Types abstraits

- Type :
  - Ensemble des valeurs codées par le type
  - Ensemble des opérations que l'on peut effectuer sur les valeurs et variables de ce type
- Il n'est pas nécessaire de connaître la manière dont les valeurs et les opérations sont codées pour pouvoir les utiliser.
- On utilise les types de façon abstraite, sans connaître leur implantation interne
- Ex : Entier + - * /

- Il existe un certain nombre de types scalaires de base
- On souhaite construire de nouveaux types abstraits :
  - Constructibles à partir de types existants
  - Manipulables à travers un jeu d'opérations, **sans avoir à connaître leur structuration interne**
- Ex : Les chaînes de caractères C/C++ sont un piètre exemple de type abstrait
  - Leur utilisation est étroitement liée à leur implantation

- **Description des Types Abstraits dans le cadre de modules**
- Module :
  Regroupe un ensemble de définitions de constantes, de variables globales, de **types**, de procédures et de fonctions qui forment un ensemble cohérent.
  - **Interface du module :**
    Présentation claire des constantes, variables, **types**, procédures et fonctions offertes par le module
  - **Implantation du module :**
    Mise en oeuvre des **types**, procédures et fonctions proposées dans l'interface. Définition des constantes et variables globales du module.

- Pourquoi la séparation interface / implantation ?
  (Déclarations / définitions)
  - Eviter l'introduction (parfois inconsciente) de dépendances entre l'utilisation d'un type et son implantation
    (idem pour les procédures et les fonctions)
  - Possibilité de modifier l'implantation du module sans toucher à son interface
- Si mise en œuvre en C/C++
  - Possibilité de compiler l'implantation du module indépendamment des programmes utilisateurs
  - Possibilité de transmettre uniquement l'interface et l'implantation compilée au programme utilisateur d'un module

- **Modules et types abstraits : Pseudo langage utilisé**
- **module** nom_module { rôle du module}
  - **importer**
    Déclaration des éléments de modules extérieurs **utilisés dans l'interface** de nom_module
  - **exporter**
    Déclaration des éléments **offerts** par nom_module
  - **implantation**
    - **Déclaration** des éléments de modules extérieurs **utilisés dans l'implantation** de nom_module
    - **Définition** éventuelle d'éléments internes au module (utiles pour l'implantation de nom_module mais non exportés)
    - **Définition** des éléments offerts par nom_module
  - **initialisation**
    - Actions à exécuter au début du programme pour garantir une utilisation correcte du module

  finmodule

Eléments = constantes, variables globales (au module), types, procédure et fonction

module nombre_complexe

- **importer**
  Module nombre_réel {offrant le type MonRéel}
- **exporter**
  constante PI : MonRéel
  variable cpteComplexe : entier {nombre de Complexes manipulés initialisés}
  Type Complexe

  procédure initialiser(Résultat c : Complexe) {pré/postconditions ...}
  procédure initialiser(Résultat c1 : Complexe, Donnée c2 : Complexe) {pré/postconditions ...}
  procédure initialiser(Résultat c : Complexe, Donnée a,b : MonRéel) {pré/postconditions ...}
  procédure testament (donnée-résultat Complexe c) {pré/postconditions}

  procédure affectation (donnée-résultat Complexe c1, donnée Complexe c2) {pré/postconditions...}
  procédure affiche (donnée Complexe c) {pré/postconditions ...}
  Complexe **fonction** addition(Complexe c1, Complexe c2) {préconditions et résultat...}
  ...

- **implantation**

```
constante PI : MonRéel
variable cpteComplexe : entier
Type Complexe = structure
                    x  : MonRéel
                    y  : MonRéel
      fin Complexe
procédure initialiser(Résultat c : Complexe, Donnée a,b : MonRéel)
   début
     c.x ←a  c.y←b  cpteComplexe++
   fin initialiser
procédure initialiser(Résultat c : Complexe)
   début
     initialiser(c,0,0)
   fin initialiser
procédure initialiser(Résultat c1 : Complexe, Donnée c2 : Complexe)
   début
     initialiser(c1,c2.x,c2.y)
   fin initialiser
procédure testament (donnée-résultat Complexe c2 )
   début
     cpteComplexe--
   fin testament
```



```
procédure affectation(donnée-résultat  Complexe c1,
                      donnée Complexe c2)
   début
     c1.x ←c2.x
     c1.y←c2.y
   fin affectation
procédure affiche (donnée Complexe c )
   début
     affiche (c.x), affiche(‘’ +i ‘’), affiche (c.y)
   fin affiche
Complexe fonction addition (Complexe c1, Complexe c2)
   début
     Complexe c
     c.x←c1.x+c2.x
     c.y←c1.y+c2.y
     resultat c
   fin addition
```

- **Initialisation**

  PI : MonRéel ← **3,141 59**

  cpteComplexe ← 0

finmodule nombre_complexe

Programme utilisateur

- **importer**

  module nombre_réel, nombre_complexe

- **variable**

  r : MonRéel

  z1 : Complexe

  z2 : Complexe

  z3 : Complexe

- Début

  r←5

  initialiser(z1,r,PI)

  initialiser(z2,z1)

  initialiser(z3)

  affectation(z2, z3)

  affectation(z3,addition(z1,z2))

  affiche(z2)

  testament(z1)

  testament(z2)

  testament(z3)

