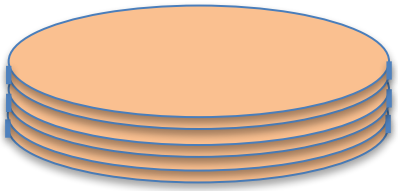
ADT *STACK (PILA)*

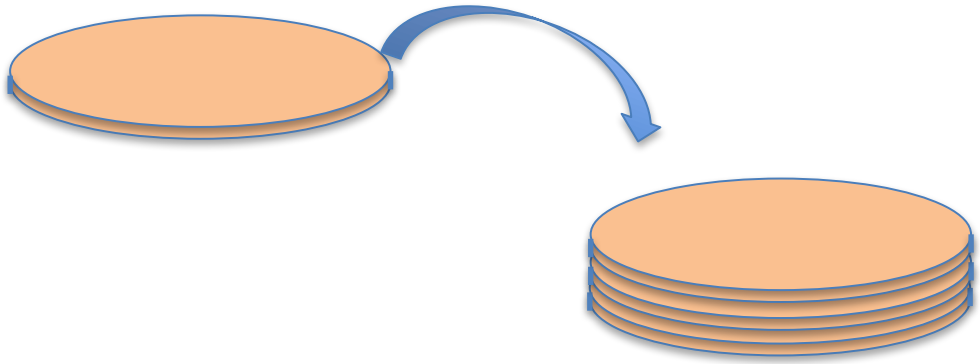
Il tipo di dati astratto *Stack (pila)*

- Una ***pila*** *(spesso chiamata anche* ***stack****)* è una sequenza di elementi di un determinato tipo, in cui è possibile aggiungere o togliere elementi, uno alla volta, esclusivamente da un unico lato (***top*** dello stack).
- Questo significa che la sequenza viene gestita con la modalità detta ***LIFO (Last-in-first-out)*** cioè l'ultimo elemento inserito nella sequenza sarà il primo ad essere eliminato.
- La pila è una struttura dati *lineare*, *omogenea*, a *dimensione variabile* in cui si può accedere direttamente solo al ***primo*** elemento della lista.
- Non è possibile accedere ad un elemento diverso dal primo, cioè quello che è stato inserito per ultimo, **se non dopo** aver eliminato tutti gli elementi che lo precedono (cioè che sono stati inseriti dopo).

