Laboratorio di Programmazione 1

Docente: dr. Damiano Macedonio

Lezione 20 – 12/05/2014

Allocazione Dinamica Memoria

- Fino ad ora abbiamo utilizzato una *allocazione statica (automatica)* della memoria.
  - Le variabili vengono allocate automaticamente in memoria (lo *stack*) quando si entra in un blocco di codice (es. una funzione), e corrispondentemente vengono *distrutte* automaticamente quando si esce dal blocco stesso.
  - L'allocazione statica richiede che la dimensione delle variabili sia nota a tempo di compilazione (es. la dimensione degli array deve essere definita con costanti)
- L'*allocazione dinamica* della memoria consente di determinare lo spazio necessario a certe variabili durante l'esecuzione del programma.
  - Una variabile viene allocata dinamicamente in memoria (*heap*) attraverso specifiche istruzioni, e rimane tale finché non viene esplicitamente deallocata.

Funzione `malloc()`

- La funzione `malloc()` della libreria standard `<stdlib.h>` consente di allocare dinamicamente la memoria.

  - Richiede un solo argomento: il numero totale di byte da allocare in memoria.
  - Restituisce un puntatore (indirizzo) all'area di memoria allocata (di tipo `void`). Il puntatore è `NULL` se, per qualche motivo, l'allocazione non è stata possibile.

Operatore `sizeof()`

- L'operatore `sizeof()` restituisce la dimensione in byte dell'elemento specificato.
- L'argomento dell'operatore `sizeof()` può essere
  - Una variabile
    - `int x;`
    - `sizeof(x)`: numero di byte per memorizzare un intero
  - Il nome di un array
    - `int array[10];`
    - `sizeof(array)`: numero di byte per memorizzare 10 interi
  - Il nome di un tipo di dati fondamentale
    - `sizeof(int)`: numero di byte per memorizzare un intero
  - Il nome di un tipo di dati derivato (es. struttura)
    - `sizeof(struct date)`: numero di byte per memorizzare una struttura `date`.
  - Un'espressione.

Operatore `sizeof()`

- Si tratta di un *operatore* e non di una funzione, quindi viene valutato durante la compilazione e sostituito con il risultato del calcolo, a meno che il suo argomento non sia un array di dimensione variabile.

Uso di `sizeof()` in `malloc()`

- L'operatore `sizeof()` è utilizzato per determinare la dimensione degli elementi da riservare con la funzione `malloc()` in modo indipendente dalla macchina.

```
int *int_ptr;
...
int_ptr = (int *) malloc(sizeof(int));
```

- Il cast `(int *)` è necessario poichè la funzione `malloc()` restituisce un puntatore a `void`.

Allocazione Dinamica Memoria

- Se si chiede più memoria di quella disponibile, la funzione `malloc()` restituisce un puntatore `NULL`.

- Esaminare il puntatore restituito consente di verificare che l'operazione sia andata a buon fine.

```
int *int_ptr;
...
int_ptr = (int *) malloc(sizeof(int));
if (int_ptr == NULL) {
   ...
}
```

- Se l'allocazione va a buon fine `int_ptr` punta ad un intero.

Funzione `free()`

- La funzione `free()` permette di restituire la memoria allocata dinamicamente attraverso la funzione `malloc()`.
  - Una buona regola è restituire sempre la memoria allocata dinamicamente quando questa non è più utilizzata, in modo da poterla riutilizzare per successive allocazioni.

- L'argomento della funzione `free()` è un puntatore alla memoria allocata (quello restituito dalla funzione `malloc()`).

```
int_ptr = (int *) malloc(sizeof(int));
...
free(int_ptr);
```

- La funzione `free()` non ritorna nessun risultato.

Esercizio 1

- Si scrivano i file `polinomio.h` e `polinomio.c` che definiscono un polinomio (struttura) a cofficienti interi in una sola variabile e di grado massimo 2.

