UE Socle Commun Informatique.
Programmation Impérative

Yamine Ait-Ameur, Xavier Crégut, Katia Jaffrès-Runser

ENSEEIHT
Département Sciences du Numérique,
{yamine, cregut, kjr}@n7.fr

Année 2020-2021

Plan

Plan

- Définitions et terminologie
- Développement de programmes
- Exemples de programmes complexes
- Le cours PIM

Plan

Définitions : informatique

Qu'est-ce que l'Informatique ?

- Traitement automatique de l'information grâce à
  - l'exécution
  - de programmes informatiques
  - sur des ordinateurs, robots (machines)

On retiendra de cette définition

- **information** : il faut la représenter, la modéliser, la formaliser, la coder, etc. par l'humain.
- **programme** : il faut les écrire, par l'humain ou bien les générer par d'autres programmes (compilateur, traducteur etc. par exemple)
- **exécution** : réalisation des actions élémentaires. Il faut rendre les programmes compréhensibles par les machines

Définitions : Processeur

Processeur

- Machine abstraite ou concrète
- Associée à des entités, des concepts qu'il manipule,
- Exécute des actions élémentaires **comprises** par ce processeur ou cette machine

Exemple

- Un ordinateur $\Longrightarrow$ Machine
- Des constantes et des variables $\Longrightarrow$ Entités et concepts
- Instructions d'un langage de programmation $\Longrightarrow$ Actions élémentaires

Définitions : Programme

Définition

Un programme est une suite finie d'instructions pré-déterminées destinées à être exécutées de manière automatique par un processeur en vue d'effectuer des traitements, impliquant généralement une interaction avec son environnement.
Synonymes : Application, logiciel.

Exemples de programmes

- Toutes les applications qui s'exécutent sur un ordinateur : traitement de texte, navigateur Internet, messagerie, ...
- Toutes les applications qui s'exécutent sur un smartphone,
- ou sur des consoles de jeu, des imprimantes, une carte réseau, un GPS, ... ,
- les guichets automatiques bancaires (GAB), le système automobile (injection, ABS...), un pilote automatique, ou un robot.

Définitions : Programme

Définition

Un programme est une suite finie d'instructions pré-déterminées destinées à être exécutées de manière automatique par un processeur en vue d'effectuer des traitements, impliquant généralement une interaction avec son environnement.
Synonymes : Application, logiciel.

Exemples d'environnements

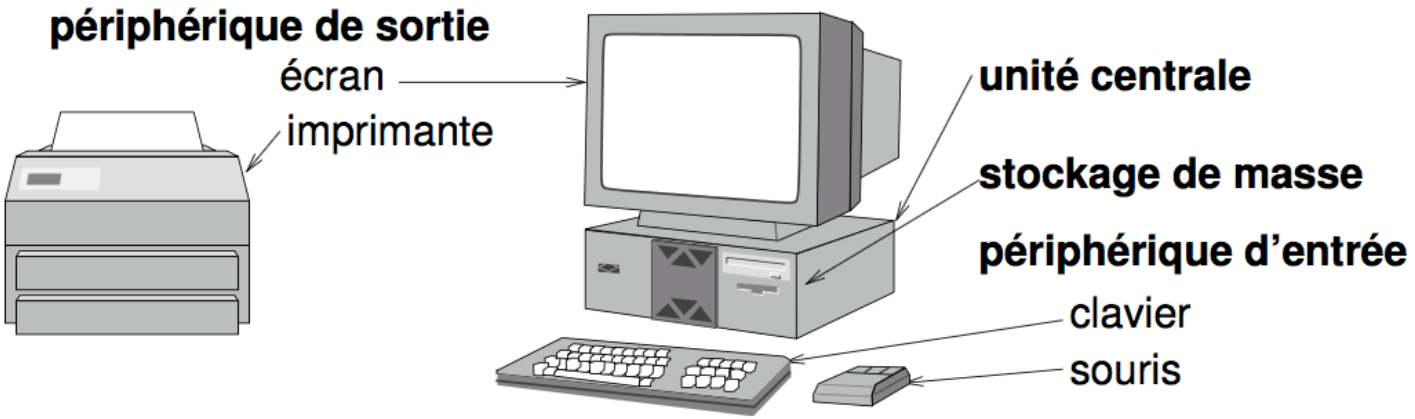
- Utilisateur humain : traitement de texte, SMS..
- Un autre système informatique : navigateur internet, guichet automatique bancaire, réseau, etc.
- Des éléments physiques : capteurs et actionneurs (ABS, régulateur vitesse).

Programme Impératif

- Programme constitué d'une suite d'ordres (actions) exécutés par un ordinateur.
  On parle aussi de style **impératif**. Il existe d'autres types de programmes qui ne sont pas impératifs.

Définitions : Ordinateur

Point de vue de l'utilisateur final



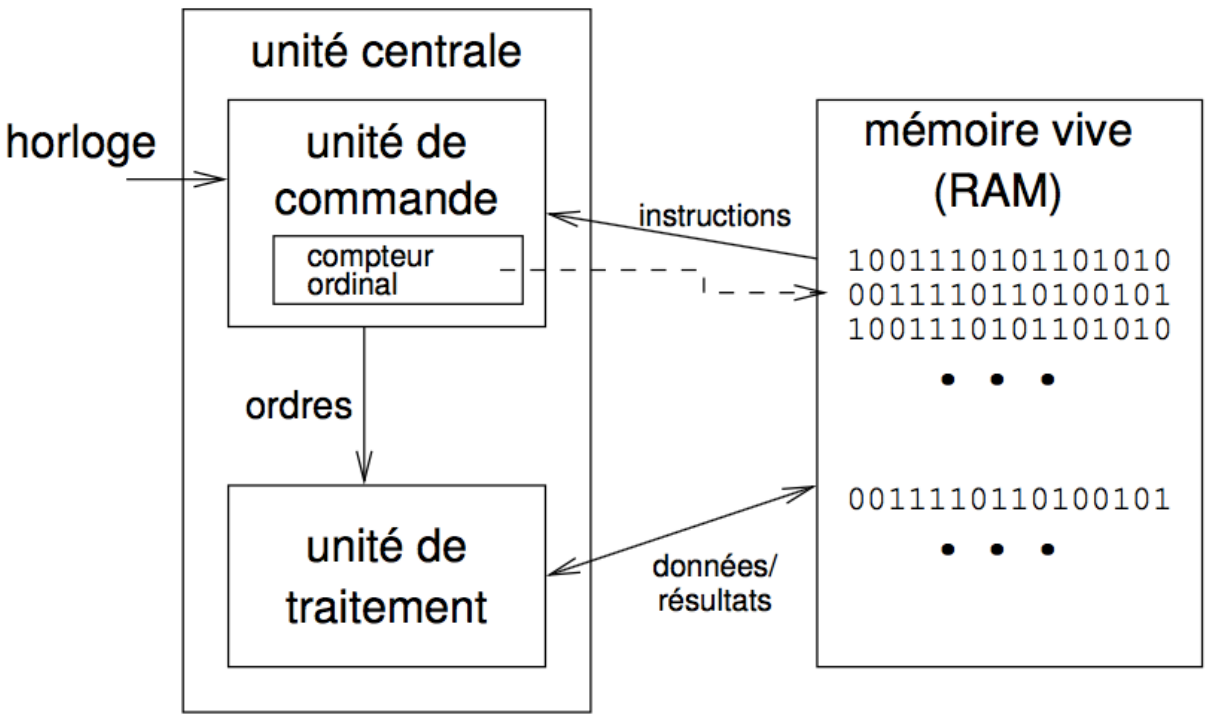
1SN "Environnement informatique"

- Environnement graphique multi-fenêtré
- Terminal et interpréteur de commandes
- Principales commandes

Définitions : Ordinateur

Architecture simplifiée d'un ordinateur

Architecture de von Neumann



http://www.grappa.univ-lille3.fr/polys/intro-info/index.html

Définitions : Langage machine

Les instructions d'un programme sont enregistrées dans la mémoire vive en *langage machine*.

En binaire :

```
11100000000000000000000000001000
11110001000000000000000000000000
11100000000000000000000001000111
11110001000000000000000000000001
11100000000000000000000001101111
11110001000000000000000000000010
11100000000000000000000000100000
11110001000000000000000000000011
11100000000000000000000000100001
11110001000000000000000000000100
11100000000000000000000000001010
11110001000000000000000000000101
11100000000000000000000000000000
11110001000000000000000000000110
11100000000000000000000000001010
11110001000000000000000000000111
11100000000000000000000000000000
11110001000000000000000000001000
10000000000000000000000000010011
11100001000000000000000000000000
11000010000000000000000000000000
11001000000000000000000000000111
01000010000000000000000000000000
10010100000000000000000000010100
11001000000000000000000000000001
```

En hexadécimal :

```
E0000008
F1000000
E0000047
F1000001
E000006F
F1000002
E0000020
F1000003
E0000021
F1000004
E000000A
F1000005
E0000000
F1000006
E000000A
F1000007
E0000000
F1000008
80000013
E1000000
C2000000
C8000007
42000000
94000014
C8000001
```

Définitions : Langage de programmation

Dans un pseudo-langage assembleur

Le même programme est ici énoncé dans un langage de programmation assembleur.

```
; Auteur   : Xavier Crégut <Prenom.Nom@enseeiht.fr>
; Version  : 1.2
; Objectif : Réaliser un décompte (de taille DUREE) et afficher TEXTE.
        JMP MAIN        ; Aller au réel début du programme
DUREE   DW  1   8       ; Durée du décompte
TEXTE   DW  0   "Go␣!\n"        ; Message à afficher
RL      DW  0   "\n"    ; Un retour à la ligne

MAIN:   LD DUREE ACC    ; Charger la valeur de durée dans l'accumulateur
DEBUT:  PRi ACC         ; Afficher le temps restant
        PRs RL          ; Afficher un retour à la ligne
        DEC ACC         ; Décrémenter le compteur
        BNE DEBUT       ; Continuer le décompte ?
        PRs TEXTE       ; Afficher le texte
        STOP
```

Définition

Un langage de programmation est une notation conventionnelle destinée à formuler des algorithmes et produire des programmes informatiques qui les appliquent.

Définitions : Langage de programmation

Quelques types de langages

- Les *langages machine* : définissent le jeu d'instructions élémentaires correspondant aux capacités d'un processeur.
- Les *langages assembleur* : gèrent les adresses logiques (étiquettes), déchargent le programmeur du positionnement du programme en mémoire.
  → "Architecture des ordinateurs".
- Les *langages structurés* : s'appuient sur les structures de contrôle (conditionnelle, répétition) pour éviter la profusion de branchements (et les programmes spaghettis). Structuration en sous-programmes et modules.
  Exemples : Pascal (1969), C (1972), Ada (1983), etc.
  → "Programmation Impérative".
- Les *langages objets* (années 80) : regrouper les données et les traitements.
  Exemples : SmallTalk (1980), C++ (1983), Java (1995), Python (1990)...
  → "Technologies Objets".
- Les *langages dédiés* (Domain specific langagen, DSL) : dédiés à des technologies spécifiques, contrairement aux langages généralistes (General purpose langage, GPL).
  Exemples : Matlab, VHDL, JavaScript, etc.
  Ingénierie des DSL → "Génie du Logiciel et des Systèmes".

Définition : Langage de programmation

Langages et styles de programmation

Ces langages de programmation permettent d'utiliser un ou plusieurs styles de programmation de façon plus ou moins aisée.
Une classification possible de différents styles de programmation :

- Programmation impérative
  - Ada, Pascal, C, Python, Fortran, $\cdots$
- Programmation fonctionnelle
  - Caml, Lisp, Haskell, $\cdots$
- Programmation objet
  - Java, C++, Ada, Python, Fortran, $\cdots$
- Programmation logique
  - Prolog, Datalog, $\cdots$

On notera par exemple qu'Ada permet d'adopter un style de programmation impératif mais aussi un style objet.

Définitions : Compilateur / Interpréteur

Compilateur

Il traduit un programme écrit dans un langage L1 en un programme *équivalent* écrit dans un langage L2.
Exemple : Le plus souvent, on l'utilise pour traduire un langage de haut niveau (Ada, C, ..) en un langage que l'on peut exécuter sur une machine donnée (le langage machine par exemple).
cf. 1SN "Programmation Impérative" **PIM**.

Interpréteur

Un interpréteur est capable de comprendre (interpréter) un programme écrit dans un langage L1 et de l'exécuter directement sur une machine donnée.
Exemples : L'interpréteur de commandes (shell), interpréteur Python.
cf. 1SN "Architecture des Ordinateurs" (Novembre)

Note : Compilateur et interpréteur sont eux-mêmes des programmes.

Définitions : Algorithme et langage algorithmique

Algorithme

- Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions élémentaires qui permettent de réaliser une action abstraite.

Un algorithme est une méthode générale pour résoudre un type de problèmes. Il est dit correct lorsque, pour chaque instance du problème, il se termine en produisant la bonne sortie, c'est-à-dire qu'il résout le problème posé.

On retrouve des algorithmes dans différents domaines scientifiques, mais aussi dans la vie courante (recettes de cuisine, notices d'assemblage, etc).

Langage algorithmique

L'algorithme peut se formaliser de différentes manières selon l'usage. Nous utiliserons un langage algorithmique dans ce cours.

Définitions : Algorithme et langage algorithmique

Exemple d'algorithme

```
1  Algorithme périmètre_cercle
2
3      -- Déterminer le périmètre d'un cercle à partir de son rayon
4      -- Attention : aucun contrôle sur la saisie du rayon ==> non robuste !
5
6  Constante
7      PI = 3.1415
8
9  Variable
10     rayon: Réel          -- le rayon du cercle lu au clavier
11     périmètre: Réel      -- le perimètre du cercle
12
13 Début
14     -- Saisir le rayon
15     Écrire("Rayon = ")
16     Lire(rayon)
17
18     -- Calculer le périmètre
19     périmètre <- 2 * PI * rayon          -- par définition
20     { périmètre = 2 * PI * rayon }
21
22     -- Afficher le périmètre
23     Écrire("Le périmètre est : ", périmètre)
24 Fin
```

Nous introduirons plus tard le langage algorithmique utilisé pour ce cours.

Définitions : Algorithme et langage algorithmique

Pourquoi utiliser un langage algorithmique dédié ?

- Pour bien faire la différence entre programmation et construction d'une solution algorithmique,
- Être indépendant des contraintes d'un langage de programmation particulier,
- Utiliser (intégrer) des concepts/constructions remarquables issus de plusieurs langages :
  - Les structures de contrôle de Modula 2,
  - Les modes **in** , **out** et **in out** de Ada,
- Favoriser la créativité avec un langage plus souple (permettant par exemple de définir des instructions abstraites) mais suffisamment rigoureux pour que tout le monde puisse comprendre un algorithme écrit dans ce langage.

Plan

Développement de programmes

Comment obtenir un programme ? $\Longrightarrow$ par un **développement**

- À partir d'un **cahier des charges** exprimant **besoins et exigences**,
- Une **spécification**, la plus précise possible, est produite
- Elle est **décomposée, raffinée** (en plusieurs étapes) jusqu'à obtenir
- Un **algorithme** qui répond à la spécification.
- Il est ensuite traduit ou codé dans un **langage de programmation**.

Durant les différentes étapes de développement, des activités de

- validation
- et de vérification

sont nécessaires afin de garantir la **qualité** du programme obtenu

Le développement d'un programme est une activité d'ingénierie
$\Longrightarrow$ besoin d'**ingénieurs**

Génie Logiciel

Développement de programmes

Démarche générale

- Poser le problème
- Le résoudre
- Démontrer la correction

En Informatique

- poser le problème = **Spécification**
- le résoudre = **Conception et Codage**
- démontrer la correction = **Vérification & Validation**

Cette démarche sera détaillée dans méthode des raffinages (D-2).

Développement de programmes

Cycle de développement d'un programme

Principales étapes

- **Spécification.** Il s'agit de définir clairement le problème en langue naturelle ou à l'aide d'une spécification formelle.
- **Conception.**
  - Organiser les données et entités manipulées
  - Concevoir les algorithmes par la méthode des raffinages,
  - et les Écrire/Présenter en langage algorithmique
- **Codage.** Écrire les algorithmes et données dans un langage Informatique (Ada, Python, Java, Caml ...).
- **Vérification et Validation.** Tests unitaires, fonctionnels et de couverture des programmes.

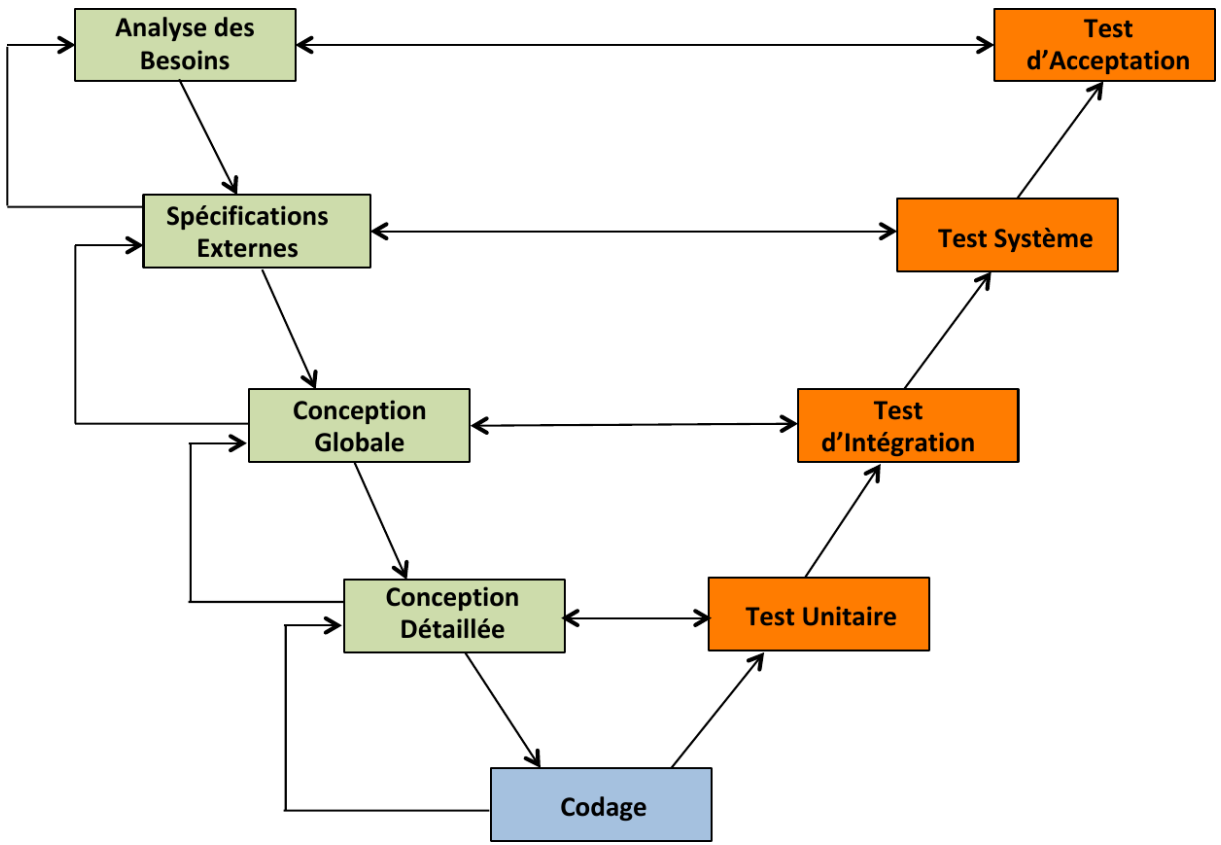
Note :

D'autres étapes sont également présentes dans le cycle de développement. Elles ne sont pas abordées dans ce cours (maintenance, simulation, etc.).

Développement de programmes

Cycle de développement

- Cycle en V : étapes nécessaires pour construire un programme, le valider et le vérifier
- Deux branches : **conception** (Gauche) et **validation et vérification** (Droite)
- Validation et Vérification sont des activités transversales
- Activité de l'ingénieur. **Ingénierie du logiciel**, **Génie Logiciel**, etc.



Développement de programmes : Spécification

Enoncé informel (pour l'humain)

Un algorithme produisant, à partir d'une donnée (entrée E), un résultat S en résultat (sortie)

Enoncé Logique (pour la validation et la vérification)

$$\forall E \ \ \exists S. \quad P(E, S)$$

$$\forall E \ \ \exists S. \quad S = F(E)$$

Enoncé formel (pour l'ordinateur)

```
Begin
   Lire (E) ;
   Programme_Calcul;
   S := Result_Calcul;
   Ecrire (S);
End;
```

Développement de programmes : Conception

Exemples d'algorithmes (1)

- Considérons deux **actions abstraites**

Action 1	**Action 2**
Calculer la somme de deux nombres à deux chiffres non signés	Tracer un carré de 100 unités par côté

- Il s'agit de la description d'une action à réaliser par le processeur
- Si le processeur à notre disposition est capable de réaliser ces actions abstraites, alors l'algorithme correspond à cette action.
- Sinon, il faut **décomposer/raffiner** jusqu'à atteindre une définition formée d'actions élémentaires comprises par le processeur.

Développement de programmes : Conception

Exemples d'algorithmes (1)

- Processeur

  Humain || Table traçante

- Objets manipulés

  Chiffre || Colonne

- Actions du processeur

  additionner des chiffres, écrire placé, tracer un trait || avancer, tourner droite

Développement de programmes : Conception

Exemples d'algorithmes (1 - Suite)

- Algorithme
  - Ecrire les deux nombres l'un sous l'autre, alignés
  - Tracer un trait sous le deuxième nombre
  - Additionner les chiffres des unités
  - Ecrire dans la colonne des unités, sous le trait, le nombre des unités de la somme
  - Ecrire dans la colonnes des dizaines, au dessus du premier chiffre, la retenue (i.e. le nombre des dizaines de la somme : 1 ou 0)
  - Additionner les chiffres des dizaines et la retenue
  - Ecrire dans la colonne des dizaines, sous le trait, le nombre des unités de la somme
  - Ecrire dans la colonnes des centaines, sous le trait, la retenue (i.e. le nombre de dizaines de la somme : 1 ou 0)
  - AVANCER 100
  - TOURNER DROITE 90
  - AVANCER 100
  - TOURNER DROITE 90
  - AVANCER 100
  - TOURNER DROITE 90
  - AVANCER 100
  - TOURNER DROITE 90
  - POUR i DE 1 À 4 FAIRE
    - AVANCER 100
    - TOURNER DROITE 90
  - FIN POUR

Développement de programmes : Conception

Exemples d'algorithmes (2)

- Action abstraite : calculer une factorielle
- S'il existe un processeur capable de réaliser ces actions abstraites, alors l'algorithme correspond à cette action.

  Par exemple une calculatrice, une bibliothèque de fonctions, etc.

- Sinon, il faut **décomposer/raffiner** jusqu'à atteindre une définition formée d'actions élémentaires comprises par le processeur.

Développement de programmes : Conception

Exemples d'algorithmes (2)

- Processeur : Ordinateur
- Objets manipulés : entiers, chaînes de caractères, ...
- Actions du processeur : instructions d'un langage impératif, affectation, séquence, conditionnelle, répétition
- Algorithme écrit dans un langage algorithmique (pseudo-langage) pour être ensuite transcrit dans un langage de programmation
- Correction de l'algorithme

  $\Longrightarrow$ il faut garantir que $resultat = n!$ en fin de programme

Développement de programmes : Conception

Le programme de la factorielle en langage algorithmique :

```
1   PROCEDURE Appel_Factorielle EST
2
3       -- Calculer la factorielle d'un entier positif.
4       -- Paramètre N, N entier, donnée
5       -- Précondition N >= 0
6       FONCTION Factorielle(N : IN Entier) RETOURNE Entier EST
7           Résultat : Entier     -- produit des entiers de 1 à N
8       DÉBUT
9           Résultat <-- 1
10          POUR i DE 2 À N FAIRE
11              Résultat <-- Résultat * i
12          FIN POUR
13          RETOURNE  Résultat
14      FIN Factorielle
15
16  DÉBUT
17      Écrire ("4! = ", Factorielle (4))
18  FIN Appel_Factorielle
19
```

- Il s'agit d'un algorithme/programme écrit dans un langage algorithmique pour être ensuite transcrit dans un langage de programmation.
- Cet algorithme utilise un sous-programme, ici une fonction.

Développement de programmes : Codage

Le programme de la factorielle en Ada

```
1   with Ada.Text_IO;           use Ada.Text_IO;
2   with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;
3
4   procedure Appel_Factorielle is
5
6       -- Calculer la factorielle d'un entier positif.
7       -- Paramètre N, N entier, donnée
8       -- Précondition N >= 0
9       function Factorielle (N: in Integer) return Integer is
10          Resultat: Integer ; -- produit des entiers de 1 à N
11      begin
12          Resultat := 1 ;
13          for I in 2..N loop
14              Resultat := Resultat * I;
15          end loop;
16          return Resultat;
17      end Factorielle;
18
19  -- Programme principal
20  begin
21      Put ("4! = ");
22      Put (Factorielle (4), 1);
23      New_Line;
24  End Appel_Factorielle;
25
```

Développement de programmes : Codage

Le programme de la factorielle en Python

```
1   def factorielle(n):
2       '''
3       Calculer la factorielle d'un entier positif.
4       Paramètre n, n entier
5       Précondition n >= 0
6       '''
7
8       resultat = 1
9       for i in range (2, n + 1):
10          resultat = resultat * i
11      return resultat
12
13
14  print("4! =", factorielle(4))
```

Développement de programmes : Codage

Le programme de la factorielle en C

```c
#include <stdio.h>
#include <stdlib.h>

/**
 * Calculer la factorielle d'un entier positif.
 * \param[in] n, l'entier positif
 * \return la factorielle de n
 * \pre n positif : n >= 0
 */
int factorielle(int n) {
    int resultat = 1;
    for (int i = 2; i <= n; i++) {
    resultat = resultat * i;
    }
    return resultat;
}


int main() {
    printf("4! = %d\n", factorielle(4));
    return EXIT_SUCCESS;
}

```

Développements de programmes : problèmes fondamentaux

Problèmes fondamentaux en algorithmique

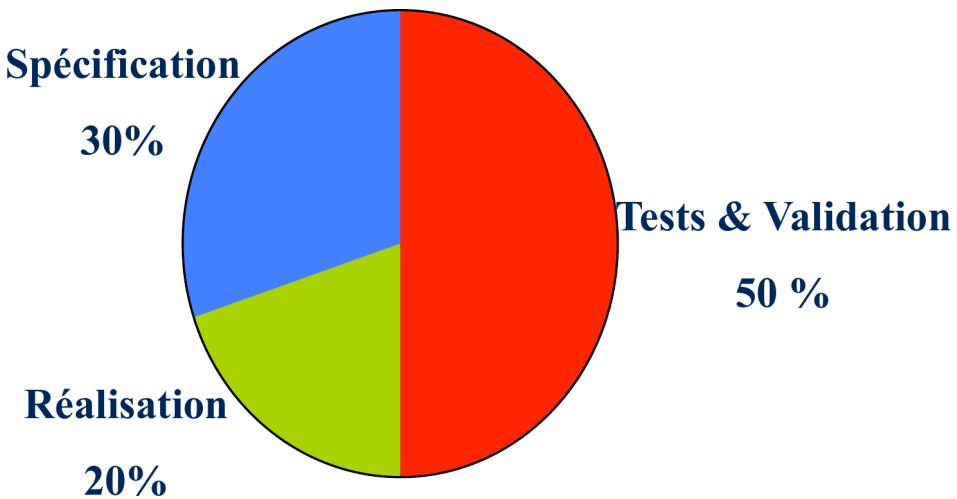
- Modélisation
- Validation
- Vérification
- Complexité
- Calculabilité

Plan

Exemples de programmes complexes : Bilan d'un développement

Cas de l'avionique du Rafale (source Dassault Aviation)

- Pour une étape de développement de 60 000 lignes de code Ada
  - 24 mois de la spécification détaillée jusqu'à la livraison (code + calculateur)
  - 30 à 40 personnes
  - 260 fiches de modification
- Répartition des activités de développement



Exemples de programmes complexes : Ordres de grandeur

Mesure en nombre de lignes de code (Loc)

- 1 Million de loc $\simeq$ 18 000 pages imprimées

Des exemples de gros programmes

- Google Chrome $\simeq$ 34 Millions de Loc
- Avionique et systèmes embarqués du Boeing 787 $\simeq$ 35 Millions de Loc
- Android $\simeq$ 120 Millions de Loc
- MySQL $\simeq$ 120 Millions de Loc
- Linux 3.1 $\simeq$ 170 Millions de Loc
- Microsoft office 2013 $\simeq$ 1, 1 Milliards de Loc

Exemples de programmes complexes : Ordres de grandeur

Mesure en nombre de lignes de code (Loc)

- 1 Million de loc $\simeq$ 18 000 pages imprimées

Des exemples de gros programmes

- Google Chrome $\simeq$ 34 Millions de Loc
- Avionique et systèmes embarqués du Boeing 787 $\simeq$ 35 Millions de Loc
- Android $\simeq$ 120 Millions de Loc
- MySQL $\simeq$ 120 Millions de Loc
- Linux 3.1 $\simeq$ 170 Millions de Loc
- Microsoft office 2013 $\simeq$ 1, 1 Milliards de Loc

Ces programmes constituent des produits à part entière nécessitant

- des techniques de l'ingénieur
- la gestion de projets

Plan

Méthode de conception

Méthode des raffinages

- Mise en oeuvre de méthode des raffinages pour concevoir des programmes de style impératif.

Pratique

- Utiliser cette méthode pour des problèmes immédiats
  - Conception de *"petits"* programmes : Programmation structurée
  - Conception d'architectures de sous-programmes
  - Conception *"modulaire"*
  - Conception *"orientée objet"* (2ème semestre)

Le langage de programmation ADA

Choix du langage Ada

- Initié par le DoD dans les années 70 développé par l'équipe de Jean ICHBIAH
- Aboutissement à Ada83, puis Ada95

Quelques points forts

- Langage avec typage fort et statique
- Lisibilité des programmes écrits en Ada
- Qualité et rigueur du compilateur
- Traitement de la généricité et des exceptions
- Encapsulation
- Séparation de la spécification et de l'implantation

Environnements de développement

- Disponibilité de compilateurs GNAT(Linux, MacOS) ou AdaGide (windows)
- Utilisation de GPS (GNAT Programming Studio) diffusé par AdaCore

Compétences acquises

Compétences attendues à l'issue de ce cours

1. Algorithmique, Raffinages et Programmation
   - Etre capable de comprendre un problème,
   - Etre capable de structurer une solution : raffinages,
   - Etre capable de le programmer : langage algorithmique et Ada.
2. Architecture d'un programme
   - Savoir structurer les données : types utilisateurs (énumérés, enregistrements et tableaux),
   - Savoir structurer les traitements : sous-programmes (procédures et fonctions),
   - Savoir structurer les applications : les modules (paquetages).
3. Concepts avancés
   - Gestion de la mémoire dynamique,
   - Connaitre les structures de données usuelles (pile, liste, arbre ...),
   - Savoir utiliser la programmation par contrat,
   - Savoir utiliser les exceptions,
   - Avoir une notion de la complexité d'un algorithme,
   - Mener des tests pour valider son application.

Lien avec les autres UE

La matière "programmation impérative" est en relation avec les UE d'informatique de base

- Architecture des ordinateurs
- Programmation orientée objets (TOB)
- Programmation fonctionnelle
- Modélisation

Les notions suivantes seront abordées dans cette UE :

- Spécification
- Raffinage
- Preuve
- Test

Plan

- Anatomie d'un programme simple
- Expressions
- Instructions simples
- Entrées / Sorties
- Structures de contrôle
- Bonnes pratiques de programmation

Plan

Exemple de programme

```
1   Algorithme périmètre_cercle
2
3       -- Afficher le périmètre d'un cercle dont le rayon du cercle est saisi.
4       -- Attention : aucun contrôle sur la saisie du rayon ==> non robuste !
5
6   Constantes
7       PI = 3.1415
8
9   Variables
10      Rayon: Réel         -- le rayon du cercle lu au clavier
11      Périmètre: Réel     -- le perimètre du cercle
12
13  Début
14      -- Saisir le rayon
15      Écrire("Rayon ? ")
16      Lire(Rayon)
17
18      -- Calculer le périmètre
19      Périmètre <- 2.0 * PI * Rayon    -- par définition
20
21      { Périmètre = 2.0 * PI * Rayon }
22
23      -- Afficher le périmètre
24      Écrire("Périmètre = ", Périmètre)
25  Fin
```

Commentaires

- **Définition :** Annotations **dans un programme** à destination d'un lecteur humain
- **Objectif :** Aider le lecteur à **comprendre** le programme et à **raisonner** dessus

Commentaires pour comprendre le programme

- **Exemple :** `-- Ceci est un commentaire de type explication`
- **Notation :** de deux tirets consécutifs jusqu'à la fin de la ligne
  - Placé devant un groupe de lignes, il explique l'objectif de ces lignes

```
-- Saisir le rayon
Écrire("Rayon ? ")
Lire(Rayon)

-- Calculer le périmètre
Périmètre <- 2.0 * PI * Rayon

-- Afficher le périmètre
Écrire("Périmètre = ", Périmètre)
```

  - Placé en fin de ligne, il explique, justifie le code de cette ligne

```
Périmètre <- 2.0 * PI * Rayon    -- par définition
```

  - Ils sont écrits en langage naturel, voir la méthode des raffinages (p. D-2)
- **Attention :** Le commentaire ne doit ni paraphraser, ni être redondant avec le code !

Commentaires (2)

Commentaires pour raisonner sur le programme

1. **Exemple :** *{ Ceci est un commentaire de type propriété}*
2. **Notation :** entre accolades
3. **But :** Exprimer une propriété sur le programme (généralement, écrit formellement) :
   - `{ Périmètre = 2.0 * PI * Rayon }`
   - `{ n < 0 ET x = 0.0 }`
4. Voir programmation par contrat (p. E-2)

Commentaires pour les apprenants

- Nous utiliserons les commentaires `--!` pour donner une explication qui n'aurait normalement pas sa place dans un programme.
- Exemple :

```
Lire(Rayon)    --! demander à l'utilisateur la valeur du rayon
```

  - Ce commentaire paraphrase le code et est donc à ce titre inutile.
  - Il ne devrait donc pas apparaître dans un vrai programme.
  - Cependant, dans le cadre d'un cours, il peut être intéressant de donner cette information pour expliquer `Lire`.
  - Nous utiliserons donc `--!` dans ce cas.

