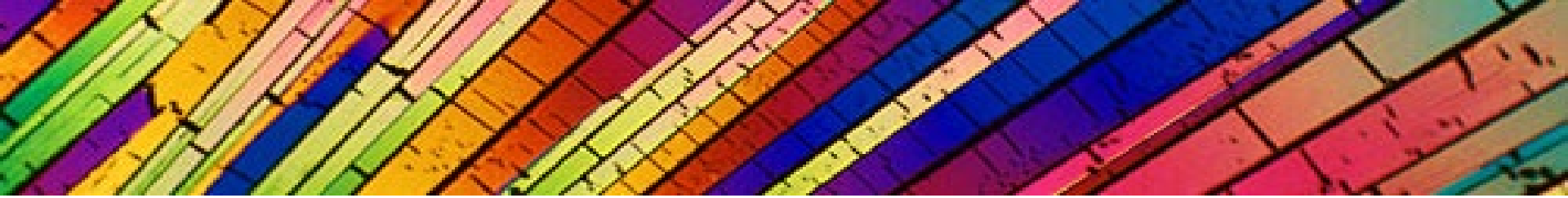

# Abstact Data Type
## (introduzione)

Ver. 3

# Abstact Data Type

- Un ADT definisce un concetto astratto e il suo comportamento
- Viene utilizzato come una "scatola nera" (*oggetto*) di cui il programma chiamante conosce solo che cosa fa e non come
- Ha funzioni (dette *metodi* nella programmazione a oggetti) che possono essere chiamate per far uso dell'ADT
- Il contenuto dell'ADT dovrebbe essere inaccessibile ai programmi che ne fanno uso

# Pubblico e privato

- Le parti **pubbliche** dell'ADT sono visibili e accessibili da parte di tutte le funzioni del programma (linkage esterno)
- I metodi per usare l'ADT sono pubblici
- Le parti **private** dell'ADT sono accessibili solo dalle funzioni definite nello stesso file dove è definito l'ADT (linkage interno al file di implementazione)
- Le strutture dati e le funzioni di supporto ai metodi sono *private:* non sono accessibili dall'esterno dell'ADT

# Struttura di un ADT

- Un ADT è costituito da:
  - un'*implementazione:* uno o più file che contengono il codice che realizza le funzioni/metodi (pubbliche e private) e le strutture dati interne private
  - un'*interfaccia:* file che descrive le funzioni pubbliche (metodi) che il resto del programma può chiamare per utilizzare l'ADT

# Sostituibilità

- Se ADT diversi risolvono lo stesso scopo (es. gestiscono uno stack) e hanno la stessa interfaccia (es. le funzioni `push` e `pop` con lo stesso prototipo), possono essere utilizzati uno in sostituzione dell'altro
- La scelta di utilizzare un'implementazione piuttosto che un'altra per risolvere un problema è dettata dalle migliori caratteristiche di una di esse in relazione al problema specifico (ad es. un'implementazione potrebbe essere più veloce, ma meno capiente; un'altra più lenta ma più capiente)

# Esempio
## Specifiche richieste

- Si vuole definire il tipo di dato **Counter**
- Deve poter essere:
  - visualizzato
  - azzerato
  - incrementato di 1
- I metodi forniti devono essere:
  - `getCounter()`
  - `resetCounter()`
  - `incCounter()`

# Esempio – 1a soluzione
## Scelta della struttura dati

- Serve una variabile intera da incrementare: la struttura dati che conterrà il contatore sarà in questo esempio di tipo `int`:

```
typedef int Counter;
```

# Esempio – 1a soluzione
## counter.h – Interfaccia

- L'interfaccia (file `counter.h`) contiene le informazioni pubbliche: il tipo di dato e i prototipi dei metodi
- ```
  typedef int Counter;
  int getCounter(Counter c);
  void resetCounter(Counter *c);
  void incCounter(Counter *c);
  ```
- Per modificare il parametro `c`, questo deve essere passato tramite puntatore
  ```

# Esempio – 1a soluzione
## counter.c – Implementazione

```
typedef int Counter;
int getCounter(Counter c)
{
   return c;
}
void resetCounter(Counter *c)
{
   *c = 0;
}
void incCounter(Counter *c)
{
   (*c)++;
}
```

# Esempio – 1a soluzione
## main()

