IFT339
Structures de données

Thème 1 : Types abstraits et Structures de données

Aïda Ouangraoua
Département d'informatique

UNIVERSITÉ DE
SHERBROOKE

Objectifs

Types Abstraits de Données (TAD) : généralisations des types prédéfinis (int, float, bool, array, etc.)
Permettent de définir de nouveaux types avec les opérations possibles sur ces types.

Définition

Type Abstrait de Données (TAD) : description d'un ensemble fini de données et d'opérations possibles sur ces données en faisant abstraction de la structure des données (détails d'implémentation)
Vue logique d'un ensemble de données associé à la spécification des opérations possibles sur ces données (création, accès, modifications)

Exemple : Entier

Nom du TAD
Ensemble des données
Prototypes des opérations possibles sur les données.

Entier
Entiers entre -2^{31} et $2^{31}-1$
Entier : Chaîne_carac → Entier **==** : Entier → Booléen **!=** : Entier → Booléen **<** : Entier → Booléen **>** : Entier → Booléen **+** : Entier → Entier **-** : Entier → Entier **/** : Entier → Entier ***** : Entier → Entier

Note : dans ce cours, nous utilisons une syntaxe orientée objet : l'objet est toujours un paramètre/retour implicite des opérations. La valeur de retour implicite est notée en ajoutant le suffix __ : Exemple : x.operateur()__ désigne le retour implicite de l'appel x.operateur().

Définition

Type Abstrait de Données (TAD) : description d'un ensemble fini de données et d'opérations possibles sur ces données en faisant abstraction de la structure des données (détails d'implémentation)
Vue logique d'un ensemble de données associé à la spécification des opérations possibles sur ces données (création, accès, modifications)

Exemple : Point2D

Nom du TAD
Ensemble des données
Prototypes des opérations possibles sur les données.

Point2D
Points dans un plan
Point2D : Réel x Réel → Point2D **==** : Point2D → Booléen **Distance** : Point2D → Réel **Modifier** : Point2D → Ø

Note : dans ce cours, nous utilisons une syntaxe orientée objet : l'objet est toujours un paramètre/retour implicite des opérations. La valeur de retour implicite est notée en ajoutant le suffix __ : Exemple : x.operateur()__ désigne le retour implicite de l'appel x.operateur().

Exemple:
Point2D a = Point2D (1,1)
a.Modifier(Point2D (2,2))

Définition

Type Abstrait de Données (TAD) : description d'un ensemble fini de données et d'opérations possibles sur ces données en faisant abstraction de la structure des données (détails d'implémentation)
Vue logique d'un ensemble de données associé à la spécification des opérations possibles sur ces données (création, accès, modifications)

Exemple : Ensemble d'objets

TAD
Type des données
Ensemble d'opérations possibles sur les données.

Ensemble_b
Ensemble de Billes
Ensemble_b : Ø → Ensemble_b **tirer** : Ø → Bille **ajouter** : Bille → Ø **taille** : Ø → Entier

Définition

TAD caractérisé par:

- Son identité (nom, adresse)
- Type des données (domaine)
- Prototype des opérations possibles sur les données (**Noms** et types des valeurs de paramètres/retours)
- Sémantique des opérations (description du comportement)

Point2D
Points dans un plan
Point2D : Réel x Réel → Point2D **==** : Point2D → Booléen **Distance** : Point2D → Réel **Modifier** : Point2D → Ø

Définition

TAD caractérisé par:

- ❑ Son identité (nom, adresse)
- ❑ Type des données (domaine)
- ❑ Prototype des opérations possibles sur les données (**Noms** et types des valeurs de paramètres/retours)
- ❑ Sémantique des opérations (description du comportement)

Point2D
Points dans un plan
Point2D : Réel x Réel → Point2D **==** : Point2D → Booléen **Distance** : Point2D → Réel **Modifier** : Point2D → Ø

Indépendant de:

- ❑ Représentation utilisée pour les données et algorithmes utilisés pour les opérations

  Exemple pour Point2D : représentation avec coordonnées polaires (rayon, angle) ou coordonnées cartésiennes (abscisse, ordonnée) ?

- ❑ Langage de programmation utilisée pour l'implémentation

Définition

TAD caractérisé par:

- ❑ Son identité (nom, adresse)
- ❑ Type des données (domaine)
- ❑ Prototype des opérations possibles sur les données (**Noms** et types des valeurs de paramètres/retours)
- ❑ Sémantique des opérations (description du comportement)

Point2D
Points dans un plan
Point2D : Réel x Réel → Point2D **==** : Point2D → Booléen **Distance** : Point2D → Réel **Modifier** : Point2D → Ø

Indépendant de:

- ❑ Représentation utilisée pour les données et algorithmes utilisés pour les opérations

  Exemple pour Point2D : représentation avec coordonnées polaires (rayon, angle) ou coordonnées cartésiennes (abscisse, ordonnée) ?

- ❑ Langage de programmation utilisée pour l'implémentation

Impossible pour un utilisateur de modifier un TAD (risque d'introduire des incohérences)

Spécification de la sémantique des opérations

- Comportement des opérateurs spécifié par des axiomes :
  - Équations de la forme e = e' telles que e et e' sont deux expressions donnant des valeurs de même type
  - Permettent de définir la relation entre les valeurs des paramètres et retours des opérations

Exemples avec le TAD Ensemble_b

Ensemble_b
Ensemble de Billes
Ensemble_b : Ø → Ensemble_b **tirer** : Ø → Bille **ajouter** : Bille → Ø **taille** : Ø → Entier

- Ensemble_b().taille() = 0
- Ensemble_b().tirer() = Erreur
- (x.tirer()__).taille() = x.taille() - 1
- (x.ajouter(b)__).taille() = x.taille() + 1
- x.ajouter(x.tirer())__ = x

Exercice 1

Le bottin d'une entreprise contient le répertoire des employés de l'entreprise. Pour chaque employé, on y trouve : nom, prénom, département, téléphone, et courriel.