Identificateurs

- **Définition :** C'est le nom donné à une entité définie dans un programme
- **Intérêt :** nommer les entités (programme, constante, type, variable...) pour permettre :
  - à l'ordinateur de distinguer les entités (nom différents)
  - aux humains de comprendre le rôle, l'objectif de ces entités (noms significatifs)

Règles

- Un identificateur commence par une lettre, suivie de chiffres, lettres ou soulignés (_)
- Un identificateur doit être significatif (i.e. avoir du sens pour le lecteur)
- Pas de distinction majuscule/minuscule en langage Algorithmique (ni en Ada)
  - un souligné entre chaque partie (`Prix_HTT`)
  - une majuscule au début de chaque partie
- Exemples : `Rayon, Prix_TTC, N1, N2, Element_Absent`
- Identificateurs à éviter : `R` (trop court), `Le_Rayon_Du_Cercle` (trop long)
- Mauvais identificateurs :
  - `5N` (ne commence pas par une lettre)
  - `F(2)` (pas de parenthèses possibles)

Constituants

Le reste de ce premier exemple montre les éléments qui seront décrits dans la suite :

- des constantes littérales : `3.1415`, `"Rayon ? "`
- des constantes symboliques : `PI`
- des variables : `Rayon`, `Périmètre`
- des instructions : `Écrire`, `Lire`, `<-`
- des expressions : `2.0 * PI * Rayon`

Plan

Types

- Un type définit et un **ensemble de valeurs** et les **opérateurs** associés.
- Un type est identifié par son **nom** (identificateur)

Exemple : le type Entier

- valeurs : les entiers relatifs (-5, 0, 1, 2048...)
  - généralement bornés (par exemple intervalle $[-2^{31} .. 2^{31} - 1]$)
- opérateurs : arithmétiques (`+`, `-`, `*`...), comparaison (avec les booléens) (`<`, `=`, `>`...), etc.

Typage statique

- Les types sont connus au moment de l'écriture du programme
  $\Longrightarrow$ Possibilité de vérifier que les opérandes des opérateurs sont du bon type (compilateur)
  $\Longrightarrow$ Permet de détecter des erreurs avant l'exécution du programme
  - `5 > "dx"` provoque un erreur à la compilation (typage statique)
  - `5 > 4.0` provoque un erreur à la compilation (typage statique fort)
  - **Typage fort :** Pas de conversion implicite/coercition entre types (ici entier et réel)

Types simples (ou scalaires) prédéfinis (ou fondamentaux)

	Réel	Entier	Booléen	Caractère
valeurs	-7.3, 1.5e-3, 3.0	-15, 0, 18	FAUX, VRAI	'A', '$', '5', ' ;'
opérateurs arithmétiques	+, - unaires +, - binaires *, **			
	/	Div, Mod		
op. relationnels	< > <= >= = /=			
opérateurs logiques			Non, Et, Ou EtAlors, OuSinon (p. B-13)	
autres opérateurs				Succ, Pred Ord, Chr (p. B-11)

- Types discrets : Entier, Booléen, Caractères.
- Types continus : Réel (simple ou double précision)
- Voir la matière "Architecture des ordinateurs" pour leur représentation
- Le type Booléen n'a que deux valeurs avec FAUX $<$ VRAI

Type caractère

Le type `Caractère` caractérise les caractères.
Chaque caractère est associé à un code entier (code ASCII, UTF-8, etc.).

Prédécesseur et successeur

- Succ(C) : le caractère après C. Exemple : `Succ('A') = 'B'`
- Pred(C) : le caractère avant C. Exemple : `Pred('8') = '7'`

Les opérateurs de conversion

`Ord` : le code du caractère (dans l'encodage utilisé)

`Chr` : le caractère correspondant à un code

Pour tout `C : Caractère`, on a : `Chr(Ord(C)) = C`

Remarque : Ne pas écrire 48 mais `Ord('0')` !

Les lettres majuscules, les lettres minuscules et les chiffres sont consécutifs

- Le caractère C est une majuscule ssi `(C >= 'A') Et (C <= 'Z')`
- L'entier qui correspond au chiffre C (un caractère) est : `Ord(C) - Ord('0')`.
- Le caractère qui correspond au chiffre N (dans [0..9]) est : `Chr(Ord('0') + N)`

Types définis ou utilisateurs

Il s'agit de **nouveaux** types que le programmeur peut **définir** (détaillés p. F-2)

- **Simples**
  - Type énuméré noté `Enumération`
    - Définit une énumération de valeurs discrètes.
    - Exemples : les jours de la semaine, les couleurs, etc.
  - Le type **Pointeur** noté `Pointeur` (voir p. I-2)
    - Permet de décrire des adresses en mémoire
- **Structurés**
  - Type **tableau** noté `Tableau`
    - contient un nombre fini et fixe de valeurs d'un type donné
    - Décrit une séquence indexée (par un type discret) de valeurs d'un type quelconque
    - Exemple : les notes d'un élève
  - Le type **chaine de caractères** noté `Chaine`.
    - Il s'agit d'un cas particulier de tableau dont les valeurs sont des caractères
    - Exemples : `"Bonjour"` ou `"bonjour"`
  - Le type **Enregistrement** noté `Enregistrement`
    - Permet de décrire des produits cartésiens
    - Exemple : Un nombre complexe est défini par une partie réelle et une partie imaginaire

Remarque : On définira les opérations sur ces types grâce aux sous-programme (p. E-2)

Opérateurs logiques (rappels)

Table de vérité des opérateurs logiques

A	B	A **Et** B	A **Ou** B	**Non** A
VRAI	**VRAI**	**VRAI**	**VRAI**	**FAUX**
VRAI	**FAUX**	**FAUX**	**VRAI**	**FAUX**
FAUX	**VRAI**	**FAUX**	**VRAI**	**VRAI**
FAUX	**FAUX**	**FAUX**	**FAUX**	**VRAI**

Lois de De Morgan

Non (A **Et** B) = (**Non** A) **Ou** (**Non** B)

Non (A **Ou** B) = (**Non** A) **Et** (**Non** B)

Non (**Non** A) = A

Détails et autres propriétés dans la matière ≪ Modélisation ≫.

Autres formulations

Si A est de type Booléen, on a :

A	équivalent à	`A = VRAI`
Non A	équivalent à	`A = FAUX`

Conseil : La notation de gauche est à préférer !

Évaluation en court-circuit (paresseuse ou partielle)

L'évaluation d'une expression booléenne s'arrête dès que le résultat est connu :

FAUX Et expr -- toujours FAUX sans avoir a évaluer expr

VRAI Ou expr -- toujours VRAI sans avoir a évaluer expr

Intérêt : Pouvoir évaluer `(N` $\neq$ `0) Et ((S Div N)` $\geq$ `10)` pour $N = 0$

Attention : Tous les langages n'ont pas d'évaluation partielle.

Dans notre langage algorithmique

- Évaluation partielle : **EtAlors** et **OuSinon**
- Évaluation totale : **Et** et **Ou**
- Exemple : `(N` $\neq$ `0) EtAlors ((S Div N)` $\geq$ `10)`

Constantes

Une constante est une information dont la valeur ne change pas au cours de l'exécution du programme.

Propriétés

Une constante a :

- un nom (identificateur) et une sémantique (commentaire)
- une valeur **non modifiable**
- un type (celui de sa valeur)

Exemples

```
PI = 3.1415              -- constante d'Archimède
MAJORITE = 18            -- Age de la majorité
TVA = 20                 -- Taux de TVA
CAPACITE_PROMO = 120     -- Nombre maximum d'élèves par promo
ACRONYME = "PIM"         -- Acronyme de l'UE
```

Constantes (2)

Règles

- On accède à (la valeur d')une constante par son nom (identificateur).
- Une constante **doit** être définie (rubrique constantes).
- Ne jamais utiliser une constante littérale sans la nommer (sauf -1, 0 et 1).
- Définir et utiliser une constante symbolique a plusieurs avantages :
  - Si la valeur doit changer, il suffit de la changer à un seul endroit
    - précision de π, capacité de la promotion, etc.
  - Le code est plus lisible car le nom de la constante est (devrait être !) explicite
  - Que signifie la constante littérale 12 dans un programme ?
    - le mois de décembre,
    - la capacité d'une boîte d'œufs,
    - l'âge maximal pour bénéficier de la gratuité,
    - les noces de soie,
    - etc.
  - De plus, la constante peut se trouver sous des formes un peu différentes ce qui rend difficile le changement de la valeur de la constante.
    - Exemple : 11 (12 - 1), 13 (12 +1), "12 ans", etc.

Variables

Une variable représente une donnée du monde réel qui sera manipulée dans le programme.
Exemples : le rayon d'une cercle, les coordonnées d'un point, la meilleure note...

Caractéristiques

Une variable est caractérisée par :

- un **rôle** indiquant sa sémantique, son utilité (représenté par un commentaire)
- un **nom :** identificateur, il doit être significatif !
- un **type :** caractérise **toutes** ses **valeurs** possibles
- une **valeur** associée à la variable (forcément de son type), **connue à l'exécution**

Remarque : Lors de la conception du programme, elles sont identifiées dans cet ordre

Techniquement, à l'exécution d'un programme, le nom de la variable désigne la zone de la mémoire de l'ordinateur où la valeur de cette variable est stockée
Au lancement du programme, les variables n'ont pas de valeur associée (**indéterminée**)

Variables (2)

Déclaration de variables

- Toute variable doit être déclarée dans une rubrique « Variables » :

```
Variables
    nom : Type    -- rôle
```

- Exemples

```
1  Variables
2      Rayon : Réel     -- rayon du cercle
3      X1 : Réel        -- abscisse du premier point
4      Y1 : Réel        -- ordonnée du premier point
5      X2, Y2 : Réel    -- coordonnées du deuxième point
6      Initiale : Caractère  -- initiale du nom
```

Conseil : Ne pas déclarer plusieurs variables sur la même ligne (sauf commentaire commun).

Règles de manipulation des variables

- Le type d'une variable ne peut pas changer... Et son **rôle** non plus !
- Si sa valeur ne change pas, il aurait fallu utiliser une constante !
- Le nom d'une variable peut apparaître :
  - dans une expression : on accède à sa valeur (p. B-19)
  - à gauche d'une affectation : on modifie sa valeur (p. B-23).

Expression

Une expression est une phrase du programme qui a une valeur et donc un type.

Remarque : Le type de l'expression (donc de sa valeur) est connu à la compilation.

Une expression est soit :

- une constante, littérale ou symbolique
- une variable
- un opérateur appliqué à des opérandes, elles-mêmes des expressions
- un appel de fonction (voir p. E-2).
- une expression entre parenthèses (voir évaluation d'une expression, p. B-20)

Exemples

```
3,   "Bonjour",   -3.6   -- trois expressions constantes littérales
Rayon,   X,   PI         -- deux variables et une constante symbolique
2 * PI * Rayon           -- opérateur * (2 fois) sur constantes et vari
Sin (X * 2 + Y)          -- un appel de fonction
(1 + TVA) * Prix_HT      -- avec des parenthèses
```

Évaluation d'une expression

Principe

Une expression est évaluée :

- en commençant par les opérateurs de priorité la plus élevée,
- à priorité égale, les opérateurs les plus à gauche (associativité à gauche).
  `a + b ** 2 * c * d` ≡ `(a + (((b ** 2) * c) * d))`
- Les parenthèses permettent de définir l'ordre de calcul : `prix_ht * (1 + TVA)`
- Une constante symbolique ou une variable est remplacée par sa valeur
- ⚡ les variables doivent avoir été initialisées (sinon erreur dans le programme)

Table de priorités des opérateurs

`and, or`	1 (moins prioritaire)
`<, >, <=, >=, = , /=`	2
`+ - binaires, &`	3
`*, /, div, mod`	4
`+ - unaires`	5
`**, abs, not`	6 (plus prioritaire)

Évaluation d'une expression : exemple

Pour déterminer l'ordre d'évaluation de l'expression : `a + b ** 2 * c * d`, on peut :

1. Identifier les ≪ mots ≫ (unités lexicales ou lexèmes) :

```
a   +   b   **   2   *   c   *   d
-   -   -   --   -   -   -   -   -
```

2. Identifier la priorité de chaque opérateurs

```
a   +   b   **   2   *   c   *   d
-   3   -    6   -   4   -   4   -
```

   - Les - marquent les expressions parenthésées, pour l'instant des expressions élémentaires

3. Identifier l'opérateur de priorité la plus forte, le plus à gauche : `**`
4. Lui associer ses opérandes (ici b à gauche et 2 à droite) $\Longrightarrow$ ajout de parenthèses.

```
a   +   (b   **   2)   *   c   *   d
-   3   ----------   4   -   4   -
```

5. Continuer avec le premier `*`.

```
a   +  ((b   **   2)   *   c)   *   d
-   3  ----------------------   4   -
```

6. Continuer avec l'autre `*`.

```
a   + (((b   **   2)   *   c)   *   d)
-   3 ------------------------------
```

7. Terminer avec le dernier opérateur : `+`.

```
(a   + (((b   **   2)   *   c)   *   d))
----------------------------------------
```

Plan

- Affectation
- Assertion
- Rien

Affectation

- L'affectation est l'instruction qui **associe** une valeur à une variable.
- Syntaxe : `Ident <-- Expression`
- Exemple : `X := 5`
  - on dit : ≪ la variable X **reçoit** la valeur 5 ≫ (ou ≪ X **reçoit** 5 ≫)
- Contrainte :
  - `Ident` doit être une variable
  - `Expression` est une expression dont le type doit être le même que celui de `Ident`
- Exécution :
  1. évaluation de la valeur de l'expression
  2. associer cette valeur à la variable
- L'affectation provoque un **effet de bord** (modification de l'état du programme)

Exemples

Soient `A`, `B`, `C`, `X` et `Y` des variables de type entier déjà initialisées.

```
1  A <-- -3          --! ok { A = -3 }
2  B <-- 3.6         --! erreur de compilation : 3.6 n'est pas un entier
3  C <-- X + Y       --! ok
4  A <-- A + 1       --! ok { A = -2 }  (la valeur de A devient -2)
5  A + B <-- X * Y --! erreur de compilation : A + B n'est pas une variable !
```

Assertion

- **Observation de variables** à l'aide d'assertions (commentaire formel, logique).

```
{ X = Val }
X <-- X + 3
{ X = Val + 3 }
```

```
{ X = Val et Y = 8 }
X <-- X + 3
Y <-- X + Y
{ X = Val + 3 et Y = Val + 3 + 8 }
```

- On utilise ce type **d'assertion pour vérifier la correction** de programmes
- Ces assertions permettent **d'observer les variables d'un programme** qui constituent **l'état d'un programme**
- Assertions utilisées pour décrire toutes les **propriétés pertinentes** pour la **vérification, la validation, la compréhension** d'un programme comme des propriétés, **préconditions, postconditions, invariants, variants,** etc.

Instruction Rien

- L'instruction Rien est une instruction qui ne fait rien.
- Syntaxe : `Rien`
- Contrainte : aucune
- Exécution : aucun effet
- Intérêt :
  - Utile pour dire explicitement qu'il n'y a rien à faire

Plan

Les entrées / Sorties

Les instructions d'entrée / sorties permettent au programme d'interagir avec son environnement extérieur, en particulier avec les périphériques :

- de sortie : écran, imprimante, disque, actionneur, etc.
- d'entrée : clavier, souris, disque, écran tactile, capteur, etc.

Les instructions Écrire (sortie) et Lire (entrée) font abstraction du dispositif utilisé.

Nous nous limiterons à l'écran (périph. de sortie) et au clavier (périphérique d'entrée).

En général, un programme est constitué de trois parties :

- **Acquisition :** Lire les données de manière conviviale et fiable
- **Traitement :** Traiter le problème posé
- **Restitution :** Afficher les résultats obtenus de manière conviviale

Interface Humain Système (IHM)

L'objectif de ce cours n'est pas de traiter les IHM.
Aussi, elles seront limitées au minimum, en mode texte (pas de mode graphique).

Sorties / Écritures

Instruction Écrire

- Transfère (affiche, imprime) sur le périphérique de sortie une **valeur**.
- Syntaxe : `Écrire(Expression)`
- Contraintes :
  - `Expression` est une expression d'un type prédéfini (Entier, Réel, Caractère ou Chaîne).
  - Seule une `Expression` (donc une variable, une constante, etc.) est affichée (écrite)
- Exécution :
  1. la valeur de `Expression` est évaluée
  2. cette valeur est traduite en caractères qui sont transmis au périphérique de sortie

Rq : Écrire modifie (≪ affecte ≫) la sortie avec une donnée du programme.

Exemples

```
1  Rayon <-- 5.0
2  Écrire(10)                  --! affiche "10"
3  Écrire(2 * PI * Rayon)      --! affiche "31.415"
4  Périmètre <-- 2 * PI * Rayon
5  Écrire(Périmètre)           --! affiche "31.415"
6  Écrire("Périmètre")         --! affiche "Périmètre"
7      --! Ne pas confondre variable et constante littérale de type Chaîne
```

Entrées / Lectures

Instruction Lire

- Traduit les caractères du périphérique d'entrée en une valeur pour le programme.
- Syntaxe : `Lire(Variable)`
- Contraintes :
  - `Variable` est une variable d'un type prédéfini (Entier, Réel, Caractère ou Chaîne).
  - Seule une `Variable` peut être *lue*
- Exécution :
  1. les caractères de l'entrée sont interprétés en une valeur V du type de `Variable`
  2. la valeur de `Variable` devient cette valeur V
  3. si les caractères ne correspondent pas à une valeur du type de `Variable`, une exception se produit, `Variable` est inchangée

Remarque. `Lire` se comporte comme une affectation : elle donne une valeur à `Variable`.

Exemple

```
1  Variables
2      Rayon : Réel          -- rayon du cercle
3  Début
4      -- Demander le rayon du cercle
5      Écrire("Rayon du cercle ? ")  -- afficher la consigne à l'utilisateur
6      Lire(Rayon)          --! L'utilisateur devra saisir un réel, e.g. : 5.0
7      Lire(Rayon + 2.0)    --! Erreur : Taille + 2 n'est pas une variable
8  Fin
```

Lectures et Écritures

Remarques

- Les instructions de lecture (entrée) `Lire` et d'écriture (sortie) `Ecrire` définies précédemment sont utilisées pour
  - lire et écrire des données typées par des types de base : `Caractère`, `Booléen`, `Entier`, `Réel` et `chaîne de caractères`.
  - Ces instructions sont disponibles dans la plupart des langages de programmation

- Pour les autres types de données (Voir p. F-2), il faudra définir des actions (sous-programmes, voir p. E-2)) spécifiques pour les entrées et sorties associées à ces types de données.

Plan

Structures de contrôle

Principe

L'exécution d'un programme est une séquence d'affectations (`<-`, `Lire` et `Écrire`) qui font passer d'un état initial (valeurs indéterminées) à un état final (résultat).

Structures de contrôle

Les **structures de contrôle** définissent l'ordre d'exécution des affectations.

1. **Séquence** : plusieurs instructions exécutées à la suite
2. **Décision** (ou conditionnelle ou alternative) : choisir quelle séquence exécuter
   - Si
   - Selon
3. **Répétition (ou** itération ou boucle) : exécuter plusieurs fois une séquence
   - TantQue
   - Répéter $\cdots$ JusquÀ
   - Pour
4. **Appel de sous-programme** (voir chapitre sous-programmes p. E-2)
5. **Exception** (voir chapitre Exception p. J-2)

Séquence

La **séquence** est la structure de contrôle implicite qui définit que les instructions s'exécutent les unes à la suite des autres (en séquence).

Exemple

```
1 I1
2 I2
3 I3
4 ...
5 In
```

```
1 -- permuter les valeurs de X et Y
2 { X = a et Y = b }
3 Mémoire <-- X
4 X       <-- Y
5 Y       <-- Mémoire
6 { X = b et Y = a }
```

Exécution

- Les instructions sont exécutées l'une après l'autre, dans l'ordre.

Règles

- Une **seule** instruction par ligne.
- Les instructions d'une même séquence doivent avoir la **même indentation**.

Décision Si

Définition

La **décision** `Si` permet de choisir la séquence à exécuter.

```
1  Si Condition Alors
2      Séquence_Alors
3  Sinon
4      Séquence_Sinon
5  FinSi
6  Suite
```

```
Si A < B Alors
    Min <- A
Sinon
    Min <- B
FinSi
```

Contrainte

- `condition` est une expression booléenne

Exécution

1. La `condition` est évaluée
2. Si la `condition` s'évalue à **VRAI**, alors `Séquence_Alors` est exécutée, puis `Suite`
3. Si la `condition` s'évalue à **FAUX**, alors `Séquence_Sinon` est exécutée, puis `Suite`

Variantes de la décision Si

`Sinon` optionnel

La partie `Sinon` est optionnelle

La mettre avec `Rien` est conseillé (cas explicité, couverture total du type Booléen de la condition)

```
Si Valeur < Min Alors
    Min <- Valeur
FinSi
```

```
Si Valeur < Min Alors
    Min <- Valeur
Sinon
    Rien
Finsi
```

`SinonSi`

Il est possible d'ajouter des `SinonSi`.

Peut être réécrit avec `Si Sinon`

- ☺ les conditions sont au même niveau
- ☺ les séquences alternatives ont même indentation
- ⚠ les conditions devraient porter sur les mêmes variables

```
1 Si N > 0 Alors
2     Signe = 1
3 SinonSi N < 0 Alors
4     Signe = -1
5 Sinon
6     Signe = 0
7 FinSi
```

Lisibilité de la décision Si

- Indenter les séquences des `Si`, `SinonSi` et `Sinon` (voir exemples)
- Ne pas écrire `Si X = Vrai Alors ...` mais `Si X Alors ...`
- Ne pas écrire `Si X = Faux Alors ...` mais `Si Non X Alors ...`
- Ne pas écrire :

```
Si Condition Alors
    Une_Variable <- True
Sinon
    Une_Variable <- Faux
FinSi
```

  mais :

```
Une_Variable <- Condition
```

- De la même manière, ne pas écrire :

```
Si Condition Alors
    Une_Variable <- Faux
Sinon
    Une_Variable <- Vrai
FinSi
```

  mais :

```
Une_Variable <- Non Condition
```

Décision Selon

La **décision** `Selon` permet de **choisir la séquence** à exécuter en comparant la **valeur d'une expression** à plusieurs **choix disjoints**.

Contraintes

- Le type de `Expression` est un type discret
- Les différents choix sont disjoints
- L'ensemble des choix doit couvrir toutes les valeurs du type de `Expression` (sauf si `Autres` est utilisé).

Exécution

1. `Expression` est évaluée (à V)
2. `Séquence_i` si V est dans `Choix_i` sinon `Séquence_Autre` est exécutée.
3. L'exécution continue à Suite.

Syntaxe

```
1   Selon Expression Dans
2       Choix_1 => Séquence_1
3       Choix_2 => Séquence_2
4       ...
5       Choix_n => Séquence_n
6       Autres  => Séquence_Autre
7   FinSelon
8   Suite
```

Exemple

```
1   Selon x + 3 Dans
2       6 =>
3           x <-- 8
4           y <-- x + 5
5
6       9..10 =>
7           x <-- 3 * x
8
9       8, 7, 25 =>
10          x <-- 7
11          x <-- x + 9
12
13      Autres =>
14          x <-- 0
15  FinSelon
```

Répétition TantQue

La répétition `TantQue` exécute zéro ou plusieurs fois une séquence.

Syntaxe

```
1  TantQue Condition Faire
2      Séquence
3  FinTQ
4  { Non condition }
5      --! sortie du TantQue
6  Suite
```

Exemple

```
1  Lire (Valeur)
2  TantQue Valeur /= 0 Faire
3      Valeur <-- Valeur * 2
4      Écrire (Valeur)
5      Lire (Valeur)
6  FinTQ
7  { Valeur = 0 }  --! sortie du TantQue
```

Contraintes

- Condition est un expression booléenne
- Séquence doit modifier Condition

Exécution

1. Évaluer la condition
2. Si sa `valeur` est **Vrai**, `séquence` est exécutée et l'exécution continue en 1.
3. Si sa `valeur` est **Faux**, l'exécution continue à `Suite`.

Répétition Répéter

La répétition Répéter exécute une ou plusieurs fois une séquence.

Syntaxe

```
1  Répéter
2      Séquence
3  JusquÀ Condition
4  Suite
```

Exemple

```
1  Somme <-- 0
2  Valeur <-- 1
3  Répéter
4      Somme <-- Somme + Valeur
5      Valeur <-- Valeur + 1
6  JusquÀ Somme > Max
```

Contraintes

- Condition est un expression booléenne
- Séquence doit modifier Condition

Exécution

1. Exécuter Séquence.
2. Évaluer la condition
3. Si sa `valeur` est **Vrai**, l'exécution continue à Suite.
4. Si sa `valeur` est **Faux**, l'exécution continue en 1.

Répétition Pour

La répétition Pour exécute un nombre connu de fois une séquence.

Syntaxe

```
1  Pour VarBoucle De Début À Fin Pas P Faire
2      Séquence
3  FinPour
4  Suite
```

Exemple

```
1  Somme <-- 0
2  Pour N De 1 À 9 Faire
3      Somme <-- Somme + N
4  FinPour
```

Contraintes

- Début et Fin sont du m̂ type scalaire discret T
- VarBoucle est une variable de ce même type T :
  - déclarée implicitement
  - utilisable que par Séquence
- P est un entier non nul (1 par défaut)
- Séquence ne doit pas modifier VarBoucle

Exécution

1. Début, Fin et P sont évalués
2. VarBoucle est initialisée à Début
3. Tant que VarBoucle ne dépasse pas Fin :
   1. Séquence est exécutée et
   2. VarBoucle est augmentée de P

On sait dès le début combien de fois Séquence sera exécutée :

$$\text{Max}(0, \text{Fin - Début} + 1) \text{ si } P = 1$$

Terminaison

- La terminaison est une propriété fondamentale des algorithmes : est-ce que le programme se terminera.
- La terminaison est garantie pour les :
  - instructions simples (affectation, rien)
  - entrées/sorties (si le périphérique fonctionne et l'utilisateur joue le jeu)
  - séquence et décisions : l'exécution avance dans le programme et s'approche de la fin
- Le problème se pose pour les répétitions :
  - On exécute plusieurs fois la même séquence.
  - Le programme pourrait ne pas s'arrêter. On dit qu'il boucle.
- Un moyen pour montrer sa terminaison et d'exhiber un **variant** :
  - Un entier naturel (donc positif !)
  - qui décroit strictement à chaque itération
- Remarque : la terminaison est garantie pour la répétition `Pour`
  - car on sait par construction combien de fois la boucle sera exécutée

- Variant, invariants, etc. seront étudiés dans la matière ≪ Modélisation ≫

Choisir la bonne structure de contrôle

Principe

Il faut à chaque fois choisir l'outil le plus adapté.

Quelle décision choisir ?

- Préférer un Selon quand il est possible.
- Remarque : pour une expression booléenne, on utilisera toutefois un Si.

Quelle répétition choisir ?

1. On commence par identifier la séquence que l'on va répéter
2. On regarde combien de fois on la répète :
   - Un nombre connu de fois $\Longrightarrow$ on utilise un Pour
     - Sauf si on peut connaître le résultat avant $\Longrightarrow$ TantQue
   - Un nombre inconnu de fois $\Longrightarrow$ TantQue ou Répéter
     - Le TantQue conduit souvent à dupliquer des instructions
     - Le Répéter conduit souvent à dupliquer une condition

   Attention : Si la séquence peut ne pas être exécutée $\Longrightarrow$ TantQue

Plan

Normes de programmation

Les règles ci-dessous devront être respectées impérativement.
Elles seront complétées tout au long du cours.

- Utiliser des identificateurs **significatifs** et **commentés** pour les noms de constantes, variables, types (identificateurs de types), sous-programmes (on en parlera plus tard)
- Toute variable **doit être initialisée** avant de pouvoir en utiliser la valeur.
  En principe, on donnera une valeur initiale à la variable uniquement au moment où on en a besoin (et pas nécessairement à la déclaration)
- On choisira, pour un problème donné, les **bonnes structures de contrôle** (ne pas utiliser un `TANT QUE` à la place d'un `POUR` etc.)
- **Indenter** le code, écrire **une seule** instruction par ligne
- On vérifiera que les répétitions (boucles) sont bien écrites, à savoir :
  - La boucle **termine**
  - Les **variables** des instructions de la boucle et de la condition **ont bien une valeur**
  - Le **commentaire de fin de boucle** sera systématiquement écrit
- Les lectures de données (ou affichages de résultats) **doivent** être **conviviales** et **fiables**

Plan

Structure d'un programme Ada

```
1   -- Nom auteur, groupe de TP
2   -- sémantique du programme
3
4   with Text_Io;              use Text_Io;  -- E/S sur les chaînes de caractères
5   with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;    -- E/S sur les entiers
6   with Ada.Float_Text_IO;   use Ada.Float_Text_IO;      -- E/S sur les réels
7
8   procedure Nom_Programme is
9       -- déclarations de constantes
10      -- déclarations de types
11      -- déclarations de sous-programmes
12      -- déclarations des variables du programme principal
13  begin
14      -- instructions
15  end Nom_Programme;
```

Exemple de programma Ada simple : le périmètre d'un cercle

```
1   -- NOM Prénom, Groupe X
2   -- Déterminer le périmètre d'un cercle à partir de son rayon
3   -- Attention : aucun contrôle sur la saisie du rayon => non robuste
4
5   with text_io;              use text_io ;
6   with ada.float_text_io ;   use ada.float_text_io ;
7
8   procedure Perimetre_Cercle is
9       --! définition de constantes
10      PI : constant Float := 3.1416;  --! Constante PI
11      --! déclarations des variables du programme principal
12      Rayon     : Float;          -- le rayon du cercle lu au clavier
13      Perimetre : Float;          -- le périmètre du cercle
14  begin
15      -- Saisir le rayon
16      Put ("Rayon = ");
17      Get (Rayon);
18
19      -- Calculer le périmètre
20      Perimetre := 2.0 * PI * Rayon;    -- par définition
21
22      pragma Assert (Perimetre = 2.0 * PI * Rayon);
23
24      -- Afficher le périmètre
25      Put ("Périmètre = ");
26      Put (Perimetre);
27      New_Line;
28  end Perimetre_Cercle;
```

Eléments de base du langage Ada

Caractères et identificateurs

```
Alphabet  ::= lettre  |  chiffre  |  caractère-spécial
Lettre    ::= majuscule  |  minuscule
Majuscule ::= A  |  B  |  ... |  Z
Minuscule ::= a  |  b  | ... |  z
Chiffre   ::= 0 |  1  | ...  |  9
Caractère_spécial ::= *  |  /  |  +  |  - |<  |  >  |  <=  | >=  |  =  |= |
     _ |   |  (  |  )  |  "  | '  | #  |  &  |  : | ;
Ident ::= lettre suivie d'un nombre quelconque de caractères
      parmi les lettres, les chiffres et le _
```

- Pas de distinction majuscule/minuscule
- Pas de limite sur la longueur des identificateurs

Notation dites BNF (Backus-Naur Form

- `X ::= Y` production. Le membre gauche `X` produit (peut être remplacé par) le membre droit `Y`
- Autres opérateurs : itération, optionel, etc.
- Traitment des langages formels (voir cours de modélisation, compilation, langages formels)

Eléments de base du langage Ada

Mots clés ou mots réservés

```
Mot_clé ::= and  |  or  |  not  |  begin  |  end    | mod |
            do  |  of |  for  |  in  | reverse  |  loop | while  |
            until  |  if  |  then  | else |  elsif  |  case |
            when  |  null |  constant  | is | type | array | record |
            function | procedure  | return |  in  | out |
            with | use | others
```

Commentaires

```
-- un commentaire
```

Affectation avec le symbole :=

```
X := Y + 5
```

Instruction Rien : Null

```
null;
```

Eléments de base du langage Ada

Les types

```
Short_Integer
Integer
long_Integer
Float
Long_Float
Boolean
Character
```

Déclarations de variables et de constantes

```
PI                  : constant Float := 3.1416;
Somme               :          Integer;
Montant             :          Float;
Prix_TTC, Prix_HT   :          Float;
```

- Toutes les **déclarations** se terminent par un « ; »
- Une constante se définit en ajoutant le mot-clé `constant` devant le type

Eléments de base du langage Ada

Expressions

```
Expression ::= ident | constante | op_unaire expression |
               expression op_binaire expression
Op_unaire ::= + | - | not
Op_binaire ::= + | -  | *  |  /  |
                mod |  rem |  >  |  >=  |  =  |  /=  | <  |
                <=  |  or  |  and
```

- Même priorité des opérateurs en ADA et en langage algorithmique
- Typage fort des opérateurs et opérandes :
  `1.2 + 3` n'est pas autorisé en Ada, il faut écrire `1.2 + 3.0`
- Opérateurs de conversion de type disponible.
  L'expression précédente peut s'écrire `1.2 + Float(3)` avec l'opérateur de conversion de type d'entier en floattant

Eléments de base du langage Ada

Entrées : avec Get

```
Get (X);     --! X doit être une variable ou assimilé
```

- Cette procédure est surchargée pour les types de bases.
- Seuls les caractères utiles pour construire une valeur du type de X sont consommés. Les autres restent en attente d'une prochaine lecture. `Skip_Line;` les efface.

Cas particulier de chaînes de caractères

```
Get_Line (S, N);     --! les deux paramètres sont en out
```

- `S` est une variable de type chaîne de caractères désignant la chaîne à lire
- `N` est une variable qui contient le nombre de caractères saisis dans la chaîne `S`

Sorties : avec Put

```
Put (Expression);       --! écrire la valeur de expression
Put_Line (Expression); --! en ajoutant un retour à la ligne (pour les chaînes)
New_Line ;              --! passer à la ligne
```

- Put sur les entiers a deux paramètres optionnels `Width` et `Base`

Eléments de base du langage Ada

Toute **instruction** du langage Ada se termine par un ;

Séquence

```
1       -- Séquence
2       X := 2;
3       Y := 4;
```

Conditionnelle : Si Alors Sinon

```
1       if Expression then
2           Sequence_Alors;
3       else
4           Sequence_Else;
5       end if;
```

Eléments de base du langage Ada

Conditionnelle : Si Alors SinonSi

```
1  if Condition_Si then
2      Sequence_Alors;
3  elsif Condition_Elseif_1 then
4      Sequence_Elseif_1;
5  elsif Condition_Elseif_2 then
6      Sequence_Elseif_2;
7      ....
8  else
9      Sequence_Else;
10 end if;
```

Conditionnelle : Selon

```
1  case Expression is
2      when Val_1                => Sequence_1;
3      when val_2 | ... | Val3 => Sequence_2;
4      when Val_4..Val_5         => Sequence_3;
5      when others               => Sequence_4;
6  end case;
```

Eléments de base du langage Ada

Répétition : Tant Que

```
1  while Expression loop
2      Sequence;
3  end loop;
```

Répétition : Répéter Jusqu'à

```
1  loop
2      Sequence;
3  exit when Condition;
4  end loop;
```

Attention : **Aucune instruction** entre `exit when` et `end loop`.

