PROGRAMMAZIONE MODULARE

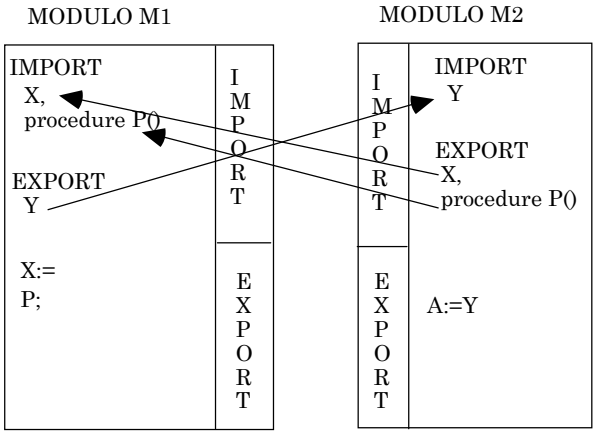
- Tecnica di suddividere un progetto software in parti il più possibile indipendenti (moduli sviluppabili ***separatamente***, con compilazione e testing separati), le cui modalità di interazione siano ben definite (***interfacce*** standard)
- Principio di ***Information hiding***
- Non è necessario conoscere i dettagli dell'implementazione
- Supporto a progettazione sia ***top-down*** che ***bottom-up***
- Permette di ottenere sia riusabilità sia estendibilità
- Concetto di ***modulo***, presente in linguaggi quali Modula (Wirth), Pascal (TurboPascal), MESA, Ada

Concetto di modulo:

- Un ***modulo*** raggruppa al suo interno un insieme di informazioni (tipi, costanti, variabili, procedure e funzioni)
- In generale conterrà ***dati*** ed ***operazioni***.
- Solo ciò che è [esplicitamente] esportato all'esterno del modulo e dualmente solo ciò che è [esplicitamente] importato è disponibile all'interno di un modulo
- Un modulo perciò definisce e confina un preciso ***ambiente di visibilità***
- I riferimenti fra moduli sono risolti staticamente durante la fase di collegamento + il programma eseguibile non distingue più le unità separate
- Linguaggi che:

  supportano il concetto di modulo a livello linguistico (Ada, Modula, etc.)

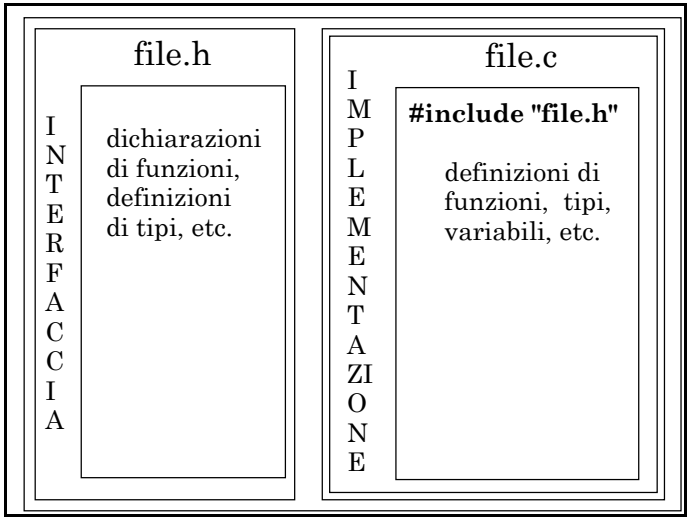
  suddividonono le applicazione in più unità (nel caso del C, file compilabili separatamente)



- Nel linguaggio C possiamo avere forme più o meno fini di import/export:
  - a livello di singolo identificatore (variabili o funzioni **extern**);
  - a livello di componente software ("modulo") costituito da una interfaccia (file .h) e una implementazione (file .c).