  Esempio: $3x^2 + 2x - 8$
- Si devono definire e implementare le seguenti funzioni:
- `struct polinomio *construct`

  `(int a, int b, int c)`
  - Costruisce un nuovo polinomio i coefficienti indicati: `a` è il coefficiente del termine di secondo grado, `b` è il coefficiente del termine di primo grado, `c` è il termine noto. Per esempio, per costruire il polinomio dell'esempio si deve poter richiamare la funzione `construct(3, 2, -8)`
- `void destruct_polinomio (struct polinomio *this)`
  - Dealloca il polinomio indicato.

Esercizio 1

- `int grado(struct polinomio *this)`
  - Restituisce il grado reale del polinomio indicato (verifica se ci sono coefficienti uguali a 0);
- `struct polinomio *add`
  `(struct polinomio *this,`
  `struct polinomio *other)`
  - Restituisce un nuovo polinomio che è la somma dei due polinomi indicati;
- `int evaluate(struct polinomio *this, int x)`
  - Restituisce il valore del polinomio indicato valutato nel punto indicato.
- `void toString(struct polinomio *this)`
  - Stampa una stringa che descrive il polinomio indicato, una cosa del tipo 3x^2 + 2x - 8 (i termini con coefficiente 0 non devono essere riportati).

Esercizio 1

```
#include ...
int main() {
  struct polinomio *poly1 = construct(-3, 4, 0);
  struct polinomio *poly2 = construct(5, -11, 2);
  struct polinomio *somma = add(poly1, poly2);

  printf("Polinomio 1: " ); toString(poly1);
  printf(" di grado %i\n", grado(poly1));

  printf("Polinomio 2: "); toString(poly2);
  printf(" di grado %i\n", grado(poly2));

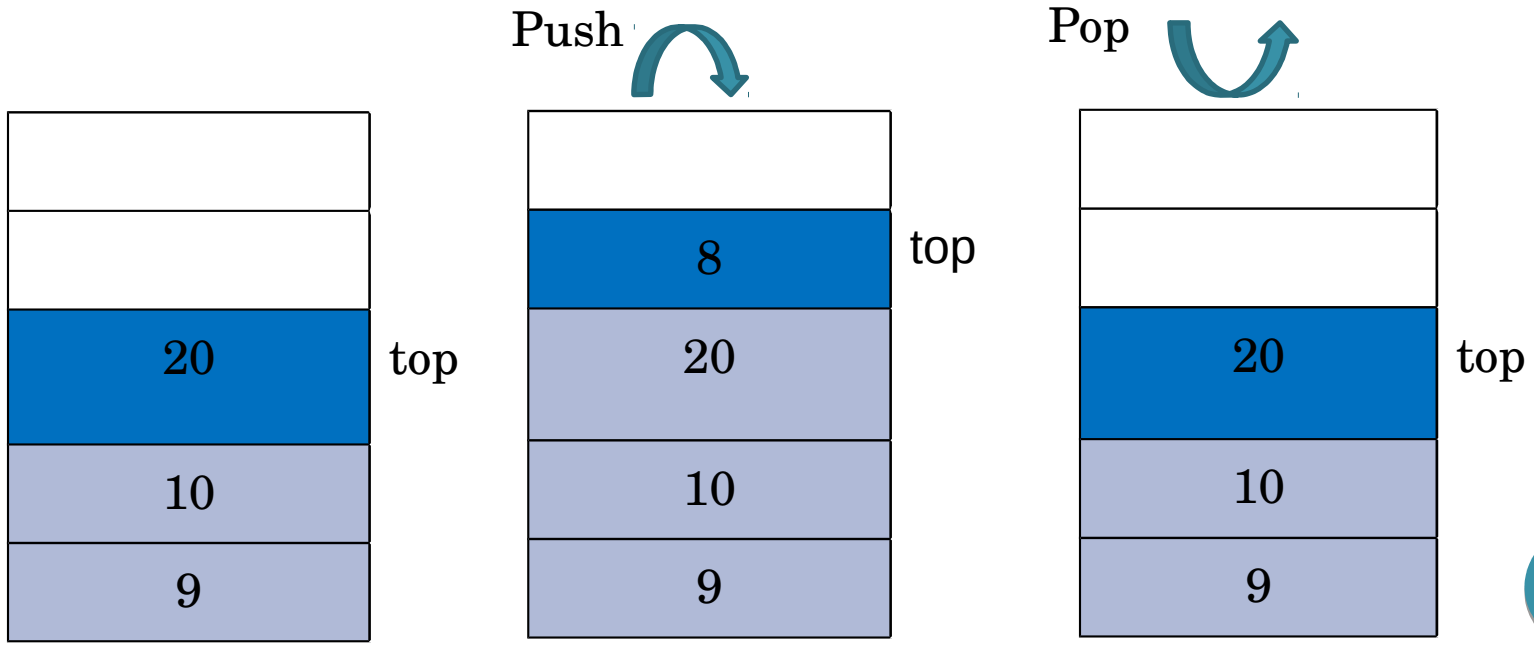
  printf("somma: " ); toString(somma);
  printf(" di grado %d\n", grado(somma));
  printf("la somma valutata in x = 7 vale %d\n", evaluate(somma,7));

  destruct_polinomio(somma);
  destruct_polinomio(poly2);
  destruct_polinomio(poly1);
  return 0;
}
```

