Programmazione

Docenti: Giovanni Da San Martino
Francesco Gavazzo
Lamberto Ballan
<lamberto.ballan@unipd.it>

Programmazione, A.A. 2023/24
SCQ0093758 - LT Informatica

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

File (binari) ad accesso casuale

- Finora abbiamo visto come accedere a file di testo in modalità sequenziale
- L'accesso ai dati però è inefficiente, e in molti casi vorremmo trattare dati grezzi e non solo testo
  - I file di testo sono formati da righe/stringhe di caratteri (ad es. con codifica ASCII) terminate dal simbolo `'\n'`
- Un'alternativa è quindi quella dei file binari: un file binario è, in pratica, una generica sequenza di byte "grezzi"

File (binari) ad accesso casuale

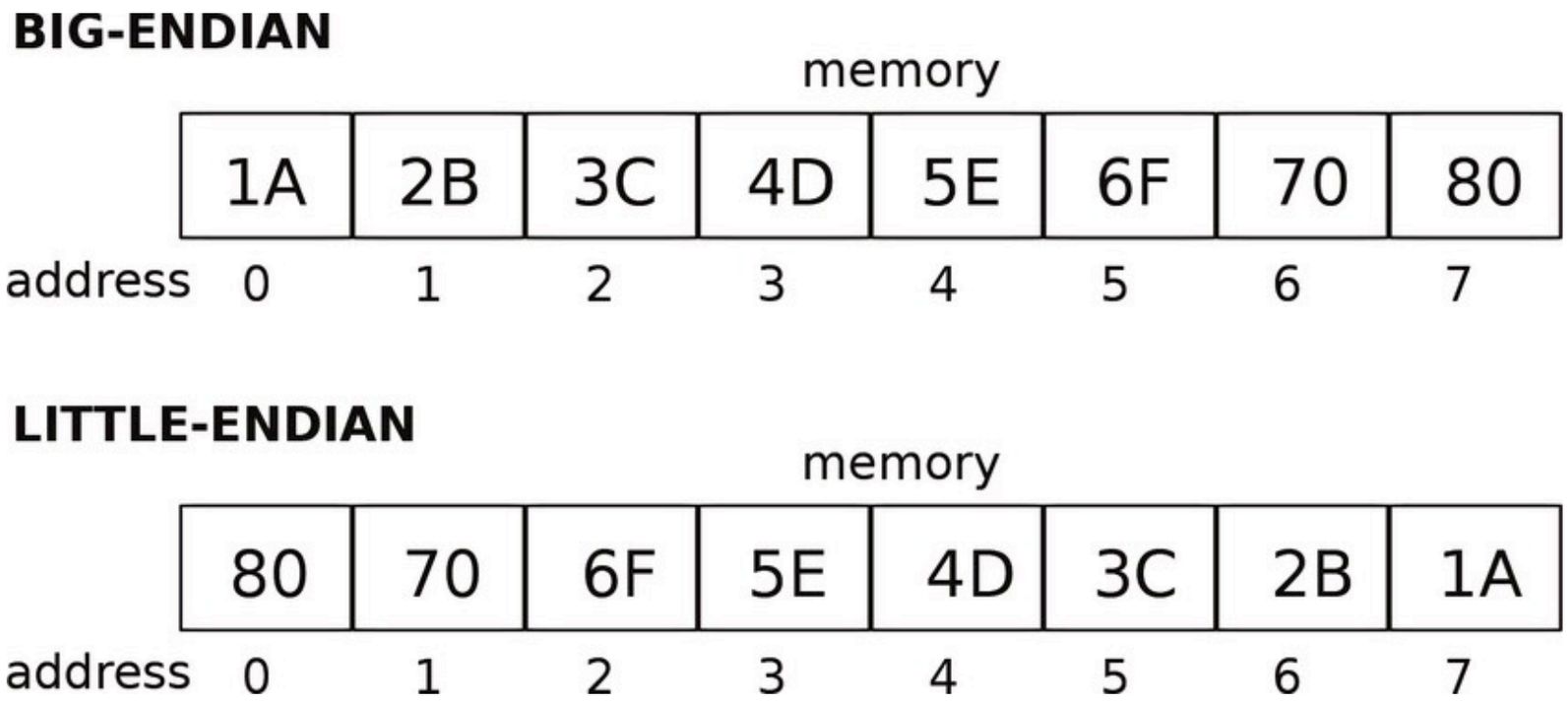
- Nei file binari si dovrebbe parlare più propriamente di byte, anche se spesso viene usato il termine carattere
  - Generalmente i byte vengono raggruppati in blocchi di byte chiamati *record*
  - Ogni record viene letto/scritto in un'unica operazione
- Quando i record hanno la stessa dimensione, l'accesso al record n-esimo è molto veloce perché (se il record occupa L byte) la sua posizione sarà n*L
  - In questo caso si parla di file ad accesso diretto (o casuale)