"Programmazione modulare" e C:

- Un "modulo" C può essere realizzato attraverso due file separati (***header*** ed ***implementation***) che riflettono l'idea di componente software dotato di interfaccia nota ed accessibile ed implementazione nascosta



Si possono importare:

- tutti gli identificatori definiti o dichiarati nel file header (***interfaccia***), se si usa la direttiva:

  #include "*file.h*"

- solo alcuni identificatori, se si usa la dichiarazione:

  extern *identificatore*

Esempio:

Gli identificatori ***esportati*** sono definiti o dichiarati in un file separato (***file header***)

```
/* ELEMENT TYPE - file el.h */

typedef   int el_type;
typedef   enum {false,true} boolean;
boolean   isequal(el_type, el_type);
boolean   isless(el_type, el_type);
void   showel(el_type e);
void init(el_type *e, el_type v);
```

Altri identificatori e la definizione delle funzioni ***esportate*** e altre funzioni sono riportate in un secondo file (***file.c***)

```
/* ELEMENT TYPE - file el.c */

#include "el.h"
#include <stdio.h>

boolean isequal(el_type e1,el_type e2)
{return (e1==e2);}

boolean isless(el_type e1, el_type e2)
{return (e1<e2);}

void showel(el_type e)
{printf("%d",el);}

void init(el_type *e, el_type v){ *e=v; }
```



```
/* main file - inizio.c */

#include "el.h" /* inclusione interfaccia */

void main(void)
{el_type Dato;
 init(&Dato,5);
 showel(Dato);
}
```

Il collegamento tra moduli viene specificato attraverso un *file progetto* (estensione ***.prj***) o *Makefile* (denominazione UNIX).

Per compilare un programma suddiviso in più file, viene creato il file progetto (estensione ***.prj***) in cui si specificano quali file (o "moduli") C costituiscono l'applicazione.

Esempio:

☞ Nel *file.prj*: **inizio.c**
el.c

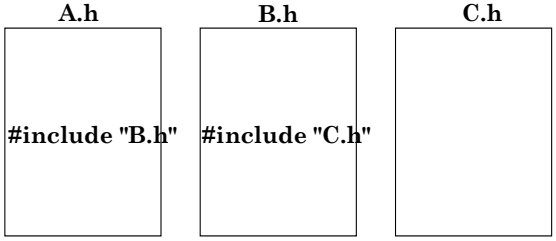
Ciascun file con estensione ***.c*** che compare nel file progetto aperto viene compilato separatamente (producendo file oggetto) e successivamente il linker collega i file oggetto in un unico eseguibile (rilocabile)

- La direttiva ***#include*** può comparire sia nella parte ***interfaccia (header)*** che nell'***implementazione*** di un "modulo".

- Non è detto che ***interfaccia*** ed ***implementazione*** usino le stesse funzionalità.

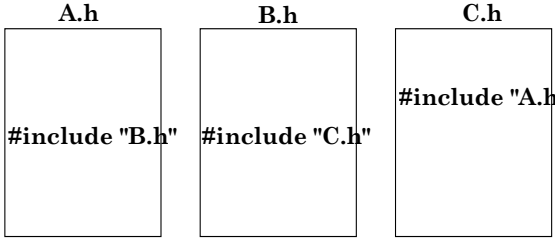
- Se cambia la sezione interfaccia di un modulo (***file header***), occorre ricompilare i moduli che lo includono.

- Un modulo può utilizzare le funzionalità messe a disposizione da altri moduli:



- Nel caso del linguaggio C, la direttiva ***#include*** stabilisce una ***relazione transitiva*** (**A** vede gli identificatori esportati da **B** e da **C**)

- Non è possibile avere riferimenti circolari nella parte ***interface***:



Esercitazione 7.1: tavola come astrazione di dato

Riprendiamo l'esercitazione 6.1 e realizziamo la tavola come astrazione di dato, nascondendone la realizzazione (in particolare il tipo **`rubrica`**, la variabile **`R`** e la dimensione corrente, rappresentata da una variabile denominata **`DIM`**) ed esportando solo i prototipi delle operazioni di **`inserimento`**, **`cancellazione`** e **`ricerca`** sul dato.

```
/* interfaccia - file tavola.h */

void inserimento(void);
void cancellazione(void);
void ricerca(void);
```

```
/* implementazione – file tavola.c */

#include "tavola.h"
#include <stdio.h>
#include <string.h>

typedef struct   {char nome[20];
                  char tel[16]; } elemento;

#define N 100
typedef elemento rubrica[N];
rubrica R;
int DIM=0;

int individua(elemento e);

void inserimento(void)
{   if (DIM<N-1)
    {   printf("\nInserire nome:  ");
        gets(R[DIM].nome);
        printf("\nInserire numero:  ");
        gets(R[DIM].tel);
        DIM++;
    }
    else printf("Vettore pieno!\n");
    return;
}
```



```
void cancellazione(void)
{   int k, j;
    elemento e;
    printf("\nInserire nome:  ");
    gets(e.nome);
    k=individua(e);
    if (k<N)
{printf("\nCancellazione di %s ...\n",
            R[k].nome);
        for (j=k; j<DIM-1; j++)
            R[j]=R[j+1];
        DIM--;
    }
    else
printf("\n%s\t non trovato\n", e.nome);
    return;
}

void ricerca(void)
{   int k;
    elemento e;
    printf("\nInserire nome:  ");
    gets(e.nome);
    k=individua(e);
    if (k<N)
  printf("\n%s\t%s\n",R[k].nome,R[k].tel);
    else
  printf("\n%s\t non trovato\n", e.nome);
}

int individua(elemento e)
{   int i,trovato=0;
    for (i=0; i<DIM && !trovato; i++)
        if (!strcmp(e.nome, R[i].nome))
                trovato=1;
    if (trovato) return i-1;
    else return N;}
```

