Programmazione modulare

- Inizialmente intesa come costruzione di programmi assemblando parti, di solito ***sottoprogrammi***

- Tecnica di suddividere un progetto software in parti il piu` possibile indipendenti (moduli sviluppabili ***separatamente***, con compilazione e testing separati), le cui modalita` di interazione siano ben definite (***interfacce*** standard)

- Non e` necessario conoscere i dettagli dell'implementazione

- Supporto a progettazione sia ***top-down*** che ***bottom-up***

- Permette di ottenere sia riusabilita` che estendibilita`

  ☞concetto di ***modulo***

- Presente in linguaggi quali Modula (Wirth), Pascal (TurboPascal), MESA, Ada

Prime estensioni in linguaggi tradizionali:

- ***Sviluppo di codice separato:***

- Meccanismi che operano ***a livello di codice sorgente***, ed intervengono ***prima della compilazione***

- possibile incapsulamento di insiemi di istruzioni (MACRO) espanse quando servono

- riutilizzo di parti di codice espresse in sorgenti gia` disponibili includendoli in nuovi programmi

Macro:

Ad es., nel linguaggio C si definisce una *macro* attraverso il costrutto:

#**define** <*id_macro*> <*sequenza_token*>

Sono possibili anche macro parametriche:

#**define**<*id_macro*>(<*lista_arg*>)<*sequenza_token*>

L'elenco argomenti rappresenta una lista di parametri "formali"

La chiamata e` simile a quella delle chiamate delle funzioni:

<*id_macro*> (<*lista_arg_effettivi*>)

Si possono avere anche definizioni di macro annidate.

- **Esempio:**

```
#define cubo(x) ((x)*(x)*(x))
...
int n, m, y;
n = cubo(y);
m = cubo(y+1);
```

sono espansi in:

```
n = ((y)*(y)*(y));
m = ((y+1)*(y+1)*(y+1));
```

Una chiamata ad una macro non ha verifiche di tipo

Puo` inoltre provocare effetti "collaterali" indesiderati, quando ad esempio l'argomento effettivo e` valutato piu` volte

- **Esempio:**

```
int CUBO(int x) {return x*x*x;};
#define cubo(x) ((x)*(x)*(x))
...
int n, m, y=3;
m = CUBO(y++);
            /* m vale 27, y vale 4 */
y=3;
n = cubo (y++);
/* espanso in ((y++)*(y++)*(y++))
 n vale 60, y vale 6 */
```

- ***Inclusione di sorgenti:***

  Nel linguaggio C esiste una direttiva al preprocessore (o precompilatore) che consente di ***includere il codice sorgente*** contenuto in un altro file nel programma:

  #**include** <*nome_file*>

  #**include** *"nome_file"*

  #**include** *identificatore_macro*

  Il codice sorgente contenuto nel file *nome_file* viene incluso nel punto in cui si trova la direttiva

  Si possono avere piu` inclusioni di file sorgenti annidate

- Meccanismi che operano ***a livello di codice oggetto***, ed intervengono ***durante la fase di collegamento*** (linking)

  dichiarazioni di importazione ed esportazione di entita` fra file diversi collegati a formare un unico programma

- In C la creazione di librerie collegate (***link***) ai vari programmi (senza renderne disponibile il codice sorgente) si ottiene attraverso la creazione di ***file separati***, compilati separatamente (***moduli***)

- Esistono ***librerie*** predefinite.

- La filosofia dell'uso di ***moduli*** puo` essere seguita in modo sistematico anche nello sviluppo di programmi di utente

  ☞ ***programmazione modulare***

Programmazione modulare: requisiti

Occorre che siano soddisfatti alcuni principi:

1. Moduli corrispondenti ad ***unita` sintattiche*** nel linguaggio usato (separatamente compilabili);

2. Ciascun modulo deve "comunicare" con il minor numero di moduli (solo con quelli necessari);

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

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

5. Le informazioni di un modulo sono private, a meno che il modulo non le dichiari esplicitamente pubbliche (***information hiding***).

Concetto di modulo:

- Un ***modulo*** raggruppa al suo interno un insieme di informazioni (tipi, costanti, variabili, procedure e funzioni)

- In generale conterra` ***dati*** ed ***operazioni***.

