Appunti sull'implementazione in C di tipi di dato

Alessandra Raffaetà

Università Ca' Foscari Venezia
Corso di Laurea in Informatica

Programma in un singolo modulo

Inserisce i numeri da 1 a 5 in una pila e poi li stampa

```
/* pilaSenzaModuli.c */
#include <stdio.h>
#include <stdlib.h>

/* Dichiarazione del tipo Pila */
typedef struct node{      /* tipo del nodo della lista */
  int info;
  struct node *next;
} Node;
typedef Node * List;   /* tipo lista per memorizzare gli elementi*/

struct stack{          /* tipo dello stack realizzato con puntatori */
  List contents;
  int size;
};
typedef struct stack * Stack;
```

Programma in un singolo modulo (cont.)

- ... continua

```
/* Function Definitions */

/*post: costruisce uno stack vuoto */
Stack initstack(){
  Stack s;
  s = (Stack) malloc(sizeof(struct stack));
  s->contents = NULL;
  s->size = 0;
  return s;
}

/*post: ritorna 1 se lo stack e' vuoto, 0 altrimenti */
int stackempty(Stack s){

  return s->contents == NULL;
}
```

Programma in un singolo modulo (cont.)

- ... continua

```
/*post: inserisce elem in cima allo stack */
void push(Stack s, int elem){

  List temp;
  temp = (List) malloc(sizeof(Node));
  temp->info = elem;
  temp->next = s->contents;
  s->contents = temp;
  (s->size)++;
}

/*post: ritorna il numero di elementi nello stack */
int size(Stack s){
  return s->size;
}
```

Programma in un singolo modulo (cont.)

- ... continua

```
/*pre: stack non vuoto */
/*post: ritorna e rimuove l'elemento in cima allo stack */
int pop(Stack s){
  int ris;
  List temp;
  temp = (s->contents);
  s->contents = temp->next;
  ris = temp->info;
  free(temp);
  (s->size)--;
  return ris;
}
/*pre: stack non vuoto */
/*post: ritorna l'elemento in cima allo stack */
int top(Stack s){
  return s->contents->info;
}
```

Programma in un singolo modulo (cont.)

- ... continua

```
int main(){
  Stack s;
  int i;

  s = initstack();

  for (i = 1; i <= 5; i++)
    push(s, i);

  while (!stackempty(s))
    printf("%d\n", pop(s));

  return EXIT_SUCCESS;
}
```

Programma in un singolo modulo: Problemi

- Realizzare un intero programma come un singolo modulo presenta vari inconvenienti:
  - Ogni (minima) modifica richiede la ricompilazione dell'intero programma
    - tempi di compilazione elevati !!
  - Non è facile riutilizzare funzioni (di utilità generale) definite nel programma (es. **`push, pop …`**).
  - Nota: Per il secondo problema un semplice "cut&paste" delle funzioni è una pessima soluzione per
    - manutenzione: ogni operazione di aggiornamento (es. sostituzione del codice della funzione con uno più efficiente) su ogni copia!
    - efficienza: il cut&paste è lento e ciascuna copia della funzione occupa spazio disco.

Programma in un singolo modulo: Problemi

- Chiameremo programma cliente un programma che usa un tipo di dato e implementazione un programma che specifica il tipo di dato.
- La realizzazione del tipo di dato è visibile ai programmi cliente.
- Non si può modificare la rappresentazione senza dover modificare i programmi cliente.

Suddivisione in più moduli

- Un programma C complesso è normalmente articolato in più file sorgenti
  - compilati indipendentemente
  - quindi collegati in un unico eseguibile
- Risolve i problemi menzionati precedentemente
  - Uno stesso file può essere utilizzato da diversi programmi (funzioni riusabili)
  - La rigenerazione di un eseguibile richiede la ricompilazione dei soli file sorgente modificati ed il linking.

Suddivisione in più moduli (cont.)