Définir un ensemble de types abstraits permettant de gérer des bottins d'entreprises. Pour chaque TAD, décrire le nom, le domaine, et le prototype des opérations.

Structure de données vs TAD

- **Structure de données** : Implémentation d'un TAD dans un langage donnée, incluant la représentation des données, et l'implémentation des algorithmes des opérations spécifées dans le TAD. Elle peut aussi inclure des opérations cachées aux utilisateurs (non spécifiées dans le TAD).

Structure de données vs TAD

❑**Structure de données** : Implémentation d'un TAD dans un langage donnée, incluant la représentation des données, et l'implémentation des algorithmes des opérations spécifiées dans le TAD. Elle peut aussi inclure des opérations cachées aux utilisateurs (non spécifiées dans le TAD).

Point2D
Points dans un plan
Point2D : Réel x Réel → Point2D **==** : Point → Booléen **Distance** : Point → Réel

Exemples en C++

Structure de données vs TAD

❑**Structure de données** : Implémentation d'un TAD dans un langage donnée, incluant la représentation des données, et l'implémentation des algorithmes des opérations spécifiées dans le TAD. Elle peut aussi inclure des opérations cachées aux utilisateurs (non spécifiées dans le TAD).

Point2D
Points dans un plan
abscisse (Réel) **ordonnée** (Réel) **Point2D** : Réel x Réel → Point2D **==** : Point → Booléen **Distance** : Point → Réel **Changer_abs** : Réel → Ø **Changer_ord** : Réel → Ø

Exemples en C++

Structure de données vs TAD

❑**Structure de données** : Implémentation d'un TAD dans un langage donnée, incluant la représentation des données, et l'implémentation des algorithmes des opérations spécifiées dans le TAD. Elle peut aussi inclure des opérations cachées aux utilisateurs (non spécifiées dans le TAD).

Point2D
Points dans un plan
abscisse (Réel) **ordonnée** (Réel) **Point2D** : Réel x Réel → Point2D **==** : Point → Booléen **Distance** : Point → Réel **Changer_abs** : Réel → Ø **Changer_ord** : Réel → Ø

Exemples en C++

Fichier Point.h:
I- Identité et domaine visibles (public)
I- Prototypes des opérations spécifiées dans le TAD (public)
I- Représentation des données et prototypes des opérations non-spécifiées dans le TAD cachés (privé)

Fichier Point.cpp:
Implémentation des algorithmes des opérations cachée (abstraction procédurale)

Vue utilisateur vs Vue concepteur

❑Deux vues pour les programmeurs

- **Vue Utilisateur (TAD)**
  - ne connaît que les informations et opérations spécifiées dans le TAD
  - pas d'accès à la représentation et aux détails internes
- **Vue Concepteur (Structure de données)**
  - conçoit et implémente le TAD; Accès à tous les détails
  - accès à la représentation et aux détails d'implémentation

Point2D
Points dans un plan
abscisse (Réel) **ordonnée** (Réel) **Point2D** : Réel x Réel → Point2D **Point2D** : Ø → Point2D **==** : Point → Booléen **Init** : Réel x Réel → Ø **Distance** : Point → Réel **Changer_abs** : Réel → Ø **Changer_ord** : Réel → Ø

Point2D
Points dans un plan
rayon (Réel) **angle** (Réel) **Point2D** : Réel x Réel → Point2D **Point2D** : Ø → Point2D **==** : Point → Booléen **Init** : Réel x Réel → Ø **Distance** : Point → Réel **Changer_ray** : Réel → Ø **Changer_angl** : Réel → Ø

- Même TAD

- Deux implémentations différentes

Vue utilisateur vs Vue concepteur

❑Deux vues pour les programmeurs

- **Vue Utilisateur (TAD)**
  - ne connaît que les informations et opérations spécifiées dans le TAD
  - pas d'accès à la représentation et aux détails internes
- **Vue Concepteur (Structure de données)**
  - conçoit et implémente le TAD; Accès à tous les détails
  - accès à la représentation et aux détails d'implémentation

Point2D
Points dans un plan
abscisse (Réel) **ordonnée** (Réel) **Point2D** : Réel x Réel → Point2D **Point2D** : Ø → Point2D **==** : Point → Booléen **Init** : Réel x Réel → Ø **Distance** : Point → Réel **Changer_abs** : Réel → Ø **Changer_ord** : Réel → Ø

Informations et spécifications du TAD

Représentation des données

Implémentation des algorithmes des opérations

Abstraction : séparation entre la spécification du TAD et son implémentation. Cela permet une conception modulaire, rigoureuse, et l'intégration de nouveaux types à partir des types pré-définis.

Programmation par type abstrait

- **Respecte les principes de:**
  - **Modularité** : division des objets en modules (types)
    - Exemple pour un ensemble de TAD permettant de gérer un ensemble de polygones particuliers (rectangle, carré, triangle équilatéral, losange) dans un plan et les manipuler : translation en spécifiant un vecteur (point), rotation autour du centre du polygone ou autour de l'origine du plan en spécifiant un angle, calcul de l'aire, de la circonférence, affichage d'une figure, affichage de l'ensemble des figures.