La variabile **R** di tipo **rubrica** è incapsulata all'interno della parte implementazione.

Il cliente (**main.c**) conosce solo l'interfaccia del componente (i prototipi delle operazioni invocabili sul dato).

```
/* utilizzo – file main.c */
#include "tavola.h"
#include <stdio.h>

/* menu . . . */

main()
{   int scelta, fine;

    fine=0;
    do
    {scelta=menu();
     switch(scelta){
      case 1:inserimento();  break;
      case 2: cancellazione();  break;
      case 3: ricerca();  break;
      case 4: fine=1; break;
      default:printf("Sceltabagliata\n");
        }
    }while (!fine);
}
```


Tipo di dato astratto (Abstract Data Type, ADT):

- Entità descrittiva che specifica le caratteristiche (rappresentazione dei dati e operazioni) di ogni dato generato da essa

- Un tipo di dato astratto T (una *classe*) permette una condivisione di codice da parte di ogni dato dichiarato di tipo T (*istanza* di T) con conseguente non replicazione di informazioni.

- Tipicamente nei linguaggi di programmazione sono a disposizione:

  - TIPI PRIMITIVI con operazioni predefinite associate (ADT);

  - variabili multiple istanziate da questi.

- L'utente può utilizzare costruttori di tipo, ma non estendere il linguaggio con nuove astrazioni.

- SIMULA67 è il progenitore dei linguaggi di programmazione che rendono disponibile un ***costrutto di classificazione*** per definire nuove astrazioni (anche con *ereditarietà*)

Esercitazione 7.2.0: Tavola come ADT

Riprendiamo l'esercitazione 6.1 e 7.1 e realizziamo un nuovo tipo di dato, il tipo **tavola**, con associate le operazioni di **inserimento**, **cancellazione** e **ricerca**.

Il componente esporta quindi attraverso la sua interfaccia sia l'identificatore di tipo sia i prototipi delle operazioni sui dati che verranno dichiarati di quel tipo:

```
/* interfaccia - file tavola_adt.h */

typedef struct   { char nome[20];
                   char tel[16]; } elemento;

#define N 100
typedef elemento rubrica[N];
typedef struct {rubrica R;
       int DIM;  } table;
typedef table tavola;

void init (tavola * T);
void inserimento(tavola * T);
void cancellazione(tavola * T);
void ricerca(tavola * T);
```



```
/* implementazione – file tavola_adt.c */
#include "tavola_adt.h"
#include <stdio.h>
#include <string.h>

int individua(tavola * T,elemento e);

void init (tavola * T)
{(*T).DIM=0; }          /* T->DIM */

void inserimento(tavola * T)
{   if (((*T).DIM)<N)
    {   printf("\nInserire nome:  ");
        gets(((*T).R[(*T).DIM]).nome);
        printf("\nInserire numero:  ");
        gets(((*T).R[(*T).DIM]).tel);
        ((*T).DIM)++;
    }
    else printf("Vettore pieno!\n");
    return;
}
```

```
void cancellazione(tavola * T)
{   int k, j;
    elemento e;
    printf("\nInserire nome:  ");
    gets(e.nome);
    k=individua(T,e);
    if (k<N)
{printf("\nCancellazione di %s ...\n",
          ((*T).R[k]).nome);
       for (j=k; j<(*T).DIM-1; j++)
          (*T).R[j]=(*T).R[j+1];
       (*T).DIM--;
    }
    else
printf("\n%s\t non trovato\n", e.nome);
    return;
}
```



```
void ricerca(tavola * T)
{   int k;
    elemento e;
    printf("\nInserire nome:  ");
    gets(e.nome);
    k=individua(T,e);
    if (k<N)
  printf("\n%s\t%s\n",(*T).R[k].nome,
              (*T).R[k].tel);
    else
  printf("\n%s\t non trovato\n", e.nome);
}


int individua(tavola * T, elemento e)
{   int i,trovato=0;
    for (i=0; i<((*T).DIM) && !trovato; i++)
       if (!strcmp(e.nome,((*T).R[i]).nome))
              trovato=1;
    if (trovato) return i-1;
    else return N;}
```

