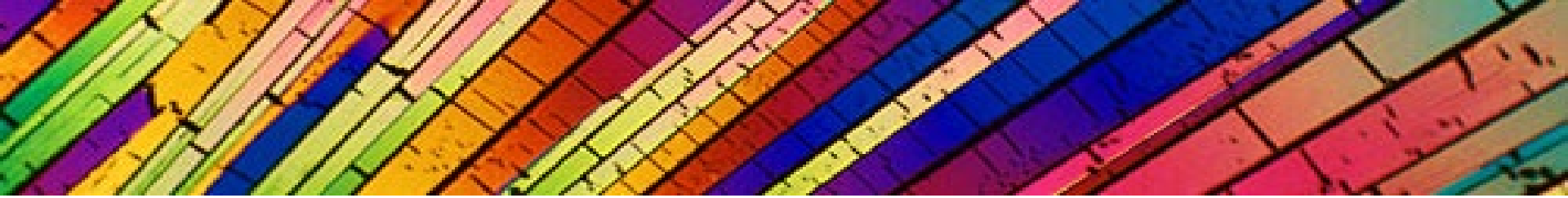


Programmi composti da più file

Ver. 3

© 2021 - Claudio Fornaro - Corso di programmazione in C (C89)

Suddivisione in più file

- Il codice di un programma può essere suddiviso su più file `.c`
- Un solo file `.c` conterrà il `main`, gli altri conterranno definizioni di funzioni "accessorie", richiamate direttamente o indirettamente da esso
- Ogni file è compilato separatamente e produce un file oggetto, il linker assembla gli oggetti e le librerie per costituire un unico file eseguibile
- Quando si usa un'IDE, l'insieme dei file sorgenti viene spesso chiamato ***progetto***

Suddivisione in più file

- Ciascuno dei file "accessori" si comporta come una libreria di funzioni: vi si collocano funzioni e strutture dati che insieme realizzano una certa funzionalità, ad esempio la gestione di uno stack
- Un file (o un insieme di file) che fornisce una determinata funzionalità può essere facilmente riutilizzato in altri programmi: basta includerlo nel progetto; ma la soluzione più corretta è trasformarlo in una libreria (la modalità dipende dal compilatore, spesso bastano opportune opzioni di compilazione)

Suddivisione in più file

- Quando si crea un file `.c` "accessorio" con funzioni che verranno usate da altri sorgenti e variabili condivise, è utile creare un header file `.h` con lo stesso nome contenente i prototipi di tutte le funzioni (`extern` è opzionale), tutte le `#define` con simboli da condividere e la dichiarazione delle variabili condivise precedute da `extern`
- I file `.c` "accessori" non dovranno contenere `#define` con simboli da condividere perché includeranno il loro stesso header file

Suddivisione in più file

- Se si vogliono utilizzare le funzioni del file "accessorio" in un altro progetto, lo si includerà nel progetto perché sia "linkato" con gli altri file, inoltre tutti i file `.c` che ne usano le funzioni dovranno includere l'header file (`.h`) corrispondente con le virgolette, non le <>:
  `#include "mioheader.h"`
- I nomi di file degli include possono contenere un percorso (sconsigliabile), non essendo letterali stringa eventuali caratteri '\' non serve siano indicati doppi:
  `#include "..\mioheader.h"`

Suddivisione in più file

- È utile che ogni file `.c` "accessorio" includa il suo stesso file `.h` in modo che il compilatore e il linker verifichino la coerenza tra la *definizione* delle variabili nel file `.c` e le *dichiarazioni* `extern` nel file `.h` che tutti i file includeranno
- Bisogna proteggere i file `.h` dall'inclusione multipla (vedere slide sul preprocessore), dichiarare più volte gli stessi identificatori `extern` non è un problema, ma lo è ad esempio ridefinire simboli definiti con `#define`: i simboli non sono esportabili con `extern`, devono essere definiti in ogni file

Suddivisione in più file

- Quando un file `.c` viene compilato ("accessoiro" o principale), inizialmente viene eseguito il preprocessore che include i file `.h` ed esegue le sostituzioni di macro (`#define`), questo file intermedio viene chiamato ***translation unit***

include e define condivisi

- Ciascun file accessorio ha bisogno delle sole direttive `#include` e `#define` e delle librerie necessarie al codice di quel file, ma eventuali elementi in più non creano problemi
- Per usare una funzione definita in un altro file (purché non `static`), è necessario indicarne il prototipo (`extern` è opzionale), eventualmente in un header file