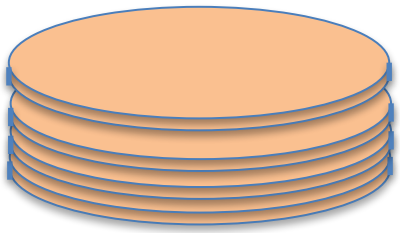
Un esempio di stack: una pila di monete



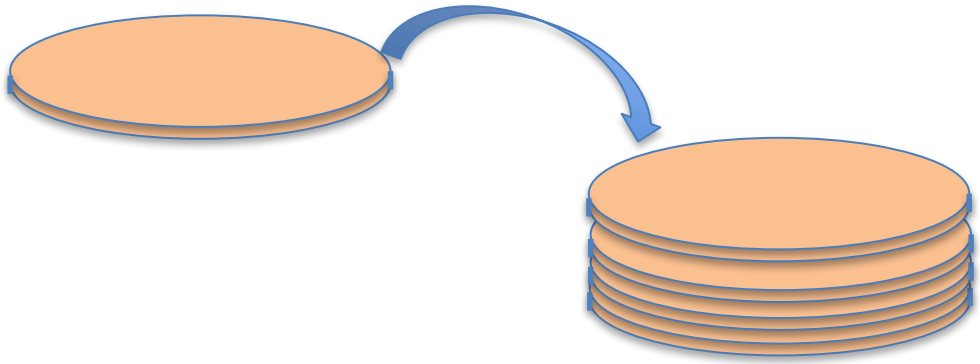
Un esempio di stack: una pila di monete



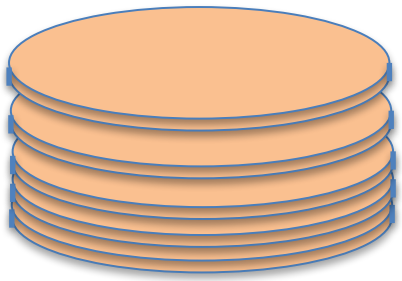
Un esempio di stack: una pila di monete



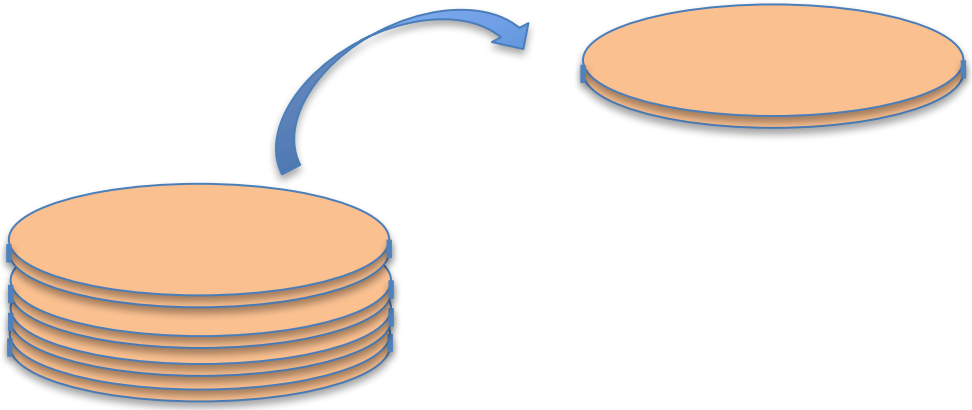
Un esempio di stack: una pila di monete



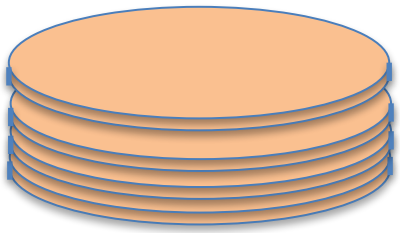
Un esempio di stack: una pila di monete



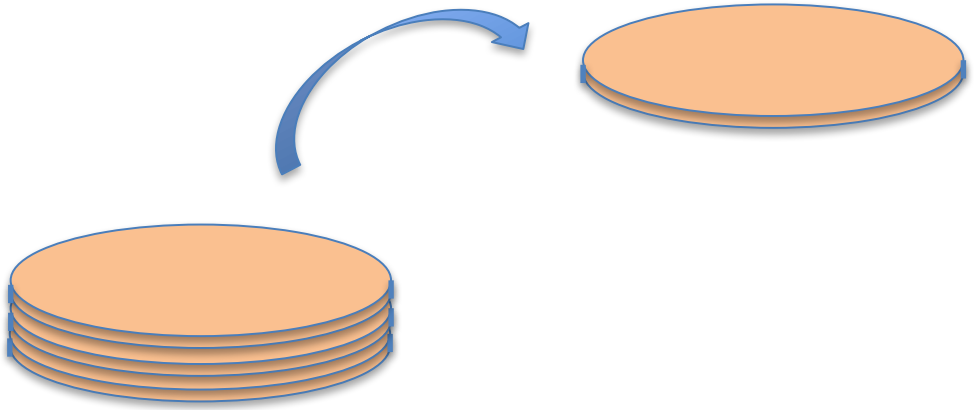
Un esempio di stack: una pila di monete



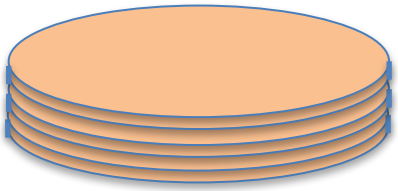
Un esempio di stack: una pila di monete



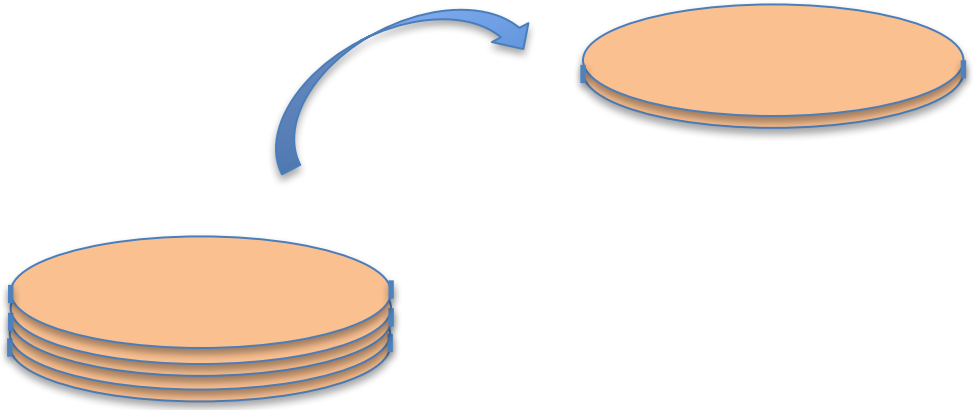
Un esempio di stack: una pila di monete



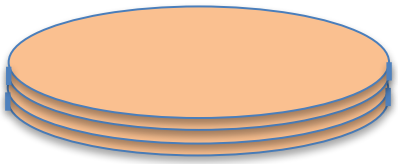
Un esempio di stack: una pila di monete



Un esempio di stack: una pila di monete



Un esempio di stack: una pila di monete



ADT *Stack*: Specifica sintattica

- **Tipo di riferimento**: stack
- **Tipi usati**: item, boolean

- **Operatori**
  - newStack() → stack
  - emptyStack(stack) → boolean
  - push(item, stack) → stack
  - pop(stack) → stack
  - top(stack) → item

ADT *Stack*: Specifica semantica

- **Tipo di riferimento stack**
  - stack è l'insieme delle sequenze *S=a1,a2,…,an* di tipo ***item***
  - L'insieme stack contiene inoltre un elemento ***nil*** che rappresenta la pila vuota (priva di elementi)

ADT *Stack*: Specifica semantica

- **Operatori**
  - newStack() → s
    - Post: s = nil
  - emptyStack(s) → b
    - Post: se s=nil allora b = true altrimenti b = false
  - push(e, s) → s'
    - Post: s = <a1, a2, … an> AND s' = <e, a1, a2, …, an>
  - pop(s) → s'
    - Pre: s = <a1, a2, …, an> n>0
    - Post: s' = <a2, …, an>
  - top(s) → e
    - Pre: s = <a1, a2, …, an> n>0
    - Post: e = a1

Implementare il tipo astratto *Stack*

- Tra le possibili implementazioni, le più usate sono realizzate tramite:
  - **Array**
  - **Lista concatenata**

Implementazione semplice di stack con array

- Lo stack è implementato come un puntatore ad una struct c_stack che contiene due elementi:
  - Un array di MAXSTACK elementi
  - Un intero top che indica la posizione successiva a quella dell'elemento in cima allo stack, e quindi anche il numero di elementi presenti
  - Quando lo stack si riempie, non è più possibile eseguire l'operazione push …

Implementazione di *Stack* con array: header file stack.h

```
// file stack.h

typedef struct c_stack *stack;

// prototipi

stack newStack(void);

int emptyStack(stack s);

int pop(stack s);

int push(item val, stack s);

item top (stack s);
```

L'ADT stack è realizzato in modo da non dipendere dal tipo degli elementi contenuti.

Utilizza il tipo generico item già visto in precedenza

pop e push restituiscono un intero che indica l'esito dell'operazione

file stack.c (versione con uso di array)

```
#include <stdio.h>
#include <stdlib.h>
#include "item.h"
#include "stack.h"
#define MAXSTACK 50


struct c_stack {
    item vet[MAXSTACK];
    int top;
};

stack newStack(void)
{
    stack s;
    s = malloc (sizeof(struct c_stack));
    if (s == NULL) return NULL;
    s->top = 0;
    return s;
}
```

```
int emptyStack(stack s)
{

    return s->top == 0;
}


int push(item val, stack s)
{
    if (s->top == MAXSTACK)
        return 0;
    s->vet[s->top] = val;
    (s->top)++;
     return 1;
}
```

file stack.c (versione con uso di array)

```
int pop(stack s)
{
      if (s->top == 0)
            return 0;
      (s->top)--;
      return 1;
}
```

```
item top(stack s)
{
      item e;
      if(s->top > 0)
            e = s->vet[s->top-1];
      else
            e = NULLITEM;
      return e;
}
```

Esercizio sull'uso di stack: Espressioni aritmetiche con parentesi bilanciate

- **Verificare se una data espressione aritmetica è ben bilanciata rispetto a tre tipi di parentesi: ( ) , [ ] , { }**

  (4 + a) * {[1 – (2/x)] * (8 – a)} è ben bilanciata

  [x – (4y + 3] * (1 – x)) non è ben bilanciata

N.B.: per semplicità supponiamo che non esista un ordine di priorità fra i tre tipi di parentesi