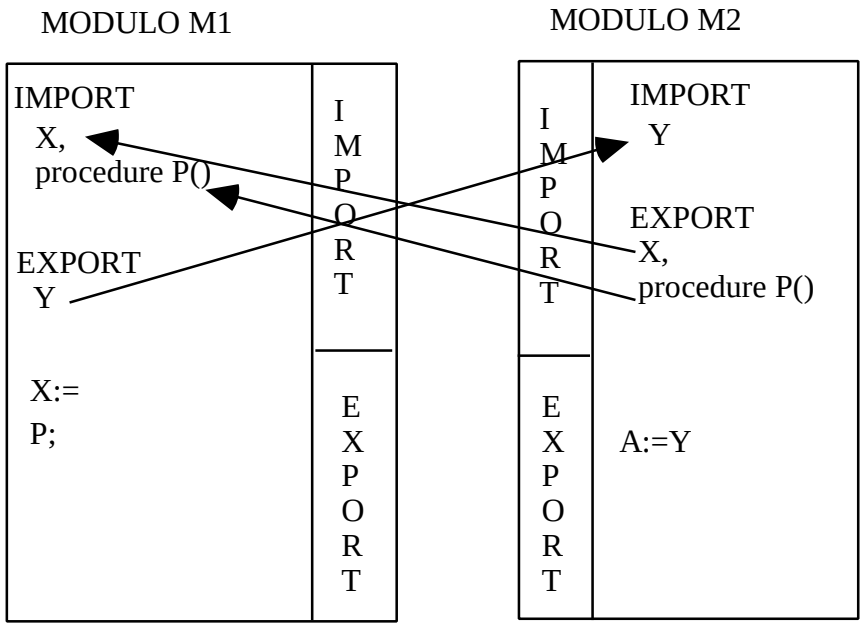
- Solo cio` che e` [esplicitamente] esportato all'esterno del modulo e dualmente solo cio` che e` [esplicitamente] importato e` disponibile all'interno di un modulo

- Un modulo percio` definisce e confina un preciso ***ambiente di visibilita`***

- I riferimenti fra moduli sono risolti staticamente durante la fase di collegamento ☞ il programma eseguibile non distingue piu` le unita` separate

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

- Applicazione suddivisa in piu` unita` (nel caso del C, file compilabili separatamente)



- Nel linguaggio C possiamo avere forme piu` 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 ed una implementazione.

  La seconda ci consente di seguire una migliore ***metodologia*** nello sviluppo del software.

Import/export a livello di singolo identificatore:

- Le entita` ***importate*** sono determinate mediante la dichiarazione **extern**.

Esempio:

File "AAA.c"

```
 extern void fun2(...);

...

int ncall = 0;

...

fun1(...)
   {
   ncall++;
   ...
   }
```

File "BBB.c"

```
extern fun1(...);

void fun2(...);

...

extern int ncall;

...

void              fun2(...)
   {
   ncall++;
   ...
   }
```

Il file "BBB.c" importa la variabile ncall ed il file "AAA.c" importa la funzione fun2.

File progetto

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

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

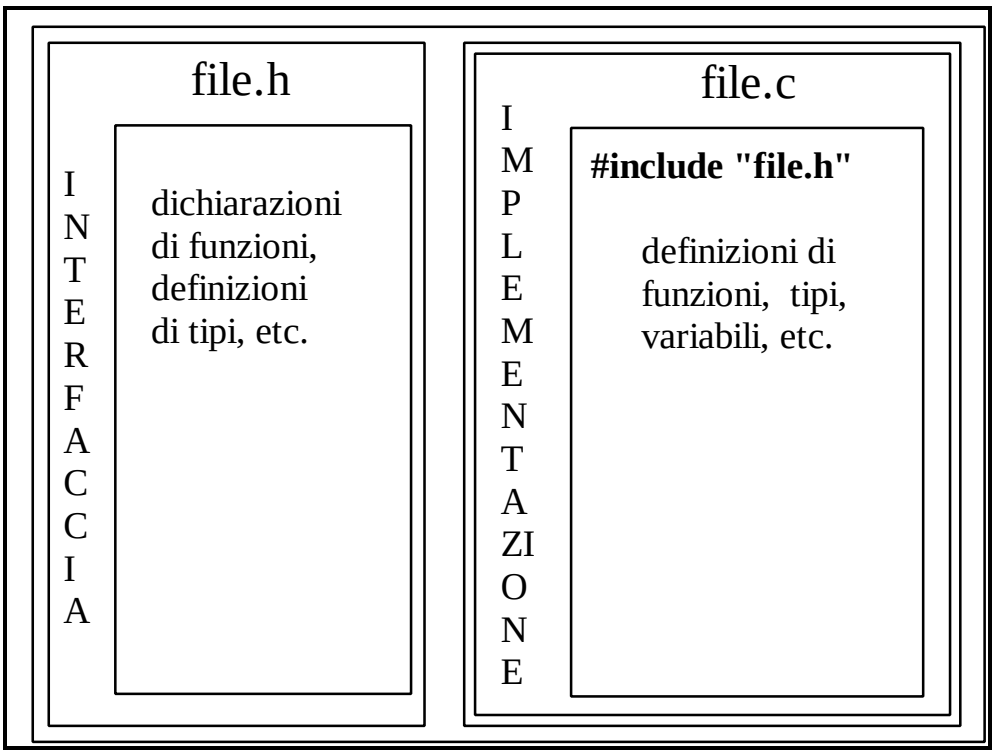
Esempio:

☞Contenuto di *file.prj*: AAA.c
BBB.c

Ciascun file ***.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)

Import/export a livello di componenti software

- Un "modulo" C puo` 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 (C):

- 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);
```

- 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);}
```

```
/* main file - inizio.c */
#include "el.h"           /* utilizzo */
void main(void)
{el_type Dato=5;
 showel(Dato);
}
```

- Il collegamento tra moduli viene specificato anche in questo caso attraverso un file progetto (estensione ***.prj***)