Point2D
Points dans un plan

Polygone
Polygones dans un plan

EnsemblePolygones
Ensembles de polygones dans un plan

Rectangle
Rectangles dans un plan

Triangle
Triangles dans un plan

Losange
Losanges dans un plan

Carré
Carrés dans un plan

Programmation par type abstrait

- **Respecte les principes de:**
  - **Modularité** : division des objets en modules (types)
  - **Réutilisabilité** : un même module peut être ré-utilisé à différents endroits
    - Exemple : Le type Point2D est réutilisé pour tous les types de figures

Programmation par type abstrait

- **Respecte les principes de:**
  - **Modularité** : division des objets en modules (types)
  - **Réutilisabilité** : un même module peut être ré-utilisé à différents endroits
    - Exemple : Le type Point2D est réutilisé pour tous les types de figures
  - **Encapsulation** : regroupement des données et leurs opérations
    - Regroupement de la représentation et des opérations dans chaque module

Point2D
Points dans un plan

Polygone
Polygones dans un plan
Translation : Point → Ø Rotation : Réel → Ø Aire : Ø → Réel Circonference : Ø → Réel Affichage : Ø → Ø ...

EnsemblePolygones
Ensembles de polygones dans un plan
Affichage : Ø → Ø

Rectangle
Rectangles dans un plan

Triangle
Triangles dans un plan

Losange
Losanges dans un plan

Carré
Carrés dans un plan

Programmation par type abstrait

- **Respecte les principes de:**
  - **Modularité** : division des objets en modules (types)
  - **Réutilisabilité** : un même module peut être ré-utilisé à différents endroits
    - Exemple : Le type Point2D est réutilisé pour tous les types de figures
  - **Encapsulation** : regroupement des données et leurs opérations
    - Regroupement de la représentation et des opérations dans chaque module
  - **Abstraction** : séparation entre spécification et implémentation du type, et accès protégé aux détails d'implémentation

Programmation Orientée Objet (POO)

- POO : réalise la programmation par type abstrait en utilisant l'abstraction procédurale

- Ensemble de types (classes) interagissant entre eux ➔ **modularité**

- Types caractérisés par la représentation des données et les opérations encapsulés dans la définition des types ➔ **encapsulation**

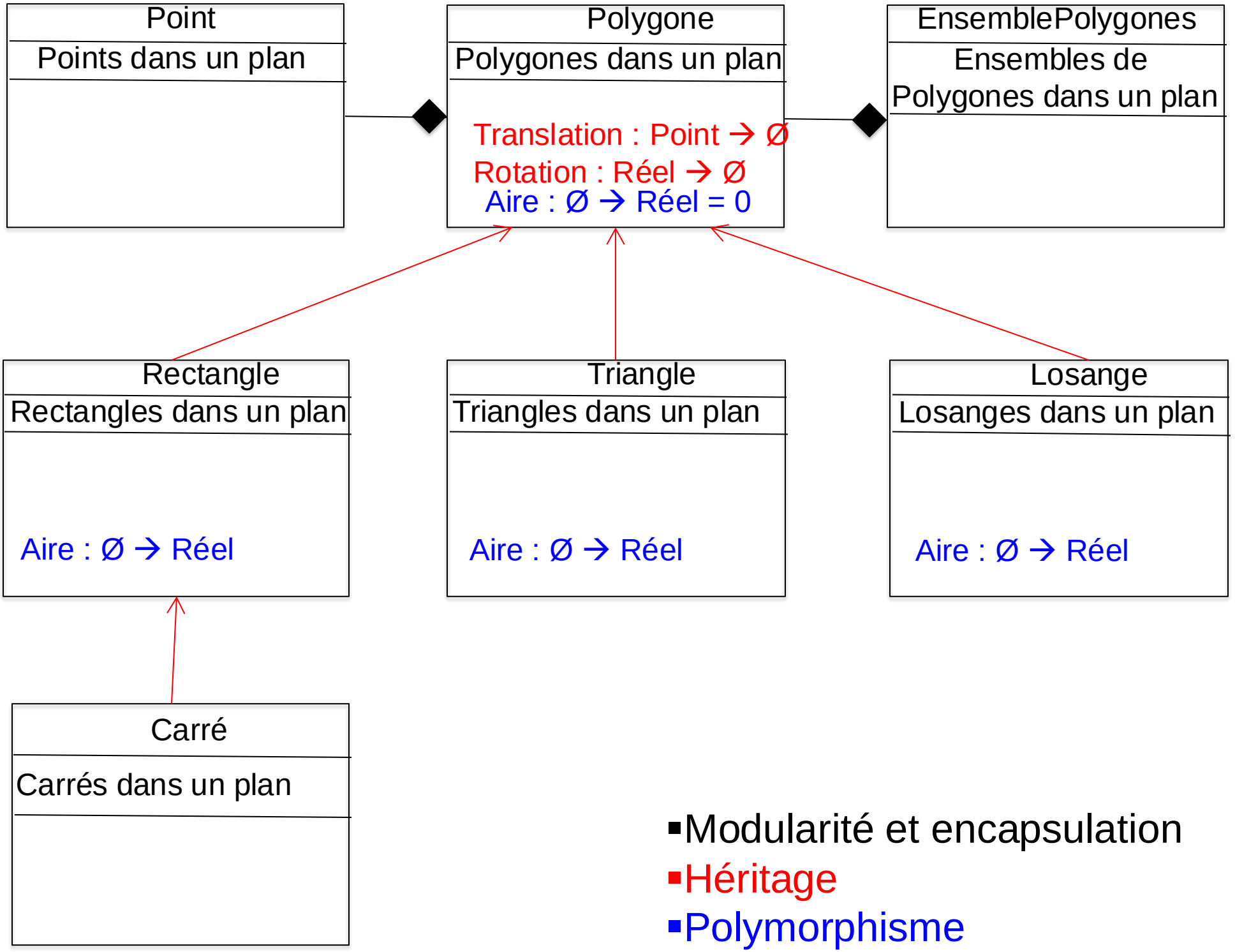
- Conception en séparant la description du comportement des types (prototypes et sémantiques des opérations) et les algorithmes des opérations ➔ **abstraction procédurale**