(a + {b -1}) / [b + 2] è ammessa come valida

Parentesi bilanciate: analisi del problema (1 di 2)

- **Vogliamo solo verificare se una data espressione aritmetica è ben bilanciata rispetto alle parentesi, non ci interessa sapere se gli operatori in essa contenuti sono corretti e se hanno il giusto numero di operandi**

- **Possiamo estrarre dall'espressione solo le parentesi, cancellando tutto il resto**

  se l'espressione (4 + a) * {[1 – (2/x)] * (8 – a)} è ben bilanciata, lo valutiamo dalla sua versione semplificata:

  ( ) { [( )] ( ) }

Parentesi bilanciate: analisi del problema (2 di 2)

- **Dati in ingresso:**
  una stringa **exp**

- **Dati in uscita:**
  un valore booleano **ris**

- **Precondizione:**

- **Postcondizione:**
  - se la stringa exp è vuota, allora ris = true
  - se la stringa exp non è vuota, allora ris=true se le parentesi in essa presenti sono bilanciate, ris=false se non lo sono

Parentesi bilanciate: progettazione

Step 1: prendere in input una stringa exp

Step 2: se exp è vuota, dare in output true

Step 3: sia S uno stack di caratteri inizialmente vuoto

Step 4: per ogni carattere car della stringa
se car == ‘(‘ or car==‘[‘ or car==‘{‘
inserisci car in S
se car == ‘)‘ or car==‘]‘ or car==‘}‘
se S è vuoto dare in output false
estrarre un elemento da S e metterlo in t
se car non corrisponde a t dare in output false

Step 5: se S è vuoto dare in output true
altrimenti dare in output false

Parentesi bilanciate: codifica

```
// file: parentesi.c

#include <stdio.h>
#include <stdlib.h>

#include "item.h"
#include "stack.h"

#define MAX_S_size 81

int verifica (char *expr);
int corrisp(char a, char b);

/* vanno aggiunte al modulo item:

char itemtoch (item e);
item chtoitem (char c);

*/
```

```
int main (void)
{
    int ris;
    char expr[MAX_S_size];

   printf ("Inserire una espressione ");
   printf ("senza spazi\n");
   scanf ("%s", expr);

   ris = verifica (expr);

   if (ris)
      printf ("parentesi bilanciate");
  else
     printf ("parentesi sbilanciate");

  return 0;

}
 //   … continua prossimo lucido
```

Parentesi bilanciate: codifica

```
int verifica (char *expr)
{
    stack S = newStack();
    int c, i = 0;
    item top_el;

    if (!expr)   return 1;

    while (expr[i] != '\0')
    {
       c = expr[i];
       if (c=='(' || c=='['  ||  c=='{')
           push (chtoitem(c), S);
       if (c==')' || c==']'  ||  c=='}')
       {
           if (emptyStack(S))
               return 0;
           top_el = top(S);
           pop(S);
           if (!corrisp(itemtoch(top_el), c))
               return 0;
       }
       i++;
    }
    if (emptyStack(S))
            return 1;
    else
            return 0;
}


int corrisp(char a, char b)
{
     if (a=='(' && b==')')
         return 1;
     if (a=='{' && b=='}')
         return 1;
     if (a=='[' && b==']')
         return 1;
     return 0;
}
```

ADT *STACK (PILA)*

Altre implementazioni

Esercizio: Implementare lo stack senza dimensione max

- Come facciamo ad evitare che lo stack abbia una capienza massima ?
  - Bisogna usare l'allocazione dinamica della memoria e due costanti
  - La prima STARTSIZE definisce la dimensione iniziale dello stack
  - La seconda ADDSIZE definisce di quanto allargare lo stack nel caso in cui si riempia
  - Questo significa che ci occorre anche una variabile size che ci dica quanti elementi può contenere lo stack in ogni momento

file stack.c (versione senza dimensione max)

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

#define STARTSIZE 50  // dimensione iniziale stack
#define ADDSIZE 20    // dimensione da aggiungere se pieno

struct c_stack {
    item *vet;
    int size;   // serve a mantenere la dimensione corrente
    int top; };

stack newStack(void) {
    stack s = malloc (sizeof(struct c_stack));
    if (s == NULL)
        return NULL;
    s->vet = malloc(STARTSIZE * sizeof(item));
    if (s->vet == NULL)
        return NULL
    s->size = STARTSIZE;
    s->top = 0;
    return s;
}
```

file stack.c (versione senza dimensione max)