Non è un vero e proprio ADT perché esternamente al modulo viene fornito non solo l'identificatore di tipo **`tavola`** (e anche **`rubrica`** ed **`elemento`**), ma anche la struttura dei tipi.

In questo caso, le variabili di tipo **`tavola`** sono ***dichiarate da chi utilizza il modulo***, ovvero il cliente **`main.c`**.

Sta al cliente un uso disciplinato attraverso le sole operazioni rese visibili nel file header.

```
/* utilizzo - file main.c */
#include "tavola_adt.h"
#include <stdio.h>
#include <stdlib.h>

main()
{   tavola T1, T2;
    int scelta, fine;

    init(&T1);        /* inizializzazione */
    fine=0;
    do
    {scelta=menu();
     switch(scelta){
      case 1:inserimento(&T1);  break;
      case 2: cancellazione(&T1);  break;
      case 3: ricerca(&T1);  break;
         case 4: fine=1; break;
      default:printf("Sceltabagliata\n");
        }
    }while (!fine);

    init(&T2);
    ...
}

/* menu . . . */
```


Programmazione modulare: requisiti

Occorre che siano soddisfatti alcuni principi:

- Moduli corrispondenti a ***unità sintattiche*** nel linguaggio usato (separatamente compilabili);
- Le informazioni di un modulo sono private, a meno che il modulo non le dichiari esplicitamente pubbliche (***information hiding***);
- Ciascun modulo deve "comunicare" con il minor numero di moduli (solo con quelli necessari);

  Se due moduli si interfacciano tra loro, le loro interfacce devono scambiarsi il minor numero di informazioni (solo quelle necessarie);

  Se due moduli comunicano tra loro, questo deve essere evidente dal loro codice.

In C questi requisiti sono soddisfatti solo parzialmente.

Esercitazione 7.2.1: tavola come ADT

Riprendiamo l'esercitazione 7.2 e realizziamo il nuovo tipo di **tavola** come puntatore a un tipo **table** (con associate le usuali operazioni di **inserimento**, **cancellazione** e **ricerca**).

```
/* interfaccia - file tavola_adt_p.h */

typedef struct   {char nome[20];
                  char tel[16]; } elemento;

#define N 100
typedef elemento rubrica[N];
typedef struct {rubrica R;
      int DIM;  } table;
typedef table * tavola;

void init (tavola T);
void inserimento(tavola T);
void cancellazione(tavola T);
void ricerca(tavola T);
```

In alcuni linguaggi a moduli (Ad es. Modula 2) la definizione del tipo **table** è posta nel file implementazione.


```
/* implementazione – file tavola_adt_p.c */

#include "tavola~1.h"
#include <stdio.h>
#include <string.h>

int individua(tavola T,elemento e);

void init (tavola T)
{(*T).DIM=0; }          /* T->DIM */

void inserimento(tavola T)
{   if (((*T).DIM)<N)
    {   printf("\nInserire nome:  ");
        gets(((*T).R[(*T).DIM]).nome);
        printf("\nInserire numero:  ");
        gets(((*T).R[(*T).DIM]).tel);
        ((*T).DIM)++;
    }
    else printf("Vettore pieno!\n");
    return;
}
```

```
void cancellazione(tavola T)
{   int k, j;
    elemento e;
    printf("\nInserire nome:  ");
    gets(e.nome);
    k=individua(T,e);
    if (k<N)
{printf("\nCancellazione di %s ...\n",
          ((*T).R[k]).nome);
       for (j=k; j<(*T).DIM-1; j++)
          (*T).R[j]=(*T).R[j+1];
       (*T).DIM--;
    }
    else
printf("\n%s\t non trovato\n", e.nome);
    return;
}
```



```
void ricerca(tavola T)
{   int k;
    elemento e;
    printf("\nInserire nome:  ");
    gets(e.nome);
    k=individua(T,e);
    if (k<N)
  printf("\n%s\t%s\n",(*T).R[k].nome,
              (*T).R[k].tel);
    else
  printf("\n%s\t non trovato\n", e.nome);
}


int individua(tavola T, elemento e)
{   int i,trovato=0;
    for (i=0; i<((*T).DIM) && !trovato; i++)
       if (!strcmp(e.nome,((*T).R[i]).nome))
              trovato=1;
    if (trovato) return i-1;
    else return N;}
```