Apertura di un file binario

- I file binari si aprono con `fopen()` ma indicando i modi binari (simili ai file di testo ma con lettera `'b'`)
  - "rb", "wb", "ab", "r+b" o "rb+", "w+b" o “wb+", "a+b" o “ab+"
  - La differenza tra “r+b” e “w+b” è solo nell’apertura (come nel caso dei file di testo)
- Anche i file binari si chiudono con `fclose()`

File binari e compatibilità

- I file binari possono avere problemi di portabilità tra architetture (causati da diverse strategie di memorizzazione)
  - Ad es. codifica *big-endian* vs *little-endian*:



Lettura in file binari

- La lettura di blocchi di byte si ottiene con la funzione `fread(p,<dim_record>,<num_record>,fp);`
  - Legge dal file fp un numero di record pari a `<num_record>` ciascuno di `<dim_record>` byte e li colloca nella struttura puntata da p
  - Restituisce `size_t` che contiene il numero di record letti
- Fa avanzare il *file position pointer* del numero di byte letti

Lettura in file binari

- La lettura di blocchi di byte si ottiene con la funzione `fread(p,<dim_record>,<num_record>,fp);`
  - Vediamo un primo esempio:

```
struct studente stud;
fp = fopen("studenti.dat","rb");
fread(&stud, sizeof(struct studente), 1, fp);
```

  - Vediamo un altro esempio:

```
struct studente inf22_23[180];
fp = fopen("studenti.dat","rb");
fread(inf22_23, sizeof(struct studente), 180, fp);
```

Scrittura in file binari

- La funzione di scrittura è analoga a quella di lettura
- La scrittura di blocchi di byte si ottiene con la funzione `fwrite(p,<dim_record>,<num_record>,fp);`
  - Scrive dal file fp un numero di record pari a `<num_record>` ciascuno di `<dim_record>` byte, prelevandoli dalla struttura (o array) puntata da p
  - Restituisce `size_t` che contiene il numero di record scritti; può essere un numero inferiore a `<num_record>` in caso di errore in scrittura
  - Fa avanzare il *file position pointer* del numero di byte scritti

Scrittura in file binari

- La scrittura di blocchi di byte si ottiene con la funzione `fwrite(p,<dim_record>,<num_record>,fp);`
  - Vediamo un esempio:

```
struct studente stud;
…
fwrite(&stud, sizeof(struct studente), 1, fp);
```

Scrittura in file binari

- La scrittura di blocchi di byte si ottiene con la funzione `fwrite(p,<dim_record>,<num_record>,fp);`
  - E se volessi scrivere su file due volte i dati dello stesso `stud`?

```
struct studente stud;
…
fwrite(&stud, sizeof(struct studente), 1, fp);
fwrite(&stud, sizeof(struct studente), 1, fp);
```

  - Domanda: potrei scrivere la stessa istruzione come segue?

```
…
fwrite(&stud, sizeof(struct studente), 2, fp);
```