Nel *file.prj*:

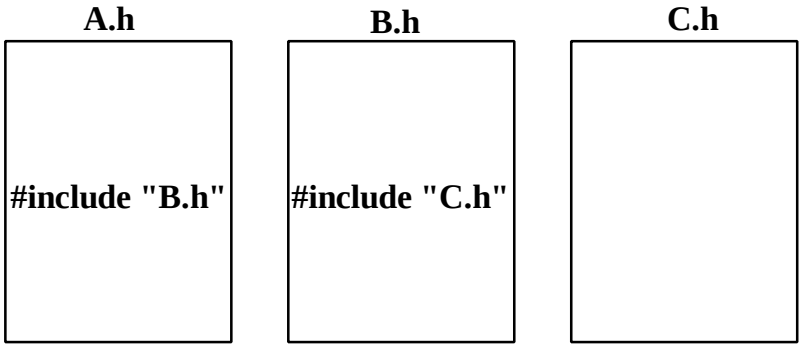
```
inizio.c
el.c
```

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

- Non e` detto che ***interfaccia*** ed ***implementazione*** usino le stesse funzionalita`.

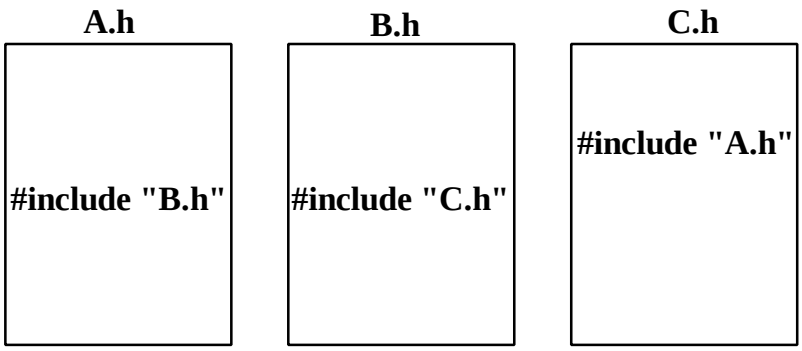
- Se cambia la sezione interfaccia di un modulo (***file header***), occorre ricompilare i moduli che lo includono.

- Un modulo puo` utilizzare le funzionalita` 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 e` possibile avere riferimenti circolari nella parte ***interface***:


A.h
#include "B.h"
B.h
#include "C.h"
C.h
#include "A.h"

Componenti software:

- Ci sono diversi livelli di utilizzo di moduli per realizzare:

- ***Librerie:***

  Il modulo rende visibili procedure e funzioni che fanno uso solo di variabili locali. Il modulo e` una collezione di ***operazioni*** (ad esempio, funzioni matematiche).

- ***Astrazioni di dato:***

  Il modulo ha dati locali (nascosti) e rende visibili all'esterno i prototipi delle operazioni invocabili (procedure e funzioni) su questi dati locali, ma non gli identificatori dei dati. Attraverso una di queste operazioni si puo` assegnare un valore iniziale ai dati locali nascosti.

- ***Tipo di dato astratto:***

  Il modulo esporta un identificatore di tipo T ed i prototipi delle operazioni eseguibili su dati dichiarati di questo tipo. I "clienti" del modulo dichiarano e controllano quindi il tempo di vita delle variabili di tipo T.

Esempio: tavola come astrazione di dato

```
/* interfaccia - table.h */

typedef char     Key[7];
typedef struct
             {Key chiave;
             char Nome[20];
             char Cognome[20];
             int      Reddito;
             int  Aliquota;} Persone;
void init;
void inserisci(Persone P);
void cancella(Persone P);
int esiste(Key K);
int ricerca(Key K, Persone *el)
```

```
/* implementazione - table.c */
#include "table.h"
#include <string.h>
#define dummy " "
#define  N 100
typedef Persone  Tavola[N];
Tavola T;    /* definizione del dato tavola */
void init
{int i;
 for(i=0;i<N;i++) T[i].chiave=dummy; }
```

```
void inserisci(Persone P);
{int i, trovato=0;
 for(i=0;(i<N) && !trovato; i++)
  if (strcmp(T[i].chiave,dummy)==0)
    trovato=1;
 if (trovato) T[--i]=P; /*inserimento */
}

int esiste(Key K)
{int i, trovato=0;
 for(i=0;(i<N) && !trovato; i++)
  if (strcmp(T[i].chiave, K) == 0)
    trovato=1;
 if (trovato) return i;
 else return 0; }

void cancella(Persone P);
{int i,j;
 if (i=esiste(P.chiave))
  {for(j=i-1;j<N-1;j++)
    T[j]=T[j+1]; /* shift verso l'alto */
   T[j].chiave=dummy; }

int ricerca(Key K, Persone *el)
{int i;
 if (i=esiste(K))
    *el=T[i];        /* el=&T[i] */
 return i; }
```

La variabile di tipo Tavola e` incapsulata all'interno della parte implementazione

```
/* main.c */
#include "table.h"

main()
{Persone P;
 init;
 while (!feof(stdin))
  {leggi_dati(P);
   inserisci(P);}
}
```

Tipo di dato astratto:

- Entita` descrittiva che specifica le caratteristiche (rappresentazione dei dati ed 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 puo' utilizzare costruttori di tipo, ma non estendere il linguaggio con nuove astrazioni.