- Se un programma è diviso in più moduli sorgente
  - se un file sorgente utilizza una funzione, non definita nello stesso file, deve contenere la dichiarazione del prototipo della funzione
    - Es: per utilizzare **push** occorre dichiarare

```
void push(Stack s, int elem)
```

  - direttive per il pre-processore / definizioni di tipo devono essere presenti in ogni file che le utilizza.
    - Es. (non legate all'esempio)
      - `#define MAX 100`
      - `typedef struct stack * Stack;`

Suddivisione in più moduli (cont.)

- Per rendere un modulo **module.c** facilmente riusabile si predispone un header file (file di intestazione) **module.h** contenente:
  - direttive, definizioni di tipo
  - prototipi delle funzioni definite
- Il file modulo **module.c** include il proprio header

```
#include “modulo.h”
```

- Ogni file che utilizza il modulo ne include l'header

```
#include “modulo.h”
```

Suddivisione in più moduli: Esempio

```
/* stack.h */
    typedef struct stack * Stack;

    Stack initstack();
    int stackempty(Stack s);
    void push(Stack s, int elem);
    int pop(Stack s);
    int top(Stack s);
    int size(Stack s);
```

Implementazione di un Tipo di dato astratto in C

- Utilizzo di un handle (aggancio, maniglia) per descrivere un riferimento a un oggetto astratto.
- Lo handle è definito come puntatore a una struttura che è specificata solo attraverso una etichetta.
- Il programma cliente non può accedere a un membro della struttura per dereferenziazione di un puntatore perché non conosce i nomi dei membri della struttura.
- Tutte le informazioni specifiche per la struttura dati sono incapsulate nell'implementazione.

Suddivisione in più moduli: Esempio

```
/* implPilaPunt.c */

#include <stdlib.h>
#include "stack.h"  /* contiene il tipo e i prototipi per Stack */

    typedef struct node{      /* tipo del nodo della lista */
      int info;
      struct node *next;
    } Node;
    typedef Node * List;   /* tipo lista per memorizzare gli elementi*/

    struct stack{      /* tipo dello stack realizzato con puntatori */
      List contents;
      int size;
    };
```

Suddivisione in più moduli: Esempio

- ... continua

```
/*post: costruisce uno stack vuoto */
Stack initstack(){
  Stack s;
  s = (Stack) malloc(sizeof(struct stack));
  s->contents = NULL;
  s->size = 0;
  return s;
}

/*post: ritorna 1 se lo stack e' vuoto, 0 altrimenti */
int stackempty(Stack s){

  return s->contents == NULL;
}

.........
```

Suddivisione in più moduli: Esempio

```
/* usaPila.c */
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"  /* contiene il tipo e i prototipi per Stack */

    int main(){
      Stack s;
      int i;
      s = initstack();
      for (i = 1; i <= 5; i++)
        push(s, i);
      while (!stackempty(s))
        printf("%d\n", pop(s));

      return EXIT_SUCCESS;
    }
```

Suddivisione in più moduli (cont.)

- Ogni file sorgente può essere compilato separatamente

```
gcc -c modulo.c
```

- La compilazione produce il file oggetto modulo.o che contiene
  - codice macchina
  - tabella dei simboli
- La tabella dei simboli permette di ricombinare (tramite il compilatore **gcc**) il codice macchina con quello di altri moduli oggetto per ottenere file eseguibili.

Compilazione separata: Esempio

- Con l'opzione **-c** ...

```
$ gcc -c implPilaPunt.c
$ gcc -c usaPila.c
```

si generano i file oggetto **implPilaPunt.o** e **usaPila.o**

- Alternativamente, con un solo comando

```
$ gcc -c implPilaPunt.c usaPila.c
```

Compilazione separata: Esempio

- Infine il linking dei file oggetto, risolve i riferimenti incrociati e crea un file eseguibile **outPila**

```
$ gcc implPilaPunt.o usaPila.o -o outPila

$ ./outPila
  5
  4
  3
  2
  1
```

- Se volessimo cambiare **implPilaPunt.c** dovremmo ricompilare solo questo file e poi rifare il linking.