Polinomio 1: - 3x^2 + 4x di grado 2
Polinomio 2: + 5x^2 - 11x + 2 di grado 2
somma: + 2x^2 - 7x + 2 di grado 2
la somma valutata in x = 7 vale 51

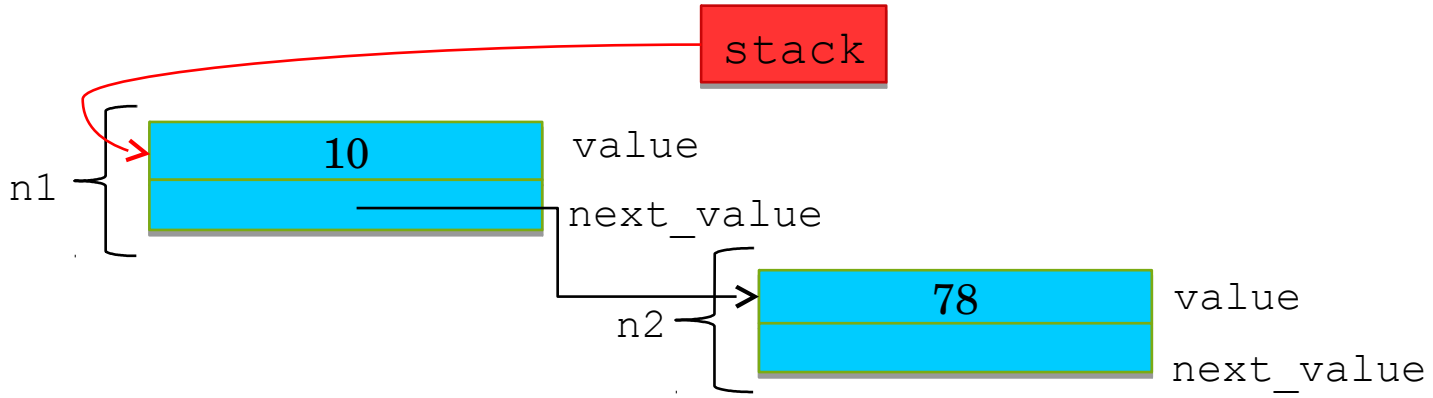
Esercizio 2 – stack

- Uno *stack* o *pila* è una struttura dati nella quale gli elementi vengono gestiti con una modalità di accesso LIFO (Last-In-First-Out).
  - L'ultimo elemento inserito è il primo ad essere rimosso.



