COMPONENTI SOFTWARE

Diverse tipologie di componenti:

- ***Librerie***

  semplici ***collezioni di procedure e funzioni***. Non fanno uso di variabili globali o statiche e non definiscono nuovi tipi (visibili all'esterno)

  Il file header di una libreria contiene quindi *solo dichiarazioni*. Esempio: `math.h`

- ***Tipi di dato astratto (ADT)***

  definiscono nuovi tipi $T_1,...,T_N$ e ***dichiarano le operazioni*** eseguibili su di essi; il programmatore può ***definire variabili (istanze)*** di tipo T

  Il file header contiene *una definizione di tipo* e le dichiarazioni delle operazioni esportate. Esempio: `element.h`, `list.h`, ...

- ***Singole astrazioni di dato***

  costituiscono ***la realizzazione di uno specifico oggetto*** con certe proprietà; i dati necessari a implementare l'oggetto sono ***pre-definiti al suo interno*** sotto forma di ***variabili statiche***

  Non viene definito alcun nuovo tipo: come una libreria, il componente ***dichiara solo le operazioni fornite;*** tali funzioni agiscono solo sui dati interni all'oggetto, non su variabili passate dal chiamante

  Il programmatore ***non definisce alcuna variabile***: si limita a usare ***unica istanza dell'oggetto così com'è***

IL COMPONENTE STACK (pila)

Contenitore di elementi gestito con politica LIFO (***Last-In-First-Out***): il primo elemento entrato è l'ultimo a uscire

Formalmente:

$$\textbf{stack} = \{ \textbf{D}, \ \Im, \ \Pi \}$$

dove:

- **D** (il *dominio base*) può essere qualunque
- $\Im$ (funzioni)= { push, pop, emptyStack }

  push: $D \times stack \rightarrow stack$ (*inserimento*)

  pop: $stack \rightarrow D \times stack$ (*estrazione*)

  emptyStack: $\rightarrow stack$ (*costante stack vuoto*)
- Π (predicati)= { empty, full }

  empty: $stack \rightarrow boolean$ (*test di stack vuoto*)

[full: $stack \rightarrow boolean$ (test di stack pieno)]

Due approcci possibili:

- realizzare un ***tipo di dato astratto stack***
- realizzare ***uno stack come singola astrazione di dato*** (cioè come singolo oggetto)

Per l'implementazione, un ***vettore*** o una ***lista***

I DUE APPROCCI

1) realizzare un ***tipo di dato astratto stack***

2) realizzare uno ***stack come singola astrazione di dato***

Nel primo caso:

- è necessario che il programmatore ***crei espressamente*** uno stack prima di poterlo usare
- è possibile definire ***più stack distinti***
- lo stack su cui si opera figura ***esplicitamente*** fra i parametri delle operazioni

Nel secondo caso:

- ***NON*** è necessario ***creare espressamente*** uno stack, perché il componente è già un (singolo) oggetto stack
- ***NON*** vi è possibilità di avere ***più stack distinti***
- lo specifico stack su cui si opera ***NON figura*** fra i parametri delle operazioni (riferimento implicito all'unico oggetto stack)

DIFFERENZA FONDAMENTALE:

- nel primo caso, il file ***non contiene variabili***, ma definisce (ed esporta) un tipo
- nel secondo caso, il file contiene invece anche le ***variabili globali (statiche)*** che realizzano l'oggetto stack, ma non ha bisogno di definire nessun tipo

1) STACK COME ADT

Articolazione del progetto

Due file per il tipo `element` (`element.h`, `element.c`)

Due file per il tipo **`stack`** (`stack.h`, `stack.c`)

Operazioni

operazione	*descrizione*
push : D × stack → stack	inserisce un elemento nello stack dato (modificando lo stack)
pop : stack → D × stack	estrae (e rimuove) un elemento dallo stack dato (modificando lo stack)
newStack : → stack	crea e restituisce uno stack vuoto
isEmpty : stack → bool	Restituisce *vero* se lo stack dato è vuoto, *falso* altrimenti

OSSERVAZIONE

Idealmente, uno stack ha ampiezza illimitata
→ può essere vuoto, ma non *pieno*

Tuttavia, alcune ***implementazioni*** potrebbero porre ***limiti*** all'effettiva dimensione di uno stack → ulteriore primitiva:

isFull : stack → bool	Restituisce *vero* se lo stack dato è pieno, *falso* altrimenti

In uno stack illimitato, **`full`** restiturà sempre ***falso***

STACK COME ADT

Possibili implementazioni per `stack`

- un vettore + un indice
- una lista

File header nel caso "vettore + indice":

```
#include "element.h"
```

```
typedef struct {
   element val[MAX];
   int sp;
} stack;
```

```
void push(element, stack);
element pop(stack);
boolean isEmpty(stack);
stack newStack(void);
```

File header nel caso "lista":

```
#include "element.h"
#include "list.h"
```

```
typedef list stack;
```

```
void push(element, stack);
element pop(stack);
boolean isEmpty(stack);
stack newStack(void);
```

CORRETTO?