```
#include <stdio.h>
#include "counter.h"
main()
{
   Counter x;

   resetCounter(&x);
   incCounter(&x);
   printf("Valore: %d\n",
                    getCounter(x));
   printf("Valore: %d\n", x);
   return 0;
}
```

# Esempio – 1a soluzione
## Problema

- La definizione di Counter è pubblica, chiunque può accedere al valore di una variabile di tipo Counter anche senza tramite delle funzioni (ad es. la seconda `printf`)
- Se nel programma si accede direttamente al valore interno di Counter, viene meno la sostituibilità e non si può sostituire con un altro ADT implementato in un altro modo
- In sostanza: non è Abstract, non è un ADT
- Funziona, ma solo grazie all'autodisciplina del programmatore

# Richiamo sulla typdef

- Si può definire (ma non deferenziare) una ***variabile puntatore*** ad una `struct` *`TAG`* (Tag non omesso) che sarà dichiarata successivamente o in un altro file
- Si può dichiarare un ***tipo puntatore*** (con `typedef`) ad una `struct` *`TAG`* che sarà dichiarata successivamente o in un altro file
- *File 1:*

```
struct TAG
{ int t; };
```

- *File 2:*

```
typedef struct TAG *p;
```

Questo secondo file non conosce i dettagli interni di `struct TAG`

# Struttura di un ADT in C

- L'**interfaccia** viene realizzata da un header file (`.h`) contenente:
  - la definizione del nome dell'ADT
  - i prototipi dei metodi pubblici dell'ADT
- L'interfaccia viene inclusa con `#include` (con le virgolette) in tutti i file che fanno uso dell'ADT e dai file di implementazione dell'ADT stesso

# Struttura di un ADT in C

- L' **implementazione** viene realizzata da uno o più file (`.c`) contenenti:
  - l' `#include` dell'interfaccia (il file `.h`)
  - strutture di immagazzinamento dei dati (private)
  - funzioni di creazione di oggetti di quel tipo (*costruttori*, pubblici)
  - *funzioni di accesso* per leggere i dati (pubbliche)
  - *funzioni di manipolazione* per modificare i dati (pubbliche)
  - funzioni di rimozione di oggetti di quel tipo (*distruttori*, pubblici)
  - altre funzioni e variabili di supporto (private)

# Esempio – 2a soluzione
## Scelta della struttura dati

- Serve una variabile intera da incrementare: la struttura dati che conterrà il contatore sarà di tipo `int`
- Per evitare che sia accessibile senza il tramite delle funzioni, la si "nasconde" in una `struct` locale dichiarata dentro il file **`counter.c`**:

```
struct CounterInnerType {int x;};
```

# Esempio – 2a soluzione
## Scelta della struttura dati

- Il tipo Counter deve essere noto a tutte le funzioni in tutti i file, quindi viene definito in **`counter.h`** che viene incluso in tutti i file `.c`
- Perché una funzione possa modificare i valori dell'oggetto **Counter** (ossia modificare il membro nascosto `x`) bisogna passarle il puntatore all'oggetto stesso, definiamo il tipo Counter già come puntatore:

```
typedef
struct CounterInnerType *Counter;
```

- Si accederà a ciascun oggetto di tipo Counter mediante questo puntatore detto ***handler***

# Esempio – 2a soluzione
## I metodi

- L'accesso all'ADT avviene sempre attraverso il corrispondente **handler**:
  - i **costruttori** costruiscono un oggetto (o *istanza*) e ne restituiscono l'**handler**
  - i **distruttori** distruggono un oggetto, quello di cui viene passato l'**handler**
  - i **metodi di accesso** accedono tramite l'**handler** all'oggetto, non lo modificano
  - i **metodi di manipolazione** accedono tramite l'**handler** all'oggetto, lo modificano

# Esempio – 2a soluzione
## counter.h

```
/* Tipo Counter */
typedef struct CounterInnerType *Counter;
/* Costruttore di un Counter = 0 */
Counter newCounter(void);
/* Costruttore di un Counter con val */
Counter newInitializedCounter(int initialValue);
/* Restituisce il valore del Counter */
int getCounter(Counter c);
/* Restituisce il numero di Counter */
int howmanyCounter(void);
/* Azzera un Counter */
void resetCounter(Counter c);
/* Incrementa un Counter */
void incCounter(Counter c);
/* Distruttore di un Counter */
void deleteCounter(Counter c);
```

# Esempio – 2a soluzione
## counter.c

```
#include <stdlib.h>
#include "counter.h" /*prototipi e tipo Counter*/

/* Struttura su cui si basa Counter */
   /* è definito qui e non nell'header perche' non
      sia visibile fuori dal file */
struct CounterInnerType
{
    int x;
};

/* Tipo Counter, così come visto dal programma */
/* deve essere visibile pubblicamente quindi
     viene definito nell'header e non qui */
/* typedef struct CounterInnerType *Counter; */
```

# Esempio – 2a soluzione
## counter.c

```
/*****************************/
/* VARIABILI PRIVATE LOCALI */
/*****************************/

   /* Variabile (globale) che tiene conto di quanti
      Counter sono stati creati. Deve essere privata
      dell'ADT, quindi static */
static int numCounters = 0;
```

# Esempio – 2a soluzione
## counter.c