Programmation Orientée Objet (POO)

- **Caractéristiques de la POO :**
  - Modularité, encapsulation : facilitent la maintenance
  - Héritage : un type parent peut transmettre ses propriétés à un type enfant (réutilisabilité)
  - Polymorphisme : un même opérateur à des actions différentes en fonction du type sur lequel il agit (abstraction)

Programmation Orientée Objet (POO)

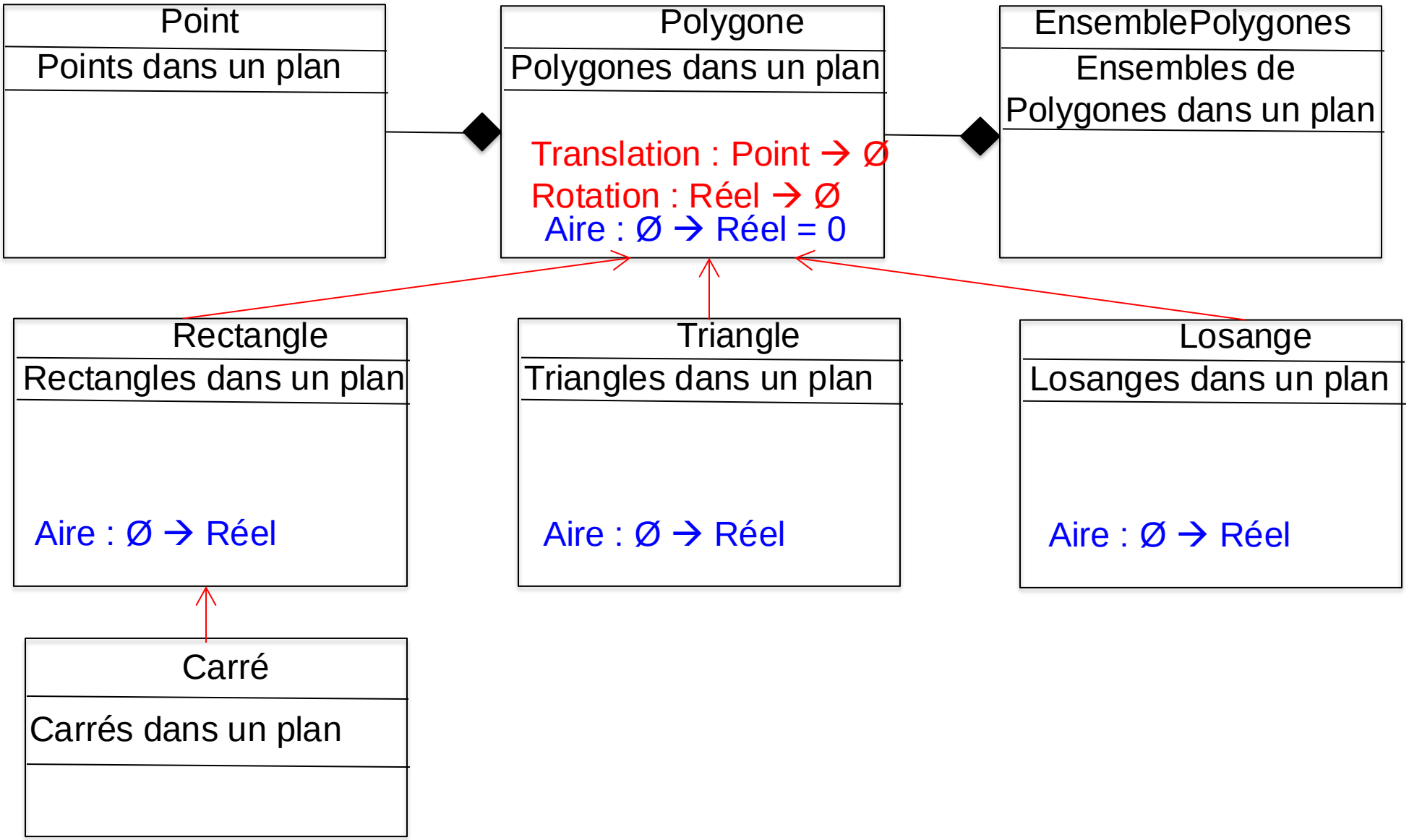
- **Caractéristiques de la POO :**
  - Exemple pour l' application permettant de définir des polygones et les manipuler
    - Modularité, encapsulation
    - Héritage
    - Polymorphisme



- Modularité et encapsulation
- Héritage
- Polymorphisme

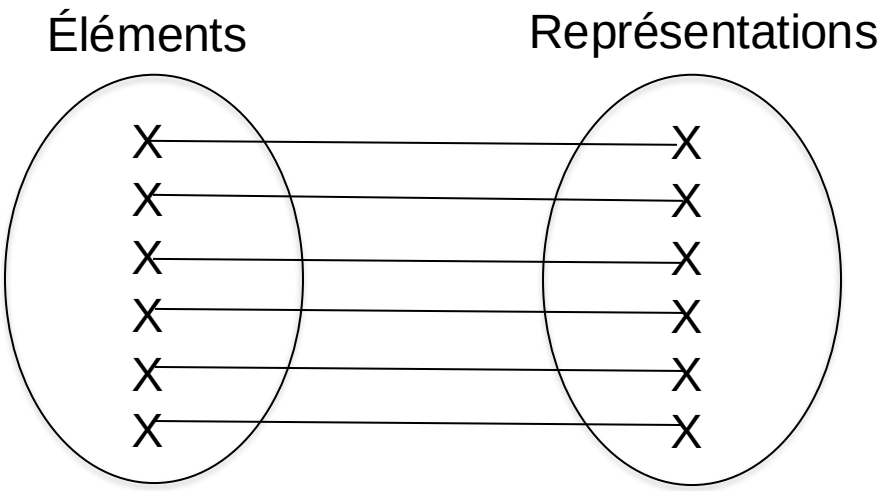
Exercice 2

Compléter la définition des TAD ci-dessous en ajoutant les prototypes des fonctions manquantes dans chaque TAD, tout en respectant les principes de la programmation par TA et de la POO.