- Fin

- Précondition de affectation :
  c1 doit avoir été préalablement initialisée
  (sans quoi la valeur de cpteComplexe n’est plus pertinente)

```
procédure affectation (donnée-résultat  Complexe c1,
                       donnée Complexe c2)
   {précondition : c1 initialisé
    postcondition : c1+ = c2}
```

- Pourquoi la procédure initialisation n’a-t-elle pas été appelée dans l’implantation de la fonction addition?

```
Complexe fonction addition (Complexe c1, Complexe c2)
   début
     Complexe c
     c.x←c1.x+c2.x
     c.y←c1.y+c2.y
     résultat c
   fin addition
```


- Parce que la fonction testament ne peut pas être invoquée sur la valeur de retour de l’addition…

Mise en oeuvre en C/C++

- Répartition du module sur 2 fichiers
  - **Interface du module :** Fichier d'entête (.H) Contenant la **déclaration** des éléments importés et exportés

    Le fichier d'entête contient malheureusement aussi les définitions de type …
  - **Implantation du module :** Fichier source (.C)

NombreComplexe.H
Fichier de promesses!

```
#ifndef __NBCOMP
#define __NBCOMP
#include ''NombreReel.H'' //type MonReel
  extern const MonReel PI;
  extern int cmpteComplexe;
  struct Complexe
   {
     MonReel x,y;     //données membres
   };

// Procédures d'initialisation
    void initialiser(Complexe & c);
     // Pré/postconditions…initialisation par défaut
   void initialiser(Complexe & c1, const Complexe & c2);
    //Pré/postconditions initialisation par copie
   void initialiser(Complexe & c, const MonReel & a, const MonReel & b);
      // Pré/postconditions initialisation à partir de 2 réels

   void testament(Complexe & c);
   //Pré/postconditions…

    void affectation (Complexe & c1, const Complexe & c2);
   //Pré/postconditions…

   Complexe addition(const Complexe & c1, const Complexe & c2);
   void affiche(const Complexe & c); //Préconditions et résultat
#endif
```


NombreComplexe.C

```
#include ''NombreReel.H''
#include ''NombreComplexe.H ''
  const MonReel PI=3,141 59;
  int cmpteComplexe=0;

  void initialiser( Complexe &c ) {c.x=c.y=0; cmpteComplexe++;}
  void initialiser( Complexe &c1, const Complexe & c2) {…}
  void initialiser( Complexe &c, const MonReel& a, const MonReel& b)
  {…}

  void testament(Complexe & c) {cmpteComplexe--;}

  void affectation(Complexe& c1, const Complexe & c2)
    {
      c1.x=c2.x;
      c1.y=c2.y;
    }

  Complexe addition(const Complexe & c1, const Complexe & c2)
    {…}

  void affiche(const Complexe & c)
    {…}
```


main.C

```
#include ''NombreReel.H''
#include ''NombreComplexe.H ''
int main()
{
   MonReel r=5;
   Complexe z1, z2, z3;
   initialiser(z1,r,PI);
   initialiser(z2,z1);
   initialiser(z3);
   affectation(z2,addition(z1,z3));
   affiche(z2);
   testament(z1);
   testament(z2);
   testament(z3);
   return 0;
}
```


Mot clef *static*

- Utilisation pour définir des éléments **internes** à un module
  - Éléments non exportés (non utilisables dans d'autres modules)
- Exemple :
  - Pour l'instant la variable globale cmpteComplexe du module NombreComplexe est exportée,
  - Possibilité de la modifier depuis un programme utilisateur et donc de lui faire perdre son intégrité
  - Il serait préférable :
    - Que cmpteComplexe soit une variable globale interne au module NombreComplexe
    - et que l'on exporte juste une fonction permettant d'accéder à sa valeur depuis un programme utilisateur

NombreComplexe.H
Fichier de promesses!

```
#ifndef __NBCOMP
#define __NBCOMP
#include ''NombreReel.H'' //type MonReel

   extern const MonReel PI;

   // La variable globale cmpteComplexe a été retirée de l'interface
   // au profit d'une fonction d'accès à sa valeur

   struct Complexe
   {
    MonReel x,y;//données membres
   };

   // Procédures, fonctions et opérations offertes sur le type abstrait
   Complexe
    …
   int nbComplexesVivants();
    //Préconditions : aucune
   //Résultat : nombre de Complexes existant en mémoire au moment de
   l'appel
#endif
```

NombreComplexe.C

```
#include ''NombreReel.H''
#include ''NombreComplexe.H ''
#include <iostream>

  const MonReel PI=3,141 59;
  static int cmpteComplexe=0; //définition variable globale interne

   //On peut aussi définir des fonctions internes à un module
   static int exempleFonctionInterne()
    {
      std::printf ( '' Hello \n'');
      return 8;
     }

  //Définition des procédures, fonctions et opérations
  // offertes sur le type abstrait Complexe
  ...
  int nbComplexesVivants()
  {
    return cmpteComplexe;
  }
```

cmpteComplexe et exempleFonctionInterne accessibles **uniquement** dans NombreComplexe.C