Risposta: **No!!!**

Lettura e scrittura di blocchi di byte

- Ricapitolando quanto detto:
  - Le funzioni `fread` e `fwrite` leggono e scrivono un numero di byte pari a `<dim_record>*<num_record>`
  - Le funzioni ricevono un semplice puntatore al blocco e non hanno alcuna informazione sul contenuto del blocco stesso
  - Solo quando il blocco viene letto da `fread` e viene fatto un cast (anche implicito) assegnandolo ad una variabile, esso assume un significato specifico

Lettura e scrittura di blocchi di byte

- Vediamo un esempio:

```
struct persona {
   char nominativo[30];
   unsigned int annoNascita;
} iscritto;
fread(&iscritto, sizeof(iscritto), 1, fp);
```

- Solo quando il blocco viene letto e collocato in `iscritto` che il suo contenuto è identificabile come variabile `nominativo` di tipo puntatore a `char` (stringa di 30 caratteri) e come variabile `annoNascita` di tipo `unsigned int`

File binari e posizionamento

- Come per i file di testo, le funzioni `fread` e `fwrite` posizionano il file position pointer (offset) all'inizio del record successivo
- Il posizionamento generico si ottiene con la funzione `fseek(fp, <offset>, <origine>);`
  - `offset` (di tipo long): indica il numero di caratteri/blocchi in cui deve essere spostato il file position pointer rispetto ad <origine> (nota: può essere un valore negativo)
  - `origine`: indica la posizione da cui calcolare l'offset:
    - `SEEK_SET`: da inizio file
    - `SEEK_CURR`: dalla posizione corrente
    - `SEEK_END`: dalla fine del file (offset negativo)

File binari e posizionamento

- Ad esempio:

```
fseek(fp, 10, SEEK_SET);
```

  - Sposta il *file position pointer* al 11° blocco (il 1° ha posizione 0)

- Per conoscere la posizione del *file position pointer* si può usare la funzione `ftell`:

```
pos = ftell(fp);
```

  - `pos` è una variabile di tipo `long` (in caso di errore la funzione `ftell` restituisce -1L)

- In alternativa si possono usare `fgetpos` e `fsetpos`

Considerazioni sull'efficienza

- Come detto più volte, le operazioni di I/O su file sono estremamente lente
- I file infatti sono un classico esempio di memoria di storage (archivio) e non di lavoro
- Conviene quindi limitare il più possibile l'accesso a file effettuando le operazioni di I/O strettamente necessarie
  - Solitamente si caricano i dati in memoria (in opportune strutture dati) e si aggiorna il file solo alla fine dell'elaborazione dei dati

