(5,sizeof(int));
a[0] = 5;
```



Vantaggi della dichiarazione dinamica

- La dichiarazione dinamica degli array comporta diversi vantaggi
  - il numero di celle dell'array può essere specificato tramite una variabile o un'espressione generica
  - l'indirizzo dell'array può essere modificato
  - il numero delle celle dell'array può essere modificato

```
int* a;
a = (int *)calloc(5,sizeof(int));
a[0] = 100;
…
a = realloc(a,10);
a[8] = a[0];            /* cioè 100 */
```


Esercizi

1. Implementa in linguaggio C uno stack di interi con le funzioni `NEW_STACK`, `IS_EMPTY`, `PUSH`, e `POP` e con la gestione telescopica della memoria
   - operazione `PUSH` con complessità ammortizzata O(1)
2. Implementa in linguaggio C una coda di interi con le funzioni `NEW_QUEUE`, `IS_EMPTY`, `ENQUEUE`, e `DEQUEUE` e con la gestione telescopica della memoria
   - operazione `ENQUEUE` con complessità ammortizzata O(1)

