TIPI DI DATO ASTRATTO

Un *tipo di dato astratto (ADT)* definisce una categoria concettuale con le sue proprietà:

- una ***definizione di tipo***
  - **implica un dominio, D**
- un ***insieme di operazioni ammissibili* su oggetti di quel tipo**
  - **funzioni: *calcolano valori sul dominio D***
  - **predicati: *calcolano proprietà vere o false su D***

ADT – Abstract Data Type

- Specifica un insieme di dati e un insieme di operazioni applicabili a tali dati
- *Astratto* ➔ "funzionalmente" indipendente dalle varie possibili implementazioni
  - Le operazioni applicabili sono l'interfaccia dell'ADT
  - Un ADT può essere implementato in vari modi pur mantenendo la **stessa** interfaccia
- L'interfaccia fornisce un ***costruttore***, che restituisce un ***handle*** astratto, e varie operazioni, rappresentate come funzioni che accettano come argomento tale *handle* astratto
- *handle* è di solito un puntatore a "qualcosa"

ADT – Abstract Data Type

- ADT è rappresentato da un'interfaccia, che nasconde l'implementazione corrispondente
- Gli utenti di un ADT utilizzano solo l'interfaccia ma non accedono (almeno non dovrebbero) direttamente all'implementazione poiché questa in futuro può cambiare
- Tutto ciò è basato sul principio di ***information hiding***, ovvero proteggere i "clienti" da decisioni di design che in futuro possono cambiare
- La potenza del concetto di ADT sta nel fatto che l'implementazione è **nascosta** al cliente: viene resa pubblica solo l'interfaccia
- Questo significa che un ADT può essere implementato in vari modi ma, finché non viene cambiata l'interfaccia, i programmi che lo utilizzano non devono essere alterati

TIPI DI DATO ASTRATTO IN C

In C, un *ADT* si costruisce definendo:

- ***il nuovo tipo con `typedef`***
- ***una funzione per ogni operazione***

Esempio: il contatore

una entità caratterizzata da un valore intero

```
typedef int counter;
```

con operazioni per

- inizializ. contatore a zero `reset(counter*);`
- incrementare il contatore `inc(counter*);`

ORGANIZZAZIONE DI ADT IN C

La struttura di un ADT comprende quindi:

1. ***un file header,*** contenente
   - ***`typedef`***
   - ***dichiarazione delle funzioni***
2. ***un file di implementazione***, contenente
   - ***direttiva #include per includere il proprio header*** (per importare la definizione di tipo)
   - ***definizione delle funzioni***

ADT `counter`

1. ***file header `counter.h`***

```
typedef int counter;
void reset(counter*);
void inc(counter*);
```

Definisce in astratto che cos'è un counter e che cosa si può fare con esso

2. ***file di implementazione `counter.c`***

```
#include "counter.h"
void reset(counter *c){ *c=0; }
void inc(counter* c){ (*c)++; }
```

Specifica *come funziona (quale è l'implementazione) di counter*

ADT `counter`: un cliente

Per usare un `counter` occorre:

- ***includere il relativo file header***
- ***definire una o più variabili di tipo `counter`***
- ***operare su tali "oggetti" mediante le sole operazioni (funzioni) previste***

```
#include "counter.h"
int main(){
  counter c1, c2;
  reset(&c1); reset(&c2);
  inc(&c1); inc(&c2); inc(&c2);
}
```


OPERAZIONI DI UN ADT

Quali operazioni definire per un ADT?

- **costruttori** (***costruiscono un oggetto*** di questo tipo, a partire dai suoi "costituenti elementari")
- **selettori** (restituiscono uno dei ***"mattoni elementari"*** che compongono l'oggetto)
- **predicati** (verificano la ***presenza di una proprietà*** sull'oggetto, restituendo *vero* o *falso)*
- **funzioni** (***agiscono*** in vario modo sugli oggetti)
- **trasformatori** (***cambiano lo stato*** dell'oggetto)

Matrici come ADT

- Definire *handle* (tipo di base)
- Definire costruttore (e distruttore…)
- Definire operazioni

```
typedef double Element;
typedef struct DynMatrixStruct
{
    Element* matrix;
    int rows, cols;
} *DynMatrix;
```