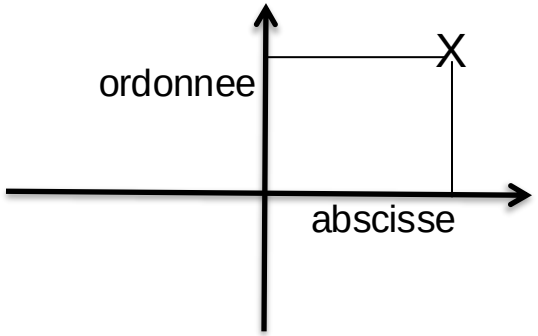


Implémentation : choix d’une bonne représentation

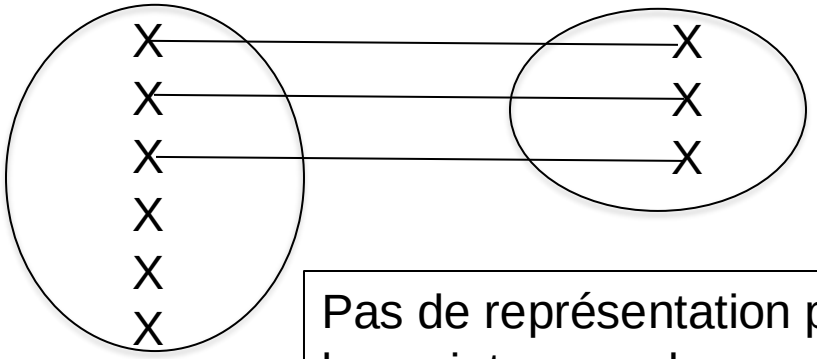
- Choix en fonction de l'utilisation(problème à traiter)
- Représentation avec laquelle les opérations utilisées seront les plus efficaces
- Bijection entre l’ensemble des éléments du type et l’ensemble des représentations



Bijection : exemple de mauvaise représentation

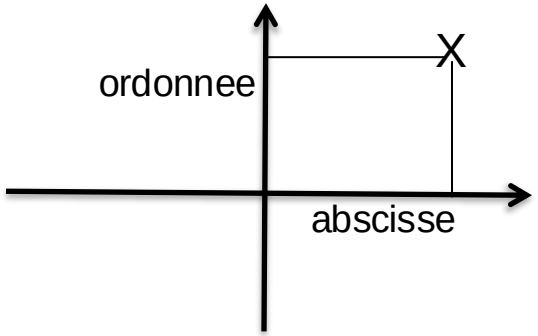


Point
Points dans un plan
abscisse (Entier) **ordonnée** (Entier) **Point** : Réel x Réel → Point **Point** : Ø → Point **==** : Point → Booléen **Init** : Réel x Réel → Ø **Distance** : Point → Réel

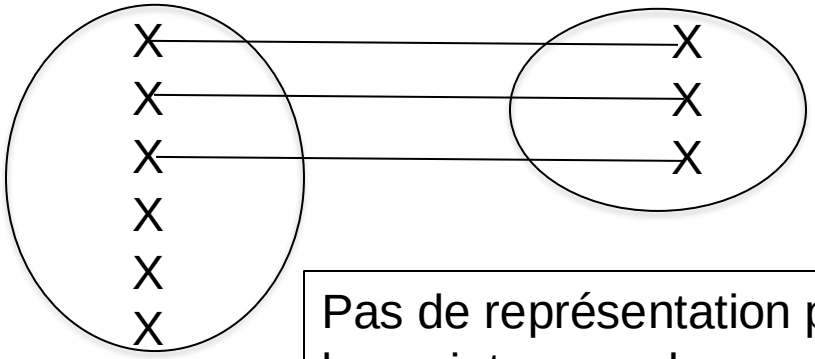


Pas de représentation pour les points avec des coordonées de type Réel. Ex: (1, 1.5)

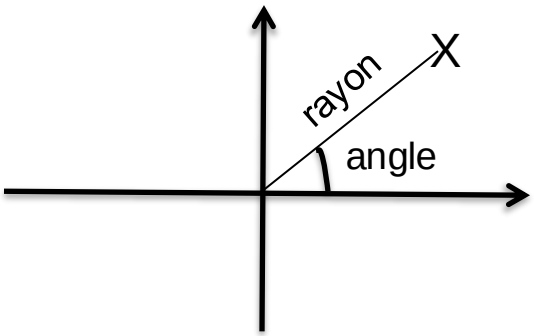
Bijection : exemple de mauvaise représentation



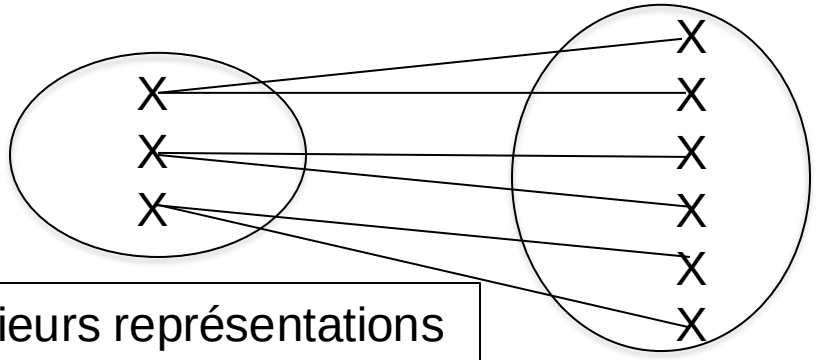
Point
Points dans un plan
abscisse (Entier) **ordonnée** (Entier) **Point** : Réel x Réel → Point **Point** : Ø → Point **==** : Point → Booléen **Init** : Réel x Réel → Ø **Distance** : Point → Réel