Linkage di variabili e funzioni

- Lo scope di una variabile o di una funzione è la parte di codice all'interno di un unica translation unit (file) dove è visibile e usabile
- Quando si ha a che fare con più file, entra il gioco anche il concetto di *linkage*, nome che richiama il link, ossia quella parte del processo di creazione dell'eseguibile che collega ("to link") più translation units
- Il linkage determina quali dichiarazioni e definizioni in scope differenti (file o moduli) si riferiscono alla stessa entità
- Lo scope è gestito dal compilatore, il linkage dal linker

Linkage di variabili e funzioni

- Un identificatore ha ***linkage esterno*** se è visibile in altre translation units (es. variabili esterne e funzioni purché non `static`)
- Un identificatore ha ***linkage interno*** se è visibile solo nella translation unit dove è definito (variabili esterne e funzioni con lo specificatore `static`)
- Un identificatore ***non ha linkage*** se è locale al blocco dove è definito (es. variabili locali senza `extern`, parametri di funzioni, tag, membri, etc.)

Linkage di variabili e funzioni

- Un identificatore con *linkage esterno* è significativo almeno nei primi 6 caratteri case insensitive
- Un identificatore con *linkage interno* è significativo almeno nei primi 31 caratteri case sensitive

Uso di extern con più file

- Una variabile *definita* esternamente alle funzioni in un file ha automaticamente *linkage esterno* se non ha lo specificatore `static`:

```
int x;
```

- Altri file possono ricollegarsi ad essa **dichiarandola** esternamente alle funzioni preceduta da `extern`:

```
extern int x;
```

- Il linker le considererà la stessa variabile

static su identificatori esterni

- Lo storage class specifier `static` su identificatori con *scope di file* (variabili definite esternamente alle funzioni e i nomi stessi delle funzioni) non richiede una duration statica (che c'è già, essendo identificatori esterni), ma *linkage interno*, ossia li rende visibili e utilizzabili solo alle funzioni definite in quello stesso file (successive alla definizione):

```
static int stack;
static int funzione(int x)
{…definizione della funzione…}
```

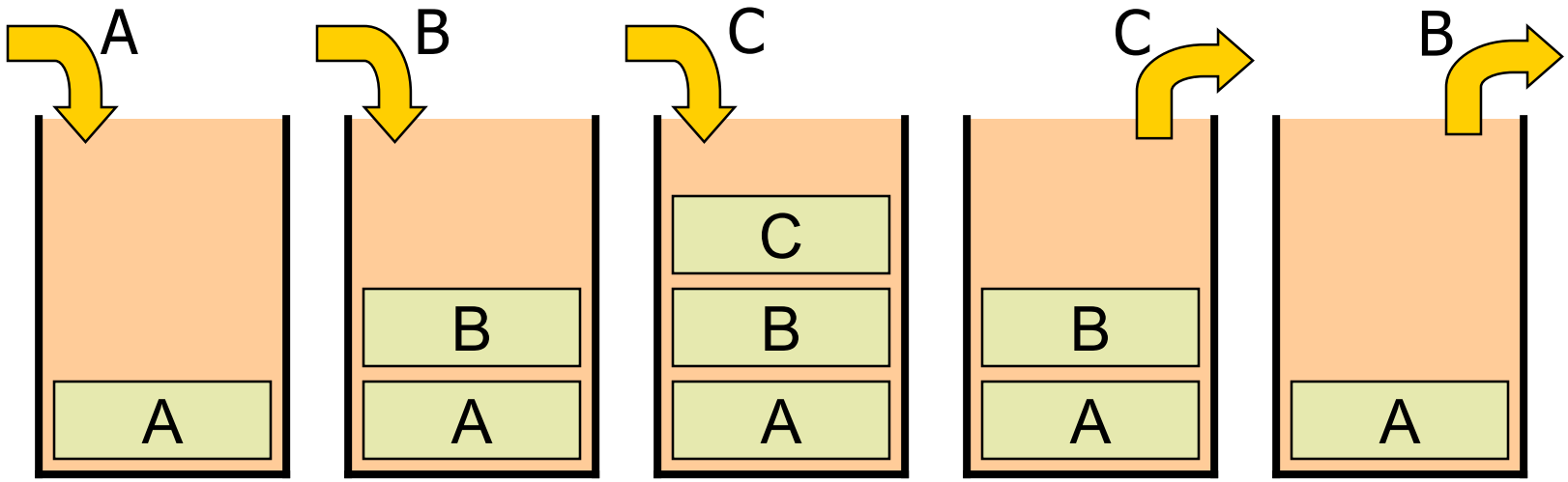
`static` si riferisce al nome della funzione, non al valore restituito

