L03.1. ABSTRACT DATA TYPES

Abbiamo detto che l'**astrazione** è un procedimento mentale che ci consente di evidenziale le caratteristiche principali e trascurare aspetti "secondari", e che esistono due tipi: **funzionale e procedurale**, che ha la finalità di ampliare l'insieme dei modi di operare sui tipi di dati già disponibili attraverso la definizione di nuovi operatori (delegano operazioni a nuovi sotto-programmi), e quella sui **dati**, che ha la finalità di ampliare i tipi di dati disponibili attraverso l'introduzione sia di nuovi tipi di dati che di nuovi operatori.

Un **tipo di dati** è definito da un **dominio di valori** e un insieme di **operazioni** previste su quei valori, ad esempio il tipo interi può essere definito da un range di valori come da 1 a 9, mentre le operazioni possono essere "+, -, *, /, …". Nel linguaggio C esistono vari tipi di dati:

- **Dati primitivi**, forniti direttamente dal linguaggio: int, char, float, double;
- **Dati aggregati**, array, strutture, enumerazioni, unioni;
- **Puntatori**.

Unendo questi due concetti, ovvero astrazione e tipi di dati, otteniamo i "**tipi di dati astratti**" (ADT), quest'ultimo è un tipo di dati che estende dei dati esistenti, definito distinguendo **specifica** e **implementazione**:

- **Specifica**, viene fatta definendo il tipo dei dati e l'insieme degli operatori di cui si disporrà su questi dati. Esistono due tipi di specifiche:
  1. **Sintattica**, vengono definite delle regole, ovvero ***nomi*** e ***tipi***;
  2. **Semantica**, vengono definiti i significati, ovvero ***valori*** e ***vincoli***.
- **Implementazione**, indipendente dalla specifica, è la codifica di quanto viene definito nella specifica, questa fase è spesso nascosta al programmatore, seguendo il principio dell'**incapsulamento** (information hiding).

Definiremo i tipi di dati astratti usando una tabella che ci semplifica il lavoro della specifica, ed è strutturata in questo modo:

	Sintattica	Semantica
Tipi di dati	• Nome dell'ADT • Tipi da dati già usati	• Insieme dei valori
Operatori: Per ogni operatore	• Nome dell'operatore • Tipi di dati di input e di output	Funzione associata all'operatore • Precondizioni: definiscono quando l'oeratore è applicabile • Postcondizioni: definiscono relazioni tra dati di input e output

Esempio tipo dati astratto chiamato "punto":

Sintattica	Semantica
Nome del tipo: Punto Tipi usati: Reale	Dominio: Insieme delle coppie (ascissa, ordinata) dove ascissa e ordinata sono numeri reali
creaPunto (reale, reale) → punto	creaPunto(x, y) = p • pre: true • post: p = (x, y)
ascissa (punto) → reale	ascissa(p) = x • pre: true • post: p = (x, y)
ordinata (punto) → reale	ordinata(p) = y • pre: true • post: p = (x, y)
distanza (punto, punto) → reale	distanza(p1, p2) = d • pre: true • post: d = sqrt((ascissa(p1)-ascissa(p2))^2 + (ordinata(p1)-ordinata(p2))^2)

ADT STRUCT:

Per implementare gli ADT in C, un primo costrutto è la **Struttura (struct)**, ovvero un tipo dati composito che include un elenco di variabili fisicamente raggruppare in un unico blocco di memoria, ha il vantaggio che migliora la leggibilità dei programmi:

```
struct point {        //Definizione della struttura
   float x;           //Campi della struttura
   float y;
};
```

```
int main(){
   struct point p;                          //Variabile di tipo struttura
   p.x= 2.0;                                //Per accedere ai campi si usa la "dot sintax"
   p.y= 3.0;
   printf("coordinate del punto: (%.1f, %.1f)", p.x, p.y);
   struct point p1 = {2.0, 3.0};            //Inizializzazione della struttura
}
```

Si può usare il **typedef** sulla struttura precedentemente definita:

```
typedef struct point Point;
```

Oppure usare in combinazione il typedef e la definizione della struttura:

```
typedef struct{
   float x;
   float y;
} Point;
```

```
int main(){
   Point p = {2.0, 3.0};
   printf("coordinate del punto: (%.1f, %.1f)", p.x, p.y);
}
```

Possiamo anche allocare memoria dinamica per l'ADT struttura utilizzando le consuete funzioni:

```
Point *p = malloc(sizeof(Point));
```