Pas de représentation pour les points avec des coordonées de type Réel. Ex: (1, 1.5)



Point
Points dans un plan
rayon (Réel) **angle** (Réel) **Point** : Réel x Réel → Point **Point** : Ø → Point **==** : Point → Booléen **Init** : Réel x Réel → Ø **Distance** : Point → Réel



Plusieurs représentations pour un même point
Ex: (1, π/4) et (1, 5π/4)

Exemple : les types primitifs

- **Types primitifs :** types prédéfinis dans le langage
- **Types usagers** : nouveaux types intégrés dans le langage
- Cohérence dans le traitement des objets par le compilateur (vérifie que l' objet placé dans une variable correspondent à son type déclaré)
- Objectif de la POO : intégration des types usagers de façon harmonieuse (en utilisant l'architecture et les types existants)

Types primitifs en C++

- **Scalaire**
  - **Types numériques discrets (dénombrables) :** char, short, int, long int, long long int (signé et non-signé)
  - **Types numériques flottants (indénombrables) :** float, double, long double (signé et non-signé)
    - **À virgule flottante** : représentation à 3 composantes (signe, mantisse, exposant)

    Nombre = signe x (1+mantisse) x $2^{exposant}$
    - **À virgule fixe** : représentation similaire aux entiers (partie entière et partie fractionnaire)
  - **Type booléen :** boolean (True ou False)

Types primitifs en C++

- **Composé**
  - **Conteneurs :** array (contenant des données de même type), string (chaîne de caractères), enum (type enuméré)
  - **Adresse :** pointeur (adresse d'un objet en mémoire)

Choix de représentation lié à l'architecture (existant)

- **Deux ressources**
  - **Temps :** unité centrale de traitement
  - **Mémoire (espace) :** suite d'octets
    - 1 octet = 8 bits
    - adresse d'un octet : sa position en mémoire

Exercice 3

Proposer une implémentation pour le TAD Ensemble_b ci-dessous : décrire une représentation utilisant des types primitifs, et des algorithmes pour les opérations du TAD.

Ensemble_b
Ensemble de Billes
Ensemble_b : Ø → Ensemble_b **tirer** : Ø → Bille **ajouter** : Bille → Ø **taille** : Ø → Réel

- Ensemble_b().taille() = 0
- Ensemble_b().tirer() = Erreur
- (x.tirer()__).taille() = x.taille() - 1
- (x.ajouter(b)__).taille() = x.taille() + 1
- x.ajouter(x.tirer())__ = x

Représentation des types entiers

- Représentation dans le système binaire (chaîne de 0 ou 1)

- Définitions de short, int, long dépendent de l'architecture de la machine
  - short ≤ int ≤ long
  - architecture à 16 bits : short = int = 16 ; long = 32
  - architecture à 32 bits : short = 16 ; int = long = 32

Représentation des types entiers

- Sur n bits : 2^n représentations possibles

$$b_{n-1}b_{n-2} \ldots b_1b_0$$

- Entiers non signés $[0, 2^n-1]$

Chaque nombre calculé par une somme de puissances de 2

$$\text{Nombre} = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots + b_1 \times 2^1 + b_0 \times 2^0$$

Représentation des types entiers

- Exemple sur 4 bits : $2^4 = 16$ représentations

1111	$1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 =$	15
1110	$1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 =$	14
1101	.	13
1100	.	12
1011	.	11
1010	.	10
1001	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 =$	9
1000	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 =$	8
0111	$0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 =$	7
0110	$0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 =$	6
0101	.	5
0100	.	4
0011	.	3
0010		2
0001	$0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 =$	1
0000	$0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 =$	0

Représentation des types entiers

- Sur n bits : 2^n représentations possibles

$$b_{n-1}b_{n-2} \ldots b_1b_0$$

- Entiers signés $[-2^{n-1}, 2^{n-1}-1]$
  - La moitié pour représenter les entiers positifs ou nul $[0, 2^{n-1}-1]$ (représentations telles que $b_{n-1} = 0$)
  - L'autre moitié pour représenter les entiers négatifs $[-2^{n-1}, -1]$ (représentations telles que $b_{n-1} = 1$)

Représentation des types entiers

- Exemple sur 4 bits : $2^4 = 16$ représentations

0111	$0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 =$	7
0110	$0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 =$	6
0101	.	5
0100	.	4
0011	.	3
0010		2
0001	$0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 =$	1
0000	$0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 =$	0
1111	$1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 - 2^4 =$	-1
1110	$1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 - 2^4 =$	-2
1101	.	-3
1100	.	-4
1011	.	-5
1010	.	-6
1001	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 - 2^4 =$	-7
1000	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 - 2^4 =$	-8

Représentation des types entiers

- À partir de C++11 : système de noms pour différentier les types d'entiers

[u]int[8 | 16 | 32 | 64]_t

- Exemples:
  - uint8_t : entiers non signés sur 8 bits
  - Int16_t : entiers signés sur 16 bits

- Suffixe pour préciser le type des constantes:

<val>[U][L | LL]

- Exemples:
  - 0U : constante de valeur 0 de type entiers non signés
  - 1L : constante de valeur 1 de type entiers longs signés
  - 1ULL : constante de valeur 1 de type entiers longs longs non signés

Représentation des types entiers

- Sur n bits, pour trouver la représentation binaire d'un entier x >= 0

Pour i de n-1 à 0, faire:

$b_i \leftarrow x / 2^i$ (division entière)

$x \leftarrow x - b_i * 2^i$

- Exemple sur 8 bits [-128,127], représentation de x = 46:

$b_7 \leftarrow x / 2^7 = 46 / 128 = 0$

$b_6 \leftarrow x / 2^6 = 46 / 64 = 0$

$b_5 \leftarrow x / 2^5 = 46 / 32 = 1$

$x \leftarrow x - 2^5 = 46 - 32 = 14$

$b_4 \leftarrow x / 2^4 = 14 / 16 = 0$

$b_3 \leftarrow x / 2^3 = 14 / 8 = 1$

$x \leftarrow x - 2^3 = 14 - 8 = 6$

$b_2 \leftarrow x / 2^2 = 6 / 4 = 1$

$x \leftarrow x - 2^2 = 6 - 4 = 2$

$b_1 \leftarrow x / 2^1 = 2 / 2 = 1$

$x \leftarrow x - 2^1 = 2 - 2 = 0$

$b_0 \leftarrow x / 2^0 = 0 / 1 = 0$

Représentation (46) = 00101110

Représentation des types entiers

- Sur n bits, pour trouver la représentation d'un entier négatif x

Prendre le complément à 1 du representant de –x

Additionner le représentant de 1 (somme en base binaire)

- Exemple sur 8 bits [-128,127], représentation de x = -46:

Représentation (46) = 00101110

Complément à 1 11010001

+ 00000001

Représentation (-46) = 11010010

- Pour passer d'un type entier à un type plus long, propager le bit de poids fort

- Exemple pour passer de 8 à 16 bits:

(46) 00101110 0000000000101110

(-46) 11010010 1111111111010010

Opérations sur les types entiers

Entier
Entiers compris entre -2^{n-1} et 2^{n-1}
chaine (Chaine_bin) **Entier** : Chaine_dec → Entier **+** : Entier → Entier **- unaire** : Ø → Entier **-** : Entier → Entier * : Entier → Entier ***/*** : Entier → Entier

Opérations sur les types entiers

Entier
Entiers compris entre -2^{n-1} et 2^{n-1}
chaine (Chaine_bin) **Entier** : Chaine_dec → Entier **+** : Entier → Entier **- unaire** : Ø → Entier **-** : Entier → Entier * : Entier → Entier ***/*** : Entier → Entier

Chaine_bin
Chaines binaires de longueur n
chaine (Chaine de 0 ou 1) **Chaine_bin** : Chaine_dec → Chaine_bin **complement** : Ø → Chaine_bin **decaler** : Entier → Chaine_bin **+** : Chaine_bin → Chaine_bin **- unaire** : Ø → Chaine_bin **-** : Chaine_bin → Chaine_bin * : Chaine_bin → Chaine_bin ***/*** : Chaine_bin → Chaine_bin

Opérations sur les types entiers

❑ **Chaine_bin** : Chaine_dec → Chaine_bin

Entree : $d=d_m d_{m-1} \ldots d_0$
res ← Chaine_bin(0)
$x \leftarrow \Sigma_{i=0..m}\ d_m \times 10^m$
Pour i de n-1 à 0, faire:
    $b_i \leftarrow x / 2^i$
    $x \leftarrow x - b_i * 2^i$
res.chaine = $b_{n-1} b_{n-2} \ldots b_1 b_0$
Sortie : res

Exemple sur 8 bits [-128,127]:

Entree : ''46''
$x = 4 \times 10^1 + 6 \times 10^0$
$b_7 \leftarrow x / 2^7 = 46 / 128 = 0$
$b_6 \leftarrow x / 2^6 = 46 / 64 = 0$
.
.
y.chaine : 00101110

Chaine_bin
Chaines binaires de longueur n
chaine (Chaine de 0 ou 1) **Chaine_bin** : Chaine_dec → Chaine_bin **complement** : Ø → Chaine_bin **decaler** : Entier → Chaine_bin **+** : Chaine_bin → Chaine_bin **- unaire** : Ø → Chaine_bin **-** : Chaine_bin → Chaine_bin * : Chaine_bin → Chaine_bin **/** : Chaine_bin → Chaine_bin

Opérations sur les types entiers

❑ **complement** : Ø → Chaine_bin

Entree : Ø
res ← Chaine_bin(0)
$b_{n-1}b_{n-2} \dots b_1b_0$ ← __.chaine
Pour i de n-1 à 0, faire:
    c_i ← 1- b_i
res.chaine = $c_{n-1}c_{n-2} \dots c_1c_0$
Sortie : res

Exemple sur 8 bits [-128,127]:

x.chaine : 00101110
x.complement().chaine : 11010001

Chaine_bin
Chaines binaires de longueur n
chaine (Chaine de 0 ou 1) **Chaine_bin** : Chaine_dec → Chaine_bin **complement** : Ø → Chaine_bin **decaler** : Entier → Chaine_bin **+** : Chaine_bin → Chaine_bin **- unaire** : Ø → Chaine_bin **-** : Chaine_bin → Chaine_bin * : Chaine_bin → Chaine_bin **/** : Chaine_bin → Chaine_bin

Opérations sur les types entiers

❑ **decaler** : Entier → Chaine_bin

Entree : k
res ← Chaine_bin(0)
$b_{n-1}b_{n-2} \dots b_1b_0$ ← __.chaine
Pour i de n-1+k à 0+k, faire:
    c_i ← b_i
Pour i de k-1 à 0, faire:
    c_i ← 0
res.chaine = $c_{n-1}c_{n-2} \dots c_1c_0$
Sortie : res

Exemple sur 8 bits [-128,127]:

x.chaine : 00101110
x.decaler(3).chaine : 00101110000

Chaine_bin
Chaines binaires de longueur n
chaine (Chaine de 0 ou 1) **Chaine_bin** : Chaine_dec → Chaine_bin **complement** : Ø → Chaine_bin **decaler** : Entier → Chaine_bin **+** : Chaine_bin → Chaine_bin **- unaire** : Ø → Chaine_bin **-** : Chaine_bin → Chaine_bin * : Chaine_bin → Chaine_bin */* : Chaine_bin → Chaine_bin