Linkage di variabili con extern

- Una variabile con scope di file o di blocco dichiarata con lo specificatore `extern` ha normalmente *linkage esterno*, salvo che si riferisca a una variabile definita nello stesso file esternamente alle funzioni con specificatore `static`, nel qual caso ha *linkage interno*

Lo stack

- Uno *stack* (o *pila*) è una struttura dati di tipo LIFO (Last In First Out) in cui i valori vengono prelevati nell'ordine inverso a quello di introduzione
- L'inserimento di un dato viene detto `push`
- L'estrazione di un dato viene detta `pop`



Esercizi

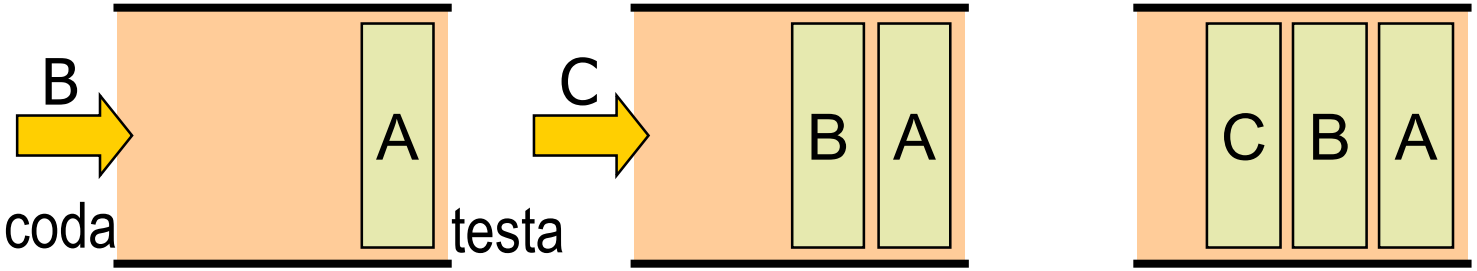
1. Scrivere in un file separato la realizzazione di uno stack basato su un *vettore di interi*. Si realizzino le funzioni `push` e `pop` (con *opportuni parametri*) che restituiscano 1 in caso di errore (stack pieno o vuoto) e 0 altrimenti (non devono fare I/O). Si scriva un main a menu in grado di verificarne il funzionamento. I prototipi siano in un file `.h`.

   Nota. Il vettore deve essere `static` perché solo `push` e `pop` ne abbiano accesso. Si utilizzi un puntatore `p` che punti alla prima locazione libera e si utilizzino le seguenti espressioni:

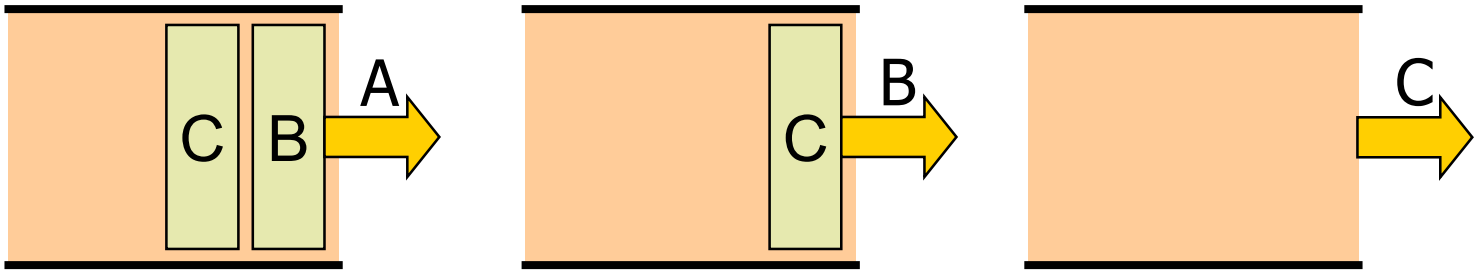
   per push: `*p++ = val;`
   per pop: `val = *--p;`

La coda

- Una coda è una struttura dati di tipo FIFO (First In First Out) in cui i valori vengono prelevati nello stesso ordine di introduzione
- Introduzione in coda: `enqueue`