```
int emptyStack(stack s)
{
    return s->top == 0;
}


int push(item val, stack s)
{
    if (s->top == s->size) {  // necessario il resizing dello stack
        item *tmp = realloc(s->vet, (s->size + ADDSIZE) * sizeof(item));
        if (tmp == NULL)
            return 0;
        s->vet = tmp;
        s->size = s->size + ADDSIZE;
      }

     s->vet[s->top] = val;
     (s->top)++;
     return 1;
}
```

file stack.c (versione senza dimensione max)

```
int pop(stack s)
{
	if (s->top == 0)
		return 0;
	(s->top)--;
	return 1;
}

item top(stack s)
{
	item e;
	if(s->top > 0)
		e = s->vet[s->top];
	else
		e = NULLITEM;
	return e;
}
```

Implementazione dello stack con liste collegate

- Il tipo stack è definito come un puntatore ad una struct che contiene
  - Un intero numelem che indica il numero di elementi dello stack
  - Un puntatore top ad una struct nodo (come per la lista)

file stack.c (versione con lista collegata)

```
#include <stdio.h>
#include <stdlib.h>
#include “item.h”
#include "stack.h"

struct node {
    item value;
    struct node *next;
};

struct c_stack {
    struct node *top;
    int numel;
};
```

```
stack newStack(void)
{
    stack s;
    s = malloc (sizeof(struct c_stack));
    if (s == NULL)
        return NULL;
    s->numel = 0;
    s->top = NULL;
    return s;
}

int emptyStack(stack s)
{
    return s->numel == 0;
}
```

file stack.c (versione con lista collegata)

```
int push(item val, stack s)
{
    struct node *nuovo;
    nuovo = malloc (sizeof(struct node));

    if (nuovo == NULL)
            return 0;

    nuovo->value = val;
    nuovo->next = s->top;
    s->top = nuovo;
    (s->numel)++;

   return 1;
}
```

file stack.c (versione con lista collegata)

```
int pop (stack s)
{
      if (s->numel == 0)
            return 0;

      struct node *temp;
      temp = s->top;
      s->top = s->top->next;
      free(temp);
      (s->numel)--;

      return 1;
}
```

```
item top (stack s)
{
      item e;

      if(s->numel > 0)
            e = s->top->value;
      else
            e = NULLITEM;

      return e;
}
```

Implementazione dello stack basata sull'uso del modulo lista

- Il tipo stack è definito come un puntatore ad una struct che contiene
  - Un elemento top di tipo list
  - Non serve più nemmeno l'intero numelem che indica il numero di elementi dello stack …
  - Anche se abbiamo un solo elemento nella struct, continuiamo a definire il tipo stack come puntatore a struct c_stack per non cambiare la definizione nell'header file …

file stack.c (versione con uso di modulo lista)

```
#include <stdio.h>
#include <stdlib.h>

#include “item.h”
#include "stack.h”
#include “list.h”

struct c_stack {
    list top;
};

stack newStack(void)
{
    stack *s;
    s = malloc (sizeof(struct c_stack));
    if (s == NULL)
        return NULL;

    s->top = newList();
    return s;
}
```

```
int emptyStack(stack s)
{
    return emptyList(s->top);
}

int push(item val, stack s)
{
   return insertList(s->top, 0, val);
}

int pop (stack s)
{
  return removeList(s->top, 0);
}

item top (stack s)
{
   return getFirst(s->top);
}
```

E se volessimo realizzare una sola istanza di stack?

- In questo caso non abbiamo bisogno di definire ed esportare un tipo
- La struttura dati dello stack la manteniamo in variabili globali accessibili solo all'interno del modulo (dichiarazioni static)
- Gli operatori della lista non usano parametri di tipo stack, ma operano sulle variabili globali
- Quello che stiamo realizzando è un singolo oggetto stack
- Vediamo come si fa … implementiamo uno stack basato su array senza dimensione massima …

Implementazione di singola istanza di *Stack* con array