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

Esempio: tipo di dato astratto tavola

```
/* interfaccia - adt_table.h */
#define  N 100
typedef char     Key[7];
typedef struct
             {Key chiave;
             char Nome[20];
             char Cognome[20];
             int      Reddito;
             int  Aliquota;} Persone;
typedef Persone  Tavola[N];
void init(Tavola T);
void inserisci(Tavola T, Persone P);
void cancella(Tavola T, Persone P);
int esiste(Tavola T, Key K);
int ricerca(Tavola T, Key K, Persone *el)
```

```
/* implementazione - adt_table.c */
#include "adt_table.h"
#include <string.h>
#define dummy " "

void init(Tavola T)
{int i;
 for(i=0;i<N;i++) T[i].chiave=dummy; }
```

```
void inserisci(Tavola T, Persone P);
{int i, trovato=0;
 for(i=0;(i<N)&& !trovato;i++)
  if (strcmp(T[i].chiave,dummy)==0)
    trovato=1;
 if (trovato) T[--i]=P;  /* inserimento */
}

int esiste(Tavola T, Key K)
{int i, trovato=0;
 for(i=0;(i<N)&& !trovato;i++)
  if (strcmp(T[i].chiave, K) == 0)
    trovato=1;
 if (trovato) return --i;}

void cancella(Tavola T, Persone P);
{int i,j;
 if (i=esiste(T,P.chiave))
  {for(j=i;j<N;j++)
    T[j]=T[j+1]; /* shift verso l'alto */
   T[j].chiave=dummy;    }

int ricerca(Tavola T, Key K, Persone *el)
{int i;
 if (i=esiste(T,P.chiave))
    *el=T[i];        /* el=&T[i] */
 return i;   }
```

Non e` un vero e proprio tipo di dato astratto perche` esternamente al modulo viene fornito non solo l'identificatore di tipo Tavola, ma anche la struttura del tipo

In questo caso, le variabili di tipo Tavola sono ***dichiarate da chi utilizza il modulo***

```
/* main.c */
#include "table.h"
main()
{Tavola T;
 Persone P;
 init(T);
 while (!feof(stdin))
  {leggi_dati(P);
   inserisci(T,P);}  }
```

Il linguaggio MODULA-2

- Definito nel 1978 da N. Wirth, come linguaggio per una workstation mono-utente e mono-processore

- Deriva dal Pascal, al quale aggiunge il costrutto MODULO

- Information hiding (derivata da Simula)

- Moduli standard per I/O (TTIO, InOut)

- Primitive per la concorrenza (NEWPROCESS, TRANSFER)

- Struttura di un programma:

  Il "blocco" principale di un programmaModula-2 e` detto modulo principale

  Contiene tutte le sezioni di un programma Pascal ed alcune dichiarazioni che coinvolgono altri moduli

Struttura interna del modulo principale in MODULA-2

```
MODULE nomemodulo;
(* clausole di import/export *)
FROM nomelibreria1 IMPORT identif1;
FROM nomelibreria2 IMPORT identif2;

(* dichiarazioni *)
CONST  c = 2; ...;
TYPE   t = CARDINAL;
VAR        v : t; ... ;

PROCEDURE nomeproc (p1: type1;
(* dichiarazioni locali *)
 VAR p2: type2; ... ): typeret;
BEGIN
(*corpo della procedura *)
END nomeproc; ...

BEGIN
 (* corpo del programma, che
 viene eseguito *)
END nomemodulo.
```

- Modula-2 (come il Pascal) richiede che ogni dato del programma abbia un tipo associato
- Aggiunge nuovi tipi a quelli gia` presenti in Pascal
  - CARDINAL
  - OPEN ARRAY
  - PROCEDURE
  - MODULE
  - PROCESS
- ed alcuni costrutti di controllo del flusso d'esecuzione:
  - LOOP (loop infinito, che termina attraverso l'esecuzione dello statement EXIT).
- Non supporta *overloading*
- Regole di visibilita` Pascal-like che si modificano nel caso di moduli

Blocchi vs Moduli

- Un blocco [procedura] controlla la visibilita` ed il tempo di vita di un oggetto

- Nella progettazione di programmi di grandi dimensioni:
  - c'e` la necessita` di ***separare*** la ***visibilita`*** dei dati dal ***tempo di vita***;
  - c'e` la necessita` di un ***controllo piu` stretto sulla visibilita`***

```
PROCEDURE Outside;
 VAR x,y,z:INTEGER;

 (* nessun modulo *)

 a,b,c:INTEGER;
 PROCEDURE P1;
 BEGIN
 a:=a+1;
 x:=a
 END P1;
 ...
END Outside;
```

```
(* seconda versione *)

PROCEDURE Outside;
 VAR x,y,z:INTEGER;

 MODULE Mod;
 IMPORT x;
 EXPORT a,P1;
 VAR a,b,c:INTEGER;
 PROCEDURE P1;
 BEGIN
 a:=a+1;
 x:=a
 END P1;
 END Mod;
 ...
END Outside;
```