Répétition : Pour

```
1  for Indice in Initial..Final loop
2      Sequence;
3  end loop;
```

La variable de boucle d'un `for` ne se déclare pas !
Sur des `Character` : `for C in Character range 'A'..'Z' loop`

Plan

Introduction

Objectif du chapitre

Conception de programmes par raffinement

Conception : définition

La conception est une phase du cycle de développement d'un programme. Il s'agit du processus qui conduit à la construction d'un programme à partir d'une spécification

- Ascendante : construction d'un programme à partir de programmes élémentaires.
  - Conception des programmes élémentaires
  - Définition des opérations permettant d'assembler les programmes
  - Composition, bottom-up
- Descendante : construction d'un programme à partir d'une spécification
  - Une spécification est décomposée en actions, elles mêmes décomposées à leur tour, jusqu'à atteindre des actions exécutables par le processeur
  - Décomposition, raffinement, raffinage, top-down

Ce cours étudie la conception par raffinages successifs.

Raffinages : Principes

Principe

Décomposer un problème ≪ compliqué ≫ en sous-problèmes plus simples que l'on sait résoudre (que l'on pense savoir résoudre !).

C'est le principe ≪ Diviser pour régner ≫ (≪ *divide and conquer* ≫) :

- Diviser : décomposer un problème en sous-problèmes
- Régner : résoudre les sous-problèmes
- Combiner : utiliser les sous-problèmes pour résoudre le problème initial.

Exemple : Lemmes d'une démonstration.

Raffinage (ou décomposition)

Un raffinage est la décomposition d'une action complexe en une combinaison d'actions, élémentaires ou complexes, qui réalisent exactement le même objectif.

Raffinages

L'arbre de l'ensemble des raffinages utiles pour décomposer une actions complexe :

- La racine est l'action complexe initiale.
- Les nœuds sont les actions complexes introduites.
- Les feuilles sont les actions élémentaires.

Méthode des raffinages

Principales étapes de la méthode des raffinages

1. Comprendre le problème
   a. Reformuler le problème de manière précise
   b. Donner des exemples
2. Trouver une solution informelle
3. Structurer cette solution informelle
4. Produire l'algorithme
5. Tester (à chaque étape)

Mise en œuvre : Exemple fil rouge

Dans la suite, nous utilisons cette méthode pour résoudre le problème suivant.

Énoncé. Ecrire un programme qui calcule et affiche le quotient et le reste de la division entière de deux entiers a et b.

Étape 1. Comprendre le problème

- **Bien comprendre le problème** est nécessaire **pour** pouvoir écrire **le bon programme**
  - si besoin, demander des précisions au client (**QUOI ?**)
- **Deux outils :**
  - Reformuler le problème (l'action principale)
  - Spécifier au travers d'exemples (données en entrées **ET** de résultats attendus)
  - Définir des exemples d'utilisation du programme (spécifier)
- **Intérêt :**
  - Vérifier que le programme est compris, bien posé, précis...

1. a) L'énoncé est-il suffisamment précis ?

- NON car rien n'est dit sur les contraintes pour a et pour b, ni comment on connaît les valeurs de a et de b.
- On interroge le client pour préciser ces points
- On aboutit à un nouvel énoncé.

Étape 1. a) Reformuler le problème R0

Nouvel énoncé

Afficher le quotient et le reste de la division entière d'un *dividende* par un *diviseur*, entiers naturels lus au clavier.
On se limite aux cas où *dividende* est positif ($dividende \geq 0$) et *diviseur* strictement positif ($diviseur > 0$).
Les lectures doivent être conviviales (bien expliquer à l'utilisateur ce qu'on lui demande) et fiables (si les contraintes sur *dividende* ou *diviseur* ne sont pas respectées, les entiers seront lus à nouveau).
On n'utilisera ni multiplication ni division mais seulement addition et soustraction.

R0

La description complète est souvent longue. La première phrase résume l'objectif.
On l'appelle **R0**. Il s'agit de l'**action principale**.

```
R0 : Afficher le quotient et le reste de la division entière d'un dividende
     par un diviseur, entiers naturels lus au clavier.
```

Remarques

- L'énoncé répond à la question **QUOI ?** (i.e. Qu'est ce qui doit être fait ?)
- Des contrainte sur la solution peuvent être imposées (dernière phrase de l'énoncé).

Étape 1. b) Identifier des exemples

Exemples (jeux de test) : données en entrée ET résultats attendus

Cas de test	données en entrée		résultats attendus	
	Dividende	Diviseur	Quotient	Reste
Dividende ≥ 0 ET Diviseur > 0	11	4	2	3
Dividende < 0 ET Diviseur > 0	-12	45	resaisie de Dividende	
Dividende ≥ 0 ET Diviseur < 0	16	-2	resaisie de Diviseur	
Dividende ≥ 0 ET Diviseur $= 0$	15	0	resaisie de Diviseur	
...	...	...	...	...

- À utiliser pour vérifier l'algorithme pendant sa construction.
- À utiliser pour tester le programme quand il sera écrit !
- **Attention.** En général, pas facile de trouver les bons exemples...

Dialogues entre l'utilisateur et le programme

Préciser ces dialogues est aussi important (à valider avec le client). Voici un exemple :

```
Dividende (>= 0) ? 11
Diviseur (> 0) ? 0
Erreur. Le diviseur doit être strictement positif.
Diviseur (> 0) ? 4
Le quotient est 2 et le reste est 3.
```

Étape 2 : Trouver une solution informelle

Objectif

- Identifier une manière de résoudre le problème
  - Il s'agit d'avoir l'idée, l'intuition de comment traiter le problème.
  - Comment trouver l'idée ? C'est le **point difficile** !
  - S'appuyer sur ses connaissances, son expérience, la littérature...
- Répondre à la question ≪ **COMMENT ?** ≫
  - Comment ≪ afficher le quotient et le reste de la division entière... ≫ ?

Solution informelle

On demande à l'utilisateur du programme le dividende jusqu'à ce qu'il soit positif puis le diviseur qui doit être strictement positif. Ces deux informations obtenues, on peut alors calculer le reste et le quotient et, enfin, les afficher.

Remarque

Vérifier la solution informelle en s'appuyant sur les exemples identifiés

Étape 3 : Structurer la solution informelle

Objectif

- Identifier de nouvelles actions
- Les combiner en utilisant une structure de contrôle
- Raffiner les nouvelles actions qui sont complexes

Division entière : structurer la solution

- De la solution informelle, on identifie les 4 actions suivantes :
  - **Lire le Dividende** (au clavier) avec *Dividende* ≥ 0 de manière fiable et conviviale
  - **Lire le Diviseur** (au clavier) avec *Diviseur* > 0 de manière fiable et conviviale
  - **Calculer le quotient et le reste** de la division de Dividende par Diviseur, avec *Dividende* ≥ 0 et *Diviseur* > 0 (en n'utilisant que les opérations $+$ et -)
  - **Afficher le quotient et le reste** q et r (sur l'écran) de manière conviviale
- Réalisées en **séquence**, ces 4 actions réalisent bien l'action complexe R0.
- On peut donc en déduire le premier raffinage.

Remarque. En général, on doit ordonner, regrouper les actions identifiées.

Étape 3 : Structurer la solution informelle (2)

Structurer la solution informelle : premier niveau de raffinage (R1)

```
R1 : Comment ≪ afficher le quotient et le reste de la division entière... ≫ ?
    Demander le dividende (au clavier)
    { Dividende >= 0 }
    Demande le diviseur (au clavier)
    { Diviseur > 0 }
    Calculer le quotient et le reste de la division de dividende par diviseur
    { Dividende = Quotient * Diviseur + Reste ET 0 <= Reste ET Reste < Quotient }
    Afficher le quotient et le reste
    { Le quotient et le reste de la division de dividende par diviseur sont affichés }
```

Validation

- Ce raffinage fait bien ce qui était attendu et que ce qui était attendu par R0.
- Les exemples identifiés fournissent les résultats attendus.

Suite

- Ce raffinage fait apparaître des actions complexes (les 4 !).
- Il faudra les décomposer à leur tour (étape 2 puis 3 pour chacune).
- Pour un problème plus compliqué, il pourrait être utile de faire aussi l'étape 1.

Flots de données

Objectif

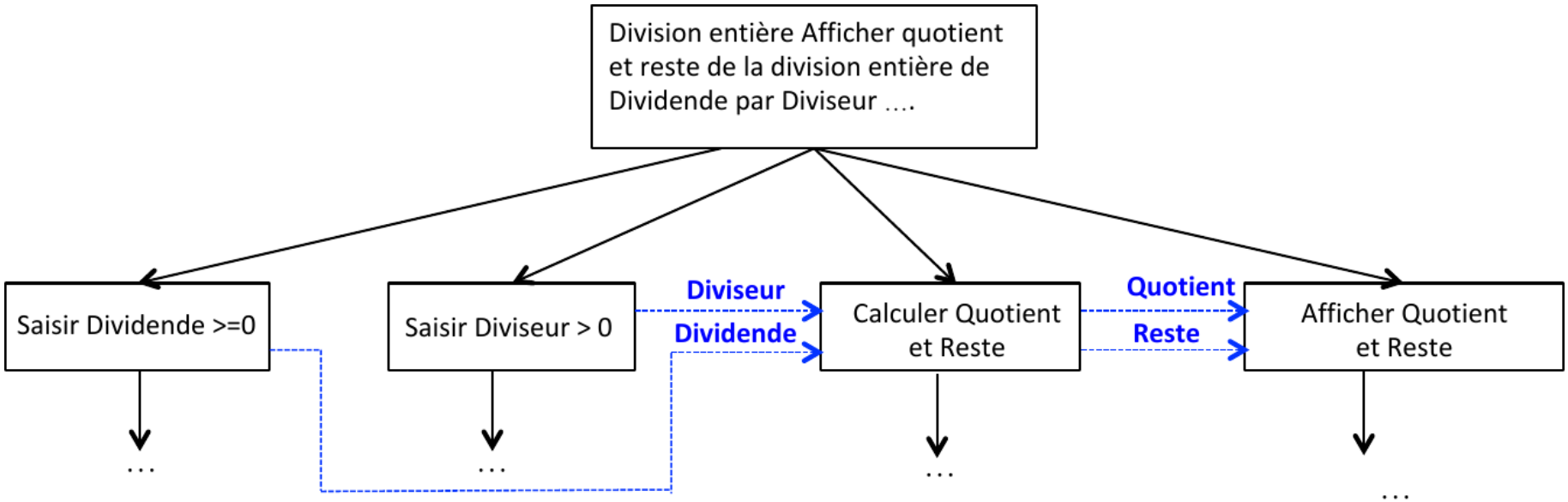
- Expliciter les données manipulées par une sous-action d'un raffinage
- Préciser ce que fait la sous-action de cette donnée :
  - **in** : l'action l'utilise sans la modifier
  - **out** : l'action produit cette donnée sans consulter à sa valeur
  - **in out** : l'action l'utilise, puis la modifie
- Contrôler le séquencement des actions :
  - une donnée doit être produite (out) avant d'être utilisée (in)

Raffinage avec flot de données

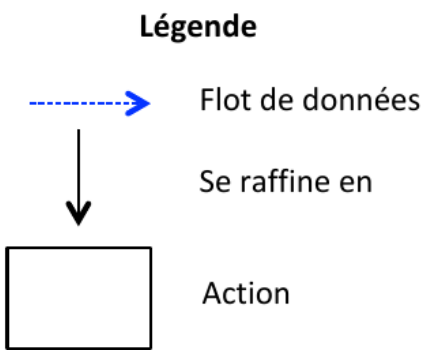
```
R1 : Comment ≪ afficher le quotient et le reste de la division entière... ≫ ?
    Demander le dividende (au clavier)        Dividende : out Entier
    { Dividende >= 0 }
    Demande le diviseur (au clavier)          Diviseur : out Entier
    { Diviseur > 0 }
    Calculer le quotient et le reste de la division de dividende par diviseur
        Dividende, Diviseur : in ; Quotient, Reste : out Entier
    { Dividende = Quotient * Diviseur + Reste ET 0 <= Reste ET Reste < Quotient }
    Afficher le quotient et le reste          Quotient, Reste: in
    { Le quotient et le reste de la division de dividende par diviseur sont affichés }
```

- "Calculer le quotient et le reste..." utilise le Dividende (in) et le Diviseur (in) pour produire le Quotient (out) et le Reste (out).

Arbre des raffinages



- Notation alternative
  - peu pratique si décisions ou répétitions
- Permet de comprende in out
  - la référence est l'action (boîte)
  - ce qui rentre (utilisé) : in
  - ce qui sort (produit) : out



Étape 4 : Production de l'algorithme

Objectif

- Produire l'algorithme à partir des raffinages.

Principe

L'algorithme est obtenu à partir d'un parcours préfixe de l'arbre des raffinages :

- les nœuds, actions complexes décomposées, apparaissent en commentaire
- les feuilles sont des instructions ou actions complexes non encore décomposées
- les structures de contrôle sont conservées
- les variables sont déduites des flots de données

Étape 4 : Production de l'algorithme (2)

Exemple de la division entière

En se limitant au niveau de raffinage R1, on obtient l'algorithme suivant.

```
1   ALGORITHME Afficher_Quotient_Reste EST
2       -- Afficher le quotient et le reste de la division entière
3       -- de Dividende par Diviseur...
4   VARIABLE
5       Dividende: Entier   -- le dividende
6       Diviseur: Entier    -- le diviseur
7       Quotient: Entier    -- le quotient de la division de Dividende par Diviseur
8       Reste: Entier       -- le reste de la division de Dividende par Diviseur
9   DÉBUT
10      Demander le dividende           Dividende: out Entier
11      { Dividende >= 0 }
12      Demander le diviseur            Diviseur: out Entier
13      { Diviseur > 0 }
14      Calculer le quotient et le reste de la division de dividende par diviseur
15          Dividende, Diviseur : in ; Quotient, Reste : out Entier
16      { Dividende = Quotient * Diviseur + Reste ET 0 <= Reste ET Reste < Quotient }
17      Afficher le quotient et le reste          Quotient, Reste: in
18      { Le quotient et le reste de la division de dividende par diviseur sont affichés }
19  FIN Afficher_Quotient_Reste
```

- Cet algorithme contient des actions complexes qu'il faudra décomposer.

Décomposer les actions complexes

Principe

- Lors de l'application de la méthode des raffinages, on introduit des actions qui peuvent être complexes.
- Il faut donc les décomposer à leur tour (avec la même méthode).
- On est libre de choisir par quelle action commencer :
  - Commencer par la plus compliquée ou moins bien comprise
  - car elle risque de remettre en cause la solution envisagée.
- Si on est plusieurs concepteurs, on peut alors se répartir les actions à décomposer

Notation

Si l'action complexe a été introduite dans un raffinage $\mathbf{R}_i$, sa décomposition sera $\mathbf{R}_{i+1}$.

- L'action principale qui apparaît comme $\mathbf{R}_0$ est décomposée dans un raffinage $\mathbf{R}_1$.
- Chaque action introduite dans $\mathbf{R}_1$ sera décomposée dans un raffinage $\mathbf{R}_2$, etc.

Division entière : décomposition des actions complexes de $\mathbf{R}_1$

Choisir les actions à décomposer

- $\mathbf{R}_1$ contient 4 actions complexes.
- Les actions de lecture (≪ Demander le dividende ≫ et ≪ Demander le diviseur ≫) et d'écriture (≪ Afficher le quotient et le reste ≫) seront traitées en TD :
  - lecture et écriture fiable et conviviale
  - nous ne les détaillons pas ici et réutilisons les raffinages connus
- Reste l'action ≪ Calculer le quotient et le reste de la division entière ≫.

Décomposition de l'action « Calculer le quotient et le reste... »

Étape 2 : Solution informelle

On retranche autant de fois que possible Diviseur à Dividende. Ce nombre de fois donne le quotient. Quand on ne peut plus retrancher diviseur, on a aussi le reste.

Structuration de la solution : réflexion

- Les actions :
  - Incrémenter Quotient : `Quotient <- Quotient + 1`
  - Soustraire Diviseur de Reste : `Reste <- Reste - Diviseur`
- On les fait plusieurs fois. Combien de fois ?
  - Plusieurs fois, éventuellement 0 (Dividende $<$ Diviseur)
    $\Longrightarrow$ La structure de contrôle est `TantQue`
  - La condition de continuation est : `Reste >= Diviseur`
- Comment initialiser les variables Quotient et Reste avant le `TantQue` ?
  - `Quotient <- 0`
  - `Reste <- Dividende`

Décomposition de l'action ≪ Calculer le quotient et le reste... ≫ (2)

Étape 3 : Structuration de la solution : formalisation

```
1   R2 : Comment ≪ Calculer le quotient et le reste de la division de Dividende
2           par Diviseur, avec Dividende >=0 et Diviseur >0 ≫ ?
3       Quotient <-- 0
4       Reste <-- Dividende
5       TANT QUE Reste >= Diviseur FAIRE
6           { Variant : Reste }
7           { Invariant : Dividende = Diviseur * Quotient + Reste }
8           Quotient <-- Quotient + 1
9           Reste <-- Reste - Diviseur
10      FIN TANT QUE
```

- On n'a pas gardé les actions complexes ≪ Incrémenter Quotient ≫ et ≪ Soustraire Diviseur de Reste ≫ car elles paraphrasent l'instruction correspondante.

Suite ?

- Toutes les actions introduites sont élémentaires
  $\Longrightarrow$ La décomposition de cette actions complexe est complètement terminée.

Division entière / Étape 4 : Production de l'algorithme

```
1   ALGORITHME Afficher_Quotient_Reste EST
2
3       -- Afficher le quotient et le reste de la division entière
4       -- de Dividende par Diviseur...
5
6   VARIABLE
7       Dividende: Entier   -- le dividende
8       Diviseur: Entier    -- le diviseur
9       Quotient: Entier    -- le quotient de la division de Dividende par Diviseur
10      Reste: Entier       -- le reste de la division de Dividende par Diviseur
11
12  DÉBUT
13      -- Demander le dividende
14      RÉPETER
15         Écrire ("Dividende (>= 0) ? ")
16         Lire (Dividende)
17         ...
18      JUSQU'À Dividende >= 0
19
20      { Dividende >= 0 }
21
22      -- Demander le diviseur
23      RÉPETER
24         Écrire ("Diviseur (> 0) ? ")
25         Lire (Diviseur)
26         ...
27      JUSQU'À Diviseur > 0
28
29      { Diviseur > 0 }
30
```

Division entière / Étape 4 : Production de l'algorithme (2)

```
31      -- Calculer le quotient et le reste de la division de Dividende par Diviseur
32      Q <--  0
33      R <-- Dividende
34      TANT_QUE R >= Diviseur FAIRE
35              Quotient <-- Quotient + 1
36              R <-- R - Diviseur
37      FIN TANT_QUE
38
39      { Dividende = Diviseur * Quotient + Reste ET 0 <= Reste < Diviseur }
40
41      -- Afficher le quotient et le reste
42      Écrire ("Le quotient est ")
43      Écrire (Quotient)
44      Écrire (" et le reste est ")
45      Écrire (Reste)
46      Écrire (".")
47
48  FIN Afficher_Quotient_Reste
```

Qualité d'un raffinage

Présentation des raffinages

- Les raffinages commencent par R0 et des exemples (R0 est une action)
- Un raffinage doit être bien présenté (indentation).

```
Ri : Comment ≪ action complexe ≫ ?
     actions combinées au moyen de structures de contrôle
```

- Tous les Ri sont écrits contre la marge.
- Un raffinage ne doit pas apparaître avant l'action complexe qu'il décompose
- Une action complexe commence par un verbe à l'infinitif

Règles

- Le vocabulaire utilisé doit être précis et clair.
- Chaque niveau de raffinage doit apporter suffisamment d'information (mais pas trop).
- Le raffinage d'une action (combinaison sous-actions) doit décrire complètement cette action.
- Le raffinage d'une action ne doit décrire que cette action.
- Éviter les structures de contrôle déguisées (si, tant que) : les expliciter (if, while. . .)
- Ne pas utiliser ≪ Comparer ≫, ≪ Vérifier ≫... et préférer ≪ Déterminer ≫, ≪ Calculer ≫...

- Certaines de ces règles sont subjectives !
- L'important est de pouvoir expliquer et justifier les choix faits

Vérifier un raffinage

Indices d'actions complexes manquantes

1. Plusieurs conditionnelles ou répétitions dans un même raffinage
2. Beaucoup d'actions dans le raffinage (par exemple plus de 5 ou 6)

Liens entre action complexe (C) et les sous-actions (A) de son raffinage

1. La combinaison d'actions du raffinage de C est la réponse à la question "Comment ≪ C ≫ ?"
   - par définition !
2. Si la réponse à ≪ Pourquoi A ? ≫ ne donne pas C, alors :
   - soit on a trouvé un meilleur nom pour A (ou C)
   - soit on a identifié une action complexe intermédiaire entre C et A (il faut l'ajouter)
   - soit A n'est pas à sa place (ne fait pas partie des objectifs de C)

Flots de données

- pour vérifier les communications entre niveaux :
  - les sous-actions doivent produire les résultats (out) de l'action ;
  - les sous-actions peuvent (doivent) utiliser les entrées (in) de l'action.
- l'enchaînement des actions au sein d'un niveau :
  - une donnée ne peut être utilisée (in) que si elle a été produite (out) avant

Bilan

Une méthode itérative en 4 étapes

- **Spécification :** description précise et claire du problème à résoudre. Elle comprend :
  - La description du problème
  - Les exemples (tests)
- $\mathbf{R_0}$ est une reformulation du problème à résoudre
  - Extrait à partir des spécifications

Principe appliqué

Une action A quelconque du raffinage de niveau i, est décomposée dans un raffinage de niveau $i + 1$ (réponse à la question **comment**) par :

- des actions de niveau de détail plus fin
- combinées grâce à une structure de contrôle

Bilan (2)

Étapes de raisonnement pour raffiner une action du niveau $\mathbf{R}_i$ dans le niveau $\mathbf{R}_{i+1}$

- **Étape 1.** Comprendre le problème décrit par l'action considérée.
  - But : Répondre à la question **QUOI ?** : "Que (**QUOI**) faut-il faire **précisément** ?" Le problème doit être bien compris, bien posé et suffisamment précis.
  - Reformuler le problème (R0)
  - Identifier des exemples (tests)
- **Étape 2.** Trouver une solution informelle (répondre à la question **COMMENT ?**)
  - Décrire **comment** résoudre le problème posé par l'action de $\mathbf{R}_i$
  - Identifier les entités manipulées
- **Étape 3.** Structurer la solution informelle
  - Identifier, ordonner et regrouper les actions
  - Identifier les flots de données (variables)
  - Identifier la **structure de contrôle** pour combiner les actions
  - Décomposer les actions identifiées qui sont complexes
- **Étape 4.** Construire l'algorithme en mettrant à plat l'arbre des raffianges :
  - parcours préfixe de l'arbre
  - les actions complexes (nœuds) deviennent des commentaires
  - les actions élémentaires (feuilles) sont les instructions du programme

Complément : La méthode des raffinages présentée comme un raffinage

Objectif

Utiliser la méthode des raffinages pour expliquer comment produire un programme pour un problème donné en utilisant la méthode des raffinages.

Étape 1 : Comprendre le problème

- Il vient d'être présenté et illustré !
  - R0 : Construire un algorithme (pour un problème posé)
- Exemple :
  - Afficher le quotient et reste de la division entière... (donnée) et le développement correspondant (résultat)
- Attention, le processeur n'est pas notre langage algorithmique mais un être humain.

Étape 2 : Trouver une solution informelle.

- C'est la méthode des raffinages expliquée dans ce chapitre...

Complément : La méthode des raffinages présentée comme un raffinage (2)

Étape 3 : Formalisation de la solution

```
1  R1 : Comment ≪ Construire un algorithme (pour un problème posé) ≫ ?
2    Comprendre le problème
3    Trouver une solution informelle
4    Structurer cette solution
5    Produire l'algorithme
6    Tester le programme
```

Étape 3 : Formalisation de la solution avec flot de donnée

```
1  R1 : Comment ≪ Construire un algorithme (pour un problème posé) ≫ ?
2    Comprendre le problème              problème: in ; R0, exemples : out
3    Trouver une solution informelle     problème, R0, exemples : in ; idée : out
4    Structurer cette solution           R0, idée, exemples: in ; raffinages : out
5    Produire l'algorithme               raffinages : in ; algorithme : out
6    Tester le programme                 algorithme, tests : in ; erreurs : out
```

Les actions identifiées sont complexes, il faut donc les détailler, raffiner (et faire apparaître les raffinages de niveau 2 et plus).

Complément : La méthode des raffinages présentée comme un raffinage (3)

Décomposition des actions complexes introduites dans $\mathbf{R}_1$

```
1   R2 : Comment ≪ Comprendre le problème ≫ ?
2          Reformuler le problème
3          Identifier des jeux de tests
4
5   R2 : Comment ≪ Structurer cette solution ≫ ?
6        Répéter
7            Construire le raffinage suivant
8            Vérifier le raffinage suivant
9            Raffiner les actions complexes
10           Vérifier l'ensemble de l'algorithme
11       JusquÀ pas d'erreurs
12
13  R2 : Comment ≪ Produire l'algorithme ≫ ?
14           :..
15
16  R2 : Comment ≪ Tester le programme ≫ ?
17         ...
```

Complément : La méthode des raffinages présentée comme un raffinage (4)

Décomposition des actions complexes introduites dans les $\mathbf{R}_2$

```
1   R3 : Comment ≪ Construire le raffinage suivant ≫ ?
2           Lister les actions
3           Identifier les flots de données
4           Ordonner les actions
5           Regrouper les actions
6           Choisir la structure de contrôle pour combiner les actions
7
8   R3 : Comment ≪ Vérifier le raffinage suivant ≫ ?
9           Vérifier que les actions introduites font R0
10          Vérifier que les actions introduites ne font que R0
11          Vérifier le niveau d'abstraction des actions
12          Vérifier l'enchaînement des actions
13
14  R3 : Comment ≪ Raffiner les actions complexes ≫ ?
15          TantQue il y a des actions complexes Faire
16                Choisir l'action la moins bien comprise
17                Construire le raffinage de cette action
18                Vérifier ce raffinage
19          FinTQ
20  ...
```

Étape 4 : Produire l'algorithme

Il suffit de mettre à plat les raffinages.

Un exercice

Recherche des diviseurs d'un nombre

Afficher les diviseurs entiers d'un nombre entier positif.

Plan

- Procédures
- Fonctions
- Surcharge
- Récursivité
- Raffinages et sous-programmes
- Bonnes pratiques
- Fonctions et Procedures en Ada
- Exercices

Introduction

Objectif : définir des parties de programmes sous forme d'opérateurs ou de fonctions à l'image des

- fonctions mathématiques : `Sin(x)`, `factorielle(n)`,
- fonctions associées aux types : `FLOAT(x)`
- actions d'Entrée-Sortie : `PUT(x)`, `GET(x)`
- ...

Sous-programmes

Ils permettent

- d'identifier, en les nommant, des parties de programmes qui seraient réutilisées pour un ensemble de valeurs de données ;
- de définir un mécanisme (appel) permettant d'exécuter ces parties de programme pour des valeurs particulières de cet ensemble de données

Introduction

Cas d'étude

Considérons l'équation $2 * x + 3 = 0$ ayant pour inconnue x
Problème posé : trouver la racine $x = -3/2$
Un algorithme qui résout cette équation serait composé d'une déclaration de deux constantes et d'une instruction d'affectation (`x<-- -3/2`) ou plus précisément (`x<-- -3.0/2.0`)

Abstraction

On peut définir des équations paramétrées (familles d'équations) au sens mathématique

$$\begin{aligned} a * x + 3 &= 0 \quad (1) \\ 2 * x + b &= 0 \quad (2) \\ 2 * x + 3 &= c \quad (3) \\ a * x + b &= 0 \quad (4) \\ a * x + b &= c \quad (5) \end{aligned}$$

a, b et c sont des **paramètres formels**. Ils sont issus d'une action **d'abstraction** sur l'équation $2 * x + 3 = 0$

Introduction

Application

On peut donner des valeurs aux paramètres et obtenir plusieurs équations particulières. Ainsi pour l'équation (4) $a * x + b = 0$ on peut obtenir

$$(-2) * x + 5 = 0 \quad (4.a)$$
$$3 * x + (-6) = 0 \quad (4.b)$$
$$0 * x + 2 = 0 \quad (4.c)$$
$$0 * x + 0 = 0 \quad (4.d)$$
$$4 * x + b = 0 \quad (4.e)$$
$$a * x + 4 = 0 \quad (4.f)$$

Les équations ci-dessus sont issues d'une **application** (valuations des paramètres). Il s'agit de **paramètres réels** ou de **paramètres effectifs**

Introduction

Propriétés

Est-ce que toutes les applications sont acceptables ?

- Tous les paramètres ne sont pas valués : on obtient une autre famille d'équations, moins grande.

$$4 * x + b = 0 \quad (4.e)$$
$$a * x + 4 = 0 \quad (4.f)$$

- Tous les paramètres sont valués : on obtient une SEULE équation

$$(-2) * x + 5 = 0 \quad (4.a)$$
$$3 * x + (-6) = 0 \quad (4.b)$$
$$0 * x + 2 = 0 \quad (4.c)$$
$$0 * x + 0 = 0 \quad (4.d)$$

Dans le cours de programmation impérative, nous ne considérons que les cas où **TOUS** les paramètres sont valués.

Introduction

Propriétés

Parmi les cas où TOUS les paramètres sont valués, on obtient trois catégories d'équations

- équations qui sont acceptables (correctes) pour le problème "trouver la racine de l'équation"

$$(-2) * x + 5 = 0 \quad (4.a)$$
$$3 * x + (-6) = 0 \quad (4.b)$$

  avec pour racines $5/2$ pour 4.a et $6/3$ pour 4.b

- équations qui ne sont pas acceptables (correctes) pour le problème "trouver la racine de l'équation"

$$0 * x + 2 = 0 \quad (4.c)$$
$$0 * x + 0 = 0 \quad (4.d)$$

  avec deux cas
  - équation incorrecte $\Longrightarrow$ équation sans solution pour 4.c
  - infinité de solutions pour 4.d

Il faut donc distinguer tous ces cas avant de résoudre ! ! ! !
Comment : en partitionnant l'ensemble des équations de la forme $a * x + b = 0$ ayant pour paramètres a et b

Introduction

Propriétés

Comment : en partitionnant l'ensemble des équations de la forme $a * x + b = 0$ ayant pour paramètres a et b

Faisons apparaître les trois cas précédents

- $a \neq 0 \Longrightarrow x = -b/a$
- $a = 0 \wedge b \neq 0 \Longrightarrow$ Equation erronée ou impossible
- $a = 0 \wedge b = 0 \Longrightarrow$ Infinité de solutions

Impact sur la définition de test

- La partition définie ci-dessous permet de définir des tests
- Par exemple
  - $a = 3, b = 5 \Longrightarrow$ la solution est $x = -5/3$
  - $a = 0 \wedge b = -2 \Longrightarrow$ Equation erronée ou impossible
  - $a = 0 \wedge b = 0 \Longrightarrow$ Infinité de solutions

Introduction

Abstraction – Application

- Abstraction : Identification
  - d'un ensemble de paramètres
  - de conditions sur ces paramètres pour définir différentes equations
  - du calcul d'une solution sur la base de ces conditions
- Application
  - fourniture de valeurs de paramètres d'une équation respectant les conditions définies sur les paramètres
  - obtention de la solution de l'équation définie par ces valeurs de paramètres

Contrat

Il est nécessaire de

- définir les conditions sur les paramètres afin de garantir le résultat du calcul effectué par l'abstraction
- fournir des valeurs de paramètres qui respectent ces conditions

$\Longrightarrow$ Définition de **contrat**.

Sous-Programme

Notion de contrat

Le contrat est un **accord** entre (deux) **parties**. Il définit des **obligations** pour chacune des parties.
Dans notre situation, on a deux parties :

- la personne qui souhaite résoudre une équation particulière
- le mathématicien qui résout l'équation.

Par exemple, pour résoudre $a * x + b = 0$ on passera le contrat suivant

- **Obligation** pour la personne qui souhaite résoudre l'équation : $a \neq 0$
- **Obligation** pour le mathématicien qui résout l'équation : trouver $x = -b/a$

Attention, d'autres contrats sont possibles, et donc avec d'autres obligations ?
Exemples ?

Sous-Programme

Définition

Sous-programme (appelé)

- Séquence d'instructions, exécutées à plusieurs points d'un programme (appelant), concourant à la production d'un calcul donné ou à la réalisation d'une action.

Ce principe a été transposé en algorithmique et programmation (1)

- **Sous-programmes** avec des **paramètres** pour définir des "familles" de programmes (**Abstraction**)
  - **paramètres en entrée** (avec le **mode** `In` **pour les flots en entrée(données)**). Ici, a et b
  - **paramètres en sortie** (avec le **mode** `Out` **pour les flots en sortie (résultats)**). Ici x

- **Appel** des sous-programmes avec des valuations de paramètres. On identifie alors des flots de données (**Application**)
  - les valuations de paramètres en entrée. Par exemple 2 et 3 pour a et b respectivement
  - les valuations de paramètres en sortie, une fois le sous-programme exécuté. Ici $-3/2$ pour x

Sous-Programme

Ce principe a été transposée en algorithmique et programmation (2)

- **Obligations** formalisées à l'aide de conditions (expressions logiques)
- **Pré-condition** obligation qui doit être satisfaite par l'appelant (programme qui appelle le sous-programme). On parle de
  - **Before Predicate**, **Prédicat Avant**
  - **Hypothèse**
  - **Nécessite**, **Requires**

  Bénéfice. L'appelé (le sous-programme appelé) n'a pas besoin de vérifier la pré-condition.
- **Post-condition** obligation qui est assurée par l'appelé à l'issue de son exécution. On parle de
  - **After Predicate**, **Prédicat Après**
  - **Conclusion**
  - **Assure**, **Ensures**

  Bénéfice. La post-condition est garantie pour l'appelant

Exemple : Algorithme pour guider le robot

Considérons un Robot capable d'exécuter les actions `AVANCER` d'une case et `PIVOTER` à droite de 90°.