```
// file stack.h

// prototipi

int newStack(void);

int emptyStack(void);

int pop(void);

int push(item val);

item top (void);
```

Adesso non c'è la definizione del tipo stack nell'header file

Gli operatori non hanno parametri di tipo stack

Gli operatori newStack, pop e push restituiscono un intero che indica il buon esito dell'operazione

file stack.c (versione senza dimensione max)

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

#define STARTSIZE 50  // dimensione iniziale stack
#define ADDSIZE 20    // dimensione da aggiungere se pieno

static struct c_stack {
    item *vet;
    int size;       // serve a mantenere la dimensione corrente
    int top; } *s;  // variabile statica usata dagli operatori

int newStack(void) {
    s = malloc (sizeof(struct c_stack));
    if (s == NULL)
        return 0;
    s->vet = malloc(STARTSIZE * sizeof(item));
    if (s->vet == NULL)
        return 0;
    s->size = STARTSIZE;
    s->top = 0;
    return 1;
}
```

file stack.c (versione senza dimensione max)

```
int emptyStack(void)
{
    return s->top == 0;
}


int push(item val)
{
    if (s->top == s->size) {  // necessario il resizing dello stack
        item *tmp = realloc(s->vet, (s->size + ADDSIZE) * sizeof(item));
        if (tmp == NULL)
            return 0;
        s->vet = tmp;
        s->size = s->size + ADDSIZE;
      }

     s->vet[top] = val;
     (s->top)++;
     return 1;
}
```

file stack.c (versione senza dimensione max)

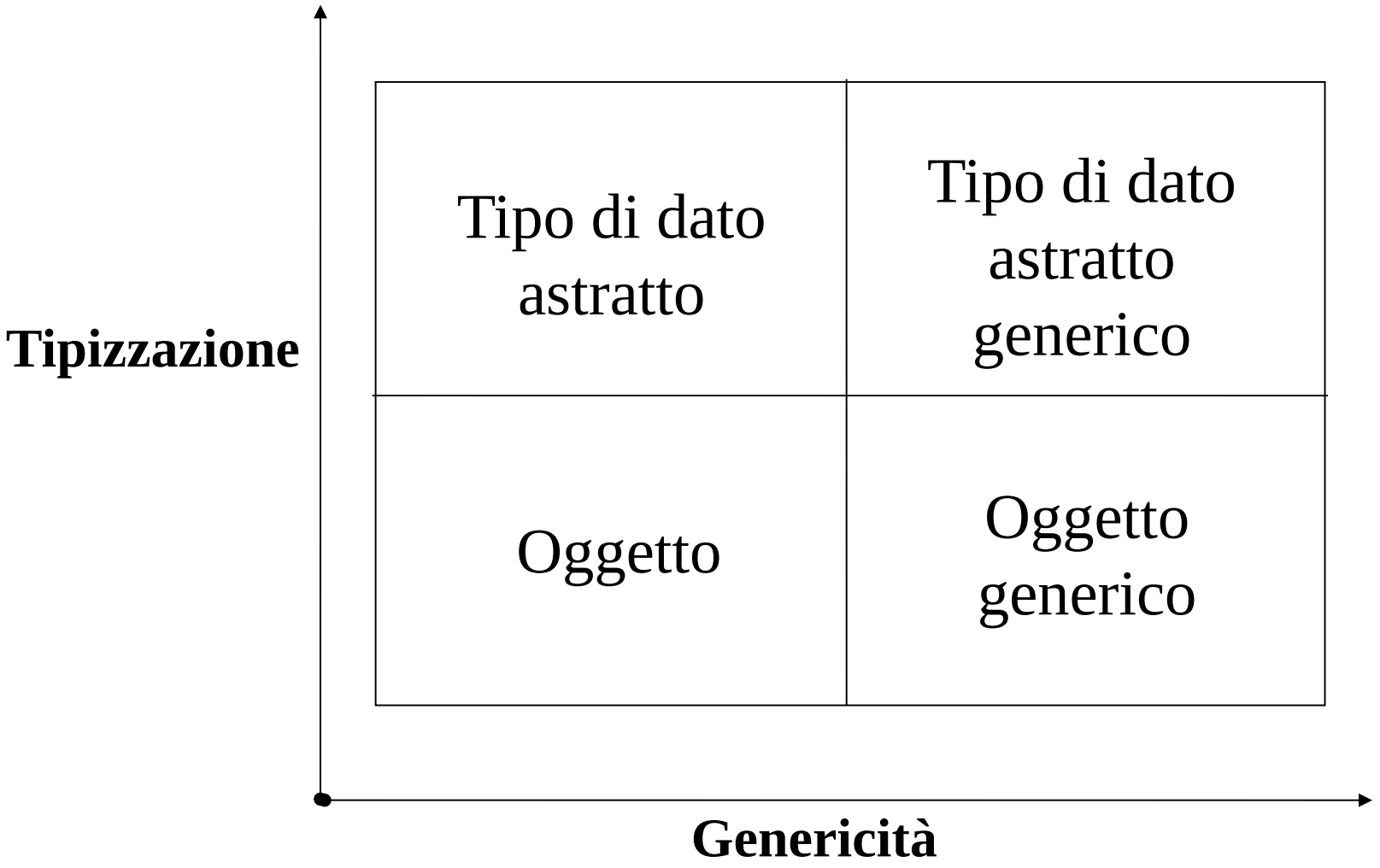
```
int pop(void)
{
	if (s->top == 0)
		return 0;
	(s->top)--;
	return 1;
}

item top(void)
{
	item e;
	if(s->top > 0)
		e = s->vet[(s->top) - 1];
	else
		e = NULLITEM;
	return e;
}
```