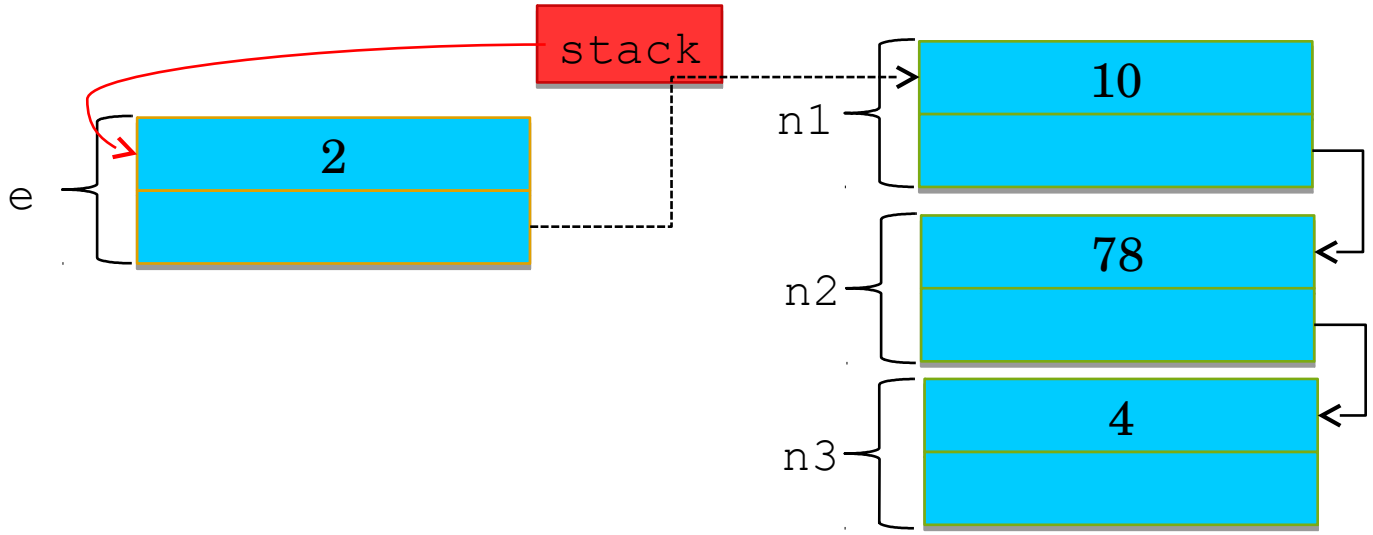
Esercizio 2 – stak con puntatori

- Implementare la struttura dati *stack* o *pila* utilizzando strutture e puntatori:
  - Ciascun elemento della pila è una struttura formata da un valore (intero) e da un puntatore alla struttura rappresentante l'elemento successivo.
  - ```
    struct entry {
      int value;
      struct entry *next_value;
    }
    ```
  - Lo stack complessivo è rappresentato da un puntatore all'elemento in cima allo stack.


    ```

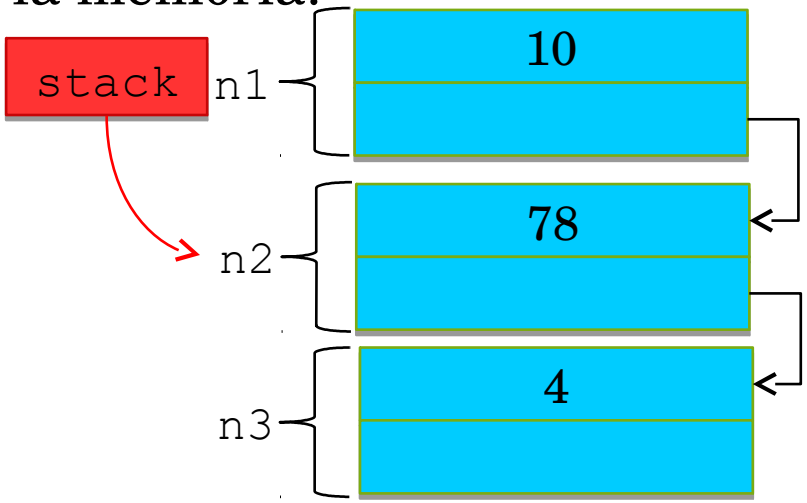
Esercizio 2 - `push`

- Sulla pila sono definite le seguenti funzioni:
  - `struct entry *push(struct entry *s, int value);`
  - Aggiunge un nuovo valore `value` in cima allo stack corrente `stack`. Per aggiungere un nuovo valore si deve crare una nuova `entry` `e` il cui valore è `value` e il cui attributo `next` punta allo stack esistente. Il valore di ritorno è un puntatore allo stack modificato.



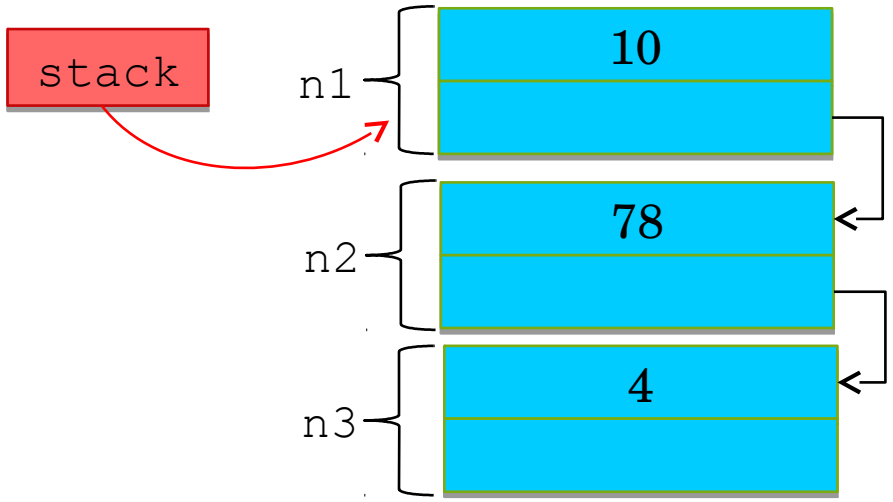
Esercizio 2 - `pop`

- `struct entry *pop(struct entry *stack);`
- Elimina l'elemento in cima allo stack corrente `stack`. Lo stack è rappresentato da un puntatore all'elemento in cima allo stack. Per eliminare un elemento è sufficiente restituire come risultato il secondo elemento in cima allo stack. Ricordatevi di liberare la memoria!



Esercizio 2 – `top`

- `int top(struct entry *stack);`
- Restituisce il valore in cima allo stack corrente `stack`.
- Non modifica lo stack corrente.



Esercizio 2

- La fine dello stack va identificata tramite un puntatore `NULL`:
  - `e.next = NULL;`
  - `if(pointer != NULL) {...}`
- La struttura e le relative funzioni vanno isolate nei file `stack.h` e `stack.c`, che saranno incluse e richiamate nel file `main.c`.
- Finché l'utente vuole continuare, si stampa un menu per scegliere una delle 3 operazioni e si visualizza il risultato. Nel caso delle operazioni di `push` e `pop` il risultato visualizzato è il contenuto dello stack aggiornato.

Esercizio 3

Seconda Prova Parziale 30/05/2010

Un *iteratore* è una struttura dati che fornisce un elemento alla volta da un insieme sorgente di elementi, fino al loro esaurimento.

Si definisca una struttura `iterator` (scrivendo quindi i file `iterator.h` e `iterator.c`) e le seguenti 3 funzioni:

`struct iterator *construct(int arr[],int length)`

Restituisce un nuovo iteratore per i `length` elementi dell'array `arr`.

Usare l'allocazione dinamica per creare la struttura ritornata.

`bool hasNext(struct iterator *this)`

Ritorna `true` se l'iteratore `this` ha ancora elementi su cui iterare.

`int next(struct iterator *this)`

Restituisce il prossimo elemento dell'iteratore `this`.

Esercizio 3

Seconda Prova Parziale 30/05/2010

- Se tutto è corretto, l'esecuzione del seguente programma:

```
#include <stdio.h>
#include "iterator.h"
int main() {
  int arr[] = {13, 4, 56, -11, 4, 25, -89};
  struct iterator *it = construct(arr, 7);
  while( hasNext(it) ){
    printf("%i ", next(it));
  }
    printf("\n");
  return 0;
}
```

- dovrà stampare:

```
13 4 56 -11 4 25 -89
```