```
/*****************************/
/* FUNZIONI PRIVATE LOCALI */
/*****************************/


/* Imposta il valore di un Counter al valore dato,
   e' una funzione di supporto: non deve essere
   utilizzata se non dai metodi di questo ADT,
   quindi static per renderla locale,
   inoltre non e' nell'interfaccia .h */
static void setCounter(Counter c, int value)
{
    c->x = value;
}
```

# Esempio – 2a soluzione
## counter.c

```
/******************/
/* COSTRUTTORI 1 */
/******************/

/* Costruttore di un Counter con val iniz 0 */
Counter newCounter(void)
{
   Counter c;

   c = (Counter)malloc(sizeof(CounterInnerType));
   resetCounter(c);
   numCounters++;
   return c;
}
```

# Esempio – 2a soluzione
## counter.c

```
/******************/
/* COSTRUTTORI 2 */
/******************/


/* Costruttore di un Counter con il valore
   iniziale indicato */
Counter newInitializedCounter(int initialValue)
{
   Counter c = newCounter();
   setCounter(c, initialValue); /* chiamata ad una
                                   funzione locale */
   return c;
}
```

# Esempio – 2a soluzione
## counter.c

```
/**********************/
/* METODI DI ACCESSO */
/**********************/

/* Restituisce il valore del Counter indicato */
int getCounter(Counter c)
{
   return c->x;
}

/* Restituisce il numero di Counter creati */
int howmanyCounter(void)
{
   return numCounters;
}
```

# Esempio – 2a soluzione

## counter.c

```
/*****************************/
/* METODI DI MANIPOLAZIONE */
/*****************************/

/* Azzera un Counter */
void resetCounter(Counter c)
{
   c->x = 0;
}

/* Incrementa un Counter */
void incCounter(Counter c)
{
   (c->x)++;
}
```

# Esempio – 2a soluzione
## counter.c

```
/****************/
/* DISTRUTTORI */
/****************/

/* Distruttore di un Counter */
void deleteCounter(Counter c)
{
   free(c);
   numCounters--;
}
```

# Esempio – 2a soluzione
## main.c

```
#include<stdio.h>
#include "counter.h"
main()
{  int v;
   Counter x = newCounter();
   Counter y = newInitializedCounter(10);
   v = getCounter(x);
   v = howmanyCounter();
   incCounter(x);
   v = getCounter(x);
   resetCounter(x);
   v = getCounter(x);
   deleteCounter(y);
   v = howmanyCounter();
   deleteCounter(x);
}
```

# Esercizi

1. Si realizzi un ADT che implementi uno stack con un vettore di interi, il numero di elementi venga passato al costruttore. Stessa interfaccia dell'esercizio visto nelle funzioni, salvo che serve passare anche l'handler allo stack da utilizzare. Il `main` deve poter agire sulle due istanze degli stack a scelta.
2. Si realizzi un ADT che implementi una coda circolare con un vettore di interi, il numero di elementi venga passato al costruttore. Il resto è come per l'esercizio precedente.

# Homework 12

Si realizzi un ADT che implementi un tipo di dato denominato `Matrice` contenente una matrice di valori `int`, con metodi per:

- creare una matrice date le dimensioni
- distruggerla dato l'handler
- azzerare una matrice
- inserire un valore in una casella (riga,colonna)
- restituire il valore di una casella
- visualizzare tutta la matrice
- sommare due matrici
- moltiplicare due matrici

Si facciano gli opportuni controlli sui possibili errori

# Homework 13

Si realizzi un ADT (con controlli di errore e documentazione) che realizzi con liste le funzionalità di uno stack e di una pila di `int`.

Si usino opportune `typedef` o `#define` per poter permettere di cambiare facilmente il tipo primitivo `int` in un altro.

Fare il test dell'ADT mediante un `main` che istanzi due o più pile e stack.

# Homework 14

Si continui l'Homework precedente aggiungendo i metodi seguenti.

- `listShowAll` (mostra tutta la lista da head)
- `listMerge` (mette insieme due liste, in modo che la prima contenga anche i valori della seconda, la seconda sarà azzerata; se una delle due è ordinata, la lista risultante deve essere ordinata)
- `listSplit` (divide una lista – ordinata o no – in due liste, viene passato come argomento l'indice dell'elemento che sarà il primo della seconda lista, restituita dal metodo come valore di ritorno)
- `listClearFrom` (elimina la parte della lista che inizia dall'elemento indicato, incluso)

# Homework 14

Inoltre si modifichi la gestione dei cursori in modo che ce ne possa essere più di uno, a tal scopo si aggiungano opportune `typedef` per il tipo e i metodi:

- `listNewCursor` (crea un nuovo cursore)
- `listDeleteCursor` (cancella un cursore)