NB: il corpo dei metodi non è cambiato in maniera sostanziale, solo che stavolta la variabile s non è un parametro di tipo stack degli operatori, ma una variabile globale static

Moduli e Astrazioni sui dati: Tipi di dati astratti e oggetti

- **Tipo di dato astratto**
  - Il modulo incapsula la definizione del tipo e gli operatori ed esporta il ***nome*** del tipo e la ***signature*** degli operatori
  - Il tipo di riferimento compare tra i parametri degli operatori
  - È consentito definire variabili di tale tipo (***istanziazione***) e utilizzarle (***referenziazione***) come parametri degli operatori
- **Oggetto (astratto)**
  - Il modulo incapsula una struttura dati (istanza) e gli operatori ed esporta la ***signature*** degli operatori
  - Non è consentito referenziare la struttura dati fuori dal modulo
  - L'uso e la manipolazione dell'oggetto consentiti unicamente attraverso i suoi operatori
  - Un oggetto ha uno stato che può cambiare in seguito all'applicazione di determinate operazioni

Moduli e Astrazioni sui dati



Moduli e astrazioni sui dati: genericità

- ***Modulo generico***: un *template* dal quale è possibile istanziare più moduli
- Tipicamente: parametrico rispetto a un tipo base o al numero di componenti di tipo base (*parametri di struttura*)
  - Esempio: l'ADT (o oggetto) stack generico
- Un modulo cliente dovrebbe prima istanziare il modulo specificando i parametri di struttura e poi …

- In C non abbiamo costrutti per fare questo ….
  - Invece dobbiamo definire un tipo generico item e delle costanti, che possiamo cambiare all'occorrenza

Esercizio su

Libretto Universitario

Problema

- Si implementi, mediante l'uso di opportune strutture dati, un programma per la gestione dei libretti universitari degli studenti.
- Specificare e implementare l'ADT libretto: ogni libretto tiene traccia dei dati dello studente (cognome, nome, matricola) e degli esami sostenuti. Questi ultimi sono caratterizzati da nome, voto e data dell'esame. L'ADT libretto dovrà consentire di aggiungere esami al libretto e di ricercare un esame in base al nome.
- Realizzare il programma di test

Libretto: Specifica sintattica

- **Tipo di riferimento**: libretto
- **Tipi usati**: lista, item, int, string

- **Operatori**
  - newLibretto(int, string, string) → libretto
  - addEsame(libretto, item) → libretto
  - dammiEsami(libretto) → lista
  - cercaEsame(libretto, string) → item

libretto: Specifica semantica

- **Tipo di riferimento libretto**
  - Libretto è l'insieme delle quadruple
    L = (mat, cogn, nom, lis)
    in cui:
    - ***mat* è un numero intero**
    - ***cogn* è una stringa**
    - ***nom* è una stringa**
    - ***lis* è una lista**

libretto: Specifica semantica

- **Operatori**
  - newLibretto(mat, cogn, nom) → lib
    - Post: lib = (mat, cogn, nom, nil)
  - addEsame(lib, es) → lib'
    - Post: se lib= (mat, cogn, nom, lis) e lis = <a1, a2, … an>
      allora lib' = (mat, cogn, nom, lis') con lis'=<es, a1, a2, … an>
  - dammiEsame(lib) → l
    - Post: se lib= (mat, cogn, nom, lis)
      allora l= lis
  - cercaEsame(lib, nom_es) → a
    - Post: se lib= (mat, cogn, nom, lis)
      se lis contiene un item a' il cui nome è nom_es
      allora a = a'
      altrimenti a = NULLITEM