Anche in questo caso non è un vero e proprio *Abstract Data Type* (ADT) perché esternamente al modulo viene fornito non solo l'identificatore di tipo **`tavola`** (e anche **`rubrica`**, **`table`** ed **`elemento`**), ma anche la struttura dei tipi.

Anche in questo caso, le variabili di tipo **`tavola`** sono ***dichiarate da chi utilizza il modulo***, ovvero il cliente **`main.c`**, che alloca anche lo spazio per loro in questa versione.

```
/* utilizzo - file main.c */
#include "tavola~1.h"
#include <stdio.h>
#include <stdlib.h>

main()
{   tavola T1;
   T1=(tavola)malloc(sizeof(table));
      /* allocazione oggetto referenziato */

   int scelta, fine;
   init(T1);    /* inizializzazione tavola*/
   fine=0;
   do
   {scelta=menu();
    switch(scelta){
     case 1:inserimento(T1);  break;
     case 2: cancellazione(T1);  break;
     case 3: ricerca(T1);  break;
     case 4: fine=1; break;
     default:printf("Scelta sbagliata\n");
       }
   }while (!fine);
  free(T1);
}

/* menu . . . */
```

E' necessaria un'allocazione esplicita dello spazio occupato dal dato referenziato dalla variabile di tipo **`tavola`** (chiamando la **`malloc`**).

Ritroveremo questo concetto nella creazione di oggetti (per es., in Java).

Anche in questo caso sta al cliente un uso disciplinato attraverso le sole operazioni rese visibili nel file header.

Esercitazione 7.3: ADT tavola come vettore ordinato

Modifichiamo l'esercitazione 7.2 in modo tale che gli elementi nella tavola siano ordinati sulla base del nome.

Al modulo si aggiungono le seguenti funzioni (in particolare, la funzione **individua_ord** che cerca la posizione in cui effettuare l'inserimento ordinato e la funzione **individua_bin** che realizza una ricerca binaria):

```
int individua_bin(tavola T, elemento e)
{int i=0,medio,trovato=0;
 int DIM=(*T).DIM;

 while ((i<DIM)&&(!trovato))
   {
    medio=(i+DIM)/2;

   if (!strcmp(e.nome,((*T).R[medio]).nome))
          trovato=1;
   else
     if
     (strcmp(e.nome,((*T).R[medio]).nome)>0)
           i=medio+1;
    else
           DIM=medio-1;
   }
   if (trovato)
       return medio;
   else
       return N;}
```

```
int individua_ord(tavola T, elemento e)
```



```
{ int i,trovato=0;
  for (i=0; i<((*T).DIM) && !trovato; i++)
  if (strcmp(e.nome,((*T).R[i]).nome)<=0)
              trovato=1;
    if (trovato)
         return i-1;
    else
         return (*T).DIM;
             /* se piena restituisce N */
}
```

```
void inserimento_ord(tavola T)
{
   int k;
   elemento e;
   printf("\nInserire nome:  ");
   gets(e.nome);
   printf("\nInserire numero:  ");
   gets(e.tel);

   k=individua_ord(T,e);
   if ((*T).DIM < N )
   {  if (k<N)
      { for (j=k; j<DIM-1; j++)
                    (*T).R[j+1]=(*T).R[j];
        (*T).R[k]=e;
        (*T).DIM++;
       }
    }
   else printf("Vettore pieno!\n");
   return;
}
```