Algorithme `Guider_robot`

```
  -- Guider le robot de la salle de cours au secrétariat.
Début
  -- Sortir de la salle de cours
  --   Progresser de 2 cases
  AVANCER
  AVANCER

  --   Tourner à gauche
  PIVOTER
  PIVOTER
  PIVOTER

  --   Progresser de 3 cases
  AVANCER
  AVANCER
  AVANCER

  -- Longer le couloir (jusqu'à la porte du vestibule)
  --   Tourner à droite
  PIVOTER
  ...
Fin
```

Les procédures : exemple

Procédures en algorithmique

Un exemple pour faire avancer le robot de N cases

```
PROCÉDURE Progresser(F_N : in Entier) EST
    -- Faire avancer le robot
    --
    -- Necessite : F_N > 0

 DÉBUT
    POUR i DE 1 A F_N FAIRE
        AVANCER
    FIN POUR
 FIN Progresser
```

Exemple : appel des sous-programmes (SP)

L'algorithme précédent peut être écrit de la façon suivante.

```
Algorithme Guider_robot
  -- Guider le robot de la salle de cours au secrétariat.
Début
  -- Sortir de la salle de cours
  progresser(2)

  --   Tourner à gauche
  tourner_gauche
  progresser(3)

  -- Longer le couloir (jusqu'à la porte du vestibule)
  tourner_droite
  ...
Fin
```

- Ici, les paramètres **réels (ou effectifs)** des appels de la procédure `progresser` sont 2 et 3, des constantes littérales.
- La procédure `tourner_gauche` n'a pas de paramètre, elle exécute une séquence de 3 actions `PIVOTER`

Sous-programmes (SP) en algorithmique

Sous-programmes

Il existe 2 types de sous-programmes :

- Les procédures
  Instructions définies par le programmeur
- Les fonctions
  Opérateurs définis par le programmeur. Cet opérateur retourne une valeur (expression), contrairement à la procédure.

Introduction

La suite ...

Définir les notions de sous-programmes (SP) en algorithmique pour

- Les fonctions
- Les procédures

Sous-programmes : Procédures et Fonctions

Deux types de sous-programmes

- **Fonction.** Vue comme **l'abstraction d'une expression**, ou **l'abstraction de données**. Elle a un nom.

  Exemple : `Sin(x + 3.2)`, `Fact(3)`

- **Procédure.** Vue comme **l'abstraction de traces d'opérations** ou **l'abstraction de contrôle**. Elle a un nom.

  Exemple : `Avancer(10)`, `Permuter(x, y)`

Dans la suite nous donnons les constructions algorithmiques permettant de

- **Spécifier** et **Déclarer** un sous-programme
- Définir un **contrat** associé à chaque sous-programme
- **Concevoir**, par raffinage, le corps d'un sous programme $\Longrightarrow$ On obtient l'**implantation**
- **Appeler** un sous-programme

Procédure

Définition

- Une procédure est une abstraction de **contrôle**. Elle définit une **action** ou une **instruction**.
- Une procédure résulte du raffinement d'une **action** complexe que le processeur n'est pas en mesure **d'exécuter**
- Une fois définie, une procédure a le statut d'une **action élémentaire** ou une **"instruction"** disponible pour le processeur
- Une procédure peut avoir un (ou des) effet(s) de bord lorsqu'elle modifie ses paramètres.
- A l'application, une procédure réalise une action

Spécifier et Déclarer une procédure

Spécification de procédure

Une procédure est **spécifiée** par la définition de

- son entête (sa signature ou son profil).
- sa sémantique (sens)
  - son contrat
  - l'ensemble des tests associés

Définition de procédure

Une procédure est **définie** par la déclaration de son

- entête
- corps

Remarque

- La spécification d'une procédure est définie par $\mathbf{R}_0$.
- Le corps d'une procédure est l'algorithme obtenu par la méthode des raffinages.

Spécifier et Déclarer une procédure

Spécification

Identificateur de procédure	Nom de la procédure
Une liste de paramètres	Séquence (ordonnée) de paramètres typés issus de l'abstraction (éventuellement vide)
Une pré-condition	Obligation de type **Prédicat avant** ou **Requires**
Une post-condition	Obligation de type **Prédicat après** ou **Ensures**
Un ensemble de tests/exemples	Tests/exemples (entrées + sorties attendues) permettant de valider la procédure
Des exceptions	On verra plus tard ...

Spécifier et Déclarer une procédure

Les paramètres

Un paramètre a

- Une sémantique ou un rôle représenté par un commentaire
- Un nom
- Un type
- Un mode de passage :
  - Donnée ou Entrée représenté par `In`
  - "Donnée/Résultat" ou Entrée/Sortie représenté par `In/Out`
  - "Résultat" ou Sortie représenté par `Out`

Remarques

- Cas d'une fonction $\Longrightarrow$ Un seul mode de passage `In`
- Cas d'une procédure $\Longrightarrow$ modes de passage `In` ou `Out` ou `In/Out`
- Les paramètres décrivent les flots de données.
- Ces flots de données doivent être respectés dans les différents raffinages

Définir le contrat associé à une procédure

Composition d'un contrat

Composants d'un contrat

- La **signature** de la procédure,
  - Le **nom** du sous-programme et sa sémantique (ou rôle)
  - La **liste des paramètres** : nom, type, mode et sémantique (ou rôle) de chaque paramètre
- La **pré-condition** : condition (requise) vérifiée par les paramètres de mode `In`, ou `In/Out` avant exécution de la procédure
- La **post-condition** : condition (assurée) vérifiée par les paramètres en mode `Out`, ou `In/Out` après exécution de la procédure si la condition était vérifiée avant son exécution.
- Les **erreurs (ou exceptions)** : on verra plus tard ...

Définir le contrat associé à une procédure

Effets d'un contrat de procédure

Le contrat assure que :

- si la pré-condition d'une procédure est assurée avant l'appel de la procedure,
- et si la procédure termine,
- alors la post-condition sera assurée après l'appel de la procédure

Si l'une des 2 hypothèses ci-dessus n'est pas respectée, alors le comportement de la procédure est imprévisible

Définir le contrat associé à une procédure

Contrat : définition

Pour toute procédure on définit un contrat sous forme de commentaire

```
1  -- nom : nom de la procédure
2  -- sémantique:  décrire ce que réalise la procédure
3  -- paramètres:
4  --  F_Param_1 : Mode (In, In/Out, Out) Type; --Rôle du paramètre
5  --  ...
6  --  F_Param_n : Mode (In, In/Out, Out) Type; --Rôle du paramètre
7  -- pré-condition: Conditions sur les paramètres en entrée (in)
8  -- post-condition: Conditions sur les paramètres en sortie (out)
```

Remarques

- Une procédure peut ne pas avoir de paramètre (0 paramètre)
- Par convention, on préfixera les paramètres formels par `F_`
  Intérêt. Faire la différence avec les paramètres réels ou effectifs lors de l'appel.

Définir la spécification d'une procédure

Spécification : exemple

- Le contrat

```
1  -- nom: Permuter
2  -- sémantique: permutation de 2 variables
3  -- paramètres:  F_X: In/Out Réel    --  première variable
4  --              F_Y: In/Out Réel    --  seconde variable
5  -- pré-conditions:  Vrai
6  -- post-condition: F_X=F_Y'Avant et F_Y=F_X'Avant
```

- Les tests

```
1  -- Tests
2  --   Entrées : F_X = 3 et F_Y = -5.  Sorties F_X = -5 et F_Y = 3
```

Concevoir le corps d'une procédure

Conception d'une procédure

Lors de la conception d'un programme, une action peut être définie par une procédure. Il faut alors

- la spécifier en donnant son contrat et les tests associés
- raffiner l'action abstraite définie par cette procédure.

Identification des procédures

- Lors du processus raffinage, le concepteur peut décider qu'une action abstraite intermédiaire, sera implantée par une procédure
- Dans ce cas,
  - cette action abstraite définit le niveau $\mathbf{R}_0$ de cette procédure,
  - les flots de données identifiés à cette étape de raffinage déterminent les paramètres et leurs modes de passage

Concevoir le corps d'une procédure ;

Corps d'une procédure : définition

- Le corps d'une procédure regroupe les actions effectuées par cette procédure
- Le corps représente l'algorithme issu du raffinement du contrat

```
1   -- nom : nom de la procédure
2   -- sémantique:  décrire ce que réalise la procédure
3   -- paramètres:
4   --   F_Param_1 : Mode (In, In/Out, Out) Type; --Rôle du paramètre
5   --    ...
6   --   F_Param_n : Mode (In, In/Out, Out) Type; -- Rôle du paramètre
7   -- pré-condition : Condition sur les paramètres en entrée (In et In/out)
8   -- post-condition : Condition sur les paramètres en sortie (Out)
9   -- Tests de la procédure
10
11  PROCÉDURE Nom_Procedure (F_Param_1 : (In, In/Out, Out) Type; ; ... ;
12                           F_Param_n : (In, In/Out, Out) Type ) EST
13
14      Déclarations de variables locales
15
16  DÉBUT
17      Instructions du corps de la procédure
18
19  FIN Nom_Procedure
```

Concevoir le corps d'une procédure ;

Corps d'une procédure : exemple

```
1
2   -- nom: Permuter
3   -- sémantique: permutation de 2 variables
4   -- paramètres:  F_X:  In/Out Réel --  première variable
5   --               F_Y:  In/Out Réel --  seconde variable
6   -- pré-conditions: Fx_Avant = a et Fy_Avant = b
7   -- post-condition: F_X = b et F_Y = a
8   -- Test Entrées : F_X=3 et F_Y = -5. Sorties F_X=-5 et F_Y=3
9
10  PROCÉDURE Permuter (F_X : IN OUT Réel;
11                      F_Y : IN OUT Réel) est
12     Memoire : Réel
13
14  DÉBUT
15     Memoire <-- F_X
16     F_X <-- F_Y
17     F_Y <-- Memoire
18
19  FIN Permuter
```

Appeler une procédure

Appel d'une procédure : une instruction

- L'appel d'une procédure est semblable à l'écriture d'une instruction
- Les paramètres **formels** sont associés aux paramètres **réels** dans l'ordre de leur déclaration dans la signature de la procédure

```
--! S'assurer que la pré-condition est satisfaite

Nom_procedure(Par_Reel_1, ..., Par_Reel_n)

--! La post-condition est satisfaite
```

A propos des pré-condition et de post-condition.

- Il faut aussi se poser la question d'écrire pré-conditions et post-conditions pour toutes les instructions d'un programme ! ! !

Appeler une procédure : exemple d'un programme de test

```
1   PROGRAMME Test_Permuter est
2
3       -- Permuter les valeurs associées aux paramètres....
4       PROCÉDURE Permuter (F_X : IN OUT Réel ; F_Y : IN OUT Réel) EST
5       DÉBUT
6           ...
7       FIN Permuter
8
9   VARIABLES
10      Val1 : Réel
11      Val2 : Réel
12  DÉBUT
13      -- Définition des données de test
14      Val1 <- 3
15      Val2 <- -5
16
17      { Val1 = 3 ET Val2 = 5 }
18      Permuter(Val1, Val2)
19      { Val1 = -5 ET Val2 = 3 } -- valeurs permutées
20
21      SI Val1 = -5 et Val2 = 3 ALORS
22          ÉCRIRE("Test réussi")
23      SINON
24          ÉCRIRE("Échec du Test")
25      FIN_SI
26  FIN Test_Permuter
```

Appeler une procédure : exemple d'un programme principal

```
1   PROGRAMME Principal EST
2
3       -- Permuter les valeurs associées aux paramètres....
4       PROCÉDURE Permuter (F_X : IN OUT Réel;
5                           F_Y : IN OUT Réel) EST
6       DÉBUT
7       ...
8       FIN Permuter
9
10  VARIABLE
11      Abscisse : Réel  -- Abscisse d'un point
12      Ordonnee : Réel  -- Ordonnée d'un point
13  DÉBUT
14    ...
15    Ecrire("Donner la valeur de l'abscisse")
16    Lire(Abscisse)
17    Ecrire("Donner la valeur de l'ordonnée")
18    Lire(Ordonnee)
19
20    { Abscisse = Val_Abs et Ordonnée = Val_Ord ont une valeur }
21    Permuter(Abscisse, Ordonnee)
22    { Abscisse = Val_Ord et Ordonnée = Val_Abs } -- valeurs permutées
23    ...
24  FIN Principal
```

Nommer explicitement les paramètres

```
1   PROGRAMME Principal EST
2       -- Permuter les valeurs associées aux paramètres....
3       PROCÉDURE Permuter (F_X : (In/Out) Réel;
4                           F_Y : (In/Out) Réel) EST
5       DÉBUT
6       ...
7       FIN Permuter
8
9   VARIABLE
10      Abscisse : Réel  -- Abscisse d'un point
11      Ordonnee : Réel  -- Ordonnée d'un point
12  DÉBUT
13      ...
14      Ecrire("Donner la valeur de l'abscisse")
15      Lire(Abscisse)
16      Ecrire("Donner la valeur de l'ordonnée")
17      Lire(Ordonnee)
18
19      { Asbcisse=Val_Abs et Ordonnée=Val_Ord ont une valeur }
20      Permuter(F_Y => Ordonnee , F_X => Abscisse)
21      { Abscisse=Val_Ord et ordonnée=Val_Abs sont permutées }
22  FIN Principal
```

- L'appel à la procédure Permuter peut s'écrire en nommant les paramètres :

```
Permuter(F_Y => Ordonnee, F_X => Abscisse)
```

- Certains langages autorisent des valeurs par défaut pour les paramètres formels.

Procédures : Quelques règles et contraintes

- **Règle1.**
  - Même nombre de paramètres effectifs que de paramètres formels
  - Mêmes types pour le ième paramètre de chaque liste, quel que soit i
- **Règle 2.** Si le paramètre formel est une **donnée (IN)** le paramètre effectif est **une expression**.
- **Règle 3.** Si le paramètre formel est une **donnée/résultat, In/Out** ou un **résultat, Out**, le paramètre effectif **doit être une variable**.
- **Règle 4.** Un paramètre **donnée, In ne peut pas être modifié dans un sous-programme** donc
  - ne peut pas être affecté
  - ne peut pas être transmis à un sous-programme avec un mode Résultat/Out
- **Règle 5.** Un paramètre **résultat/Out ne peut pas être consulté dans un sous-programme**, donc
  - ne peut pas être à droite d'une affectation
  - ne peut pas apparaître dans une expression
  - ne peut pas être transmis à un SP avec un mode IN

Fonction

Définition

- Une fonction est une **abstraction de données**. Elle définit une **expression.**
- Une fonction résulte du raffinement d'une **expression** abstraite que le processeur n'est pas en mesure **d'évaluer**
- Une fois définie, une fonction devient une **expression** comprise par le processeur
- Une fonction ne possède pas d'effet de bord à la différence d'une procédure ayant des paramètres en mode `out` ou `in out`
- L'application d'une fonction **produit** une valeur. On dit que la fonction **retourne** (ou **renvoie**) une valeur. Celle de l'expression représentée par la fonction.

Spécifier et Déclarer une fonction

Spécification de fonction

Comme une procédure, une fonction est **spécifiée** par la définition de

- son entête (sa signature ou son profil)
- sa sémantique (sens)
  - son contrat
  - l'ensemble des tests associés

Définition de fonction

Comme une procédure, une fonction est **déclarée** par la définition de

- son entête
- son corps

Spécifier et Déclarer une fonction

Identificateur de fonction	Nom de fonction
Une liste de paramètres	Séquence (ordonnée) de paramètres typés issus de l'abstraction (éventuellement vide) tous déclarés en mode `In`
Un type de retour	Type de l'expression retournée ou renvoyée par la fonction
Une pré-condition	Obligation de type **Prédicat avant** ou **Requires**
Une post-condition	Obligation de type **Prédicat après** ou **Ensures**
Un ensemble de tests	Tests (entrées + sorties attendues) permettant de valider la fonction
Des exceptions	On verra plus tard ...

Remarques

- Une fonction **retourne** la valeur d'une expression, elle **ne l'affiche pas**
- A la différence d'une fonction, une procédure **n'a pas de type de retour**
- A la différence d'une procédure, une fonction **n'a pas de paramètre en mode** `Out`

Spécifier et Déclarer une Fonction

Les paramètres

Un paramètre a

- Une sémantique ou un rôle représenté par un commentaire
- Un nom
- Un type
- Un mode de passage **UNIQUE**
  - Donnée ou Entrée représenté par `In`

Remarques

- Cas d'une fonction $\Longrightarrow$ Un seul mode de passage `In`
- Les paramètres décrivent les flots de données (idem procédures).
- Ces flots de données doivent être respectés dans les différents raffinages (idem procédures).

Définir le contrat associé à une fonction

Composition d'un contrat

Composants du contrat d'une fonction (semblables aux procédures)

- La **signature** de la fonction
  - Le **nom** du sous-programme
  - La **liste des paramètres** : nom, Type, mode `In`, sémantique (ou rôle) de chaque paramètre
  - Le **type de retour** de l'expression retournée
- La **pré-condition** condition (requise) vérifiée par les paramètres en mode `D, In` avant exécution de la fonction
- La **post-condition** : condition (assurée) vérifiée par le résultat retourné par la fonction après son exécution si la pré-condition était vérifiée avant cette exécution
- Les **erreurs (ou Exceptions)** : on verra plus tard ...

Remarque

Dans le cas d'une fonction, les paramètres n'étant pas modifiés, on peut omettre l'utilisation des attributs `'Avant` et `'Après` dans l'écriture des pré-conditions et des post-conditions.

Définir le contrat associé à une fonction

Effets d'un contrat de fonction

Comme pour une procédure, le contrat assure que :

- si la pré-condition d'une fonction est assurée avant l'appel de la fonction,
- et si la fonction termine,
- alors la post-condition sera assurée après l'appel de la fonction

Si l'une des 2 hypothèses ci-dessus n'est pas respectée, alors le comportement de la fonction est imprévisible

Définir le contrat associé à une fonction

Contrat : définition

Pour toute fonction on définit un contrat sous forme de commentaire

```
1  -- Nom : nom de la fonction
2  -- Sémantique :   sémantique de la fonction
3  -- Paramètres
4  --   F_Param_1 :  mode (in) Type -- rôle du paramètre
5  --   ...
6  --   F_Param_n :  mode (in) Type -- rôle du paramètre
7  -- Type de retour : Type du résultat retourné
8  -- Pré-condition  : Conditions sur les paramètres en entrée
9  -- Post-condition : Conditions sur le résultat retourné
```

Remarques

- Comme pour les procédures, une fonction peut ne pas avoir de paramètre (0 paramètre)
- Par convention, on préfixera les paramètres formels par `F_`

  Intérêt. Faire la différence avec les paramètres réels ou effectifs lors de l'appel.

Définir la spécification d'une fonction

Spécification : exemple

- Le contrat

```
1  -- nom : Minimum
2  -- sémantique : calculer le plus petit entier parmi 2 entiers
3  --              strictement positifs
4  -- paramètres:  F_X : (In) Entier  --  1er entier à comparer
5  --               F_Y:  (In) Entier  -- 2ème entier à comparer
6  -- Type retour: Entier représentant le plus petit des 2 nombres
7  -- précondition:    F_X > 0 et F_Y > 0
8  -- postcondition: Résultat > 0 et Résultat = Min(F_X,F_Y)
```

- Les tests

```
1  -- Tests :
2  --  Entrées (F_X < F_Y) : F_X = 2 et F_Y = 4 ==> Résultat = 2
3  --  Entrées (F_X > F_Y) : F_X = 8 et F_Y = 5 ==> Résultat = 5
4  --  Entrées (F_X = F_Y) : F_X = 3 et F_Y = 3 ==> Résultat = 3
```

Concevoir le corps d'une fonction

Conception d'une fonction

Lors de la conception d'un programme, une expression, inconnue du processeur, peut être définie par une fonction. Il faut alors

- la spécifier en donnant son contrat et les tests associés
- raffiner l'expression définie par cette fonction.

Identification d'une fonction

- Lors du processus raffinage, le concepteur peut décider qu'une expression, inconnue du processeur, sera implantée par une fonction
- Dans ce cas,
  - cette expression définit le niveau R0 de cette fonction,
  - les flots de données identifiés à cette étape de raffinage déterminent les paramètres de la fonction
  - Exemple : lors de la définition de conditions complexes

Concevoir le corps d'une fonction ;

Corps d'une fonction : définition

- Le corps comprend une instruction particulière `RETOURNE` qui retourne/renvoie le résultat de la fonction

```
1  -- nom : nom de la fonction
2  -- sémantique :   sémantique de la fonction
3  -- paramètres
4  --   F_Param_1 :  mode (In) Type --  rôle du paramètre
5  --   ...
6  --   F_Param_n :  mode (In) Type -   rôle du paramètre
7  -- Type de retour : Type du résultat retourné
8  -- Pré-condition  : Condition sur les paramètres en entrée
9  -- Post-condition : Condition sur le résultat retourné
10
11 FONCTION Nom_Fonction (F_Param_1 : IN Type ; ... ;
12                        F_Param_n : IN Type) RETOURNE Type_Retour EST
13     Déclarations de variables locales
14 DÉBUT
15
16     Instructions de calcul du résultat par une expression Reslt
17
18     RETOURNE Reslt
19
20 FIN Nom_Fonction
```

ATTENTION. Du fait de leur mode de passage en `IN`, les paramètres d'une fonction ne peuvent être modifiés dans le corps d'une fonction (Pas d'effet de bord).

Concevoir le corps d'une fonction

Corps d'une fonction : exemple

Définition du corps de la fonction `Minimum` précédemment spécifiée

```
1   -- nom : Minimum
2   -- sémantique : calculer le plus petit entier parmi 2 entiers
3   --               strictement positifs
4   -- paramètres:  F_X : (In) Entier  --  1er entier à comparer
5   --               F_Y : (In) Entier  --  2ème entier à comparer
6   -- Type retour: Entier représentant le plus petit des 2 nombres
7   -- précondition:    F_X'Avant >= 0 et F_Y'Avant >= 0
8   -- postcondition: Résultat > 0 et Résultat= Min(F_X'Avant,F_Y'Avant)
9
10  FONCTION Minimum (F_X : (In) Entier ; F_Y: (In) Entier ) RETOURNE Entier EST
11
12  VARIABLE    --// Déclarations de variables locales à la fonction
13      Résultat : Entier     -- le résultat de la fonction
14
15  DÉBUT
16
17      SI F_X > F_Y ALORS
18          Résultat <-- F_Y
19      SINON
20          Résultat <-- F_X
21      FIN SI
22
23      RETOURNE Résultat
24
25  FIN Minimum
```

Appeler une fonction

Appel d'une fonction : une expression

- le résultat d'une fonction est **une expression (valeur)**, donc l'appel d'une fonction doit **être traité comme une expression**.
- Les paramètres d'appels sont des paramètres dits **effectifs** (paramètres **réels**)
- Les paramètres **formels** sont remplacés par les paramètres **réels** dans l'ordre de la déclaration de la fonction
- La pré-condition doit être vérifiée avant l'appel. Elle porte sur les paramètres **effectifs**

Quelques formes d'appels de fonction

Soient *a* et *b* deux paramètres effectifs vérifiant la pré-condition de la fonction `Minimum` précédente. Une fonction peut être utilisée partout où une expression est possible

un résultat	`x <-- Minimum(a,b)`
une expression	`y + Minimum(a,b)`
une expression booléenne	`Minimum(a,b) > 0 ET x < Y`
un affichage	`ECRIRE(Minimum(a,b))`
une conditionnelle	`SI Minimum(a,b) >0 ET x < Y ALORS ...`
une itération	`TANT_QUE Minimum(a,b) >0 ET x < Y FAIRE ...`

Appeler une fonction : un programme de test

Appel d'une fonction dans le cas **d'un programme de test** : exemple

```
1   PROGRAMME Test_Minimum EST
2   -- informations spécifiant le contrat de la fonction ....
3   -- Test réalisé
4   --       Entrées : F_X = 2 et F_Y = 4 ==> Résultat = 2
5   FONCTION Minimum ( F_A: (D/In) Entier ; F_B: (D/In) Entier ) RETOURNE Entier ES
6       ...
7   FIN Minimum
8
9   VARIABLE
10      A, B : Entier -- ...
11
12  DÉBUT
13
14      A <-- 2
15      B <-- 4
16
17      SI Minimum(A, B) = 2 ALORS
18          ECRIRE("Test Réussi")
19      SINON
20          ECRIRE("Échec du Test")
21      FIN_SI
22
23  FIN Test_Minimum
```

- Les tests réalisés sont ceux définis lors de la spécification de la fonction
- Tous les tests doivent être réalisés à l'aide d'un programme de test

Appeler une fonction : un programme principal

Appel d'une fonction dans le cas **d'un programme principal** : exemple

```
1   PROGRAMME Principal EST
2       -- Retourner le minimum.... suite omise...
3       FONCTION Minimum (F_A: IN Entier ; F_B: IN Entier) RETOURNE Entier EST
4       ...
5       FIN Minimum
6
7   VARIABLE                              -- programme principal
8       A, B : Entier -- ...
9
10  DÉBUT
11      -- Saisir deux entiers
12      RÉPÉTER
13          lire (A, B)
14      JUSQU'A  A >= 0 et B >= 0
15
16      { A >=0 et B >= 0 }  --!  la pré-condition de min est donc vérifiée
17
18      SI Minimum (A, B) = 0 ALORS
19          ÉCRIRE ("Une des deux valeurs est nulle")
20      SINON
21          ÉCRIRE ("Aucune des deux valeurs n'est nulle")
22      FIN_SI
23
24  FIN Principal
```

Appeler une fonction : un programme principal

Nommer les paramètres

```
1   PROGRAMME Principal EST
2       -- Retourner le minimum.... suite omise...
3       FONCTION Minimum (F_A: IN Entier ; F_B: IN Entier) RETOURNE Entier EST
4       .....
5       FIN Minimum
6   VARIABLE    --! programme principal
7       A, B : Entier -- ...
8   DÉBUT
9       -- Saisir deux entiers
10      RÉPÉTER
11          lire (A, B)
12      JUSQU'A A >= 0 et B >= 0
13      { A >= 0 et B >= 0 } --! la pré-condition de min est donc vérifiée
14      SI Minimum (F_A => A, F_B => B) = 0 ALORS
15          ÉCRIRE ("Une des deux valeurs est nulle")
16      SINON
17          ÉCRIRE("Aucune des deux valeurs n'est nulle")
18      FIN_SI
19  FIN Principal
```

On peut écrire de manière équivalente cet appel à la fonction `Minimum` sous la forme

```
Minimum (F_B => B, F_A =>A)
```

Fonctions : Règles et contraintes

- **Règle 1**.
  - Même nombre de paramètres effectifs que de paramètres formels
  - Même type pour le ième paramètre de chaque liste de paramètres, quel que soit i
- **Règle 2**. Dans toute fonction, le paramètre formel est une donnée, donc le paramètre effectif est **une expression** quelconque
- **Règle 3**. Etant des données, les paramètres d'une fonction ne peuvent pas être modifiés par la fonction
  - ne peuvent pas être affectés
  - peuvent apparaître à droite d'une affectation '(<--)
- Toujours écrire une fonction lorsque le traitement est une abstraction de données (évaluation d'une expression)

Fonction ou Procédure

Quand choisir une fonction ? ou une procédure ?

- **Procédure** $\Longrightarrow$ Abstraction de contrôle, Action/Instruction abstraite.
  Exemples : `Permuter`, `Afficher`, ...
- **Fonction** $\Longrightarrow$ Abstraction de données, Expression complexe.
  Exemples : `perimetre`, `Determinant`, `Est_majoritaire`, ...
- **Composition** : à la différence des procédures, les fonctions préservent la composition

Une procédure avec un paramètre en `Out` ou bien un fonction ?

- Il est possible de remplacer un fonction de la forme

```
FONCTION F (Fpar_1: T1, ...., Fpar_n: Tn) Retourne TR
```

  et un appel de la forme

```
X <-- F(Y1, ..., Yn)
```

- par une procédure de la forme

```
PROCEDURE F (Fpar_1: T1, ...., Fpar_n: Tn, F_R: TR)
```

  et un appel de la forme

```
F(Y1, ..., Yn, X)
```

- Néanmoins, nous conserverons le critère (rappelé ci-dessus) pour décider entre procédure (abstraction de contrôle) ou fonction (abstraction de données)

Procédures et fonctions : la Surcharge

Surcharge

Surcharger la définition d'un sous-programme, c'est pouvoir utiliser le **même** nom de sous-programme pour plusieurs sous-programmes distincts.
On obtient des sous-programmes différents qui portent le même nom.

Exemples

- L'opérateur + est surchargé. Il est utilisé pour l'addition d'entiers et de flottants
- Les procédures `Get` et `Put` en Ada sont surchargées. Elle peuvent être utilisées pour lire/écrire des entiers, des flottants, des caractères et des chaînes de caractères.

Surcharge de sous-programmes

Un même nom peut être utilisé pour définir plusieurs sous-programmes à condition que l'on puisse les distinguer lors de l'appel.

On distingue des sous-programmes lorsqu'ils ont

- un nombre différent de paramètres ou
- des paramètres de types différents

La récursivité

Définition de sous-programme récursif

Il s'agit d'un sous-programme dont l'implantation contient un appel à lui même.

Une fonction récursive

- Exemple : la factorielle

```
1  -- Retourner la factorielle d'un entier...
2  -- Précondition : F_Nb >= 0
3  FONCTION Factorielle (F_Nb : in Entier) RETOURNE Entier EST
4  DÉBUT
5      SI F_Nb = 0 ALORS
6          RETOURNE 1
7      SINON
8          RETOURNE F_Nb * Factorielle(F_Nb - 1)
9      FIN SI
10 FIN Factorielle
```

Exemple de procédure récursive : Trier_Fusion

- Tri fusion : Trier des tableaux selon le principe "Diviser pour régner"
  - Si le tableau contient au plus un élément, il est trié (rien à faire).
  - Sinon on découpe le tableau en 2 sous-tableaux, on trie les deux sous-tableaux (sur le même principe, récursivité) puis on les fusionne.
- On suppose que l'on dispose de la procédure Fusion qui fusionne deux tableaux triés

```
1  -- Fusionner de deux moitiés triées du tableau
2  PROCEDURE Fusionner (F_T      : IN OUT T_Tab; -- Tableau avec 2 sous-tableaux triés
3                       F_Debut  : IN Entier;    -- Indice de début du tableau F_T
4                       F_Milieu : IN Entier;    -- Indice de fin du 1er sous-tableau
5                       F_Fin    : In entier)    -- Indice de fin du tableau F_T
```

- La procédure Tri_Fusion s'écrit de façon récursive comme suit

```
1  -- Résoudre le problème du tri-fusion...
2  PROCÉDURE Trier_Fusion(F_T   : IN OUT T_Tab; -- Tableau à trier
3                         F_Beg : IN Entier;  -- Indice de début dans le tableau F_T
4                         F_End : IN Entier;  -- Indice de fin dans le tableau
5                  ) EST
6      F_Mid : Entier ;
7  DÉBUT
8      SI F_Deb < F_Fin ALORS
9          F_Mid <-- (F_Beg + F_End)/2
10         Trier_Fusion(F_T, F_Beg, F_Mid)
11         Trier_Fusion(F_T, F_Mid + 1, F_End)
12         Fusionner(F_T, F_Beg, F_Mid, F_End)
13     SINON -- le tableau a au plus un élément
14         Rien    -- donc déjà trié
15     FIN_SI
16 FIN Trier_Fusion
```

La récursivité

Propriétés

- Les sous-programme récursifs traduisent un schéma inductif
- Pour écrire un programme récursif, il faut donc identifier
  - le ou les cas de base (arrêt de montée et descente de la pile d'exécution)
  - le ou les cas inductifs (montée dans la pile d'exécution)

  et distinguer ces deux situations dans le programme
- L'exécution d'un sous-programme récursif utilise une pile pour la gestion des appels et des retours $\Longrightarrow$ la pile d'exécution

La récursivité

Récursivité croisée

- Les exemples précédents montrent une récursivité directe
- Des cas de récursivité indirecte (mutuelle) sont possibles.
- Un sous-programme A appelle un sous-programme B qui lui même appelle le sous programme A

Exemple

```
1  FONCTION Pair (F_Nb : in Entier)
2                RETOURNE Booléen EST
3  DÉBUT
4      SI F_Nb = 0 ALORS
5          RETOURNE VRAI
6      SINON
7          RETOURNE Impair(F_Nb - 1)
8      FIN SI
9  FIN Pair
```

```
1  FONCTION Impair (F_Nb : in Entier)
2                RETOURNE Booléen EST
3  DÉBUT
4      SI F_Nb = 0 ALORS
5          RETOURNE FAUX
6      SINON
7          RETOURNE Pair(F_Nb - 1)
8      FIN SI
9  FIN Impair
```

Raffinages et sous-programmes

Description

- Une étape de raffinage peut donner lieu à un sous-programme.
- Le raffinage de cette étape donne les instructions du sous-programme.
- Ce sous-programme peut être réutilisé / appelé plusieurs fois pour factoriser le code.

Etape de raffinage et données (rappel)

- Une étape de raffinage peut
  - **accéder/consulter** des données (mode **in**),
  - **produire** des données (mode **out**) et/ou
  - **accéder** et **mettre à jour** des données (mode **in out**).
- Les données échangées représentent les flots de données (déjà abordés)
- Ces données échangées sont représentées dans le sous-programme par des paramètres.

Forme générale de programme avec appels de sous-programmes

Un programme avec des sous-programmes

```
-- sémantique de Principal
PROGRAMME Principal EST
  déclarations des constantes
  déclarations des types
  déclarations des procédures et (ou des fonctions)
  déclarations des variables locales de Principal

DÉBUT  --! début du programme principal
  ....
  Instructions classiques + appels de procédures et/ou de fonctions
  ...
FIN Principal
```

Tests de sous-programme

Un programme de test

Un programme de test est un programme dont le corps comprend (seulement) des appels au sous-programme à tester.

Il n'effectue que le test du sous-programme considéré

Recommandation

Ecrire un programme de test pour chaque sous-programme

Et passer tous les tests présents dans la spécification

Bonnes pratiques pour l'écriture de sous-programmes

Quelques règles importantes

1. Les sous-programmes sont associés à des abstractions de contrôle (Procédures) ou d'expressions (Fonctions)
2. Un sous-programme est spécifié par son contrat et les tests associés
3. Un sous-programme est déclaré dans un programme ou un autre sous-programme
4. Un sous-programme devrait être appelé (sinon code mort)
5. Un sous-programme doit être testé
6. Le choix des modes de passage des paramètres doit correspondre à la logique du traitement réalisé $\Longrightarrow$ Ne pas utiliser systématiquement le mode IN OUT
7. **Ne pas écrire une procédure à la place d'une fonction**
8. Identifiants pour les sous-programmes
   - **Procédures**. Un nom définissant une action (verbe à l'infinitif).
     Exemple. `Permuter`, `Afficher`
   - **Fonction**. Un nom décrivant l'expression et donnant une indication sur le type de retour
     Exemples. `Est_Positif`, `Factorielle`, `Meme_Age`, `Racine_Carree`

Bonnes pratiques pour l'écriture de sous-programmes

Comment concevoir un Sous-Programme ?