È possibile accedere ai campi della struttura da un puntatore usando l'operatore **freccia ->**:

```
int main(){
   Point *p = malloc(sizeof(Point));
   p->x = 2.0;
   p->y = 3.0;
   printf("coordinate del punto: (%.1f, %.1f)", p->x, p->y);
}
```

Esempio completo di implementazione dell'ADT punto:

punto.h

```
typedef struct{
   float x;
   float y;
} Punto;

Punto creaPunto(float x, float y);
float ascissa(Punto p);
float ordinata(Punto p);
float distanza(Punto p1, Punto p2);
```

Questa implementazione lascia un ***problema***, ovvero l'implementazione della struttura del tipo punto è nell'header file, visibile quindi al modulo client, che potrebbe quindi accedere direttamente ai campi della struct senza usare gli operatori dell'ADT.

Per soddisfare l'information hiding spostiamo l'implementazione della struttura Punto dall'header file al file di implementazione delle funzioni (*punto.c*).

```
typedef struct punto *Punto;

Punto creaPunto(float x, float y);
float ascissa(Punto p);
float ordinata(Punto p);
float distanza(Punto p1, Punto p2);
```

Una volta spostata l'implementazione della struttura, nell'header file definiamo il tipo Punto come ***puntatore*** alla struttura.

NOTA: All'atto della compilazione del modulo client, essendo il tipo punto un puntatore, il compilatore sa quanta memoria deve allocare per una variabile di quel tipo, indipendentemente dalla dimensione dell'elemento puntato.

L03.2. PSEUDO-GENERICS IN C

I **Generics** sono uno strumento che permette la definizione di un tipo parametrizzato, che viene esplicitato successivamente in fase di compilazione (o linkaggio) secondo le necessità, permettono di eseguire algoritmi su tipi di dati diversi e applicare ADT su tipi di dati diversi.

Per operare su tipi diversi di dati bisognerebbe modificare l'algoritmo vero e proprio, ad esempio per il ***bubble sort*** se vogliamo ordinare delle stringhe:

Bubble sort su interi:

```
void swap_int(int*a, int*b){
   int temp= *a;
   *a = *b;
   *b = temp;
}
void bsort_int(int a[], int n){
   int i, j;
   for(i=1; i<n; i++)
      for(j=0; j<n-i; j++)
         if(a[j] > a[j+1])
            swap_int(&a[j], &a[j+1]);
}
```

Main bubble sort su interi:

```
int main(){
   int i, n = 5;
   int arr[n];
   printf("Introduci il vettore: ");
   for(i=0; i<n; i++)
      scanf("%d",&arr[i]);
   bsort_int(arr, n);
   printf("Vettore ordinato: ");
   for(i=0; i<n; i++)
      printf("%d ",arr[i]);
}
```

Bubble sort su stringhe:

```
void swap_string(char**a, char**b){
   char *temp= *a;
   *a = *b;
   *b = temp;
}
void bsort_string(char*a[], int n){
   int i, j;
   for(i=1; i<n; i++)
      for(j=0; j<n-i; j++)
         if(strcmp(a[j], a[j+1])>0)
            swap_string(&a[j], &a[j+1]);
}
```

Main bubble sort su stringhe:

```
Int main(){
   Int i, n = 5;
   char*arr[5];
   printf("Introduci il vettore: ");
   for(i=0; i<n; i++){
      arr[i] = malloc(20*sizeof(char));
      scanf("%s",arr[i]);
   }
   bsort_string(arr, n);
   printf("Vettore ordinato: ");
   for(i=0; i<n; i++)
   printf("%s ",arr[i]);
}
```

Quindi il nostro obiettivo è quello di realizzare algoritmi e/o ADT che siano in grado di funzionare con tipi di dati diversi. Una **soluzione** generale è quella di generalizzare il nostro algoritmo, in modo che possa funzionare con tipi diversi (interi, stringhe o qualunque struttura). Si procede così:

1. **Creare un tipo "Item"** (interfaccia, file .h) che supporti input, output e confronto;
2. **Modificare le librerie** in modo che operino sul tipo Item;
3. **Realizzare Item in file .c** che supportano le varianti intero, stringa e struttura;
4. **Linkare ed eseguire separatamente** (con make) le varianti.

Per implementare quanto detto in C si procede on questo modo:

Questo "Item" lo definiamo come puntatore a void, quindi un generico puntatore per il quale non si specifica il tipo a cui punta: `void *p_void;`

E poi assegnarlo al tipo desiderato:

- `int* p_int= p_void;`
- `char* p_char= p_void;`
- ```
  struct studente{
     char nome[20];
     int matricola;
  };
  typedef struct studente *Studente;
  Studente p_studente= p_void;
  ```
  ```