- Gli oggetti dichiarati in **Mod** (**a,b,c,P1**) esistono allo stesso livello delle variabili **x,y,z**

- **a,b,c** sono creati allo stesso tempo di **x,y,z** ed esistono finche` la procedura **Outside** e` attiva

- **Mod** vede solo gli identificatori importati (**x**, ma non **y** e **z**)

- sono visibili esternamente a **Mod** solo gli identificatori esportati (**a** e **P1**, ma non **b** e **c**)

- Dal punto di vista di **`Outside`**, **`a`** e **`P1`** hanno la stessa visibilita` e tempo di vita di **`x`**,**`y`**,**`z`**

- Gli identificatori **`b`** e **`c`** hanno lo stesso tempo di vita, ma risultano nascosti

- I moduli influenzano la visibilita` (fenomeno a tempo di compilazione), ma non il tempo di vita (fenomeno a tempo di esecuzione)

Moduli *vs* Procedure

- Una procedura esprime una necessita' dinamica di una parte di codice
- Dati allocati dinamicamente
- Non sopravvivono alla invocazione della procedura stessa
- Un modulo rappresenta un ***confinamento di ambiente***:
  - Non influenza il tempo di vita delle entita' racchiuse
  - Modifica della visibilita'
- Un modulo puo' essere definito all'interno di una procedura ed essere soggetto alle regole di tempo di vita determinate da questa ultima (ma anche *moduli innestati*)

Regole di visibilita` e tempo di vita nei moduli

- All'interno di un modulo :
  - oggetti dichiarati localmente esistono finche` la procedura che racchiude il modulo rimane attiva;
  - oggetti dichiarati localmente sono visibili nel modulo
  - se compaiono nella export-list, sono visibili anche esternamente
  - oggetti dichiarati esternamente al modulo sono visibili solo se compaiono nella import-list del modulo
- Gli identificatori standard di Modula-2 sono importati automaticamente in ogni modulo (clausole di import)

Clausole di importazione

```
IMPORT NomeMod1, NomeMod2, Ident;
```

- import dell'intero modulo NomeMod1 e NomeMod2 e dell'identificatore Ident

```
FROM NomeMod IMPORT ident1, ident2;
```

- import degli identificatori ident1, ident2 dal modulo NomeMod, equivalenti a quelli dichiarati localmente

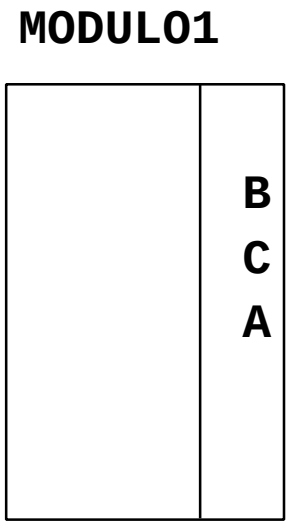
Uso degli oggetti importati

- Un identificatore I importato puo`essere individuato con il nome del modulo come prefisso (riferimento qualificato):

```
NomeModulo.I
```

- oppure essere equiparato alle entita' locali ed usato normalmente con il semplice nome (ovviamente non sono ammessi nomi uguali ripetuti):

```
I := ...
```



- Uso qualificato:

```
IMPORT MODULO1;
...;
Z := MODULO1.B + MODULO1.C();
...
```

- Uso non qualificato:

```
FROM MODULO1 IMPORT B,C;
...;
Z := B + C();
...
```

- Il modo qualificato permette di avere visibilita' di entita' con lo stesso nome.

Clausole di esportazione

```
EXPORT NomeIdent1, NomeIdent2, ...;
 (* export degli identificatori *)


EXPORT QUALIFIED NomeIdent1, NomeIdent2;
 (* export identificatori qualificati *)
```

- L'export del nome di un modulo/ record/ tipo enumerato rende visibile tutti gli identificatori esportati dal modulo/ tutti i campi del record/ tutti i valori del tipo enumerato

```
PROCEDURE Esempio1;
 VAR X:INTEGER;
 MODULE M1;
 EXPORT QUALIFIED X,Y;
 VAR X,Y:REAL;
 BEGIN
 ...
 END M1;

BEGIN
 ...;
 X:= (M1.X MOD M1.Y);
 ...
END Esempio1.
```

Moduli innestati

- In MODULA-2, i moduli possono essere *innestati*, definendo un modulo all'interno di un altro

- I moduli innestati:

  - sono costituiti da una sola parte (come il programma principale);

  - il modulo interno puo' importare tutti e soli gli oggetti che sono visibili nell'ambiente che lo racchiude;

  - il modulo esterno puo' vedere in modo qualificato gli oggetti esportati dal modulo che esso racchiude.

```
MODULE M1;
 IMPORT InOut;
 VAR a: INTEGER;
 b: CHAR;

  PROCEDURE c ();
   VAR a: BOOLEAN;

    MODULE M2;
     FROM InOut IMPORT Write, WriteLn;
     EXPORT QUALIFIED pr, c;
     VAR c, d, e: CHAR;

     PROCEDURE pr;
     BEGIN
     Write (c); Write (d);
     Write (e); WriteLn;
     END pr;
    BEGIN
     c := 'c';
     d := 'd';
     e := 'e';
    END M2;

   BEGIN (* procedure c *)
   M2.pr;
   END c;

BEGIN (* M1 *)
 c;
```

```
END M1.
```

Moduli innestati (cont.)