1. Définir la spécification du programme
   1. Définir l'objectif du SP, équivalent à **R0**
   2. Identifier les paramètres formels : rôle, mode puis identifiant
   3. Choisir entre procédure ou fonction
   4. En déduire un nom significatif pour le SP
   5. Identifier les pré-condition et post-condition sur ses paramètres
   6. Rédiger le contrat à partir des informations ci-dessus
2. Ecrire les programmes de test (appelés tests **unitaires**)
3. Rédiger la spécification à partir des informations ci-dessus
4. Définir l'implantation du SP : Appliquer la méthode des raffinages avec la spécification du SP comme **R0**
5. Tester l'implantation du SP

Bonnes pratiques pour l'écriture de sous-programmes

Recommandations sur la définition de sous-programmes (SP)

1. Eviter de mélanger traitement (calcul) et interactions avec l'utilisateur du programme (affichage ou saisie) : Les traitements sont plus stables que l'IHM.
2. Une fonction ne doit faire ni saisie, ni affichage.
3. Un SP doit être une boîte noire $\Longrightarrow$ ne pas dépendre de variables globales.
4. Un SP ne doit pas avoir trop de paramètres
   - soit mauvais découpage,
   - soit regrouper les paramètres avec un type enregistrement.
5. Un SP ne doit pas être trop long (sinon le découper en SP).
6. Un SP ne doit pas avoir trop de structures de contrôle imbriquées (faire des SP).
7. On doit être capable d'exprimer la sémantique du SP (commentaire)... sinon c'est qu'il est mal compris !
8. Formaliser la spécification des SP avec des pré- et des post-conditions.

Fonctions en Ada

- Déclaration de fonction

```
1   --  Contrat de la Fonction
2   function Nom_Fonction (
3           Fnom_param_1 : in Type_Par_1;
4            ...
5           Fnom_param_n : in Type_Par_n) return Type_De_Retour is
6       -- déclaration des variables locales si nécessaire
7   begin
8       -- instructions
9       return Resultat;
10  end Nom_Fonction;
```

- Exemples d'appels de fonction

```
Ident_Var := Nom_Fonction (Param_1,..., Param_N);

IF Nom_Fonction (Param_1, Param_2, ..., Param_N) = ...  THEN ...
```

Procédures en Ada

- Déclaration de Procédure

```
1   --  Contrat de la Procédure
2   procedure Nom_Procedure(
3           Fnom_param_1 : in  Type1 ;
4           Fnom_param_2 : out Type2 ;
5              ...
6           Fnom_param_n : in out Type3) is
7      -- déclaration des variables locales si nécessaire
8   begin
9      -- instructions
10  end Nom_Procedure;
```

- Exemples d'appels de procédure

```
Nom_Procedure (param_1, param_2, ... param_n);
```

Ada permet :

- la surcharge,
- une valeur par défaut des paramètres formels

Exercices

- Définir une fonction qui calcule la somme des nombres entiers pairs inférieurs à Nmax donné
  Ecrire le programme de test associé à cette fonction

- Ecrire une procédure qui permet de lire de manière conviviale et fiable un entier entre 1 et Nmax, sachant que NMAX est une constante ≥ 0

Plan

- Le type énumération
- Le type enregistrement
- Le type tableau
- Les types en Ada
- Exercices

Introduction

Rappel

Dans un programme, la **déclaration de type** permet de définir et de construire de nouveaux types qui pourront être utilisés pour **typer** des variables déclarées dans ce programme ou bien des paramètres des sous-programmes.

Le compilateur est capable de contrôler la **cohérence** du typage.

Rappel.

Un type est défini par la donnée d'un

- nom
- domaine ou ensemble de valeurs
- ensemble d'opérateurs applicables aux valeurs de ce type

Introduction

Types de base

A ce stade du cours, les seuls types de données utilisés sont

- Les **entiers**, les **booléens**, les **"Réels"**, les **caractères**

Il s'agit de **types de base**.

Types construits

D'autres types peuvent être construits à partir de **constructeurs de types**

- Les **énumérés**, les **enregistrements**, les **tableaux**

Ces constructeurs permettent de construire d'autres types de données **statiques** (connus avant l'exécution).
Il s'agit de **types construits**

Autres définitions de types

- Certains langages permettent de dériver de nouveaux types à partir de types existants. Exemple : sous-types, re-définition de types
- Dans le cas du typage fort, il faudra disposer d'opérateur de conversion de types. Exemple : Entiers

Introduction

Construction de types

La **constuction de type** s'appuie sur des **constructeurs de types**. Ces constructeurs

- sont présents dans les langages algorithmique et de programmation
- permettent de construire d'autres types de données statiques.

Déclaration de types contruits

- Les types construits (nouveaux types, types utilisateurs) sont définis dans la partie **déclaration de types** d'un programme. Ils sont **nommés**.

  ```
  TYPE Nom_Type EST Définition_Du_Type
  ```

  Le type ainsi défini pourra être utilisé dans d'autres types, programmes, sous-programmes, modules, ...

Introduction

Composition

Un type peut être défini à partir

- d'un type de base
- ou d'un type construit à l'aide d'un constructeur de type appliqué sur d'autres types construits ou types de base

Utilisation pour déclarer une variable

- Et une variable `Une_Var` sera déclarée comme

```
Une_Var : Nom_Type
```

Utilisée dans un programme, un sous-programme, un module, ...

Plan

Le type énumération

Définition

Un type énuméré liste (énumère) explicitement les valeurs possibles pour ce type en leur donnant un nom symbolique (énumération).

Propriétés

Un type énuméré

- est un type scalaire discret
- muni d'une relation d'ordre

Définition

Exemples de définitions de types "énumération"

```
TYPE T_Couleur EST ENUMERATION  (BLANC, BLEU, ROUGE)

TYPE T_Jour EST ENUMERATION (LUNDI, MARDI, MERCREDI, JEUDI,
                             VENDREDI, SAMEDI, DIMANCHE)
```

- `BLANC, BLEU, ROUGE` et `LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE` sont des valeurs du type
- `T_Couleur` et `T_Jour` sont des identificateurs de type

Le type énumération

Propriétés

- Le type énuméré est ordonné $BLANC < BLEU < ROUGE$
- Le type booléen est un type énuméré $FAUX < VRAI$
- Un même identificateur peut apparaître dans plusieurs énumérations

```
TYPE T_Couleur  EST ENUMERATION (BLANC, BLEU, ROUGE)
TYPE T_CouleurClaire EST ENUMERATION (BLANC, ROSE )
```

  mais `BLANC` dans le type `T_Couleur` et `BLANC` dans le type `T_CouleurClaire` sont des constantes de types différents, donc des constantes différentes
- Pour distinguer les 2 constantes `BLANC`, on utilise les notations suivantes
  - `T_Couleur'(BLANC)`
  - `T_CouleurClaire'(BLANC)`
- A l'intérieur d'un type énumération, tous les identificateurs doivent bien sûr être différents

Remarque

Des opérateurs permettent l'accès aux valeurs d'un type énuméré, la comparaison, la conversion...
Les définitions en langage algorithmique empruntent les notations du langage Ada.

Types énumération

Opérations sur le type énumération

`T` étant le nom d'un type énumération, on dispose des opérations

- `T'Image` transforme une valeur de type énumération en une valeur de type chaîne de caractères
  Exemple. `T_Couleur'Image (ROUGE)` renvoie `"ROUGE"`
- `T'Value` qui transforme une valeur de type chaîne de caractères en une valeur de type énumération
  Exemple. `T_Couleur'Value("ROUGE")` renvoie le symbole `ROUGE`
- `T'Pos(x)` qui retourne le rang de `x` dans le type `T`
  Exemple. `T_Couleur'Pos(BLANC)` renvoie 0
- `T'Val(x)` qui retourne la valeur de type T à la place x dans l'ensemble des valeurs du type T
  Exemple. `T_Couleur'Val(1)` renvoie `BLEU`

Types énumération

Opérations sur le type énumération

T étant le nom d'un type énumération, on dispose des opérations

- `T'Pred(x)` qui retourne le précédent de `x` dans l'ensemble des valeurs du type `T`
  Exemple. `T_Couleur'Pred (BLEU)` renvoie `BLANC` et `T_Couleur'Pred (BLANC)` provoque une erreur

- `T'Succ(x)` qui retourne le suivant de `x` dans l'ensemble des valeurs du type `T`
  Exemple. `T_Couleur'Succ(BLEU )` renvoie `ROUGE`
  Attention `T_Couleur'Succ(ROUGE)` provoque une erreur.

- Le type énumération est **discret** et **ordonné**. On peut donc l'utiliser pour l'indexation

```
POUR  i  DE LUNDI A DIMANCHE FAIRE
      ...
FIN POUR
```

Plan

Le type enregistrement

Définition

Un enregistrement

- définit un produit cartésien de plusieurs domaines de valeurs associés à des types
- permet de mémoriser dans une même structure des informations de types identiques ou différents, sémantiquement liées

Définition

- Définition d'un type enregistrement

```
1  TYPE Date EST ENREGISTREMENT
2       Jour  : entier
3       Mois  : entier
4       Annee : entier
5  FIN ENREGISTREMENT
```

- Déclaration de variables

```
1  d       : Date
2  d1, d2 : Date
```

Le type enregistrement

Définition : avec invariants de type

1. Les assertions associées aux champs `Jour` et `Mois` sont des **Invariants de Type**. Il s'agit **d'invariants locaux**
2. On peut écrire un **invariant global**. Par exemple `Est_Date_Valide(J,M,A)` pour indiquer la correction d'un jour dans un mois d'une année.
   Cet invariant global implique plusieurs les champs de l'enregistrement car ils sont sémantiquement liés.
3. Ces assertions s'ajoutent aux pré- et post-conditions des actions.

- Définition d'un type enregistrement

```
1  TYPE Date EST ENREGISTREMENT
2       Jour  : entier    { 1 <= Jour ET Jour <= 31 }
3       Mois  : entier    { 1 <= Mois ET Mois <= 12 }
4       Annee : entier
5                         { 1 <= Jour ET Jour <= Nb_Jour_mois(Mois, Annee) }
6  FIN ENREGISTREMENT
```

`Nb_Jour_Mois` : fonction renvoyant le nombre de jours d'un mois dans une année.
Question. Comment l'écrire ?

Le type enregistrement

Définition

- Définition d'un type enregistrement en Ada

```
type T_Date is record
    Jour : Integer;  --{ 1 <= Jour ET Jour <= Nb_Jour_Mois (Mois, Annee) }
    Mois : Integer;  --{ 1 <= Mois ET Mois <= 12 }
    Annee : Integer;
end record;
```

- Déclaration de variables

```
d      : T_Date;
d1, d2 : T_Date;
```

Remarque. On pourrait également utiliser le type intervalle

Le type enregistrement

Opérations

- Un enregistrement étant un produit cartésien, l'opération de **projection** est définie $\Longrightarrow$ Utilisation de la notation pointée.
- Si `Expression` est de type enregistrement avec un champ nommé `Nom_Champ`, alors l'accès à ce champ se fait par la projection notée

```
Expression.Nom_Champ
```

- Exemple

```
d.Mois  <-- 12

d1 <-- d2

SI d1 = d2 ALORS
...
```

- Attention
  - L'affectation `<--` est une affectation "champ par champ"
  - L'égalité `=` est une égalité "champ par champ"
  - Pas de relation d'ordre, il faut la définir.

Plan

Le type Tableau

Définition

- Un tableau permet d'indexer des informations (données) **toutes** d'un même type
- L'ensemble des index est fini et discret. Les index sont donc tous les scalaires discrets
- Un tableau T représente une fonction $E \longrightarrow F$ où E est l'ensemble des index et F l'ensemble des valeurs

Exemple

- Si $E = [1..3]$ et $F = \mathbb{R}$ alors les éléments suivants $\{1 \mapsto 2.3, \quad 2 \mapsto -3.2, 3 \mapsto 3.4\}$ représentent les éléments d'un tableau $T : E \longrightarrow F$

- Si $E = ['a'..'d']$ et $F = \mathbb{N}$ alors les éléments suivants $\{'a' \mapsto 3, \quad 'b' \mapsto 3, 'c' \mapsto 0, 'd' \mapsto 8\}$ représentent les éléments d'un tableau $T : E \longrightarrow F$

En informatique, un tableau est représenté dans une zone mémoire contiguë.
Ceci assure un accès en temps constant à tout élément du tableau à partir de son index (indice).

Le type Tableau

Accès aux éléments d'un tableau

- C'est appliquer la fonction définie par le tableau

Exemple

- Si $E = [1..3]$, $F = \mathbb{R}$ et $T : E \longrightarrow F$ avec les éléments suivants $\{1 \mapsto 2.3, 2 \mapsto -3.2, 3 \mapsto 3.4\}$ alors $T(1)$ est l'élément 2.3 et l'accès à l'élément $T(5)$ provoque une erreur

- Si $E = [\text{'}a\text{'}..\text{'}d\text{'}]$, $F = \mathbb{N}$ et $T : E \longrightarrow F$ avec les éléments suivants $\{\text{'}a\text{'} \mapsto 3, \text{'}b\text{'} \mapsto 3, \text{'}c\text{'} \mapsto 0, \text{'}d\text{'} \mapsto 8\}$ alors $T(\text{'}a\text{'})$ est l'élément 3 et l'accès à l'élément $T(\text{'}x\text{'})$ provoque une erreur

Ces erreurs sont détectées à l'exécution.

Le type Tableau

Modification du contenu d'un tableau

- Il faut utiliser une affectation

Exemple

- Si $E = [1..3]$, $F = \mathbb{R}$, $T : E \longrightarrow F$ et $T = \{1 \mapsto 2.3, 2 \mapsto -3.2, 3 \mapsto 3.4\}$ alors

```
T(2) <-- 5.6
```

  modifie le tableau T en $T = \{1 \mapsto 2.3, 2 \mapsto 5.6, 3 \mapsto 3.4\}$

- Si $E = ['a'..'d']$, $F = \mathbb{N}$, $T : E \longrightarrow F$ et $T = \{'a' \mapsto 3, 'b' \mapsto 3, 'c' \mapsto 0, 'd' \mapsto 8\}$ alors

```
T('a') <-- 25
```

  modifie le tableau T en $T = \{'a' \mapsto 25, 'b' \mapsto 3, 'c' \mapsto 0, 'd' \mapsto 8\}$

Dans la suite, on donne la définition algorithmique du type Tableau.

Le type Tableau

Définition et manipulation de tableaux à une dimension

```
1       BORNE_MAX : CONSTANTE Entier <-- 6
2       TYPE T_Notes EST TABLEAU (1.. BORNE_MAX) DE Réel
3
4       Notes: T_Notes          --// Declaration de variables
5       Nb_Elements : entier    { 0 <= Nb_Elements ET  Nb_Elements <= BORNE_MAX }
6   Début
7       --Enregitrer une première note
8       Notes (1)   <--  10.0
9       Nb_elements <--  1
10
11      --Enregitrer une nouvelle note
12      Notes (2)   <--  13.0
13      Nb_elements <--  2
```

- `BORNE_MAX` est la capacité du tableau
- `Nb_Elements` est le nombre d'éléments dans un tableau (taille effective) avec

$$Nb\_Elements \in [0..BORNE\_MAX]$$

Remarque. On peut aussi introduire une *Borne_Min* et gérer des sous-tableaux.
Il faudra déclarer le plus petit indice et introduire *Borne_Min*.

Le type Tableau

Définition et manipulation de tableaux à une dimension

Afin d'associer le nombre d'éléments et le tableau, on peut utiliser un enregistrement

```
1       BORNE_MAX : CONSTANTE ENTIER <-- 6
2
3       TYPE T_Notes EST ENREGISTREMENT
4           Elements    : TABLEAU (1.. BORNE_MAX) DE Réel
5           Nb_Elements : Entier   { 0 <= Nb_Elements <= BORNE_MAX }
6       FIN ENREGISTREMENT
7
8       Notes: T_Notes
9   Début
10       Notes.Elements (1) <--  10.0
11       Notes.Nb_Elements  <--  1
12
13       Notes.Elements (2) <--  13.0
14       Notes.Nb_Elements  <--  2
```

Remarque. Des sous-programmes pour initialiser, ajouter un élément, supprimer, obtenir l'index d'un élément, ... devaient être définis.

Le type Tableau

Affectation de tableaux à une dimension

- Soient les déclarations

```
BORNE_MAX : CONSTANTE Entier <-- 4
type Vecteur EST TABLEAU (1.. BORNE_MAX) DE REEL
T : Vecteur
```

- Affecter les cases d'un tableau une à une

```
T(3) <-- 1
```

- Affecter toutes les cases d'un tableau

```
T(1..4) <-- (1, 3, 5, 2)          -- en énumérant les valeurs
T <-- (1..2 => 0 , 3  => 1)       -- par les agrégats
T <-- (1..2 => 0 , 3  => 1, Autres => 0)  -- complet
```

- On peut aussi n'affecter qu'un sous-tableau d'un tableau

```
T(3..4 )<-- (8, 7)        -- en énumérant les valeurs
                          -- ou par les agrégats
```

Attention. Ces opérations diffèrent selon les langages de programmation.

Le type Tableau

Tableaux à deux dimensions

- Soient les déclarations

```
MAX_1 : CONSTANTE Entier <-- 4

MAX_2 : CONSTANTE  Entier <-- 6

TYPE T_Matrice EST TABLEAU (1..MAX_1, 1..MAX_2) DE Entier

T : T_Matrice
```

- Affectation

```
T(2,3) <-- 2017
```

Définition d'une fonction $T : [1..MAX\_1] \times [1..MAX\_2] \longrightarrow Entier$

Le type Tableau

Les chaînes de caractères sont des tableaux

- La définition des chaînes de caractères est très dépendante des langages de programmation.
- Chaînes de caractères déclarées sous forme de tableaux

Définition de type

```
1  BORNE_MAX : CONSTANTE ENTIER <-- 50
2
3  TYPE MesChaines EST ENREGISTREMENT
4      Caracteres : TABLEAU (1..BORNE_MAX) DE Caractère
5      Taille     : Entier     { 0 <= Nb_Elements ET Nb_Elements <= BORNE_MAX }
6  FIN ENREGISTREMENT
```

Le type Tableau

Manipulation de chaînes de caractères

```
1       BORNE_MAX : CONSTANTE ENTIER <-- 50
2       TYPE MesChaines EST ENREGISTREMENT
3           Caracteres : TABLEAU (1..BORNE_MAX) DE Caractère
4           Taille     : Entier   { 0 <= Nb_Elements ET Nb_Elements <= BORNE_MAX }
5       FIN ENREGISTREMENT
6
7       Mot : MesChaines
8   Début
9       Mot.Caracteres (1) <-- 'P'
10      Mot.Taille <--  1
11
12      Mot.Caracteres (2..3) <-- "IM"
13      Mot.Taille <--  3
14
15      SI Mot.Caracteres (1..Mot.Taille) = "PIM" ALORS
16          Écrire ("Le cours de ce jour est " & Mot.Caracteres (1..Mot.Taille))
17      SINON
18          Écrire ("Autre")
19      FIN SI
20      ...
```

Attention une chaine s'écrit entre " ("Bonjour") et un caractère s'écrit entre ' ('P')

Plan

Le type énumération en Ada

Définition

Elle suit la forme classique

```
type T_Couleur is (BLANC, BLEU, ROUGE);
```

Utilisation

- Une variable

  ```
  Une_Coul: T_Couleur;
  ```

- Une affectation

  ```
  Une_Coul := ROUGE;
  ```

- Une saisie en définissant une convention avec l'utilisateur puis une conversion
  - d'un entier (la position) lu (2) puis conversion avec `T_Couleur'Val(2)` ou
  - d'une chaîne de caractères (`"rouge"`) lue, puis `T_Couleur'Value("rouge")`

- Un affichage

  ```
  Put (T_Couleur'Image(Une_Coul));
  ```

- Un exemple d'itération

  ```
  for I in T_Couleur loop
      ...
  end loop;
  ```

Le type enregistrement

Déclaration Ada

- Soit le type suivant

```
type T_Complexe is record
    Pr : Float;     -- partie réelle
    Pi : Float;     -- partie imaginaire
end record;
```

- Deux manières d'initialiser une variable `c: T_Complexe`
  - Classique (champ par champ)

```
c.Pr := 2.0;
c.Pi := 4.0;
```

  - Par agrégat
    - (Complet) `c := (2.0, 4.0);`
    - (Complet) `c := (Pi => -2.2, Pr => 1.1);`
    - (Partiel) `c := (Pi => 1.1);`

Le type Tableau

Les tableaux en Ada

- Un exemple

```
BORNE_MAX : constant Integer := 10;
type T_Vecteur is array (1..BORNE_MAX) of Integer;
T : T_vecteur;  -- T est un tableau à 1 dimension, contenant des
                -- entiers. L'index commence à 1 et finit à BORNE_MAX
T1, T2 : T_vecteur;
```

- On peut écrire l'affectation `T1 := T2`
- Un autre exemple

```
MAX_1 : constant Integer := 12;
MAX_2 : constant Integer := 20;
type T_Matrice is array (1..MAX_1, 1..MAX_2) of Integer;
T : T_Matrice; -- T est un tableau à 2 dimensions,
               -- contenant des entiers, le 1er indice
               -- va de 1 à 12 et le 2ème de 1 à 20
T1, T2 : T_matrice;
```

- Accès et affectation avec `T1 (2,5) := T2 (3,1)`
- `String` est utilisé pour désigner le type chaîne de caractères

Plan

Exercice

Un premier exercice de modélisation

- On s'intéresse aux cartes d'un jeu de 52 cartes réparties en 4 enseignes ou couleurs (pique, coeur, carreau et trèfle) comprenant chacune 13 valeurs (de l'as au roi).

1. Définir le type `T_Carte` qui modélise une carte d'un jeu de 52 cartes.
2. Ecrire un sous-programme qui permet d'initialiser une carte à partir de son enseigne et de sa valeur.
3. Ecrire un sous-programme qui permet d'afficher une carte. On affichera d'abord la valeur de la carte sur 3 caractères, un espace et l'enseigne sur 3 caractères. Voici des exemples : `AS PIQ`, `ROI COE`, `10 CAR`, `VAL TRE`, `2 PIQ` ou `DAM CAR`.

Exercice : début de solution

Début de solution : Enseigne et valeur

Définition des types associés à une carte d'un jeu

- La couleur ou l'enseigne

```
TYPE T_Enseigne EST ENUMERATION (COEUR, CARREAU, TREFLE, PIQUE)
```

- La valeur d'une carte d'un jeu

```
TYPE T_Valeur EST ENUMERATION (
                    AS, DEUX, TROIS, QUATRE, CINQ,
                    SIX, SEPT, HUIT, NEUF, DIX, VALET,
                    DAME, ROI)
```

Dans cette déclaration, on observe que `AS < DEUX < ... < ROI`

Exercice : début de solution

Début de solution : une carte

Définition du type décrivant ce qu'est une carte d'un jeu

- Une carte est décrite par la paire (*enseigne*, *valeur*)

```
TYPE T_Carte EST ENREGISTREMENT
      Valeur   : T_Valeur
      Enseigne : T_Enseigne
FIN ENREGISTREENT
```

- Une carte, variable C, sera déclarée :

```
C : T_Carte
```

- On peut ensuite dire que C est un Roi de Trefle par

```
C.Valeur  <-- ROI
C.Enseigne <-- TREFLE
```

  ou bien

```
C <-- (Valeur => ROI, Enseigne => TREFLE)
```

- On peut dire qu'une carte C1 est plus forte qu'une carte C2 avec

```
(C1.Valeur > C2.Valeur) OU
(C1.Valeur = C2.Valeur ET C1.Enseigne > C2.Enseigne)
```

  D'autres relations d'ordre peuvent être définies en fonction du jeu.

Exercice : début de solution

Début de solution : une carte

- On peut définir une valuation d'une carte avec

```
TYPE T_Valuation EST TABLEAU (T_Valeur) DE Entier
```

- et déclarer une valuation possible pour une jeu donné

```
Valuation_Jeu : T_Valuation
```

- et effectuer l'affectation

```
Valuation_Jeu (ROI) <-- 13
```

- Ainsi pour une carte `C: T_Carte` on pourra manipuler l'entier qui représente la valeur (par exemple pour compter des points)

```
Valuation_Jeu (C.Valeur) +...
```

- Une autre possibilité pour déclarer une carte serait

```
TYPE T_Carte EST ENREGISTREMENT
    Valeur   : Entier    -- 1..13 avec Valet = 11, Dame = 12 et Roi = 13
    Enseigne : T_Enseigne
FIN ENREGISTREMENT
```

Plusieurs solutions sont donc possibles !

Exercice : début de solution

Début de solution

Définition du type décrivant ce qu'est un jeu de 52 cartes

- Un jeu est défini à l'aide d'un tableau de 52 éléments qui sont des cartes

```
NB_CARTES : CONSTANTE ENTIER <-- 52

TYPE T_Jeu EST TABLEAU (1.. NB_CARTES) DE T_Carte
                                { 52 cartes différentes }
```

- Ainsi, un jeu, sera déclaré à l'aide d'une variable `Jeu`

```
Jeu : T_Jeu
```

- On peut affecter `Pique` comme couleur de la 12ème carte du jeu par

```
Jeu (12).Enseigne <-- Pique
```

Autres exercices

Dans les exercices suivants (sauf mention particulière), on suppose que les tableaux commencent à l'indice 1, et que les informations sont rangées à partir de la 1ère case du tableau et que le tableau n'est pas forcément "rempli".
On utilisera la méthode des raffinages pour produire les algorithmes.

1. Spécifier et implanter un sous-programme permettant de saisir un tableau (TD)
2. Spécifier et implanter un sous-programme qui affiche un tableau (TD)
3. Spécifier et implanter un sous-programme qui retourne l'indice de la première occurrence d'un entier `e` dans un tableau `T` d'entiers à une seule dimension, contenant `Nb_Elements`. Le résultat sera `0` si `e` n'est pas dans le tableau.
   On écrira également un programme appelant ce sous-programme.
4. Spécifier et implanter un sous-programme qui ajoute à la fin d'un tableau `T` l'élément `e`.
5. Spécifier et implanter un sous-programme qui supprime la première occurrence d'un élément `e` dans un tableau `T`. L'ordre des éléments sera préservé.
6. Spécifier et implanter un sous-programme qui retourne le nombre d'occurrences d'un élément `e` d'un tableau `T`.
7. Spécifier et implanter un sous-programme qui supprime toutes les occurrences d'un élément `e` d'un tableau `T`.

Plan

- Motivation
- Définition
- Syntaxe sur un exemple
- Encapsulation
- Masquage d'information
- Classification des modules
- Conclusion

Le constat

A ce stade du cours, nous savons écrire des programmes composés de

- définitions de constantes,
- définitions de types
- définitions de sous-programmes ainsi que leur corps
- du programme principal (y compris la déclaration de ses variables)

Un tel programme est **monolithique** (un seul bloc, un seul fichier) :

Questions

1. Comment **définir et concevoir** des éléments de programme (constantes, types, sous-programmes) pour qu'ils puissent être utilisés par d'autres programmes ?
2. Comment **utiliser dans un programme** ces éléments réutilisables ?

Solutions

Mauvaise solution : copier / coller les éléments à réutiliser

Copier/coller dans le nouveau programme les éléments dont on a besoin.

- ☹ La réutilisation n'est pas explicitée
- ☹ Toute modification dans le code copié, devra être répercutée dans le code collé !
- ☹ ...

On regrette toujours, tôt ou tard, d'avoir fait du copier/coller !

Bonne solution : Spécifier les éléments réutilisables

Une construction syntaxique permet de décrire les éléments réutilisables :

1. **spécification** des éléments : description suffisante pour leur utilisation
2. **implantation** des éléments : détails nécessaires à leur mise en œuvre et qui n'ont pas à être connus des utilisateurs.

Remarque : C'est le cas du module `Text_IO` d'Ada : connaître la spécification des méthodes `Put` et `Get` suffit pour les utiliser même si on ne connait pas leur implantation.

C'est la notion de **module** (**paquetage**, **unité**...) qui réalise le **principe de séparation** utilisé en ingénierie (automobile, aéronautique, mécanique...) : **conception modulaire**

Module

Définition (Larousse)

- Elément **juxtaposable**, **combinable** à d'autres de même nature ou concourant à une même fonction
- Partie d'un programme constituant une **unité** à la fois **structurelle** et **fonctionnelle**

En programmation

- Ensemble de **déclarations** de constantes, de types, d'attributs et de sous-programmes ainsi que l'ensemble des **implantations (corps)** de ces sous-programmes satisfaisant le principe de **séparation**
- Notion formalisée par D.L. Parnas *"A Technique for Software Module Specification with Examples"* (1972)
- Plusieurs langages de programmation offrent la possibilité de concevoir des modules : Modula, Ada, C++, Java, Caml, Python, etc.

Module (2)

Module et abstraction

Des définitions précédentes, on retient :

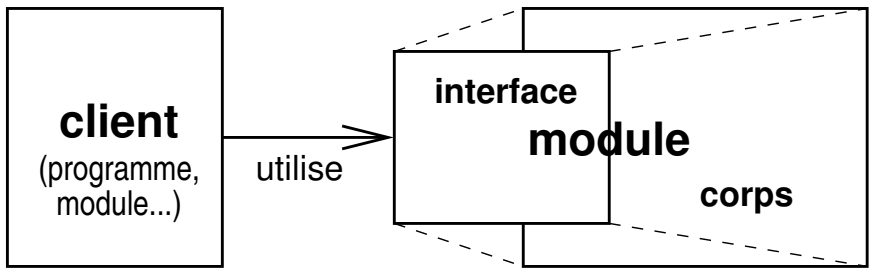
- un module regroupe donc
  - des **abstractions de données** ainsi que
  - des **abstractions de contrôle**
- le mot **unité** indique le fait que les constituants regroupés au sein d'un module traitent du **même sujet**, de la **même abstraction**
- **unité structurelle** indique que le module traite des structures (structures de données) formant une unité
- **unité fonctionnelle** indique que le regroupement des abstractions d'expressions et d'actions portant sur la structure définie

Constituants

Un module est une unité d'encapsulation qui est constutiée de :

- Une **interface**, souvent appelée **spécification**, composée de
  1. déclarations de constantes et de types
  2. spécifications des sous-programmes (signature, pré-condition, post-condition)

  L'interface offre des **services** utilisables depuis l'extérieur : autre module, sous-programme ou programme aussi appelé **client**



- Un **corps**, aussi appelé **implantation**, composé
  1. les implantations (corps) des différents sous-programmes de l'interface (obligatoire)
  2. et **éventuellement d'autres** déclarations de constantes, de types ou d'attributs (variables) ainsi que des définitions (interface et corps) d'autres sous-programmes **utiles** pour le concepteur du module mais **cachés** aux clients du module

Intérêt : évolutivité

On peut changer le corps d'un module sans impact pour ses clients.
Bien sûr ces changements doivent respecter l'interface !

Un module Dates simplifié

Nous allons montrer la syntaxe des modules en Ada sur un exemple : le module Dates

1. Sa spécification
2. Un client de ce module
3. Son corps

Ada utilise le terme **paquetage** : `package`

Module Dates

- Très simplifié, il n'offre que deux opérations principales :
  1. initialiser une date
  2. afficher une date
- La notion de Date n'est pas si simple à gérer. Elle est généralement disponible dans les bibliothèques du langage utilisé, `Ada.Calendar` pour Ada.

Conventions

- Le nom d'un module est généralement noté avec un pluriels (Dates, Complexes, etc.)
- Autres conventions possibles : `P_Date`, `P_Complexe`... ou `M_Date`, `M_Complexe`...

Interface d'un module dates simplifié : dates.ads

```
1   -- Spécification d'un module Dates très simplifié.
2
3   package Dates is
4
5       type T_Mois is (JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET,
6                       AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE);
7
8       type T_Date is
9           record
10              Jour : Integer;
11              Mois : T_Mois;
12              Annee : Integer;
13              -- Invariant : ... non fourni ici...
14          end record;
15
16
17      -- Initialiser une date à partir du jour, du mois et de l'année.
18      --
19      -- Paramètres :
20      --     Date : la date à initialiser
21      --     Jour, Mois, Annee : la valeur du jour, du mois, de l'année
22      --
23      --  Nécessite :
24      --     Jour/Mois/Annee constituent une date valide
25      --
26      --  Assure
27      --     Le_Jour (Date) = Jour
28      --     Le_Mois (Date) = Mois
29      --     L_Annee (Date) = Annee
30      --
```

Interface d'un module dates simplifié : dates.ads (2)

```
31        procedure Initialiser ( Date  : out T_Date  ;
32                                Jour  : in  Integer ;
33                                Mois  : in  T_Mois  ;
34                                Annee : in  Integer )
35        with
36            Pre => Annee >= 0 and Jour >= 1 and Jour <= 31,     -- simplifiée !
37            Post => Le_Jour (Date) = Jour and Le_Mois (Date) = Mois
38                        and L_Annee (Date) = Annee;
39
40        -- Afficher une date sous la forme jj/mm/aaaa
41        procedure Afficher (Date : in T_Date);
42
43        -- Obtenir le mois d'une date.
44        -- Paramètres
45        --     Date : la date dont on veut obtenir le moi
46        function Le_Mois (Date : in T_Date) return T_Mois;
47
48        -- Obtenir le jour d'une date.
49        -- Paramètres
50        --     Date : la date dont on veut obtenir le jour
51        function Le_Jour (Date : in T_Date) return Integer;
52
53        -- Obtenir l'année d'une date.
54        -- Paramètres
55        --     Date : la date dont on veut obtenir l'année
56        function L_Annee (Date : in T_Date) return Integer;
57
58    end Dates;
```

Explications

- L'interface du module `Nom_Module` est définie dans un fichier `nom_module.ads`
  - ads signifie Ada Specification
- L'interface est définie entre `package` et le `end` correspondant

```
1  --! Sémantique du paquetage...
2
3  package Nom_Module is
4      --! Définitions des constantes et des types,
5      --! et  spécification des sous-programmes
6      --! Pas d'ordre imposé mais tous les éléments doivent être connus
7  end Nom_Module;
```

- On peut définir des types et des constantes, spécifier des sous-programmes
- On ne donne que la spécification des sous-programmes pas leur corps !
- Il n'y a pas d'ordre imposé mais les éléments doivent être connus avant d'être utilisés (ici les types avant les sous-programmes).
  En général, l'ordre est constantes, types, spécifications des sous-programmes.