- Estrazione dalla testa: `dequeue`



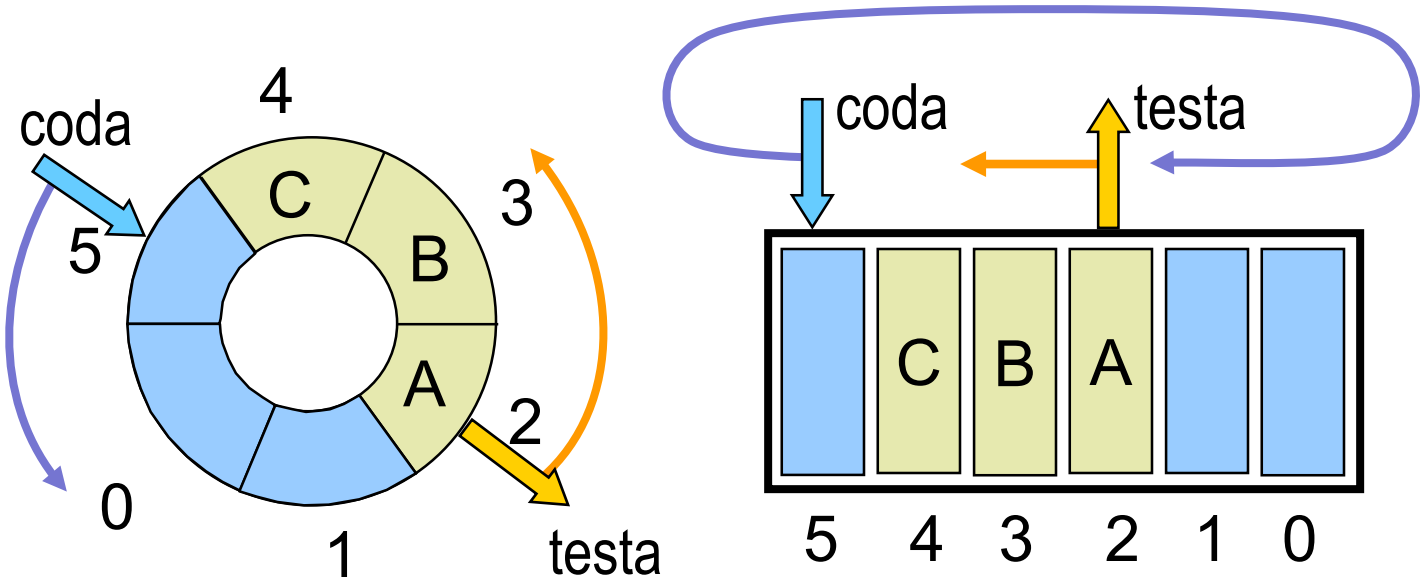
Esercizi

2. Scrivere in un file separato la realizzazione di una coda basata su un *vettore di interi*. Si scrivano le funzioni `enqueue` e `dequeue` (con *opportuni parametri*) che restituiscano 1 in caso di errore (coda piena o vuota) e 0 altrimenti (non devono fare I/O). Si scriva un main a menu in grado di verificarne il funzionamento. I prototipi siano in un file `.h`.

   Nota. Si utilizzino due puntatori: testa punti alla cella con il prossimo valore da prelevare, coda punti alla prossima cella da riempire.
   Il vettore sia `static` come per lo stack.
   Si noti che la `enqueue` è identica alla `push`.

Il buffer circolare (coda)

- Testa e coda non sono più fisse, ma si rincorrono
- È necessario un contatore delle posizioni libere per discriminare la condizione "pieno" da "vuoto": in entrambe testa e coda coincidono



Il buffer circolare (coda)

- Inserimento (in *coda*)

  if (non pieno)
  aggiungi in *coda*
  fa avanzare *coda*
  posti liberi – 1

- Estrazione (dalla *testa*)

  if (non vuoto)
  preleva valore
  fa avanzare *testa*
  posti liberi +1

Esercizio

3. Scrivere in un file separato la realizzazione di una coda come buffer circolare basato su un vettore di interi. Si scrivano le funzioni `enqueue` e `dequeue` con identico tipo di quelle dell'esercizio precedente così da utilizzare lo stesso identico `main`.

   Si noti che è necessario utilizzare un contatore delle posizioni libere (o occupate) per verificare se è possibile inserire/togliere un valore, non c'è modo di stabilirlo confrontando semplicemente i puntatori testa e coda.