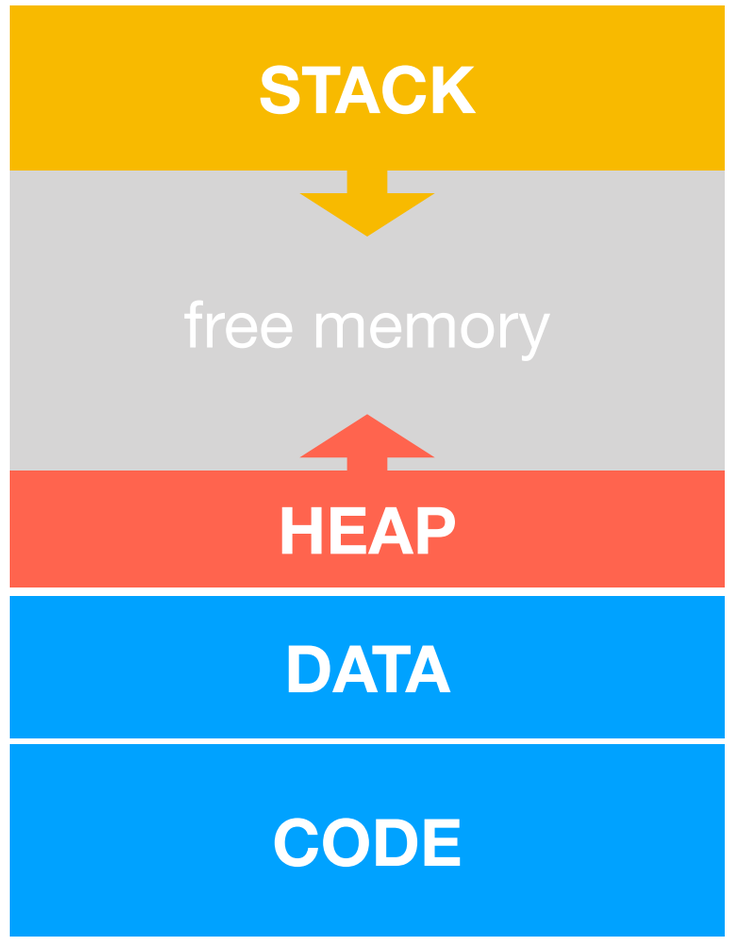
Previously on Programmazione

- Nel corso delle lezioni precedenti:
  - Abbiamo richiamato le modalità di gestione dell'allocazione della memoria (RAM) da parte del linguaggio C
  - Abbiamo visto in particolare allocazione statica ed automatica
  - Abbiamo introdotto l'allocazione dinamica della memoria e le principali funzioni C ad essa associate (`malloc` & co)

Recap: allocazione della memoria

- Il termine *allocazione* della memoria viene utilizzato per indicare l'assegnazione di un blocco di memoria RAM
- La memoria RAM può essere allocata:
  - Staticamente
  - Automaticamente
  - Dinamicamente

Variabili Globali



Recap: allocazione dinamica

- Il nucleo centrale del sistema di allocazione dinamica della memoria in C è costituito dalle funzioni:
  - `malloc()`: alloca un blocco di memoria libera
  - `free()`: libera la memoria precedentemente allocata
  - Tali funzioni sono definite nella libreria `<stdlib.h>`

- L'allocazione dinamica è fondamentale nella pratica (spesso le necessità di memoria non sono note a priori)
  - Questo vale innanzitutto per le variabili / strutture dati "semplici"
  - Ma è sostanzialmente inevitabile per le strutture dati più avanzate (liste, alberi, grafi, …)

Le liste

- Una lista è una successione finita di elementi di un tipo
- L'informazione codificata dalla lista riguarda:
  - La successione di elementi (valori)
  - La relazione di ordine tra gli elementi stessi

- La lista è qualificata non solo dai valori che rappresenta ma anche dalle operazioni che ci si eseguono
  - Inserimento e cancellazione (in testa, coda, intermedia)
  - Visita / ricerca
  - Inizializzazione

Le liste

- Possiamo ottenere diverse tipologie di lista definendo diverse modalità di inserimento/cancellazione elementi
  - pila (stack): è una struttura di tipo LIFO, i.e. “last in first out”
  - coda (queue): è una struttura di tipo FIFO, i.e. “first in first out”

Le liste

- Una lista può essere rappresentata:
  - In forma *sequenziale*: gli elementi sono rappresentati in un array e il loro ordine è codificato in modo implicito dalla posizione
  - In forma *collegata*: in questo caso la relazione è resa esplicita e ogni elemento è associato all'informazione che identifica il successore (questa può essere un indice o un puntatore)

- Una lista collegata con puntatori è una successione di elementi (nodi) connessi da puntatori

Liste collegate e array

- Le liste collegate sono strutture dinamiche e la loro dimensione può aumentare/diminuire a run time
- Gli array invece hanno dimensione pre-fissata e possono riempirsi
  - Gli elementi di un array sono memorizzati in modo contiguo;
    + questo permette l'accesso immediato ad un elemento
    - ma le inserzioni o cancellazioni di un elemento sono lente
  - Gli elementi di una lista collegata, seppur "logicamente" in sequenza, occuperanno indirizzi di memoria non contigui