- Per risolvere ambiguita` si usano riferimenti qualificati

Esempio:

- Definire le procedure Push e Pop che operano su stack di elementi di tipo REAL e su stack di elementi di tipo INTEGER.

- Moduli innestati (uno per REAL ed uno per INTEGER), che esportano Push e Pop:

```
MODULE Esempio6; (* seconda versione *)
 FROM declaration IMPORT size;
 TYPE index = [1 .. size];
 VAR b: BOOLEAN;

 MODULE RealStack;
 EXPORT QUALIFIED Push, Pop;
 TYPE RealS = ARRAY[index] OF REAL;
 VAR RS:RealS; TopRS:index;
 ...
 PROCEDURE Push(X:REAL;VAR b:BOOLEAN);
 ...;
 PROCEDURE Pop(VAR X:REAL;VAR b:BOOLEAN);
 ...;
 BEGIN
 TopRS:=1
 END RealStack;
```

```
 MODULE IntegerStack;
 EXPORT QUALIFIED Push, Pop;
 TYPE IntegerS =
        ARRAY[index] OF INTEGER;
 VAR IS:IntegerS; TopIS:index;
 ...
 PROCEDURE Push(X:INTEGER;
                  VAR b:BOOLEAN);
 ...;
 PROCEDURE Pop(VAR X:INTEGER;
                VAR b:BOOLEAN);
 ...;
 BEGIN
 TopIS:=1
 END IntegerStack;

BEGIN (* main *)
 RealStack.Push(6.7,b);
 IntegerStack.Push(7,b);...
END Esempio6.
```

Moduli innestati: regole di visibilita`

(1) La visibilita` di un identificatore dichiarato in un modulo programma o implementazione e` il blocco in cui compare la dichiarazione e tutti i blocchi di tipo procedura in esso contenuti, fatte salve le regole (2), (6) e (7).

(2) Se un identificatore I visibile nel blocco A, e` ri-dichiarato nel blocco B interno ad A, allora B e tutti i blocchi all'interno di B sono esclusi dalla dichiarazione di I visibile in A.

(3) La definizione di un identificatore standard I e` "pervasiva": l'identificatore e` visibile in qualunque blocco.

(4) Se un identificatore I e` usato nella dichiarazione dell'identificatore J e la dichiarazione di I e` nella stessa unita` di compilazione, allora la dichiarazione di I deve precedere la dichiarazione di J nel testo dell'unita`.

N.B. le liste di import/export non sono dichiarazioni. Una import list in un modulo locale puo` menzionare un identificatore la cui dichiarazione appare in seguito.

(5) La visibilita` di un identificatore (costante, tipo o variabile) dichiarato in un modulo di definizione, e` il blocco del corrispondente modulo di implementazione e tutti i blocchi procedura in esso, fatte salve le regole (2), (6) e (7).

La visibilita` di un tipo opaco e` il modulo definizione in cui e` dichiarato, fatta salva la regola (6).

La visibilita` di una intestazione di procedura in un blocco definizione e` il modulo stesso, fatta salva la regola (6).

(6) Se un identificatore I e` importato da un modulo M, l'ambito di visibilita` della dichiarazione di import comprende M e tutti i blocchi procedura in M, fatta salva la regola (2).

Esempio

```
IMPORT I
FROM N IMPORT I
```

I ed N devono essere visibili nell'ambiente di M.

L'ambiente di un modulo locale e` il blocco che lo racchiude. Per una unita` di compilazione e` l'insieme di tutte le dichiarazioni dei moduli di definizione

(7) Se un identificatore e` esportato senza qualificazione da un modulo M, il suo ambito di visibilita` include, oltre ad M, il blocco che racchiude M e tutti i blocchi procedura racchiusi da M, fatta salva la regola (2).

(8) Se un identificatore I e` esportato da M (con o senza qualificazione), dentro l'ambito di visibilita` di M posso riferirlo come M.I.

N.B. Non ci puo` essere overloading.