Exemple de client du module dates

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Dates;       use Dates;
3
4  procedure Exemple_Dates is
5      Une_Date : T_Date;
6  begin
7      -- Initialiser une date
8      Initialiser (Une_Date, 2, OCTOBRE, 2020);
9
10     -- L'afficher
11     Afficher (Une_Date);
12     New_Line;
13 end Exemple_Dates;
```

- `with` donne accès au module. On peut alors écrire :
  `Dates.T_Date, Dates.T_Mois, Dates.Initialiser, Dates.Afficher...`
- `use` donne accès au contenu du module. On peut alors simplement écrire :
  `T_Date, T_Mois, Initialiser, Afficher...`

Corps du module dates simplifié

```
1   -- Implantation d'un module Dates très simplifié.
2
3   with Ada.Text_IO;         use  Ada.Text_IO;            --! autres modules utilisés
4   with Ada.Integer_Text_IO; use  Ada.Integer_Text_IO;
5
6   package body Dates is
7
8       procedure Initialiser ( Date  : out T_Date  ;
9                               Jour  : in  Integer ;
10                              Mois  : in  T_Mois  ;
11                              Annee : in  Integer ) is
12      begin
13          Date.Jour := Jour;
14          Date.Mois := Mois;
15          Date.Annee := Annee;
16      end Initialiser;
17
18      function Le_Jour (Date : in T_Date) return Integer is
19      begin
20          return Date.Jour;
21      end Le_Jour;
22
23      function Le_Mois(Date : in T_Date) return T_Mois is
24      begin
25          return Date.Mois;
26      end Le_Mois;
27
28      function L_Annee (Date : in T_Date) return Integer is
29      begin
30          return Date.Annee;
```

Corps du module dates simplifié (2)

```
31        end L_Annee;
32
33        -- Afficher un entier sur 2 positons au moins (avec des zéros
34        -- supplémentaires si nécessaires)
35        --
36        -- Paramètres :
37        --     Nombre : le nombre à afficher
38        --
39        -- Nécessite :
40        --     Nombre >= 0
41        --
42        procedure Afficher_Deux_Positions (Nombre : in Integer) with
43            Pre => Nombre >= 0 is
44        begin
45            Put (Nombre / 10, 1);
46            Put (Nombre mod 10, 1);
47        end Afficher_Deux_Positions;
48
49        procedure Afficher (Date : in T_Date) is
50        begin
51            Afficher_Deux_Positions (Date.Jour);
52            Put ('/');
53            Afficher_Deux_Positions (T_Mois'pos (Date.Mois) + 1);
54            Put ('/');
55            Afficher_Deux_Positions (Date.Annee / 100);
56            Afficher_Deux_Positions (Date.Annee mod 100);
57        end Afficher;
58
59    end Dates;
```

Explications

- Le corps du module `Nom_Module` est défini dans un fichier `nom_module.adb`
  - adb signifie Ada Body
- Le corps est défini entre `package body` et le `end` correspondant

```
1  package body Nom_Module is
2      --! Contenu du corps :
3      --!   - Corps des sous-programmes spécifiés dans l'interface
4      --!   - Définitions de constantes et types cachés dans le corps
5      --!   - Spécification des sous-programmes cachés dans le corps
6  end Nom_Module;
```

- Le corps doit définir le corps de tous les sous-programmes spécifiés dans l'interface
- On peut définir d'autres éléments cachés dans le corps, invisibles des clients :
  - définition de constantes et types
  - spécification et implantation de sous-programmes

  Ces éléments sont nécessaires pour écrire le corps des sous-programmes de l'interface
- Ne pas rappeler la spéc. des sous-programmes de l'interface : commentaire, contrat
  - Ils sont déjà donnés dans l'interface du module
  - Il faut éviter les redondances
- Bien sûr, on doit spécifier les éléments spécifiques au corps :
  - Exemple : La spécification de `Afficher_Deux_Positions` est donnée.

Encapsulation

Encapsulation

- La notion de module réalise l'encapsulation, principe qui consiste à regrouper des données avec les sous-programmes qui permettent de les manipuler.
- L'application est alors vue comme un ensemble de modules qui s'utilisent les uns les autres (graphe de dépendance avec relation *utilise*).

Exemples de modules

- dates avec les opérations usuelles (comparaison, durée entre deux dates, etc.)
- complexes : le type complexe et les opérations usuelles
- le type vecteur et les opérations associés
- module gérant le dialogue avec l'utilisateur (IHM)...

Masquage d'information

Le masquage d'information consiste à cacher des informations aux clients pour favoriser :

- l'évolutivité : ce qui est caché peut être changé sans impact sur les clients
- la cohérence : les données internes sont contrôlées via les services offerts

Le masquage d'information est généralement associé à l'encapsulation :

- Seuls les éléments **présents dans l'interface d'un module sont visibles** à l'extérieur
- Les **éléments du corps sont cachés** (non visibles)

Cacher la définition des types

- Dans le module Dates, la définition du type Date est visible.
- L'utilisateur pourrait donc modifier directement une date D : `D.Jour := 50;`
- La date serait alors dans un état incohérent (invariant non satisfait)
- **Solution :** cacher la définition du type au client en la définissant dans le corps
- Ada impose de définir le type dans la spécification, dans une rubrique `private` (donc cachée), le type est défini `private` dans la partie publique
- Seules opérations possibles sur un type privé : affectation (`:=`) et égalité (`=` et `/=`)
- Il existe aussi un type `limited private` (très privé) qui interdit ces opérations

Masquage d'information : cacher la définition des types

```
1   -- Spécification d'un module Dates très simplifié.
2
3   package Dates is
4
5       type T_Mois is (JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET,
6                       AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE);
7
8       type T_Date is private;          --! le type T_Date est privé
9
10      -- Initialiser une date à partir du jour, du mois et de l'année.
11      --
12      -- Paramètres :
13      --     Date : la date à initialiser
14      --     Jour, Mois, Annee : la valeur du jour, du mois, de l'année
15      --
16      --  Nécessite :
17      --     Jour/Mois/Annee constituent une date valide
18      --
19      --  Assure
20      --     Le_Jour (Date) = Jour
21      --     Le_Mois (Date) = Mois
22      --     L_Annee (Date) = Annee
23      --
24      procedure Initialiser ( Date  : out T_Date  ;
25                              Jour  : in  Integer ;
26                              Mois  : in  T_Mois  ;
27                              Annee : in  Integer )
28      with
29          Pre => Annee >= 0 and Jour >= 1 and Jour <= 31,     -- simplifiée !
30          Post => Le_Jour (Date) = Jour and Le_Mois (Date) = Mois
```

Masquage d'information : cacher la définition des types (2)

```
31                      and L_Annee (Date) = Annee;
32
33      -- Afficher une date sous la forme jj/mm/aaaa
34      procedure Afficher (Date : in T_Date);
35
36      -- Obtenir le mois d'une date.
37      -- Paramètres
38      --     Date : la date dont on veut obtenir le moi
39      function Le_Mois (Date : in T_Date) return T_Mois;
40
41      -- Obtenir le jour d'une date.
42      -- Paramètres
43      --     Date : la date dont on veut obtenir le jour
44      function Le_Jour (Date : in T_Date) return Integer;
45
46      -- Obtenir l'année d'une date.
47      -- Paramètres
48      --     Date : la date dont on veut obtenir l'année
49      function L_Annee (Date : in T_Date) return Integer;
50
51  private --! rubrique accessible seulement du corps, pas des clients
52
53      type T_Date is
54          record
55              Jour : Integer;
56              Mois : T_Mois;
57              Annee : Integer;
58          end record;
59
60  end Dates;
```

Classification des modules

Les modules peuvent être classés en plusieurs catégories en fonction de ce qu'ils fournissent :

- Module type : un type (visible) et les sous-programmes pour le manipuler
- Module objet : un objet (caché) et les sous-programmes pour le manipuler
- Module utilitaire : des sous-programmes sur des types définis ailleurs (Math)
- des constantes partagées entre plusieurs modules (pas de sous-programme)

Module type

- Il est construit autour d'un type et des sous-programmes pour le manipuler
- Il permet de gérer plusieurs données de ce types.
- Exemples : Dates, Notes, Complexes, Vecteurs...

Module objet

- Il cache un objet (variable) et fournit les sous-programmes pour le manipuler
- L'objet est déclaré dans le corps ; le client le manipule via les sous-programmes
- Exemple : `Ada.Text_IO` cache le terminal que l'on manipule via les Put et Get

Conclusion

- Les modules représentent une unité logique de structuration et de traitement
- Notion courante en ingénierie
- Interface et services offerts : visible par un utilisateur
- Corps : caché par l'encapsulation et le masquage d'information
- Intérêt : réutilisation, maintenance, intégration, architecture
- Les sous-programmes d'un module sont conçus par la méthode des raffinages
- Critères de définition d'un module objet ou d'un module type

Nous avons vu deux opérations sur les modules :

1. La définition d'un module
2. L'≪ importation ≫ d'un module : `with`

D'autres opérations seraient possibles : lien de hiérarchie, extension, inclusion, visibilité...

Les modules sont la brique de base de plusieurs styles de programmation : orientée objets, à base de composants...

Plan

- Motivation
- Généricité
- Module générique
- Utilisation module générique
- Module générique
- Module utilisant un module générique
- Nature des paramètres de généricité

Et si on allait plus loin dans l'abstraction

Objectif

- On veut définir des abstractions de plus haut niveau que les fonctions et procédures

Exemples

Permuter des variables	Types des variables à permuter
Rechercher un élément dans un tableau	Types des éléments du tableau Égalité
Trier un tableau d'éléments	Type des éléments du tableau Relation d'ordre
...	...

Question

- Peut-on définir un sous-programme ou un module **générique** qui pourrait être **utilisé sur différents types** de données ?
- **Solution : Généricité**
  - Disponible en Ada, C++, Java, OCaml, etc.
  - Pour la suite, nous illustrons ce concept avec le langage Ada.

Exemple avec Permuter

Permuter les valeurs de deux variables

- Permuter deux entiers (`Integer`)

```
1  procedure Permuter_Entiers (X, Y : in out Integer) is
2      Memoire: Integer;
3  begin
4      Memoire := X;
5      X       := Y;
6      Y       := Memoire;
7  end Permuter;
```

- Permuter deux dates (`T_Date`)

```
1  procedure Permuter_Dates (X, Y : in out T_Date) is
2      Memoire: T_Date;
3  begin
4      Memoire := X;
5      X       := Y;
6      Y       := Memoire;
7  end Permuter;
```

- De la même manière, on pourrait permuter deux tableaux, deux complexes, etc.

Pourrait-on n'écrire qu'une seule fois une procédure pour permuter deux variables ?

Généricité

Principe (illustré sur Permuter)

- **Spécifier** une unité (**sous-programme ou module**) avec des paramètres de généricité, i.e. de manière **symbolique** (pour la forme)

```
1  -- Permuter deux éléments X et Y d'un type quelconque...
2  generic
3      type Un_Type is private;
4  procedure Permuter_Generique (X, Y : in out Un_Type);
```

- **Implanter** l'unité en s'appuyant sur les paramètres de généricité

```
1  procedure Permuter_Generique (X, Y : in out Un_Type) is
2      Memoire: Un_Type;
3  begin
4      Memoire := X;
5      X       := Y;
6      Y       := Memoire;
7  end Permuter_Generique;
```

- **Instancier** : fournir une valeur à chaque paramètre de généricité

```
1  procedure Permuter_Entiers is new Permuter_Generique(Un_Type => Integer);
```

- **Utiliser** l'unité ainsi obtenue

```
1      A, B : Integer;  -- deux entiers sans intérêt
2  begin
3      -- ...
4      Permuter_Entiers (A, B);
```

Module générique

Pour réutiliser le sous-programme, il faut en faire une unité semblable à un module réduit à un seul sous-programme (pas de mot-clé `package`).

Interface : `permuter_generique.ads`

```
1  -- Permuter deux éléments X et Y d'un type quelconque (Un_Type)...
2  generic
3      type Un_Type is private;
4  procedure Permuter_Generique (X, Y : in out Un_Type);
```

Corps : `permuter_generique.adb`

```
1  procedure Permuter_Generique (X, Y : in out Un_Type) is
2      Memoire: Un_Type;
3  begin
4      Memoire := X;
5      X       := Y;
6      Y       := Memoire;
7  end Permuter_Generique;
```

Utilisation du module générique

```
1   with Permuter_Generique;
2   with Dates; use Dates;
3
4   procedure Exemple_Permuter_Generique is
5       A, B : Integer;  -- deux entiers sans intérêt
6       D1, D2 : T_Date; -- deux dates sans intérêt
7
8       procedure Permuter is new Permuter_Generique(Un_Type => Integer);
9       procedure Permuter is new Permuter_Generique(T_Date);
10
11  begin
12      -- montrer la permutation de deux entiers
13      A := 1;
14      B := 2;
15      Permuter (A, B);
16      pragma Assert (A = 2);
17      pragma Assert (B = 1);
18
19      -- montrer la permutation de deux dates
20      Initialiser(D1, 1, OCTOBRE, 2020);
21      Initialiser(D2, 2, OCTOBRE, 2020);
22      Permuter(D1, D2);
23      pragma Assert (Le_Jour(D1) = 2);
24      pragma Assert (Le_Jour(D2) = 1);
25
26  end Exemple_Permuter_Generique;
```

Même nom, `Permuter`, pour les deux instances grâce à la surcharge.

Explications

- Les paramètres de généricité peuvent être fournis (comme pour les sous-programmes) :
  - soit en associant le paramètre de généricité avec le paramètre formel avec `=>`
  - soit en utilisant la position du paramètre
- La surcharge évite d'avoir à inventer des noms
  - souvent un nom avec un type en suffixe : `Permuter_Entiers`, `Permuter_Dates`...
- Pour un module générique, on fait un `with` sans `use`
  - car le module ne peut pas être utilisé tant qu'il n'est pas instancié
  - on pourra faire un `use` sur le module instancié
    (inutile ici car module particulier réduit à un seul sous-programme).

Module Vecteurs

Objectif

Définir un module Vecteurs qui fournit un tableau avec gestion de la taille tel qu'on peut le trouver en C++/Java ou en Python (list).

Opérations

- Initialiser un vecteur vide.
- Obtenir la taille d'un vecteur
- Obtenir l'élément à un indice donné
- Modifier l'élément à un indice donné
- Ajouter un élément à la fin du vecteur

Paramètres de généricité

Pour avoir un module général, on va définir deux paramètres de généricité :

- le types des éléments du vecteur : `Type_Element`
- la capacité du vecteur : `Capacite`

Interface : vecteurs.ads

```
1  generic                          -- Commentaires de spécification volontairement omis.
2    type Type_Element is private;  -- type des éléments du vecteur
3    Capacite: Integer;             -- capacité du vecteur
4  package Vecteurs is
5    subtype T_Indice is Integer range 1..Capacite;
6    type T_Vecteur is limited private;
7
8    procedure Initialiser (Vecteur: out T_Vecteur) with
9      Post => Taille (Vecteur) = 0;
10
11   function Taille (Vecteur : in T_Vecteur) return Integer with
12     Post => Taille'Result >= 0 and Taille'Result <= Capacite;
13
14   function Element (Vecteur : in T_Vecteur ; Indice : in T_Indice) return Type_Element with
15     Pre => Indice <= Taille (Vecteur);
16
17   procedure Changer (Vecteur : in out T_Vecteur ; Indice : in T_Indice ;
18                      Nouvel_Element: in Type_Element) with
19     Pre => Indice <= Taille (Vecteur),
20     Post => Element (Vecteur, Indice) = Nouvel_Element;
21
22   procedure Ajouter (Vecteur : in out T_Vecteur ; Nouvel_Element: in Type_Element) with
23     Pre => Taille (Vecteur) < Capacite,
24     Post => Element (Vecteur, Taille (Vecteur)) = Nouvel_Element;
25
26 private
27   type T_Elements is array (T_Indice) of Type_Element;
28   type T_Vecteur is
29     record
30       Elements: T_Elements;
31       Taille: Integer;
32     end record;
33 end Vecteurs;
```

Corps : vecteurs.adb

```
1  package body Vecteurs is
2
3    procedure Initialiser (Vecteur: out T_Vecteur) is
4    begin
5      Vecteur.Taille := 0;
6    end;
7
8    function Taille (Vecteur : in T_Vecteur) return Integer is
9    begin
10     return Vecteur.Taille;
11   end;
12
13   function Element (Vecteur : in T_Vecteur ; Indice : in T_Indice) return Type_Element is
14   begin
15     return Vecteur.Elements(Indice);
16   end;
17
18   procedure Changer (Vecteur : in out T_Vecteur ; Indice : in T_Indice ;
19                      Nouvel_Element: in Type_Element) is
20   begin
21     Vecteur.Elements (Indice) := Nouvel_Element;
22   end;
23
24   procedure Ajouter (Vecteur : in out T_Vecteur ; Nouvel_Element: in Type_Element) is
25   begin
26     Vecteur.Taille := Vecteur.Taille + 1;                -- Un nouvel élément
27     Vecteur.Elements (Vecteur.Taille) := Nouvel_Element; -- ajouté
28   end;
29
30 end Vecteurs;
```

Utilisation : exemple_vecteurs.adb

```
1  with Vecteurs;
2  with Ada.Integer_Text_IO;  use Ada.Integer_Text_IO;
3  with Ada.Text_IO;          use Ada.Text_IO;
4
5  procedure Exemple_Vecteurs is
6
7    package Vecteurs_Integer is new Vecteurs(Capacite => 10, Type_Element => Integer);
8    use Vecteurs_Integer;
9
10   package Vecteurs_Character is new Vecteurs(Capacite => 26, Type_Element => Character);
11   use Vecteurs_Character;
12
13   V1 : Vecteurs_Integer.T_Vecteur;
14   Lettres: Vecteurs_Character.T_Vecteur;
15 begin
16   Initialiser (V1);
17   for I in 1..5 loop
18     Ajouter (V1, I ** 2);
19   end loop;
20
21   for I in 1..5 loop
22     Put (Element (V1, I));
23   end loop;
24   New_Line;
25
26   Initialiser (Lettres);
27   for C in Character range 'A'..'Z' loop
28     Ajouter (Lettres, C);
29   end loop;
30
31   for I in 1..26 loop
32     Put (Element (Lettres, I));
33   end loop;
34   New_Line;
35 end Exemple_Vecteurs;
```

- Une fois instancié, on peut utiliser le module :

```
use Vecteurs_Integer;
use Vecteurs_Character;
```

- Le type T_Vecteur est alors ambigu
  - présent dans les deux modules instanciés.
- Il faut donc le préfixer par le nom du module instancié :

```
V1 : Vecteurs_Integer.T_Vecteur;
Lettres: Vecteurs_Character.T_Vecteur;
```

- Inutile de préfixer les noms des sous-programmes grâce à la surcharge

```
Initialiser (V1);
Initialiser (Lettres);
```

Module s'appuyant sur un module générique : Chaines

Objectif

Définir un module `Chaines` qui représente une chaîne de caractères avec les opérations d'accès et de modification d'un élément ainsi que d'ajout à la fin.

Constat

Les opérations sont les mêmes que celles du module `Vecteurs` (avec le type `Character`)

Mise en œuvre

1. Définir le type `T_Chaine` comme étant un nouveau type `T_Vecteur`
2. Spécifier les opérations sur le module avec le type `T_Chaine`
3. Dans l'implantation, convertir explicitement l'objet du type `T_Chaine` en `T_Vecteur` pour utiliser l'opération correspondante du module Vecteurs :

```
T_Vecteur (Une_Chaine)  --! Une_Chaine est du type T_Chaine
```

Interface : chaines.ads

```
1  with Vecteurs;  --! sera utilisé dans la partie `private`
2
3  generic
4    Capacite: Integer;        -- capacité de la chaîne
5  package Chaines is
6    subtype T_Indice is Integer range 1..Capacite;
7    type T_Chaine is limited private;
8
9    procedure Initialiser (Chaine: out T_Chaine) with
10     post => Taille (Chaine) = 0;
11
12   function Taille (Chaine : in T_Chaine) return Integer with
13     post => Taille'Result >= 0 and Taille'Result <= Capacite;
14
15   function Element (Chaine : in T_Chaine ; Indice : in T_Indice) return Character with
16     pre => Indice <= Taille (Chaine);
17
18   procedure Changer (Chaine : in out T_Chaine ; Indice : in T_Indice ;
19                      Nouvel_Element: in Character) with
20     pre => Indice <= Taille (Chaine),
21     post => Element (Chaine, Indice) = Nouvel_Element;
22
23   procedure Ajouter (Chaine : in out T_Chaine ; Nouvel_Element: in Character) with
24     pre => Taille (Chaine) < Capacite,
25     post => Element (Chaine, Taille (Chaine)) = Nouvel_Element;
26
27 private
28   package Vecteurs_Char is new Vecteurs(Capacite => Capacite, Type_Element => Character);
29
30   type T_Chaine is new Vecteurs_Char.T_Vecteur;
31 end Chaines;
```

Corps : chaines.adb

```
1
2  package body Chaines is
3
4    use Vecteurs_Char;  --! Aurait pu être mis dans la partie private de l'interface
5
6    procedure Initialiser (Chaine: out T_Chaine) is
7    begin
8      Initialiser(T_Vecteur(Chaine));
9    end;
10
11   function Taille (Chaine : in T_Chaine) return Integer is
12   begin
13     return Taille(T_Vecteur(Chaine));
14   end;
15
16   function Element (Chaine : in T_Chaine ; Indice : in T_Indice) return Character is
17   begin
18     return Element(T_Vecteur(Chaine), Indice);
19   end;
20
21   procedure Changer (Chaine : in out T_Chaine ; Indice : in T_Indice ;
22                      Nouvel_Element: in Character) is
23   begin
24     Changer(T_Vecteur(Chaine), Indice, Nouvel_Element);
25   end;
26
27   procedure Ajouter (Chaine : in out T_Chaine ; Nouvel_Element: in Character) is
28   begin
29     Ajouter(T_Vecteur(Chaine), Nouvel_Element);
30   end;
31
32 end Chaines;
```

Client : exemple_chaines.adb

```
1  with Chaines;
2  with Ada.Text_IO;           use Ada.Text_IO;
3
4  procedure Exemple_Chaines is
5
6    package Chaines26 is new Chaines(Capacite => 26);
7    use Chaines26;
8
9    Lettres: Chaines26.T_Chaine;
10 begin
11   Initialiser (Lettres);
12
13   for C in Character range 'A'..'Z' loop
14     Ajouter (Lettres, C);
15   end loop;
16
17   for I in 1..26 loop
18     Put (Element (Lettres, I));
19   end loop;
20   New_Line;
21
22 end Exemple_Chaines;
```

Nature des paramètres de généricité

Un paramètre de généricité peut être :

- Un type (par exemple `Type_Element` sur `Vecteurs`)
- Une constante (par exemple `Capacite` sur `Vecteurs`)
- Un sous-programme (à suivre)
- Un paquetage (à suivre)

Exemple : Nouvelles opérations sur des vecteurs

Sans modifier le module déjà écrit, on souhaite écrire de nouveaux sous-programmes :

- Fréquence d'un élément dans un vecteur avec comme paramètre
  - un vecteur de type `T_Vecteur` et
  - l'élément cherché
- Minimum d'un vecteur avec comme paramètre :
  - un vecteur de type `T_Vecteur`

  mais aussi des paramètres de généricité pour :
  - la relation d'ordre à utiliser : fonction `Est_Inferieur`
  - la valeur à retourner si le vecteur est vide : constante `Defaut`

Principes de la mise en œuvre

Interface du module `Vecteurs_Operations`

1. Définir un paramètre de généricité qui est une instance du paquetage Vecteurs :

```
1  generic
2      with package P_Vecteurs is new Vecteurs (<>);
3  package Vecteurs_Operations is
4
5      use P_Vecteurs;          --! pour utiliser le contenu de ce module
```

2. Le `use P_Vecteurs` permet d'avoir un accès au contenu de ce paquetage.
3. Définir le contenu de l'interface, ici les deux fonctions : voir page suivante.
   - On peut utiliser `T_Vecteur` et `Type_Element`
   - Attention : on n'a pas accès à ce qui est privé de `Vecteurs`
   - La fonction Minimum a deux paramètres de généricité supplémentaires :

```
1  generic
2      Defaut: Type_Element;
3      with function Est_Inferieur (Gauche, Droit: in Type_Element)
4          return Boolean;
5  function Minimum (Vecteur: in T_Vecteur) return Type_Element;
```

Corps du module `Vecteurs_Operations`

- Une seule contrainte : on n'a pas accès à ce qui est privé du module `Vecteurs`

Interface : vecteurs_operations.ads

```
1  with Vecteurs;
2
3  generic
4    with package P_Vecteurs is new Vecteurs (<>);
5  package Vecteurs_Operations is
6
7    use P_Vecteurs;          --! pour utiliser le contenu de ce module
8
9
10   -- fréquence de l'élément Elt dans un vecteur...
11   function Frequence (Vecteur: in T_Vecteur; Elt: in Type_Element)
12       return Integer;
13
14
15   -- minimum des éléments d'un vecteur...
16   generic
17     Defaut: Type_Element;
18     with function Est_Inferieur (Gauche, Droit: in Type_Element) return Boolean;
19   function Minimum (Vecteur: in T_Vecteur) return Type_Element;
20
21 end Vecteurs_Operations;
```

Corps : vecteurs_operations.adb

```
1  package body Vecteurs_Operations is
2
3      function Frequence (Vecteur: in T_Vecteur; Elt: in Type_Element) return Integer is
4          Resultat: Integer; -- fréquence de Elt dans Vecteur
5      begin
6          Resultat := 0;
7          for I in 1..Taille (Vecteur) loop
8              if Element (Vecteur, I) = Elt then
9                  Resultat := Resultat + 1;
10             end if;
11         end loop;
12         return Resultat;
13     end Frequence;
14
15     function Minimum (Vecteur: in T_Vecteur) return Type_Element is
16         Resultat: Type_Element; -- Le plus petit élément de Vecteur
17     begin
18         if Taille (Vecteur) = 0 then
19             Resultat := Defaut;
20         else
21             Resultat := Element (Vecteur, 1);
22             for I in 2..Taille (Vecteur) loop
23                 if Est_Inferieur (Element (Vecteur, I), Resultat) then
24                     Resultat := Element (Vecteur, I);
25                 end if;
26             end loop;
27         end if;
28         return Resultat;
29     end Minimum;
30
31 end Vecteurs_Operations;
```

Utilisation du module Vecteurs_Operations

- Il faut commencer par instancier le paquetage Vecteurs :

```
1       package Vecteurs_Integer is
2           new Vecteurs(Capacite => 10, Type_Element => Integer);
3       use Vecteurs_Integer;
```

- Ceci permet de déclarer un T_Vecteur, l'initialiser, etc.
- On peut maintenant instancier le paquetage Vecteurs_Operations

```
1       package Vecteurs_Integer_Operations is
2           new Vecteurs_Operations (P_Vecteurs => Vecteurs_Integer);
3       use Vecteurs_Integer_Operations;
```

- À partir de là, on peut utiliser, la fonction Frequence
- La fonction Minimum étant générique, il faut l'instancier à son tour.

```
1       function Min is
2           new Vecteurs_Integer_Operations.Minimum(
3                                   Defaut => 0, Est_Inferieur => "<");
```

- On peut alors utiliser Min.
- Voir le code complet page suivante.

Client : exemple_vecteurs_operations.adb

```
1  with Vecteurs;
2  with Vecteurs_Operations;
3
4  procedure Exemple_Vecteurs_Operations is
5      package Vecteurs_Integer is
6          new Vecteurs(Capacite => 10, Type_Element => Integer);
7      use Vecteurs_Integer;
8
9      package Vecteurs_Integer_Operations is
10         new Vecteurs_Operations (P_Vecteurs => Vecteurs_Integer);
11     use Vecteurs_Integer_Operations;
12
13     function Min is
14         new Vecteurs_Integer_Operations.Minimum(
15                             Defaut => 0, Est_Inferieur => "<");
16     Valeurs: T_Vecteur;
17 begin
18     Initialiser (Valeurs);
19     Ajouter (Valeurs, 5);
20     Ajouter (Valeurs, 13);
21     pragma Assert (Min (Valeurs) = 5);
22     Ajouter (Valeurs, 3);
23     pragma Assert (Min (Valeurs) = 3);
24     Ajouter (Valeurs, 5);
25     pragma Assert (Frequence (Valeurs, 5) = 2);
26     pragma Assert (Frequence (Valeurs, 7) = 0);
27 end Exemple_Vecteurs_Operations;
```

Plan

- Introduction
- Définition du type pointeur
- Le chaînage
- Pointeur et chaînage : conclusion

Introduction

Déjà abordé : structures de données statiques

- Les structures de données définies et utilisées dans les programmes sont connues, **en totalité**, avant l'exécution du programme, par la déclaration des
  - types de données
  - variables manipulées.
- Pendant l'exécution du programme, il n'est pas possible de
  - changer la structure de données associée à une variable, ou
  - de déclarer une nouvelle variable
- Ces types et ces variables sont dits **statiques**
- Dès lors que ces variables sont déclarées, la mémoire associée **est gérée par le compilateur de façon transparente (implicite)** pour le programmeur

Introduction

Peut-on définir des données de manière dynamique ?

- Peut-on allouer dynamiquement une données associée à une variable d'un programme ?
- Réponse $\Longrightarrow$ **Oui**
- Comment ?
  - en utilisant des **types de données dynamiques** grâce à la manipulation de **pointeurs**
- Des variables de type **pointeur** peuvent être déclarées
  - La mémoire associée à ces variables **est connue**.
  - Elle est **gérée** par le programmeur de **façon explicite**
  - en utilisant des opérations associées au type **pointeur**.

Introduction

Notion de pointeur

- Un pointeur est une adresse en mémoire.
- Cette adresse désigne une **zone mémoire**
- Cette zone mémoire est **typée**

Quelques exemples

Un **pointeur** sur	est une **adresse d'une zone mémoire contenant**
un entier	un entier
un `T_Complexe`	un enregistrement de type `T_Complexe`
un tableau de `T_Complexe`	un tableau d'enregistrements de type `T_Complexe`
un pointeur	un pointeur (donc une autre adresse en mémoire) sur une zone mémoire typée

Introduction

Un pointeur permet de décrire une donnée dynamique

Dans le corps d'un programme (ou d'un sous-programme), on pourra **à la demande** (de manière explicite via une instruction)

- **allouer**, **libérer**, **accéder à**

la zone mémoire dont **l'adresse est la valeur d'un pointeur** (ou zone mémoire adressée par un pointeur).

Pointeur et abstraction

Comme pour les autres types de données, les langages de programmation supportent l'abstraction des adresses en mémoire

- Manipulation symbolique des adresses au travers des pointeurs
- Connaître la valeur de l'adresse n'a pas d'importance !
- Toutes les zones mémoires sont identiques, des bits en mémoire

$\Longrightarrow$ **Impossible de lire ou d'écrire un** pointeur

Définition de pointeurs

Définition d'un pointeur en Algorithmique : déclaration de type

Il faut

- définir le type des pointeurs à définir
- le type des données à pointer

```
TYPE T_Ptr_Sur_T_Nom_Type EST POINTEUR SUR T_Nom_Type
```

  - `T_Ptr_Sur_T_Nom_Type` est un type décrivant des données qui sont des **pointeurs**
  - Le types des données pointées/adressées est `T_Nom_Type`
  - Les valeurs du type `T_Ptr_Sur_T_Nom_Type` sont des adresses de zones mémoires contenant des valeurs de type `T_Nom_Type`
- Cette définition est décrite dans la partie déclaration de types d'un programme ou d'un module

Définition de pointeurs

Exemples

```
TYPE T_Ptr_Entier EST POINTEUR SUR ENTIER
                                  -- pointeurs sur entier
TYPE T_Ptr_T_Complexe EST  POINTEUR SUR T_Complexe
                                  -- pointeurs sur enregistrement
TYPE T_Ptr_T_Tab EST POINTEUR SUR T_Tab
                                  -- pointeurs sur tableau
```

- `T_Ptr_Entier`, `T_Ptr_T_Complexe` et `T_Ptr_T_Tab` définissent des types dont les valeurs sont des pointeurs (adresses) sur des `ENTIER`, `T_Complexe` et `T_Tab`

Remarque.

Le type pointeur peut être

- défini comme tous les autres types d'un langage de programmation. Il **supporte la composition de types**.
  Exemple. On peut définir des *tableaux de pointeurs sur des enregistrements de tableaux de pointeurs sur des* `T_complexe` ...
- utilisé pour typer des constantes, des variables ou bien des paramètres de sous-programmes

Définition de pointeurs

Déclaration de variables pointeurs

Comme toute variable, une variable pointeur est déclarée par

```
Ptr :  T_Ptr_Sur_T_Nom_Type
```

- La variable `Ptr` permet de manipuler un pointeur
- Les **valeurs** du type `T_Ptr_Sur_T_Nom_Type` sont des **adresses** de zones mémoires contenant des valeurs de type `T_Nom_Type`
- Cette définition est décrite dans la partie déclaration de variables d'un programme ou d'un module

Exemples

```
Ptr_Ent   :  T_Ptr_Entier
Ptr_Compl :  T_Ptr_T_Complexe
Ptr_Tabl  :  T_Ptr_T_Tab
```

- `Ptr_Ent`, `Ptr_Compl` et `Ptr_Tabl` : variables représentant des pointeurs (adresses)
- Les valeurs de `Ptr_Ent`, `Ptr_Complexe` et `Ptr_Tabl` sont des adresses

Opérations sur les pointeurs

Opérations sur les de pointeurs

Le pointeur étant un type de données, des **opérations** sont associées à ce type

```
TYPE T_Ptr_Sur_T_Nom_Type EST POINTEUR SUR T_Nom_Type

Ptr_T :  T_Ptr_Sur_T_Nom_Type
```

Elles sont disponibles dans le langage de programmation.

- Quelles sont les opérations définies sur une adresse, un pointeur ?
  - **Initialisation** avec une valeur constante nulle (notée `Null`). Autres initialisations possibles.
  - **Allocation**, **construction** avec l'opérateur `NEW`
  - **Affectation** pour la modification du pointeur avec l'affectation classique `<--`
  - **Accès** à **toute** la zone mémoire adressée avec `^` (exemple : `Ptr_T^`)
    - `Ptr_T^` est équivalent à une variable de type `T_Nom_Type`
  - **Destruction** ou **libération**

Opérations sur les pointeurs (suite)

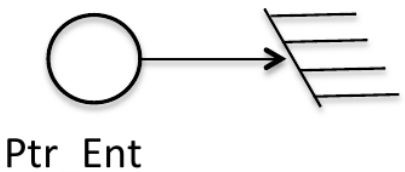
Initialisation

```
Ptr_Ent   <--  Null
Ptr_Compl <--  Null
Ptr_Tabl  <--  Null
```

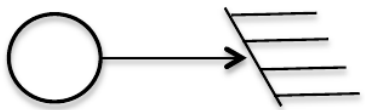
- Les variables `Ptr_Ent`, `Ptr_Compl` et `Ptr_Tabl` sont initialisées à `Null`
- Elles ne pointent sur rien.
- **Attention**. On ne peut donc pas utiliser ^

Opérations sur les pointeurs (suite)

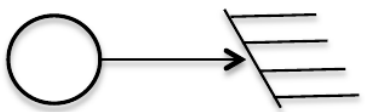
Schématisation



Ptr_Ent



Ptr_Compl



Ptr_Tabl

Opérations sur les pointeurs (suite)

Allocation et construction (opérateur `New`)