***DynMatrix** è un puntatore ad una struttura da allocare dinamicamente che contiene una matrice da allocare dinamicamente*

Costruttore

- Alloca la memoria necessaria, inizializza i dati in modo opportuno e restituisce un puntatore a *handle*

```
DynMatrix newDynMatrix(int rows, int cols)
{
    DynMatrix dm = (DynMatrix)
       malloc(sizeof(struct DynMatrixStruct));
    if (dm == NULL)
        return NULL;
    dm->matrix = (Element*)malloc(sizeof(Element)*rows*cols);
    if (dm->matrix == NULL)
        return NULL;
    dm->rows = rows;
    dm->cols = cols;
    return dm;
}
```

Distruttore

- Per deallocare un *DynMatrix* è sufficiente scrivere **`free(aDynMatrix)`**?
- Evidentemente no: occorre prima deallocare la matrice, poi la struttura contenitore → occorre una procedura che lo sappia fare

```
void destroyDynMatrix(DynMatrix m)
{
    free(m->matrix);
    free(m);
}
```


Operazioni

- Data una *DynMatrix*:
  - Verifica dei *bound* → data una coppia di valori (riga, colonna) verificarne la validità
  - Recupero numero di righe
  - Recupero numero di colonne
  - Recupero di un elemento
  - Assegnamento di un elemento
  - …altre funzioni non “primitive”

Operazioni – interfaccia

```
boolean checkBounds(DynMatrix m, int row,
                                        int col);

int getRowCount(DynMatrix m);

int getColCount(DynMatrix m);

Element getMatrixElement(DynMatrix m,
                              int row, int col);

void setMatrixElement(DynMatrix m, int row,
                         int col, Element value);
```


Verifica dei bound

```
boolean checkBounds(DynMatrix m,
                                  int row, int col)
{
    return row >= 0 && col >= 0 &&
           row < m->rows && col < m->cols;
}
```

Dimensioni

```
int getRowCount(DynMatrix m)
{
    return m->rows;
}

int getColCount(DynMatrix m)
{
    return m->cols;
}
```


Funzioni accessorie

Consentono di leggere o impostare un elemento della matrice date riga e colonna

```
Element getMatrixElement(DynMatrix m, int row, int col)
{
    if (checkBounds(m, row, col)) exit(1);
    return *(m->matrix + row * m->cols + col);
}

void setMatrixElement(DynMatrix m, int row, int col,
      Element value)
{
    if (checkBounds(m, row, col)) exit(1);
    *(m->matrix + row * m->cols + col) = value;
}
```

ESERCIZIO

Realizzare l'ADT che cattura il concetto di "stringa di al più 250 caratteri"

- **realizzazione basata su un array di caratteri**

Occorre definire le operazioni per:

- estrarre il carattere situato alla i-esima posizione - *SELETTORE*
- calcolare la lunghezza - *FUNZIONE*
- creare una nuova stringa concatenazione di due stringhe date - *COSTRUTTORE*
- confrontare due stringhe - *FUNZIONE*

ESERCIZIO

Realizzare l'ADT che cattura il concetto di "insieme" (di interi)

- **implementazione basata, ad esempio, su una struttura con un array e un indice**

Operazioni per:

- aggiungere un elemento, togliere un elemento - *TRASFORMATORI*
- verificare la presenza di un elemento - *PREDICATO*
- calcolare l'unione di due insiemi, la differenza fra due insiemi, l'intersezione, ecc. - *COSTRUTTORE?*

NOTA: su un insieme non è usualmente definita relazione di ordine e un insieme non può contenere elementi duplicati

ADT IN C: LIMITI

- **Gli ADT così realizzati funzionano, ma *molto dipende dall'autodisciplina del programmatore***
- **Non esiste alcuna protezione contro un uso scorretto dell'ADT**

  l'organizzazione ***suggerisce*** di operare sull'oggetto ***solo tramite le funzioni previste***, ma ***NON riesce a impedire*** di aggirarle a chi lo volesse
  (ad esempio: `counter c1; c1++;`)

- **La struttura interna dell'oggetto *è visibile a tutti*** (nella **`typedef`**)

ADT IN C: LIMITI

Superare questi limiti sarà uno degli obiettivi cruciali della *programmazione a oggetti*, che vedrete nel corso di Fondamenti di Informatica L-B con il linguaggio Java