Moduli locali

- Dichiarati nel programma principale

```
MODULE ModDemo;
 FROM InOut IMPORT WriteString,
 WriteInt, WriteLn;
 PROCEDURE WriteVal(X:INTEGER);
   BEGIN
   WriteString("Value is: ");
   Write(X,3);
   WriteLn
 END WriteVal;

 MODULE NumGenerator;
 EXPORT NextVal;
 VAR CurrVal:INTEGER;
 PROCEDURE NextVal():INTEGER;
   BEGIN
   INC(CurrVal);
   RETURN CurrVal
 END NextVal;

 BEGIN (* NumGenerator *)
   CurrVal:=0
   END NumGenerator;
 VAR value,count:INTEGER;

BEGIN (* ModDemo *)
 FOR count:=1 TO 10 DO
 value:=NextVal();
 WriteVal(value)
 END;
WriteString("End")
END ModDemo.
```

Moduli compilabili separatamente

- Possono essere divisi in parte di ***interfaccia*** e parte di ***implementazione***

- Le due parti sono collegate dal nome comune e sono sviluppate in modo separato e indipendente

- **Vantaggi:**

  - separazione testuale e compilazione separata;

  - costo minore in fase di modifica;

  - sviluppo parallelo di parti;

  - riutilizzo di codice

Unita` di compilazione

- Moduli programma (Nome.mod), realizzano il modulo di controllo per ciascun programma

- Moduli di definizione (Nome.def), definiscono le utilita` e le interfacce fornite da ciascun modulo server nel progetto

- Moduli di implementazione (Nome.mod), determinano come ciascun modulo server realizza le utilita` che fornisce

- Un programma/modulo importa gli oggetti dichiarati in un modulo compilato separatamente nominandoli nella sua import-list:

```
MODULE ModDemo;
 FROM InOut IMPORT WriteString,
 WriteInt, WriteLn;
```

- I moduli locali possono importare solo oggetti definiti nella export-list di moduli gia` importati dal modulo che li contiene (regole di visibilita`)

- Nell'esempio che segue, il modulo NumGenerator puo` importare alcuni identificatori dal modulo InOut, perche` questo e` stato importato dal modulo ModDemo

```
MODULE ModDemo;
 IMPORT InOut;
 VAR value, count:INTEGER;

 MODULE NumGenerator;
 FROM InOut IMPORT WriteString,
 WriteInt, WriteLn;
 EXPORT WriteVal, NextVal;
 VAR CurrVal:INTEGER;

 PROCEDURE WriteVal(X:INTEGER);
 BEGIN
   WriteString("Value is: ");
   Write(X,3);
   WriteLn
 END WriteVal;

 PROCEDURE NextVal():INTEGER;
 BEGIN
   INC(CurrVal);
   RETURN CurrVal
 END NextVal;

 BEGIN (* NumGenerator *)
   CurrVal:=0
 END NumGenerator;

BEGIN (* ModDemo *)
  FOR count:=1 TO 10 DO
  value:=NextVal();
  WriteVal(value)
  END;
WriteString("End")
```

```
END ModDemo.
```

- Nel modulo di definizione compare una lista di EXPORT QUALIFIED (identificatori visibili all'esterno del modulo) e puo` comparire una lista di import:

```
DEFINITION MODULE StringIO;

 FROM Strings IMPORT String;

 EXPORT QUALIFIED ReadString,WriteString;

 PROCEDURE ReadString(S:String);

 PROCEDURE WriteString(VAR S: String);

END StringIO;
```

- Il modulo implementazione segue la stessa sintassi del programma principale

- Puo` avere piu` liste di IMPORT, ma nessuna lista di EXPORT

- Gli oggetti dichiarati nel modulo X.def sono disponibili automaticamente nel modulo X.mod

- Gli oggetti importati dal modulo di definizione devono essere importati esplicitamente dal modulo implementazione, se necessari

- Il corpo dei moduli implementazione viene eseguito prima che il programma cominci la sua esecuzione

- Il supporto a tempo di esecuzione fa si` che i corpi dei moduli importati siano eseguiti prima di quelli dei moduli che li importano:

```
IMPLEMENTATION MODULE A;
  IMPORT B; ...; END A;

IMPLEMENTATION MODULE B;
  IMPORT C; ...; END B;

IMPLEMENTATION MODULE C;
  IMPORT D; ...; END C;
```

l'ordine di inizializzazione e`:

D C B A

- E` possibile avere IMPORT mutuo, attraverso la divisione di un modulo in definizione e implementazione:

```
IMPLEMENTATION MODULE mod1;
 IMPORT mod2;
 ...
END mod1;

IMPLEMENTATION MODULE mod2;
 IMPORT mod1;
 ...
END mod2;
```

- In questo caso l'ordine di esecuzione del corpo di mod1 e del corpo di mod2 e` indefinito

Esempio : Modulo STACK in Modula-2

```
DEFINITION MODULE stack;
EXPORT QUALIFIED push, pop;
 (* esplicita esportazione di entita' *)
FROM Declaration IMPORT size;
 (* importazione di entita' da un altro
 modulo (declaration): in particolare
 di una costante *)
PROCEDURE push(c:INTEGER; VAR b:BOOLEAN);
PROCEDURE pop(VAR c:INTEGER;
                VAR b:BOOLEAN);
END stack.

IMPLEMENTATION MODULE stack;
TYPE index = [1 .. size];
VAR stack : array [index] of INTEGER;
    top : index;
PROCEDURE push(c:INTEGER; VAR b:BOOLEAN);
 BEGIN
 IF top = size THEN (* stack pieno *)
    b := FALSE;
 ELSE stack[top] := c;
 top := top + 1;
 b := TRUE;
 END (* if *);
END push;
```

```
PROCEDURE pop(VAR c:INTEGER;
                VAR b:BOOLEAN);
 BEGIN
 IF top = 1 THEN (* stack vuoto *)
    b := FALSE;
 ELSE top := top - 1;
 c := stack[top];
 b := TRUE;
 END (* if *);
END pop;
BEGIN
 top := 1; (* inizializzazione *)
END stack.
```