```
Ptr_Ent   <--  NEW ENTIER
Ptr_Compl <--  NEW T_Complexe
Ptr_Tabl  <--  NEW T_Tab
```

- Les variables `Ptr_Ent`, `Ptr_Compl` et `Ptr_Tabl` pointent, adressent des zones mémoires contenant respectivement un `ENTIER`, un enregistrement `T_Complexe` et un tableau `T_Tab` respectivement

- L'allocation (en partie droite de `<--`) est réalisée par les opérations `NEW ENTIER`, `NEW T_Complexe` et `new T_Tab`

- **Attention.**
  Ces pointeurs sont typés, même si tous représentent des adresses.

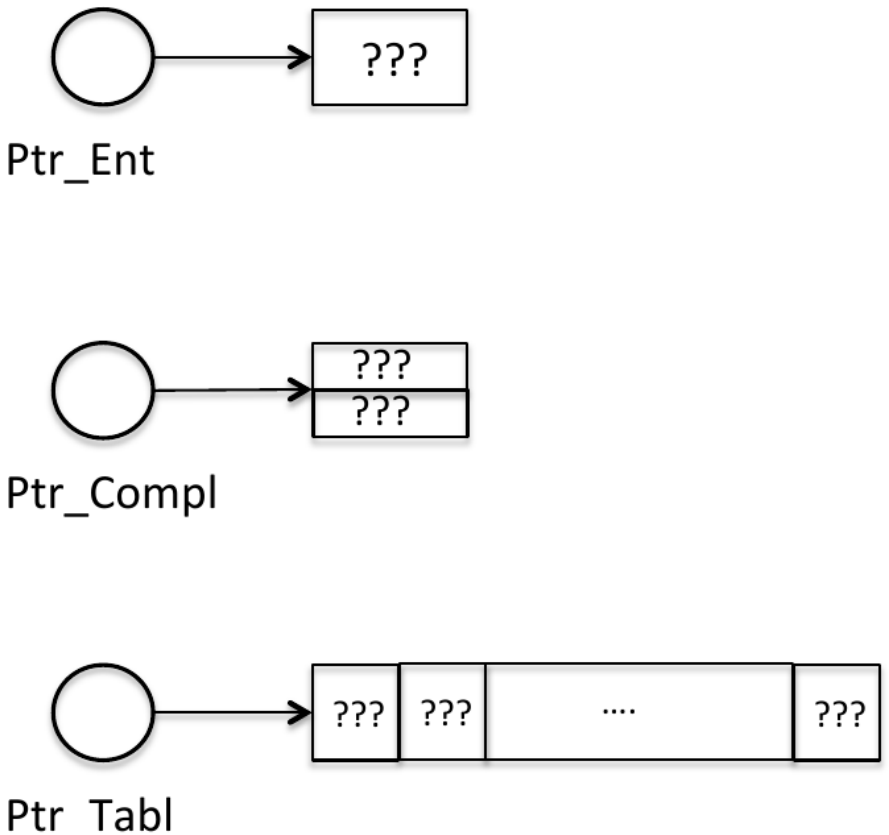
  On **ne pourra pas écrire** `Ptr_Ent <-- Ptr_Compl`

  Certains langages, sans typage permettent cela. Il faudra être vigilant.

Opérations sur les pointeurs (suite)

Schématisation

Les zones mémoire pointées ne sont pas initialisées.



Opérations sur les pointeurs (suite)

Accès et consultation (opérateur ^)

- Les expressions `Ptr_Ent^`, `Ptr_Compl^` et `Ptr_Tabl^` sont respectivement, des entiers, des enregistrement de type `T_Complexe` et `T_Tab`

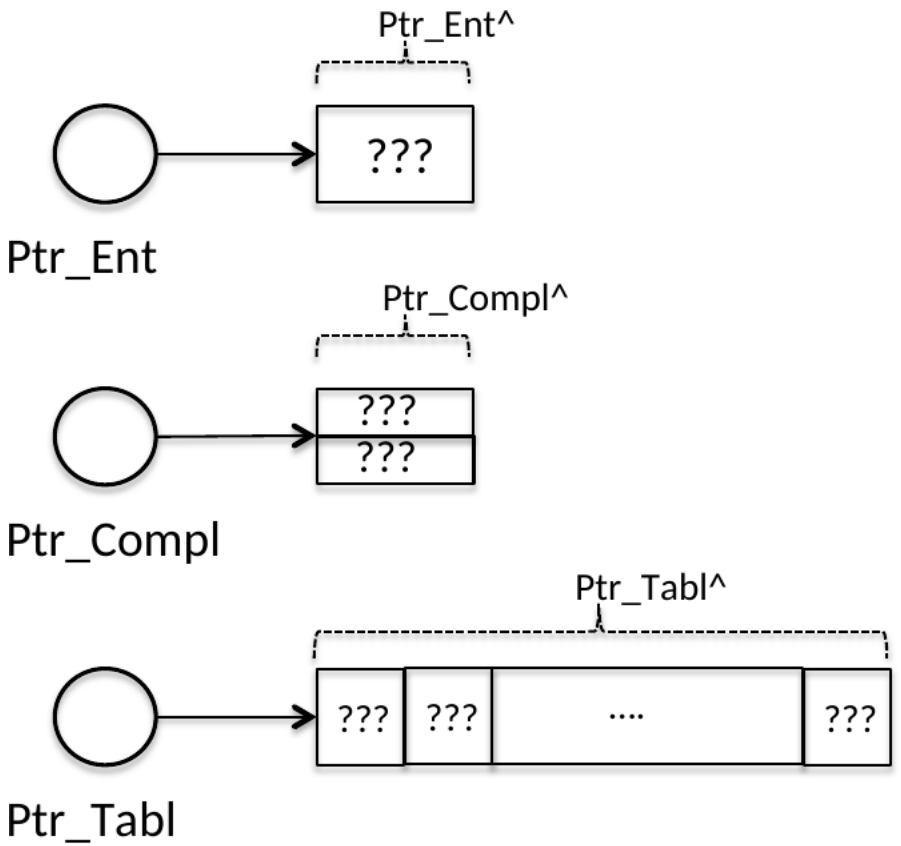
  On peut donc écrire :

```
ECRIRE(Ptr_Ent^ + 48)

SI Ptr_Compl^.Pr + 2.2 <= 3.5 ALORS
     ...

LIRE(Ptr_Tabl^.Elements(1))
```

- Les expressions `Ptr_Ent^ + 48`, `Ptr_Compl^.Pr + 2.2` et `Ptr_Tabl^.Elements(1)` sont sont traitées comme des expressions de type `ENTIER`, `REEL` et `ENTIER` respectivement

Opérations sur les pointeurs (suite)

Schématisation



Opérations sur les pointeurs (suite)

Affectation de pointeurs

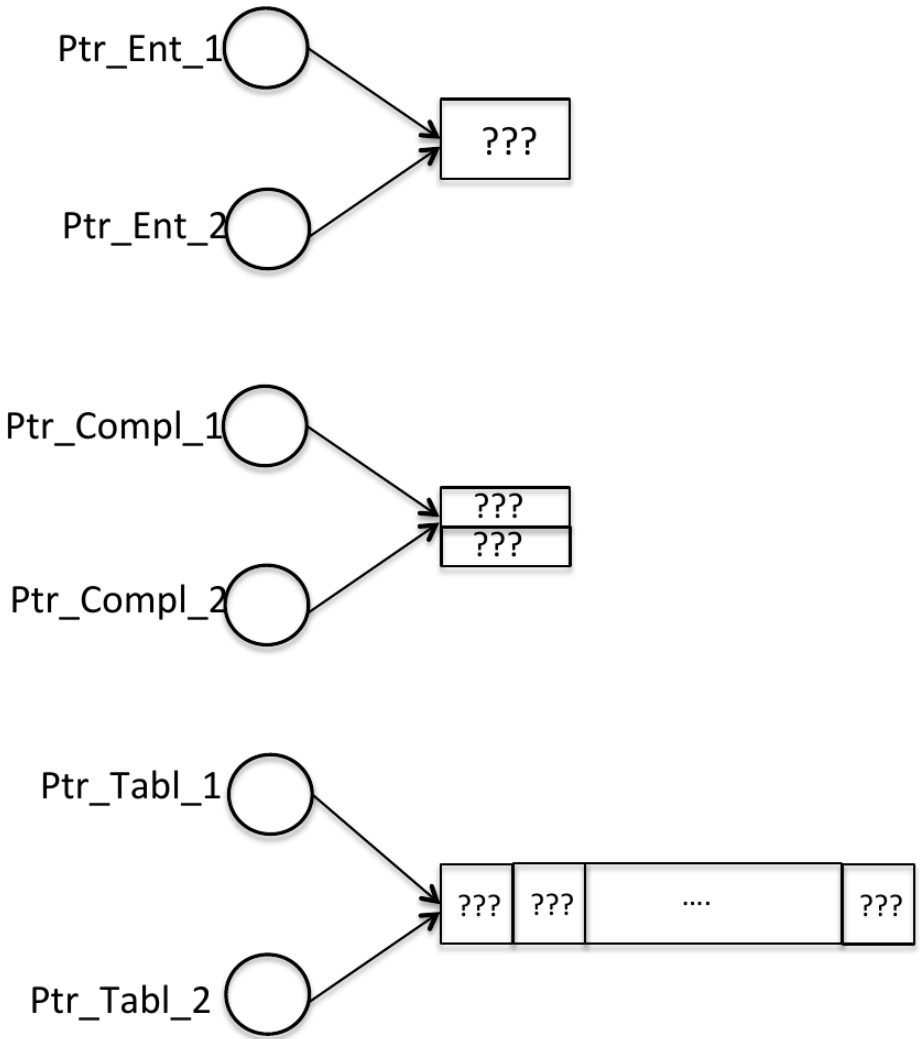
Considérons les variables `Ptr_Ent_1` et `Ptr_Ent_2`, `Ptr_Compl_1` et `Ptr_Compl_2` et `Ptr_Tabl_2`et `Ptr_Tabl_2` de type `ENTIER`, `T_Complexe` et `T_Tab` respectivement. Elles représentent des pointeurs

```
-- Allocation
Ptr_Ent_1   <--  NEW ENTIER
Ptr_Compl_1 <--  NEW T_Complexe
Ptr_Tabl_1  <--  NEW T_Tab


-- Affectations
Ptr_Ent_2   <--  Ptr_Ent_1
Ptr_Compl_2 <--  Ptr_Compl_1
Ptr_Tabl_2  <--  Ptr_Tabl_1
```

Remarque : Les zones allouées ne sont pas (encore) initialisées.

Opérations sur les pointeurs (suite)

Schématisation



Opérations sur les pointeurs (suite)

Modification de la valeur pointée

- On peut ranger une valeur de type T_Nom_Type dans la zone mémoire pointée par un pointeur sur ce type.

```
TYPE T_Ptr_Sur_T_Nom_Type EST POINTEUR SUR T_Nom_Type
Ptr_T :  T_Ptr_Sur_T_Nom_Type
```

- Si Une_Val_De_T est de type T_Nom_Type, alors on peut affecter cette valeur à la zone mémoire pointée par Ptr_T. On écrit

```
Ptr_T^ <-- Une_Val_De_T
```

Exemples

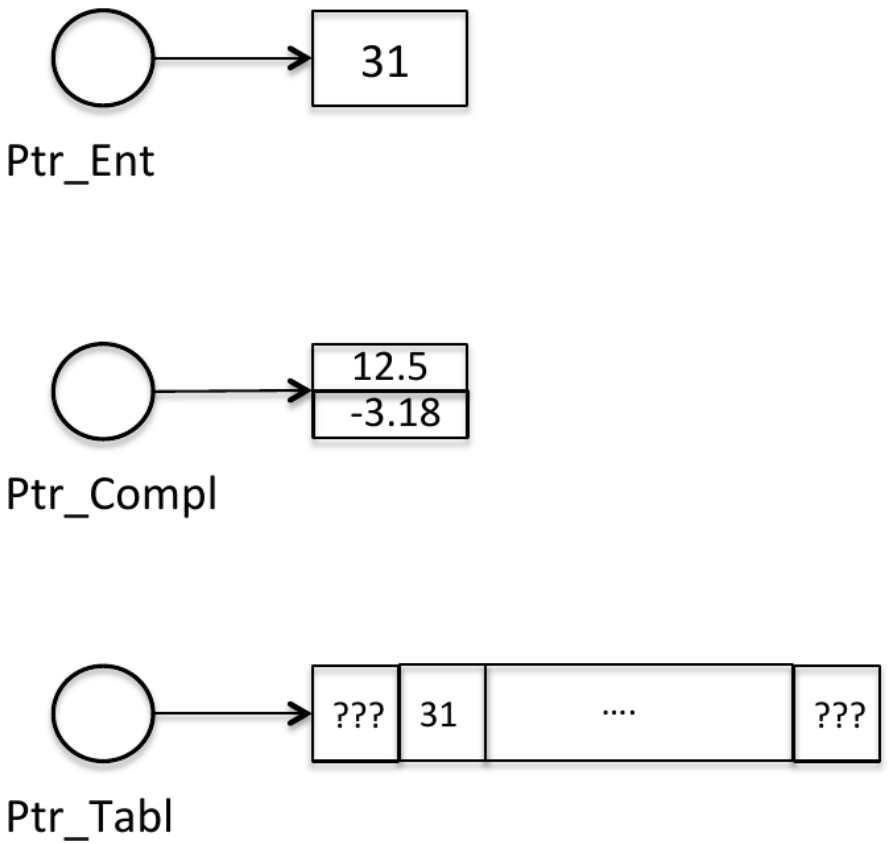
```
-- Allocations
Ptr_Ent   <--  NEW ENTIER
Ptr_Compl <--  NEW T_Complexe
Ptr_Tabl  <--  NEW T_Tab
```

```
-- Affectations
Ptr_Ent^                 <--  31
Ptr_Compl^.Pr            <--  12.5
Ptr_Compl^.Pi            <--  -3.18
Ptr_Tabl^.Elements(2)    <--  31
```

- Les variables Ptr_Ent, Ptr_Compl et Ptr_Tabl pointent (adressent) des zones mémoire modifiées par ces affectations

Opérations sur les pointeurs (suite)

Schématisation



Opérations sur les pointeurs (suite)

Affectation de pointeurs (suite)

Considérons les variables `Ptr_Ent_1`, `Ptr_Compl_1` et `Ptr_Tabl_1` de type `ENTIER`, `T_Complexe` et `T_Tab` respectivement. Elles représentent des pointeurs
Soient les affectations suivantes :

```
Ptr_Ent_1   <--  Ptr_Ent
Ptr_Compl_1 <--  Ptr_Compl
Ptr_Tabl_1  <--  Ptr_Tabl
```

- Les variables `Pt_Ent` et `Pt_Ent_1`, `Pt_Compl` et `Pt_Compl_1` et `Pt_Tabl` et `Pt_Tabl_1` ont la même valeur (elles sont égales). Elles pointent (adressent) la même zone mémoire. On a donc les propriétés suivantes :
  - `Ptr_Ent_1^ = Ptr_Ent^`
  - `Ptr_Compl_1^ = Ptr_Compl^`
    - et aussi `Ptr_Compl_1^.Pr = Ptr_Compl^.Pr`
  - `Pt_Tabl_2^ = Pt_Tabl^`
    - ou bien `Ptr_Tabl_1^.Elements(3) = Ptr_Tabl^.Elements(3)`

Opérations sur les pointeurs (suite)

Libération ou destruction

- La libération ou la destruction consiste à restituer une zone mémoire allouée dynamiquement
- Cette zone devient alors disponible pour d'autres utilisations, en particulier des allocations
- Attention, une zone mémoire libérée **ne doit plus être accédée ni manipulée**

Le ramasse miette

- La libération est à la charge de l'environnement d'exécution d'un programme
- Un composant logiciel de cet environnement, le **ramasse miette** ou **garbage collector** est chargé de récupérer toutes les zones mémoire pointées et non utilisées
- Nous considérons que le compilateur utilisée supporte un tel composant.

Le chaînage

Pointeurs

A ce stade du cours, le type de données pointeur est défini. On sait :

- Déclarer, Allouer et Manipuler un pointeur
- Accéder et Modifier le contenu de la zone pointée par un pointeur

Composition

On peut définir des structures composées de différents types de données dont les pointeurs. Par exemple

- Un tableau de pointeurs sur des entiers,
- Un enregistrement comprenant un pointeur sur entier et un tableau de pointeurs sur des entiers
- Un pointeur sur un enregistrement composé d'un entier et d'un pointeur sur un autre enregistrement
- ...

Mais, ces exemples ne permettent pas de créer un pointeur qui pointe une structure (un enregistrement par exemple) comprenant, entre autre, un pointeur de même type.
Exemples : une **liste**, une **file**, une **pile**, un **arbre**, etc.

$\Longrightarrow$ besoin de **Chaînage**

Le chaînage

Intérêt du chaînage ?

Comme son nom l'indique, le **chaînage** permet de chaîner, de relier des éléments de même type selon une structure logique.

Différentes formes de chaînage

- **linéaire** : une suite d'éléments de même type chainés l'un à la suite de l'autre
- **linéaire double** : une suite d'éléments de même type chaînés l'un à la suite de l'autre et dans les deux sens
- **hiérarchique** : des éléments de même type chaînés selon un hiérarchie
- **maillé** : des éléments de même type chaînés les uns aux autres sans contrainte particulière

Le chaînage : une liste linéaire simple

Comment réalise-t-on le chaînage ?

- principe de l'adressage indirect i.e. Une donnée qui contient une adresse, ...
- définition **circulaire** $\Longrightarrow$ Un pointeur de type `T_Liste` pointe sur une structure de type `T_Cellule` comprenant un autre pointeur de type `T_Liste`

Schéma général de déclaration (circulaire)

```
TYPE T_Liste EST POINTEUR SUR T_Cellule   --! Type pointeur sur un type T_Cellule

TYPE T_Cellule EST ENREGISTREMENT
   ...                                    --! Attributs du type T_Cellule
   Suivant : T_Liste                      --! Suivant est un pointeur sur une T_Cellule
FIN ENREGISTREMENT
```

Déclaration de variable

```
Liste : T_Liste
```

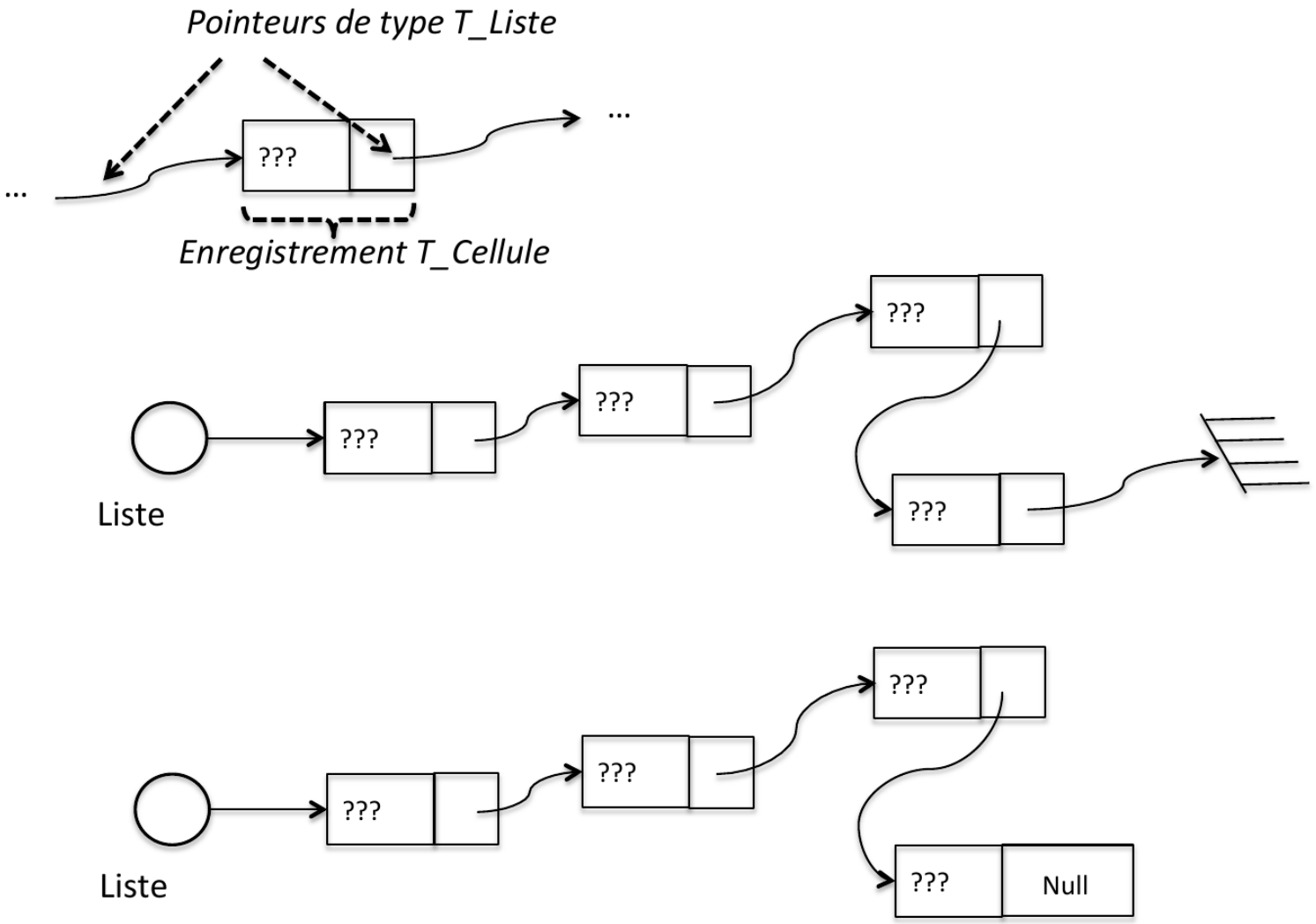
`Liste` est un pointeur sur une structure chaînée

Attention

Ce sont les opérations qui définissent la logique de construction du chaînage.

Le chaînage : une liste linéaire simple

Schématisation



Le chaînage : une liste linéaire double

Comment réalise-t-on le chaînage linéaire double ?

Besoin de deux pointeurs : un suivant et un précédent

Schéma général de déclaration

```
TYPE T_Liste_Double EST POINTEUR SUR T_Cellule  -- Type pointeur sur un type
                                                -- T_Cellule
TYPE T_Cellule EST ENREGISTREMENT
  ...                                -- Attributs du type T_Cellule
  Suivant   : T_Liste_Double         -- Suivant est un pointeur sur une T_Cellule
  Precedent : T_Liste_Double         -- Precedent est un pointeur sur une T_Cellule
FIN ENREGISTREMENT
```

Déclaration de variable

```
  Liste_D : T_Liste_Double
```

`Liste_D` est un pointeur sur une structure doublement chaînée

Attention

Ce sont les opérations qui définissent la logique de construction du chaînage.

Le chaînage : une liste linéaire double

Schématisation

Donner un schéma représentant une liste linéaire doublement chainée

Le chaînage : une structure hiérarchique

Comment réalise-t-on un chaînage hiérarchique ?

Besoin de deux pointeurs : un fils gauche et un fils droit

Schéma général de déclaration

```
TYPE T_Hierarchie EST POINTEUR SUR T_Cellule  -- Type pointeur sur un type
                                              -- T_Cellule
TYPE T_Cellule EST ENREGISTREMENT
  ...                            -- Attributs du type T_Cellule
  Fils_Gauche : T_Hierarchie     -- Fils_Gauche  est un pointeur sur une T_Cellule
  Fils_Droit : T_Hierarchie      -- Fils_Droit est un pointeur sur une T_Cellule
FIN ENREGISTREMENT
```

Déclaration de variable

```
  Liste_H : T_Hierarchie
```

`Liste_H` est un pointeur sur une structure hiérarchique

Attention

Ce sont les opérations qui définissent la logique de construction du chaînage.
En effet, cette déclaration est semblable celle de la liste linéaire double.

Le chaînage : une structure hiérarchique

Schématisation

Donner un schéma représentant un chainage hiérarchique

Retour sur la liste linéaire simple

Reprenons le cas d'une liste linéaire simple ?

Pour simplifier, nous considérons que la liste contient des caractères

Schéma général de déclaration

```
TYPE T_Liste_Caractere EST POINTEUR SUR T_Cellule_Caractere
                            -- Type pointeur sur un type T_Cellule_Caractere

TYPE T_Cellule_Caractere EST ENREGISTREMENT
  Element : Caractere       -- Contenu du type T_Cellule_Caractere
  Suivant : T_Liste_Caractere
                            -- Suivant est un pointeur sur une T_Cellule_Caractere
FIN ENREGISTREMENT
```

Soit `Liste` une variable de type `Liste : T_Liste_Caractere`

Question

1. Ecrire la séquence d'instructions permettant de construire une liste linéaire simple contenant les caractères `'P'`, `'I'`, `'M'`, puis qui affiche les éléments de cette liste sous la forme `*PIM*`

Liste linéaire simple en Ada

Déclaration d'un type pointeur : access

```
type T_Cellule_Caractere;    --! pour annoncer la référence en avant

type T_Liste_Caractere is access T_Cellule_Caractere;
                        --! Type pointeur sur un type T_Cellule_Caractere

type T_Cellule_Caractere is record
    Element : Character;
    Suivant : T_Liste_Caractere;  -- Accès à la cellule suivante
end record;
```

Déclaration de variable

```
Liste : T_Liste_Caractere;
```

Accès à la zone pointé : .All

```
Liste.All            --! équivalent à Liste^ en algorithmique
Liste.All.Element    --! équivalent de Liste^.Element en algorithmique
```

Liste linéaire simple en Ada (suite)

Manipulation

```
Liste := Null;                          -- Initialisation du pointeur Liste.
                                        -- La liste est vide
...

Liste := new T_Cellule_Caractere;       -- La variables Liste contient
                -- un pointeur sur une zone de type T_Cellule_Caractere
...

Liste.All.Element := 'U';               -- La valeur 'U' est rangée
Liste.All.Suivant := Null;              -- Le suivant est Null

...

Put (Liste.All.Element)                 -- Affichage de U

...

if Liste.All.Suivant = Null then ...
```

Retour sur la liste linéaire simple

Autres questions

Spécifier et écrire le corps d'un sous-programme permettant

1. d'initialiser une liste linéaire simple. La liste est vide
2. d'ajouter un caractère en début d'une liste linéaire simple
3. d'afficher les caractères d'une liste linéaire simple
4. d'indiquer si un caractère c est présent dans une liste linéaire simple
5. de rechercher un caractère c dans une liste linéaire simple. Le sous-programme renvoie le pointeur sur la valeur c si elle est dans la liste, sinon, elle renvoie *null*
6. d'insérer un caractère c après un caractère donné c' dans une liste linéaire simple. Si c' n'est pas dans cette liste, alors il est inséré en début de liste
7. ...

Pointeur et Chaînage : Conclusion

Une autre représentation en mémoire

- Les pointeurs et le chaînage offrent une autre représentation en **mémoire** des collections de données de même type
  - Des opérations de construction, d'accès, de manipulation, etc. doivent être définies
  - Ces opérations devront garantir un chaînage correct
- L'autre représentation en **mémoire** étant un tableau

Types abstraits de données

- Les types de données peuvent donc être implantés en **mémoire** de façons différentes tout en offrant les mêmes services
- L'encapsulation permet de cacher la représentation choisie $\Longrightarrow$ On obtient des types abstraits de données (TAD)
- La définition de TAD sur des types génériques est également possible

Plan

Exceptions

Définition : Larousse

- Ce qui est hors de la loi commune, qui paraît unique

- En droit : **Moyen soulevé** dès le début de l'instance **par le défendeur** afin de **faire reconnaître par le juge l'irrégularité** de la procédure, son incompétence ou pour **obtenir qu'il sursoie à statuer**

Définition

Une exception est un évènement qui provoque l'interruption de l'exécution d'un programme suite à la détection d'une situation anormale

Elle traduit le cas d'un fonctionnement non nominal, ou exceptionnel d'un sous-programme.

Exceptions : exemples

Quels types d'erreurs ?

Erreurs détectées par le processeur (la machine) ou les librairies (ex.modules) utilisées

- Division par zéro
- Manque de mémoire
- ...

Erreurs de programmation (dans les langages de programmation)

- Débordement de tableaux
- Accès à `null.Nom`
- ...

Erreurs issues de choix de conception (le problème étudié comporte des cas d'exception)

- Dépiler une pile vide
- Le cas a$=0$ d'une équation de la forme $a\,x + b = 0$
- Le cas de l'indice d'un tableau qui dépasse le nombre effectif d'éléments d'un tableau
- ...

Exceptions

Propriétés

Une exception est

- **déclarée** dans la partie déclaration d'un programme par un identificateur. On écrit `Ident_Except : EXCEPTION`
- **déclenchée** ou **levée** par un évènement extérieur au programme ou bien explicitement dans le programme par l'instruction `LEVER` (`raise` en Ada).
  On écrit `LEVER Ident_Except` ou `raise Ident_Except;` en Ada
- **propagée** à un autre programme ou au même programme selon des règles de propogation
- **traitée** dans un bloc de traitement d'exception (proche d'un **Selon**)
  mot-clés `EXCEPTION` et => en Algo `exception` et `when` en Ada

Elle peut aussi être **prédéfinie** dans le langage de programmation si elle est connue et consensuelle (exemples : division par zéro, débordement de tableau, ...)

Notion de bloc

Définition

Un bloc est une partie de programme composée de plusieurs instructions ou d'autres blocs.

Propriétés

- Un bloc est **délimité** par les clauses `DEBUT` et `FIN`
- Un sous-programme définit donc un bloc
- Les blocs sont **ordonnés** selon l'**ordre** des appels
  $\Longrightarrow$ Bloc précédent appelle bloc suivant
- Le bloc programme **principal** est le **premier bloc** dans cet ordre
- Chaque bloc a un **bloc précédent** (à l'exception du programme principal)

Notion de bloc

Bloc avec sous-programme

Sous programme définissant un bloc.

```
PROGRAMME test_exception EST
TYPE   ...

    PROCEDURE xx ( ......) EST
    DEBUT
    ...
    FIN xx
      ...
DEBUT
    ...
    xx ( ...)
    ...
FIN test_exception
```

Bloc d'instructions

Bloc crée au sein d'une séquence d'instructions.

```
...
DEBUT
    I1
    I2
    I3
    DEBUT
        I5
        I6
        I7
    FIN
    I8
    I9
FIN
...
```

Gestion des exceptions

Sans gestionnaire d'exceptions.

Détectée et gérée par le programmeur au sein d'un programme

```
PROGRAMME PP1 (...) EST
    X, Y, A : ENTIER
DEBUT
    ...
    SI X = 0 ALORS
        ECRIRE ("Division par zéro")
    SINON
        Y <-- A / X
    FIN SI
    ...
FIN PP1
```

Avec un gestionnaire d'exceptions

Connue du gestionnaire d'exception qui permet la levée d'une exception.