STACK COME ADT

Problema: le funzioni `push` e `pop` *devono modificare lo stack* → ***impossibile passare lo stack per valore***

Occorre passaggio per riferimento

Due scelte:

- in modo ***visibile*** (sconsigliabile)
- in modo ***trasparente***

  A questo fine, occorre **definire il tipo stack come puntatore** (a una struttura o a una lista, secondo i casi)

Nuovo header nel caso "vettore + indice":

```
typedef struct st {
   element val[MAX];
   int sp;
} *stack;
```

Nuovo header nel caso "lista":

```
typedef list *stack;
```

NOTA: in questo modo, `stack` è un puntatore a puntatore

STACK COME ADT

Implementazione nel caso “vettore + indice”:

```
#include <stdio.h>
#include “stack.h”  /* include la typedef */
stack newStack(void){
 stack s = (stack)malloc(sizeof(struct st));
 s->sp = 0;
 return s;
}

void push(element e, stack s) {
   if (!isFull(s)) s->val[s->sp++]=e;
   else perror("Errore: stack pieno”);
}

element pop(stack s) {
   if (!isEmpty(s))
          return s->val[--(s->sp)];
   else perror("Errore: stack vuoto”);
      /* e cosa restituisce ?? */
}

int isEmpty(stack s) {
   return (s->sp)==0;
}

int isFull(stack s) {
   return (s->sp)==MAX;
}
```


STACK COME ADT

Implementazione nel caso “lista”:

```
#include <stdio.h>
#include “stack.h”

stack newStack(void){
  stack s = (stack)malloc(sizeof(list));
  *s = emptyList();
  return s;
}

void push(element e, stack s) {
   if (!isFull(s)) *s= cons(e,*s);
   else perror("Errore: stack pieno”);
}

element pop(stack s) {
   if (!isEmpty(s)) {
      element e = head(*s);  *s = tail(*s);
      return e;
   }
   else perror("Errore: stack vuoto”);
      /* e cosa restituisce ?? */
}

int isEmpty(stack s) { return empty(*s); }

int isFull(stack s) {  return 0;  }
```

NOTA: il predicato **`isEmpty`** del componente stack *non può avere lo stesso nome* della funzione **`empty`** del componente lista

OSSERVAZIONE: che cosa succede se si invoca `pop()` su uno stack vuoto?

e, nel caso dell'implementazione con vettore: che cosa succede se si invoca `push()` su uno stack pieno?

Quale approccio al problema?

- "non dovrebbe succedere"
- una convenzione (quale?)
- un concetto di ***esito dell'operazione*** leggibile tramite un'opportuna primitiva **`check_result`**

Chi deve gestire il problema?

1) ***il chiamante***, facendo controlli prima di invocare una operazione "critica"
2) ***il chiamante e il chiamato***, quest'ultimo cercando di segnalare al chiamante il problema

Che cosa succede dopo?

- il chiamante continua come se niente fosse
  → lavorerà con dati o assunzioni sbagliate
- il chiamante blocca l'intero programma perché è incapace di continuare
- ***il chiamante recupera la situazione*** perché è stato informato dell'accaduto e ***sa come reagire***

→ ***concetto di eccezione e gestione delle eccezioni***

STACK: SINGOLA ASTRAZIONE DI DATO

Articolazione del progetto

Due file per il tipo `element` (`element.h`, `element.c`)

Due file per il tipo **`stack`** (`stackobj.h`, `stackobj.c`)

Il punto cruciale

Esiste *un solo oggetto stack* già definito, sotto forma di variabili statiche, all'interno di `stackobj.c`:

- ***non*** esiste un ***tipo stack***
- ***non c'è alcun parametro stack*** come argomento delle funzioni (riferimento implicito all'unico stack esistente)

File header (`stackobj.h`)

```
#include "element.h"
void push(element);
element pop(void);
int isEmpty(void);
int isFull(void);
```

Possibili implementazioni per `stackobj`

- un vettore + un indice
- una lista

In entrambi i casi, l'implementazione si baserà su ***variabili globali statiche***, *invisibili fuori da* `stackobj.c`

STACK COME SINGOLO OGGETTO

Implementazione nel caso “vettore+indice”:

```
#include <stdio.h>

#include “stackobj.h”

#define MAX 100

static int sp=0;
static element val[MAX];

void push(element e) {
   if (!isFull()) val[sp++]=e;
   else perror("Errore: stack pieno”);
}

element pop(void) {
   if (!isEmpty()) return val[--sp];
   else perror("Errore: stack vuoto");
      /* e cosa restituisce ?? */
}

BOOL isFull(void) {
   return (sp==MAX);
}

BOOL isEmpty(void) {
   return (sp==0);
}
```


STACK COME SINGOLO OGGETTO

Implementazione nel caso “lista”:

```
#include <stdio.h>

#include “stackobj.h”
#include “list.h”

static list l = emptyList();

void push(element e) { l = cons(e,l); }

element pop(void) {
  if (!isEmpty()) {
       element e = head(l);  l = tail(l);
       return e;
  }
  else perror("Errore: stack vuoto\");
       /* e cosa restituisce ?? */
}

BOOL isFull(void) { return FALSE; }

BOOL isEmpty(void) { return isEmtpy(l); }
```

NOTA: la funzione **`isFull`** restituisce sempre **`FALSE`** → efficienza maggiore con ***macro****:*

```
#define isFull(X) (FALSE)
```