Moduli Utilita`

- In Modula-2 non esistono procedure predefinite nel linguaggio per l'I/O, l'allocazione di memoria o funzioni aritmetiche

- Queste sono fornite da moduli utilita`, che si trovano nella libreria del sistema

- **Vantaggi:**

  - linguaggio piu` semplice

  - compilatore di dimensioni piu` piccole

  - sistema di supporto di dimensioni piu` piccole

  - definizioni alternative di una stessa funzionalita`

- **Svantaggi:**

  - import esplicito

Regole per la compilazione

1) M.def deve essere compilato prima dei moduli (clienti) che importano M

2) M.mod puo` essere ricompilato senza dover ricompilare altri moduli (se non cambia la sua interfaccia)

3) Se M.def (ed M.mod) sono ricompilati si devono ricompilare tutti i moduli clienti (*facilities* per forzare questa regola)

Astrazioni di Dato in Modula-2

- Il costrutto MODULE di Modula-2 non e' equivalente al concetto di astrazione di dato, ma consente di realizzarlo

Esempio: STACK come astrazione di dato

```
DEFINITION MODULE stack;
EXPORT QUALIFIED push, pop;
 PROCEDURE push (c : CHAR;
                  VAR b: BOOLEAN);
 PROCEDURE pop (VAR c : CHAR;
                 VAR b: BOOLEAN);
END stack.
```

Utilizzo:

```
 IMPORT stack; .. ; stack.push()

 FROM stack IMPORT push; .. ; push()
```

```
IMPLEMENTATION MODULE stack;
CONST ssize = 30;
TYPE st = ARRAY [1 .. ssize] OF CHAR;
     sindex = [0 .. 30];
VAR stack : st;  top : sindex;
 (* stack come array *)

PROCEDURE push(c: CHAR; VAR b: BOOLEAN);
 BEGIN
 IF top = ssize THEN (* stack pieno *)
   b := FALSE;
 ELSE
   top := top + 1;
   stack [top] := c;
   b := TRUE;
 END (* if *)
END push;

PROCEDURE pop(VAR c:CHAR; VAR b:BOOLEAN);
BEGIN
 IF top = 0 THEN (* stack vuoto *)
   b := FALSE;
 ELSE
   c := stack [top];
   top := top - 1;
   b := TRUE;
 END (* if *)
END pop;

BEGIN
 top := 0 (* inizializzazione stack *)
END stack.
```

Tipo di Dato Astratto in Modula-2

- Unica entita` descrittiva

Esempio: STACK come tipo di dato astratto

```
DEFINITION MODULE stackadt;

 EXPORT QUALIFIED  stack, push, pop,
                        create;

 TYPE stack; (* tipo opaco *)
 PROCEDURE push    (s: stack; c: CHAR;
                    VAR b: BOOLEAN);
 PROCEDURE pop (s: stack; VAR c: CHAR;
                    VAR b: BOOLEAN);
 PROCEDURE create(VAR s: stack);

END stackadt.
```

- Il tipo stack e` un ***tipo opaco***, la cui rappresentazione e`completamente nascosta

- Il nome del tipo opaco e` visibile ed esportato: gli utenti esterni possono dichiarare variabili di quel tipo senza poter operare direttamente su di esse

- Il tipo di dato astratto e` realizzato in Modula-2 attraverso un modulo compilato separatamente che esporta un insieme di *operazioni* sul *tipo opaco*

- I moduli clienti possono definire variabili di quel tipo, eseguire assegnamenti e invocare le operazioni rese disponibili

- La dichiarazione completa del tipo opaco si trova nel modulo di implementazione (***pointer***)

```
IMPLEMENTATION MODULE stackadt;
 FROM Storage IMPORT ALLOCATE;

CONST ssize = 30;
TYPE
 stackdata = ARRAY [1 .. ssize] OF CHAR;
 sindex = [0 .. 30];
 stackrec = RECORD
 sd : stackdata;
 top : sindex;
 END; (* record *)
 stack = POINTER TO stackrec;
```

```
PROCEDURE push(s: stack; c: CHAR;
 VAR b: BOOLEAN);
BEGIN
 IF s^. top = ssize THEN b := FALSE;
 ELSE s^. top := s^. top + 1;
 s^. sd [s^.top ] := c;
 b := TRUE;
 END (* if *)
END push;

PROCEDURE pop(s: stack; VAR c : CHAR;
 VAR b: BOOLEAN);
BEGIN
 IF s^. top = 0 THEN b := FALSE;
 ELSE c := s^. sd [s^.top];
 s^. top := s^. top - 1;
 b := TRUE;
 END (* if *)
END pop;

PROCEDURE create( VAR s : stack);
BEGIN
 NEW (s);
 s^. top := 0;
END create;
END stackadt.
```