```
PROGRAMME PP1(...)  EST
    X, Y, A : ENTIER
DEBUT
    ...
    Y <-- A / X
    ...
EXCEPTION
    Constraint_Error =>
        ECRIRE ("Division par zéro")
FIN PP1
```

Traitement de l'exception

Le fragment de programme suivant

```
EXCEPTION
    Constraint_Error => ECRIRE ("Division par zéro")
```

réalise le traitement de l'exception. On le désignera par un `Traite_Exception`

Catégories d'exception

Deux types d'exception

- Exceptions **prédéfinies** issues du langage de programmation et de la machine
- Exceptions **utilisateurs** définies et **déclarées** par le concepteur du programme

Exceptions prédéfinies

Quelques exemples d'exceptions prédéfinies en Ada

- `Constraint_error` lorsqu'un élément de type numérique ou non numérique prend une valeur en dehors d'un intervalle autorisé
- `Data_error` erreurs issues de la manipulation de numériques ou issues du typage (par exemple lecture au clavier d'un caractère au lieu d'un nombre)
- `Storage_error` erreurs liées au manque de mémoire
- `Status_error` erreurs liées à la manipulation de fichiers
- ...

Le manuel de définition du langage définit ces différentes exceptions... et d'autres

Catégories d'exception

Définition d'exceptions utilisateurs

Exceptions déclarées par l'utilisateur à l'aide d'identificateurs d'exceptions.
Exemples d'identificateurs d'exception

- `Pile_Pleine_Error`
- `Division_Par_Zero_Error`

Attention, il ne s'agit pas d'identificateurs de variables. **Il n'est pas affectable**

Définition d'exceptions

- A l'image des constantes et des types,
  un identificateur d'exception doit être déclaré **AVANT** les sous-programmes
- Syntaxe générale de déclaration d'une exception :

```
Identificateur_D_Exception : EXCEPTION
```

- Exemple

```
Pile_Pleine_Error : EXCEPTION
```

Gestion des exceptions

Occurrence d'une exception

- **Automatique** : la machine détecte une erreur (division par zéro ...)
  - Cas précédent de la division par zéro avec gestionnaire d'exception
- **Propagation** : programmée par le programmeur
  - Dans un cas limite, le programme souhaite "déclencher " une exception
    $\Longrightarrow$ on dit que le programme **lève** l'exception

Levée d'une exception en algorithmique

- Syntaxe `LEVER Identificateur_Exception`
- Exemple, en algorithmique

```
SI a = 0 ALORS LEVER Division_Par_Zero_Error
```

Levée d'une exception en Ada

- Syntaxe `raise Identificateur_Exception;`
- Exemple

```
if a = 0 then raise Division_Par_Zero_Error;
```

Gestion des exceptions

Traitement d'exception

Lorsqu'une exception est levée, elle peut être

- traitée au sein d'un bloc de traitement d'exception `Test_Exception`
- ou propagée si elle n'est pas traitée

Bloc de traitement d'exception

```
PROGRAMME Test_Exception
    ...
DEBUT
    I1
    I2
    I3
EXCEPTION
    Constraint_Error => écrire (''erreur'')
FIN test_exception
```

- `I1` provoque une division par 0, donc une occurrence d'exception survient
- Arrêt du bloc courant (ici du programme principal)
- Exécution du bloc `Traite_Exception`, qui interrompt l'exécution du bloc courant (donc du programme `Test_Exception`)
- Continuation au bloc précédent, s'il y en a (ici il n'y en n'a pas)

Gestion des exceptions

Traitement d'exception en présence de sous-programmes

Exception levée dans un sous-programme, propagée puis traitée

Un programme principal

```
PROGRAMME Principal EST
    I2
    Appel de Proc_1
    I3
FIN Principal
```

Un sous-programme

```
PROCEDURE Proc_1 (....) EST
DEBUT
    ...
    I  -- peut lever une exception E
    ...
EXCEPTION
    E => T1
FIN Proc_1
```

Comportement lors de l'occurrence de l'exception

- Principal a appelé Proc_1, Proc_1 s'exécute et une exception E est levée dans Proc_1
- Arrêt définitif du bloc courant (Proc_1)
- Recherche d'un Traite_Exception pour E dans le bloc courant $\Longrightarrow$ il existe
- Exécution du Traite_Exception
- Continuation au bloc précédent $\Longrightarrow$ à l'instruction I3

Gestion des exceptions

Traitement d'exception en présence de sous-programmes

Exception levée dans un sous-programme puis propagée

Un programme principal

```
PROGRAMME Principal EST
    I2
    Appel de Proc_1
    I3
FIN Principal
```

Un sous-programme

```
PROCEDURE Proc_1 (....) EST

DEBUT
    ...
    I  -- peut lever une exception E
    ...
FIN Proc_1
```

Comportement lors de l'occurrence de l'exception

- Principal a appelé Proc_1, Proc_1 s'exécute et une exception E est levée dans Proc_1
- Arrêt définitif du bloc courant (Proc_1)
- Recherche d'un Traite_Exception pour E dans le bloc courant $\Longrightarrow$ il n'en existe pas
- Propagation au bloc précédent (Programme Principal)
- Recherche d'un Traite_Exception pour E dans le bloc Principal $\Longrightarrow$ il n'en existe pas
- Arrêt définitif du programme Principal

Gestion des exceptions

Un programme principal

```
PROCEDURE Principal EST
DEBUT
    I2
    Appel de Proc_1
    I3
FIN Principal
```

Des sous-programmes

```
PROCEDURE Proc_2 (...) EST
DEBUT
    ...
    I1 -- peut lever une exception E
    ...
FIN Proc_2
PROCEDURE Proc_1 (...) EST
DEBUT
    ...
    Appel de Proc_2
    ...
EXCEPTION
    E => T1
FIN P1
```

Comportement lors de l'occurrence de l'exception

- Arrêt immédiat et définitif du bloc courant (c'est-à-dire Proc_2)
- Recherche dans le bloc courant(Proc_2) d'un Traite_Exception pour E $\Longrightarrow$ il n'y en n'a pas
- Propagation de l'exception vers le bloc précédent (Proc_1)
- Arrêt immédiat du bloc courant (Proc_1)
- Recherche dans le bloc courant(Proc_1) d'un Traite_Exception pour E $\Longrightarrow$ il existe
- Exécution du Traite_Exception du bloc courant (Proc_1)
- Continuation au bloc précédent (Principal)$\Longrightarrow$ l'éxécution se poursuit en I3

Gestion des exceptions

Un programme principal

```
PROCEDURE Principal EST
DEBUT
    I1
    Appel de Proc_1
    I2
EXCEPTION
    E => T1
FIN PP
```

```
PROCEDURE Proc_2 (...) EST
DEBUT
    ...
    I1 -- peut lever une exception E
    ...
FIN Proc_2

PROCEDURE Proc_1 (...) EST
DEBUT
    ...
    Appel de Proc_2
    ...
FIN Proc_1
```

Comportement lors de l'occurrence de l'exception

- Arrêt immédiat du bloc courant (Proc_2)
- Recherche dans le bloc courant(Proc_2) d'un Traite_Exception pour E $\Longrightarrow$ il n'y en n'a pas
- Propagation de l'exception vers le bloc précédent (Proc_1)
- Arrêt immédiat du bloc courant (Proc_1)
- Recherche dans le bloc courant(Proc_1) d'un Traite_Exception pour E $\Longrightarrow$ il n'y en n'a pas
- Propagation de l'exception vers le bloc précédent (Principal)
- Arrêt immédiat du bloc courant (Principal)
- Recherche dans le bloc courant(Principal) d'un Traite_Exception pour E $\Longrightarrow$ il y en a
- Exécution du Traite_Exception et fin du programme principal (I2 n'est pas exécutée)

Gestion des exceptions

Exemple de propagation d'exceptions

Propagation de l'exception

```
FONCTION Eq_Affine (FA, FB : IN REEL)
                RETOURNE REEL EST

DEBUT
    RETOURNE  - FB / FA

EXCEPTION
    Constraint_Error =>
        LEVER Division_Par_Zero_Error

FIN Eq_Affine
```

Traitement de l'exception

```
FONCTION Eq_Affine (FA, FB : IN REEL)
                RETOURNE REEL EST

DEBUT
    SI  FA = 0 ALORS
        LEVER Division_Par_Zero_Error
    SINON
        RETOURNE - FB / FA
    FIN SI

FIN Eq_Affine
```

Quel est l'impact de ces deux traitements ? Commenter ?

Propagation d'exceptions

Exemple de programme avec propagation d'exception déclarée

```
PROCEDURE Exemple_Exception est

  -- Déclaration d'exception
  Exception_A_Nul_Error : Exception

  -- Spécification  de la fonction ........
  FONCTION  Eq_Affine ( FA, FB : IN Reel) RETOURNE REEL EST
    DEBUT
        RETOURNE (-FB / FA )
    EXCEPTION
        Constraint_Error => LEVER Exception_A_Nul_Error
    FIN Eq_Affine


    A, B, Res : réel
  DEBUT
    ...
    Res <-- Eq_Affine(A, B)

  EXCEPTION
    Exception_A_Nul_Error => ECRIRE ("Attention, le cofficient est nul")

FIN Exemple_Exception
```

Gestion des exceptions

Conception en présence d'exceptions

- Faire apparaître les exceptions dans les contrats des sous-programmes en donnant :
  - le nom de l'exception possiblement levée
  - les conditions de levée de cette exception
- Détecter les cas d'erreurs le plus tôt possible dans un programme
- Traitement de l'exception
  - Ne traiter une exception que si le bloc courant dispose des informations nécessaires pour la traiter
  - $\Longrightarrow$ ne pas hésiter à propager des exceptions si les informations nécessaires à leur traitement sont absentes ou incomplètes

  ou bien
  - $\Longrightarrow$ ne pas hésiter à en lever de nouvelles, plus explicites, puis à les propager

Question

Comment traite-on des exceptions dans un langage qui n'offre pas de gestionnaire d'exceptions ?

Plan

- Définition
- Construction de TAD
- Services offerts par un TAD
- Programmation offensive, programmation défensive
- TAD usuels en programmation

Introduction

Déjà vu

A ce stade du cours, nous disposons de l'ensemble des constructions permettant de concevoir et d'implanter des algorithmes dans un langage de programmation impérative

- Types de données statiques et dynamiques, variables, affectations, structures de contrôle,
- sous-programmes, modules
- encapsulation, généricité

Autres thèmes abordés

- Méthodologie de conception par raffinement
- Définition de tests et écriture de programmes de test
- Programmation pas contrat
- Gestion d'exceptions
- Notions d'architecture logicielle avec les modules

Introduction

Structure de mémorisation interne

Une structure de mémorisation interne est une structure de données issue directement de l'architecture d'un ordinateur. Elle reflète le fonctionnement interne de la machine.

Deux structures de mémorisation interne sont identifiées

- Le **tableau** représentant une zone contiguë en mémoire. Il est caractérisé par
  - sa base (adresse de début) et sa capacité (taille)
  - un accès à un élément en temps constant (base + déplacement)
- Le **pointeur et le chainage qui en découle** définissant des adresses mémoire de zones mémoire. Il est caractérisé par
  - la manipulation explicite de ces adresses sans en connaître la valeur
  - une adresse d'une zone mémoire typée, il est donc typé
  - l'adressage indirect permet l'accès aux zones mémoire pointées ou adressées
  - lorsque les pointeurs définissent un chainage, l'accès aux éléments se fait en parcourant ce chainage par des indirections.

Définition

Définition d'un TAD

Un **Type Abstrait de Données** noté **TAD** est la donnée

1. d'un **Type** de données c'est-à -dire
   1. la définition des **éléments (données)** du type
   2. l'ensemble des **services (sous-programmes)** permettant de manipuler ces éléments (données)

en **cachant (en abstrayant)** la représentation interne des éléments (données) de ce type

Exemples

On peut construire des TAD pour les structures de données communément utilisées en programmation

- pile, file, arbre, graphe, ensemble, tableau, liste, fichier, etc

mais aussi pour des entités du monde réel

- une carte, un jeu de carte, un étudiant, une horloge, une distributeur de billets, un gestionnaire de fichiers, un système de réservation de billets d'avion, etc.

Construction d'un TAD

Construction en plusieurs étapes

1. **Structuration**. Choisir (dans le langage de programmation) la structure ou l'unité qui permet de décrire ce TAD.
   Exemple : module, classe, frame, etc.

2. **Identification**. Identifier le type abstrait des éléments (des données) du TAD à définir, au sens du langage de programmation.
   On utilisera pour cela les constructions de types offertes par le langage ou bien des références à d'autres TAD.

3. **Services offerts**. Etablir l'ensemble des services offerts par le TAD, c'est-à -dire, tous les sous-programmes permettant de le manipuler. Il faudra définir :
   - la spécification de chaque service,
   - les éventuelles exceptions qui pourraient apparaître.

4. **Généricité**. Abstraire partout où il est possible de paramétrer le type obtenu à l'étape 2.

Construction d'un TAD

Construction en plusieurs étapes (suite)

5. **Encapsulation**. Cacher le type associé aux éléments (données) identifié à l'étape 2. On utilisera les concepts offerts par le langage de programmation s'il en possède.

6. **Implantation**. Choisir la structure de mémorisation interne (plus éventuellement les TAD référencés) qui implantent le type identifié à l'étape 2 et encapsulé à l'étape 5.

7. **Définition**. Ecrire dans la structure choisie à l'étape 1 l'unité représentant le TAD obtenu.

8. **Test**. Définir les programmes de test du TAD obtenu, en testant chacun de ses services individuellement.

Un exemple de TAD : la file

Définition

Un TAD est associé au concept de file de la façon suivante

1. **Structuration**. Définition d'un module `P_File`
2. **Identification**. Introduction du type `T_File`
3. **Services offerts**. `File_Vide`, `Enfiler`, `Defiler`, `Tete`, `Queue` et `Est_Vide`
4. **Généricité**. Les éléments (de type `Type_Element`) contenus dans une file
5. **Encapsulation**. Elle porte sur le type `T_File` qui ne doit être manipulé qu'avec les services du TAD.
6. **Implantation**. `T_File` est implanté par la structure de mémorisation interne *Tableau* ainsi que **deux indices pour repérer la tête et la queue** de la file
7. **Définition**. Le paquetage `P_File` obtenu.
8. **Test**. Ensemble de programmes de test pour les services

Un exemple de TAD : la file – Suite

Une autre implantation

1. **Structuration**. Définition d'un module `P_File`
2. **Identification**. Introduction du type `T_File`
3. **Services offerts**. `File_Vide`, `Enfiler`, `Defiler`, `Tete`, `Queue` et `Est_Vide`
4. **Généricité**. Les éléments (de type `Type_Element`) contenus dans une file
5. **Encapsulation**. Elle porte sur le type `T_File` qui doit être manipulé qu'avec les services du TAD.
6. **Implantation**. `T_File` est implanté par la structure de mémorisation interne *Tableau* ainsi qu'un **seul indice pour repérer la queue** de la file, la tête étant toujours dans la première case du tableau
7. **Définition**. Le paquetage `P_File` obtenu.
8. **Test**. Ensemble de programmes de test pour les services

Remarque

Nous disposons de **deux implantations encapsulées** (vues en TP) pour les **mêmes service**.

Un autre exemple de TAD : la pile

Définition

Un TAD est associé au concept de pile de la façon suivante

1. **Structuration**. Définition d'un module `P_Pile`
2. **Identification**. Introduction du type `T_Pile`
3. **Services offerts**. `Pile_Vide`, `Empiler`, `Depiler`, `Sommet` et `Est_Vide`
4. **Généricité**. Les éléments (de type `Type_Element`) contenus dans une pile
5. **Encapsulation**. Elle porte sur le type `T_Pile` qui ne doit être manipulé qu'avec les services du TAD.
6. **Implantation**. `T_Pile` est implanté par la structure de mémorisation interne *liste chainée simple*
7. **Définition**. Le paquetage `P_Pile` obtenu est décrit (voir transparent suivant)
8. **Test**. Ensemble de programmes de test pour les services

Remarques

- Comme cela a été vu dans un cours précédent, une autre **implantation** est possible avec des tableaux.

Un autre exemple de TAD : Interface du TAD Pile

```
1   generic
2       type Type_Element is private;
3   package P_Pile is
4
5       type T_Pile is  private;
6
7       -- Spécification de la fonction Init_Pile
8       function Init_Pile return T_Pile;
9
10      -- Spécification de la procédure empiler
11      procedure Empiler(Fp: in out T_Pile ; Fe : in Type_Element);
12
13      -- Spécification de la fonction Est_Vide
14      function Est_Vide (Fp: in T_Pile) return Boolean;
15
16      -- Spécification de la procédure dépiler
17      procedure Depiler(Fp: in out T_Pile);
18
19      -- Spécification de la fonction sommet
20      function Sommet(Fp: in T_Pile) return Type_Element;
21
22  private
23      type T_Cellule_Pile is record
24          Element : Type_Element;
25          Suivant : T_Pile;
26      end record;
27
28      type T_Pile is access T_Cellule_Pile;
29
30  end P_Pile;
```

Services associés à un TAD

Définition des services associés à un TAD

Dans la définition des services, il faudra s'assurer de la **complétude** des services. Il s'agit de définir un ensemble de services permettant de manipuler les éléments ou valeur d'une TAD **sans avoir recours à sa représentation interne**.

De quels services doit-on disposer ?

Trois catégories principales de services

- **Constructeurs** $\Longrightarrow$ Construire un élément du TAD.
- **Accesseurs ou Observateurs** $\Longrightarrow$ **Accéder** à tout élément du TAD et **observer** certaines propriétés des éléments du TAD.
- **Modifieurs ou Transformateurs** $\Longrightarrow$ **Modifier** les éléments du TAD, voire **supprimer** tout ou partie de ces éléments ou tous ou bien définir une opération qui **transforme** les éléments du TAD (en général dans un autre type ou TAD)

Services associés à un TAD (Suite)

De quels services doit-on disposer ?

Trois catégories principales de services

- **Constructeurs**
  - Exemples. `Pile_Vide` `Empiler`, `Initialiser`, `Ajouter`, `Insérer`, etc.
- **Accesseurs ou Observateurs**
  - Exemples d'accesseurs $\Longrightarrow$ `Sommet`, `Premier` ou plus généralement `Rend_XXX` ou `Get_XXX`
  - Exemples d'observateurs $\Longrightarrow$ `Egalite`, `Element_De`, `Appartient`, `Plus_Petit`, ou plus généralement `Is_xxx`
- **Modifieurs ou Transformateurs**
  - Modifieurs. On distingue souvent deux types de modifieur
    - Modifieurs qui **affectent de nouvelles valeurs** à des éléments de TAD déjà existants. Exemples $\Longrightarrow$ `Modifier_Nom`, ou plus généralement `Set_XXX`
    - Modifieurs qui ont pour effet de **supprimer** un élément de TAD
  - Transformateurs. Ils correspondent à des traitements particuliers associés au type (service utilitaires). Exemples $\Longrightarrow$ nombre d'éléments dans une file, fusion de deux files, etc.

Services associés à un TAD (Suite)

De quels services doit-on disposer ?

Trois catégories principales de services

- **Constructeurs**. Le TAD est en résultat
- **Accesseurs ou Observateurs**
  - Accesseurs. Le TAD est en donnée seulement
  - Observateurs. Le TAD est en donnée seulement
- Modifieurs et transformateurs
  - Modifieurs. Le TAD est en donnée et en résultat
  - Transformateur. Le TAD est en donnée et en résultat

Programmation offensive, programmation défensive

Retour sur la spécification d'un sous-programme

Lors de la spécification d'un sous-programme, on distingue deux types de comportement

- **nominal** correspondant au fonctionnement *normal* du sous-programme
- **non nominal** correspondant au fonctionnement *exceptionnel* du sous-programme

Ces comportements sont pris en compte dans la spécification comme suit.

- Lorsque seuls les **cas nominaux** sont pris en compte alors la pré-condition doit décrire les obligations des entrées (IN) pour que seuls les comportements nominaux soient conservés $\Longrightarrow$ **Couverture partielle**

  C'est l'appelant qui doit s'assurer **avant l'appel** que la pré-condition est vraie

- Lorsque les **cas nominaux** ainsi que les **cas non nominaux** sont pris en compte alors, en général, la pré-condition n'impose pas d'obligation, mais la post-condition prévoit la levée d'une ou de plusieurs exceptions pour les cas non nominaux $\Longrightarrow$ **Couverture totale**

  C'est l'appelant qui doit, **après l'appel**, **décider** de **traiter** ou de **laisser se propager** les exceptions qui résulteraient de cet appel.

Programmation offensive, programmation défensive

Retour sur la spécification d'un sous-programme (Suite)

Considérons la procédure `Depiler` du paquetage `P_Pile`

- Avec pré-condition sur le cas **nominal**

```
-- Spécification de la procédure Depiler
-- ...
-- Précondition :   Non Est_Vide(Fp)
-- ...

PROCEDURE Depiler (Fp : IN OUT T_Pile ) EST
DEBUT

   Fp <-- Fp^.Suivant

FIN
```

Programmation par contrat. Contrat de confiance entre appelé et appelant

Programmation offensive, programmation défensive

Retour sur la spécification d'un sous-programme (Suite)

Considérons la procédure `Depiler` du paquetage `P_Pile`

- **1er Cas.** Avec exception sur le cas **non-nominal**

```
-- Spécification deê la procédure Depiler
-- ...
-- Précondition : Vrai
-- ...
-- Exception : Pile_Vide lorsque la pile est vide
PROCEDURE Depiler (Fp : IN OUT T_Pile ) EST
DEBUT
    Fp <-- Fp^.Suivant
EXCEPTION
    Null_Pointer_Exception ==>
        LEVER Pile_Vide
FIN Depiler
```

- Traitement explicite par le sous-programme des cas anormaux (levée d'une exception, retour d'un code d'erreur, traitement particulier, etc.)
- Besoin de connaître l'exception du langage de programmation. Elle est propagée avec l'exception du concepteur du TAD
- **Attention**, plusieurs instructions pourraient lever la même exception issue du langage de programmation alors qu'elles correspondent à des exceptions conceptuellement différentes au niveau du TAD.

Programmation offensive, programmation défensive

Retour sur la spécification d'un sous-programme (Suite)

Considérons la procédure `Depiler` du paquetage `P_Pile`

- **2nd Cas.** Avec exception sur le cas **non-nominal**

```
-- Spécification deê la procédure Depiler
-- ...
-- Précondition : Vrai
-- ...
-- Exception : Pile_Vide lorsque la pile est vide
PROCEDURE Depiler (Fp : IN OUT T_Pile ) EST
DEBUT
    Si Non Est_Vide(Fp) ALORS
        Fp <-- Fp^.Suivant
    SINON
        LEVER Pile_Vide
    FIN_SI
FIN
```

- Traitement explicite par le sous-programme des cas anormaux (en levant une exception, en retournant un code d'erreur, en faisant un traitement particulier, etc.)
- Besoin de faire le test de l'erreur qui peut être coûteuse en temps d'exécution et/ou en occupation mémoire

Programmation offensive, programmation défensive

Programmation défensive

La programmation **défensive** est un style de programmation qui correspond à l'écriture de sous-programmes (donc de spécifications) qui prennent en compte les cas d'erreur pour les situations suivantes

- **nominale** qui correspond au fonctionnement *normal* du sous-programme
- **non nominale** qui correspond au fonctionnement *exceptionnel* du sous-programme

Par *"exceptionnel"* on entend tous les cas d'erreurs qui correspondent aussi bien **aux erreurs de conception et de programmation** qu'aux **autres erreurs** (exemple : erreur de saisie)

On dit que le programme **se défend contre les utilisations erronées**

Programmation offensive, programmation défensive

Programmation offensive

La programmation **offensive** est un style de programmation qui correspond à l'écriture de sous-programmes (donc de spécifications) qui font confiance aux programme(s) appelant ces sous-programmes.

- Le **respect des pré-conditions et post-conditions** avant et après les appels permet de garantir la **correction des programmes** conçus.
- Les **pré-conditions** sont décrites pour **garantir l'exclusion des appels correspondants à ces cas d'erreurs**
- En programmation offensive, **seules les erreurs qui ne peuvent être prises en compte** par le programmeur sont **traitées de manière** *"défensive"* à **l'aide d'exceptions** par exemple
  Exemple : saisie d'un caractère au lieu d'un entier
- La **programmation par contrat** correspond à de la **programmation offensive**

Programmation offensive, programmation défensive

Remarques

- La **programmation offensive** est un cas de **programmation défensive** dans laquelle les **erreurs de conception et de programmation sont prises en compte** dans le programme (la spécification). On décrit
  - une pré-condition pour définir la partie du domaine de valeurs des paramètres qui ne correspond pas au traitement nominal
  - les traitements des cas d'erreurs en dehors du domaine par des exceptions (en général des levées d'exceptions)
- En **programmation défensive**, des **exceptions** doivent être **prévues pour traiter les cas d'erreurs qui n'auraient pas été pris en compte**. La pré-condition peut être toujours vraie.

Programmation offensive, programmation défensive (suite)

Remarques

- Si on montre **(vérification)** que chaque appel respecte les pré-conditions du sous-programme appelé et que chaque sous-programme établit ses post-conditions alors on dit que le programme est **correct par construction**.

  La preuve de programme est un moyen de réaliser la vérification.

- **Attention**. Un **programme vérifié n'est pas forcément valide**. Il faut s'assurer que le programme final satisfait les exigences du cahier des charges **(validation)**.

  Les tests, la relecture de spécifications sont deux moyens de réaliser la validation. D'autres moyens de validation sont abordés dans d'autres enseignements.

Programmation offensive, programmation défensive

Mise en œuvre

Les comportements des programmes sont pris en compte dans la spécification comme suit.

- Lorsque seuls les **cas nominaux** sont pris en compte alors la pré-condition doit décrire les obligations des entrées (IN) pour que seuls les comportements nominaux soient conservés

  $\Longrightarrow$ **Couverture partielle**

  L'appelant qui doit s'assurer **avant l'appel** que la pré-condition est vraie
  La post-condition doit prévoir la levée d'exception pour les cas d'erreurs qui ne peuvent être pris en compte dans les pré-conditions

  $\Longrightarrow$ **Couverture totale**

- Lorsque les **cas nominaux** ainsi que les **cas non nominaux** sont pris en compte alors, en général, la pré-condition n'impose pas d'obligation, et la post-condition doit prévoir **la levée d'une ou de plusieurs exceptions** pour les cas non nominaux

  $\Longrightarrow$ **Couverture totale**

C'est l'appelant qui doit, **après l'appel**, **décider** de **traiter** ou de **laisser se propager** les exceptions qui résulteraient de cet appel.

Plusieurs TAD sont couramment utilisés en programmation

TAD usuels en programmation

La conception fait appel à l'utilisation de types de données spécifiques pour représenter les concepts, entités ou données manipulées par ces programmes.

- Enregistrements
- Tableaux
- Tableaux associatifs (dictionnaires)
  - table de hachage
- Listes linéaires simples, liste linéaires doubles
- Piles
- Files
- Arbres
  - Arbres binaires
    - Arbres binaires de recherche (ABR)

Plusieurs TAD sont couramment utilisés en programmation

TAD usuels en programmation

La conception fait appel à l'utilisation de types de données spécifiques pour représenter les concepts, entités ou données manipulées par ces programmes.

- Graphes
- Fichiers
- Ensembles

Remarque

Cette liste concerne les TAD couramment utilisés en programmation.

En général **d'autres TAD sont définis pour des concepts, entités ou données manipulés par un programme** (exemple : un TAD pour une carte, un jeu de carte). Ils sont en général **représentés à l'aide d'un des TAD** énumérés ci-dessus.

Exemple de TAD

Le TAD Pile

Compléter la spécification du TAD Pile défini sur le transparent 13 en décrivant les exceptions associées

Plusieurs TAD sont couramment utilisés en programmation

Exercices

1. Pour les TAD suivants
   - Files
   - Arbres binaires de recherche
   - Graphes
   - Ensembles

   dire quelles sont les structures de mémorisation internes qui peuvent les implanter.

2. Spécifier et implanter le TAD représentant les files dans un module. On procédera
   - au choix de la structure de mémorisation interne
   - à la définition d'un type générique pour les éléments de la file

Plan

Introduction

Architecture (Larousse)

Plusieurs définitions.

- Art de construire les bâtiments.
- Caractère, ordonnance, style d'une construction : Monument d'une belle architecture.
- Ce qui constitue l'ossature, les éléments essentiels d'une œuvre ; structure. L'architecture d'un roman.
- **Organisation des divers éléments constitutifs d'un système informatique, en vue d'optimiser la conception de l'ensemble pour un usage déterminé.**

Architecture logicielle (D. Garlan et. al)

L'architecture logicielle décrit d'une manière **symbolique et schématique** les différents **éléments** d'un ou de plusieurs systèmes informatiques, leurs **interrelations** et leurs **interactions**.

Définition. Entité Logicielle

Entité Logicielle

Une entité logicielle est une unité de programme représentant un **traitement** ou une **ensemble de traitements** identifiés.

Exemples

- programme principal, sous-programme procédure ou fonction, module comme package ou package body, classe, composant, processus, interface, etc.

Représentation

On utilise un rectangle dans lequel est inscrit le nom (identifiant) de l'entité.

Autres entités

D'autres entités peuvent être identifiées dans un système d'information : "réseau", entité "matérielle", "physique" ou entité "utilisateur".
Elles ne sont pas étudiées dans ce cours. Elles sont abordés dans d'autres enseignements

Entités Logicielles et Raffinement

Entités Logicielles et Raffinement

La **méthode des raffinages** est mise en œuvre pour **concevoir** des traitements qui sont **structurés** au sein d'**entités logicielles**.

- Programme principal et sous-programmes **sont conçus** par la **méthode des raffinages**
- Modules, classes, composants, etc. **intègrent** plusieurs programmes et sous-programmes **conçus par la méthode des raffinages**

Entité logicielle et Raffinement sont deux concepts orthogonaux

- Le raffinement permet de **concevoir les entités logicielles de type programme ou sous-programme**.
  - Il définit/identifie les **traitements** à réaliser et leur **contrôle**
    $\Longrightarrow$ Enchainement de traitements, traces d'exécution
- Les entités logicielles **structurent** les traitements en **briques, pièces, composants** d'un système d'information. **Ces traitements sont identifiés par le raffinement.**
  - Elles définissent **l'architecture** d'un système d'informations
    $\Longrightarrow$ Représentation spatiale

Définition. Liens/Relations entre Entités Logicielles

Liens entre entités logicielles

Des liens structurels lient les entités logicielles entre elles.

- Ils mettent en relation une ou plusieurs entités logicielles
- Ces liens sont **orientés** et définissent une **dépendance**
- Ainsi, on peut associer un graphe à chaque application.
- Une application peut être décrite par
  - un ensemble d'entités logicielles qui définissent les **sommets d'un graphe**
  - et de liens entre ces entités qui définissent les **arêtes d'un graphe**

$\Longrightarrow$ Obtention d'un **Graphe de dépendances** associé à chaque application.

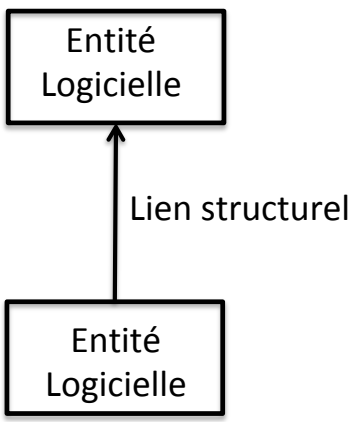
Ce graphe peut être exploité pour le test, la réutilisation, la documentation, etc.

Définition. Liens/Relations entre Entités Logicielles (suite)

Exemples de lien entre entités logicielles

- Liens étudiés dans ce cours.
  Exemples : *Appel*, *réalisation (ou implantation)*, *utilisation*, *instanciation*
- Autres liens non étudiés dans ce cours (abordés dans d'autres enseignements).
  Exemples : *héritage*, *abstraction*, *inclusion*,

Représentation



- Les sommets sont des entités et les arcs sont des liens structurels orientés

Liens/Relations entre Entités Logicielles

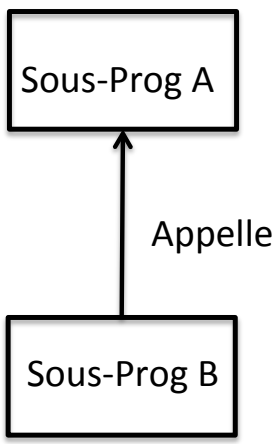
Le lien d'appel `Appelle`

Il indique le lien d'appel entre sous-programmes ou entre programme principal et sous-programmes

Exemple

La procédure de tri par sélection `Tri_Par_Selection` appelle la fonction de recherche du plus petit élément `Recherche_Min`

Représentation



- L'entité logicielle sous-programme *Sous_Prog B* appelle l'entité logicielle sous-programme *Sous_Prog A*

Liens/Relations entre Entités Logicielles

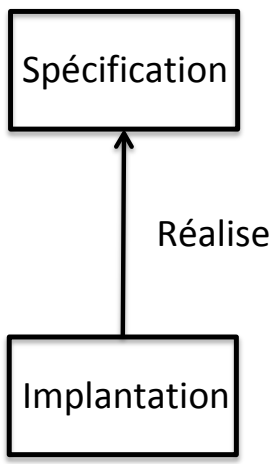
Le lien de réalisation ou d'implantation `Réalise`

Il indique le lien entre une entité de spécification et une entité qui l'implémente

Exemple

Le module corps Ada `package body Liste_Chainee` réalise le module spécification `package Liste_Chainee`

Représentation



- L'entité *Implantation* réalise, implémente l'entité *Spécification*

Liens/Relations entre Entités Logicielles

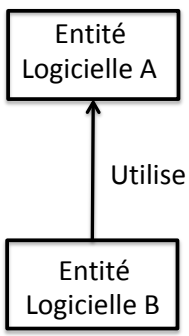
Le lien d'utilisation `Utilise`

Il indique le lien entre une entité logicielle qui utilise ou importe ou invoque une autre entité logicielle.

Exemple

Le module Ada `package Arbre` utilise le module ada `package Liste_Chainee`

Représentation



- L'entité *Entité Logicielle B* utilise, importe, invoque l'entité *Entité Logicielle A*

Liens/Relations entre Entités Logicielles

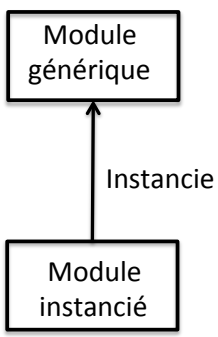
Le lien d'instantiation `Instancie`

Il indique le lien entre une entité logicielle générique et une entité logicielle qui l'implémente.

Exemple

Le module Ada `package Liste_Entier` instancie le module générique `package Liste_Chainee`

Représentation



- L'entité *Module Instancié* instancie l'entité *Module Générique*

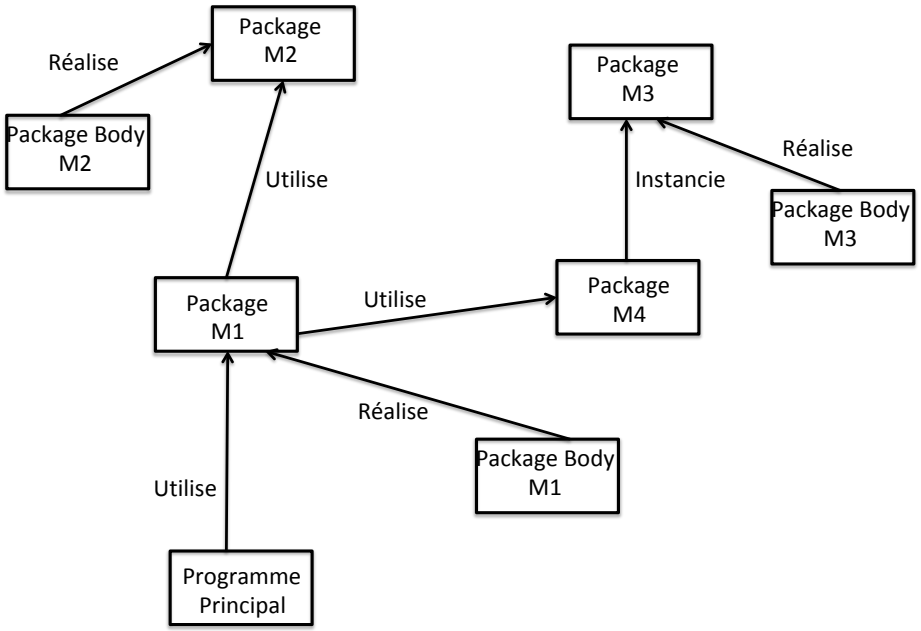
Représentation d'une application logicielle

Architecture : représentation spatiale

- L'architecture d'une application logicielle est définie par l'ensemble des entités logicielles qui la composent et des liens (relations) entre ces entités.
- Graphe connexe avec les entités comme sommets et les relations comme arêtes.
- Lorsqu'il existe, le programme principal représente le *"point d'entrée"*.

Représentation d'une application logicielle (Suite)

Exemple



- Représentation d'une application avec les modules M1, M2, M3 et M4 et un programme principal.

- L'entité `Programme Principal` représente le point d'entrée
- Les entités `Package Body M_i` définies par le lien *Réalise* peuvent être omises.

On utilisera ce type de représentation pour décrire l'architecture d'une application.

Remarque

Des langages de description d'architectures logicielles (ADL - Archtecture Description Language) ont été définis dans la littérature

Plan

Conclusion

- Les principes de programmation impérative étudiés dans ce cours s'appliquent à tous les langages de programmation utilisant le style impératif
- Ce cours est en lien avec plusieurs autres enseignements de l'informatique
  - la programmation en langage C, abordée ensuite, est en lien directe avec la programmation impérative
  - la programmation orientée objets abordée au semestre suivant
  - programmation concurrente ou distribuée, la programmation à base d'évènements, les systèmes états-transitions, la preuve de programme, le test de programmes, le génie logiciel, etc.
- Ce cours met en avant la nécessite pour un ingénieur d'utiliser des méthodes rigoureuses
  - pour la conception de produits que sont les programmes et
  - de pouvoir raisonner sur ces produits

FIN

{yamine, cregut, kjr}@enseeiht.fr