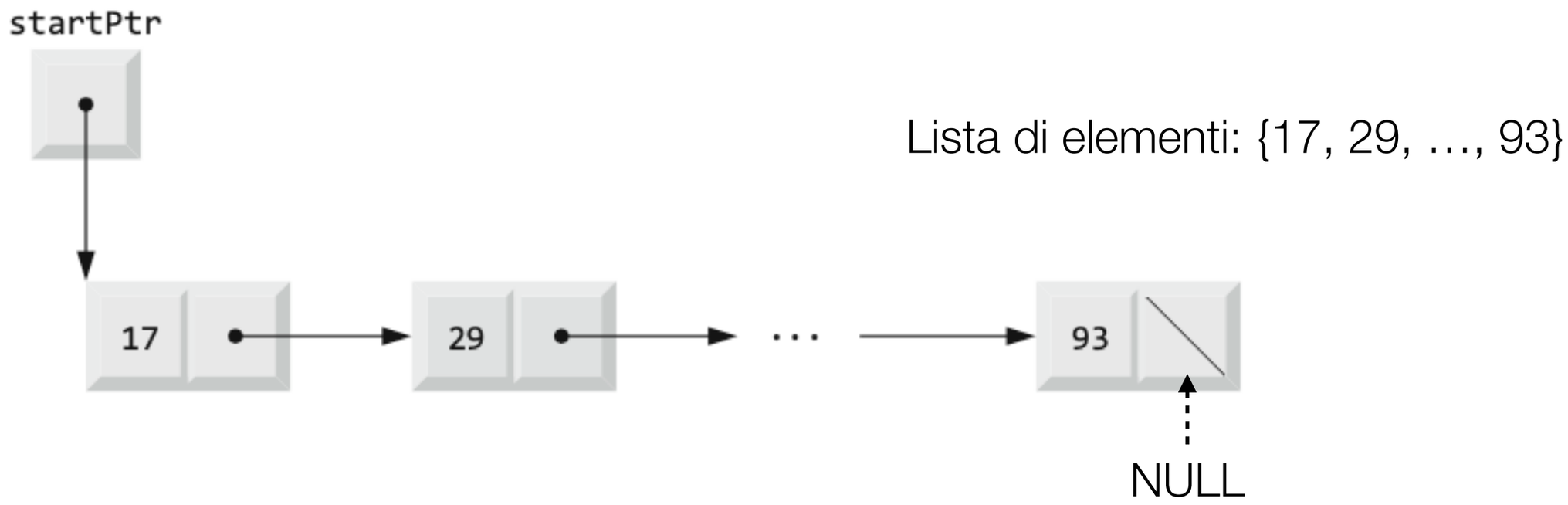
Liste collegate con puntatori

- Le liste collegate non offrono un accesso immediato ai loro elementi, ma inserimento/cancellazione di elementi possono seguire diverse strategie
  - LIFO, FIFO, in posizione “generica”
- L’elemento atomico di una lista è il *nodo* che può essere definito da una struttura “autoreferenziale”:

```
struct nodo {
   int dato;
   struct nodo *nextPtr;
};
```

Liste collegate con puntatori

- Esempio / rappresentazione grafica di una lista collegata:



Lista di elementi: {17, 29, …, 93}

```
struct nodo {
   int dato;
   struct nodo *nextPtr;
};
```

Funzioni su liste collegate

- L'utilizzo di liste collegate richiede la definizione di alcune funzioni di base:
  - inizializzazione
  - visita (solitamente stamperà i valori dei nodi)
  - ricerca di un elemento
  - inserimento
  - cancellazione

Contatti

- **Ufficio:** Torre Archimede, ufficio 6CD3
- **Ricevimento:** ~~Venerdì 9:00-11:00;~~ fino alla fine delle lezioni: Giovedì 8:30-10:30 *(inviare cmq e-mail per conferma)*

✉ lamberto.ballan@unipd.it
🏠 http://www.lambertoballan.net
🏠 http://vimp.math.unipd.it
@ twitter.com/lambertoballan