Opérations sur les types entiers

❑ **+** : Chaine_bin → Chaine_bin

Entree : m
res ← Chaine_bin(0)
$b_{n-1}b_{n-2} \dots b_1b_0$ ← __.chaine
$c_{n-1}c_{n-2} \dots c_1c_0$ ← m.chaine
retenue = 0
Pour i de 0 à n-1, faire:
    somme ← $b_i + c_i$ + retenue
    d_i ← somme % 2
    retenue ← 1 si somme ≥ 2, 0 sinon
res.chaine = $d_{n-1}d_{n-2} \dots d_1d_0$
Sortie: res

❑ **Exemple sur 8 bits [-128,127]:**

```
   00101110   (46)
 + 01101101   (109)
```

Chaine_bin
Chaines binaires de longueur n
chaine (Chaine de 0 ou 1) **Chaine_bin** : Chaine_dec → Chaine_bin **complement** : Ø → Chaine_bin **decaler** : Entier → Chaine_bin **+** : Chaine_bin → Chaine_bin **- unaire** : Ø → Chaine_bin **-** : Chaine_bin → Chaine_bin * : Chaine_bin → Chaine_bin */* : Chaine_bin → Chaine_bin

Opérations sur les types entiers

❑ **+** : Chaine_bin → Chaine_bin

Entree : m
res ← Chaine_bin(0)
$b_{n-1}b_{n-2} \dots b_1b_0$ ← __.chaine
$c_{n-1}c_{n-2} \dots c_1c_0$ ← m.chaine
retenue = 0
Pour i de 0 à n-1, faire:
    somme ← $b_i + c_i$ + retenue
    d_i ← somme % 2
    retenue ← 1 si somme ≥ 2, 0 sinon
res.chaine = $d_{n-1}d_{n-2} \dots d_1d_0$
Sortie: res

❑ **Exemple sur 8 bits [-128,127]:**

```
  01101100  retenue
  00101110  (46)
+ 01101101  (109)
  10011011  (-101) au lieu de (155)
```

Chaine_bin
Chaines binaires de longueur n
chaine (Chaine de 0 ou 1) **Chaine_bin** : Chaine_dec → Chaine_bin **complement** : Ø → Chaine_bin **decaler** : Entier → Chaine_bin **+** : Chaine_bin → Chaine_bin **- unaire** : Ø → Chaine_bin **-** : Chaine_bin → Chaine_bin * : Chaine_bin → Chaine_bin */* : Chaine_bin → Chaine_bin

Dépassement de domaine se traduit par un signe faux

Opérations sur les types entiers

❑ **+** : Chaine_bin → Chaine_bin

Entree : m
res ← Chaine_bin(0)
$b_{n-1}b_{n-2} \dots b_1b_0$ ← __.chaine
$c_{n-1}c_{n-2} \dots c_1c_0$ ← m.chaine
retenue = 0
Pour i de 0 à n-1, faire:
    somme ← $b_i + c_i$ + retenue
    d_i ← somme % 2
    retenue ← 1 si somme ≥ 2, 0 sinon
res.chaine = $d_{n-1}d_{n-2} \dots d_1d_0$
Sortie: res

Chaine_bin
Chaines binaires de longueur n
chaine (Chaine de 0 ou 1) **Chaine_bin** : Chaine_dec → Chaine_bin **complement** : Ø → Chaine_bin **decaler** : Entier → Chaine_bin **+** : Chaine_bin → Chaine_bin **- unaire** : Ø → Chaine_bin **-** : Chaine_bin → Chaine_bin * : Chaine_bin → Chaine_bin */* : Chaine_bin → Chaine_bin

❑**Exemple sur 8 bits [-128,127]:**

```
   01101100  retenue                    00101110
retenue
    00101110  (46)                      00101110  (46)
+   01101101  (109)             +       00100011  (35)
```

Opérations sur les types entiers

❑ **- unaire** : Ø → Chaine_bin

Entree : Ø
res ← __.complement() + Chaine_bin(1)
Sortie: res

Chaine_bin
Chaines binaires de longueur n
chaine (Chaine de 0 ou 1) **Chaine_bin** : Chaine_dec → Chaine_bin **complement** : Ø → Chaine_bin **decaler** : Entier → Chaine_bin **+** : Chaine_bin → Chaine_bin **- unaire** : Ø → Chaine_bin **-** : Chaine_bin → Chaine_bin * : Chaine_bin → Chaine_bin */* : Chaine_bin → Chaine_bin

Opérations sur les types entiers

❑ **-** : Chaine_bin → Chaine_bin

Entree : m
res ← __ + -(m)
Sortie: res

Chaine_bin
Chaines binaires de longueur n
chaine (Chaine de 0 ou 1) **Chaine_bin** : Chaine_dec → Chaine_bin **complement** : Ø → Chaine_bin **decaler** : Entier → Chaine_bin **+** : Chaine_bin → Chaine_bin **- unaire** : Ø → Chaine_bin **-** : Chaine_bin → Chaine_bin * : Chaine_bin → Chaine_bin */* : Chaine_bin → Chaine_bin

Exercice 4

❑ ***** : Chaine_bin → Chaine_bin

Entree : m

Donner un algorithme
Indice: utiliser les opérations : decaler et +

Sortie: y

Chaine_bin
Chaines binaires de longueur n
chaine (Chaine de 0 ou 1) **Chaine_bin** : Chaine_dec → Chaine_bin **complement** : Ø → Chaine_bin **decaler** : Entier → Chaine_bin **+** : Chaine_bin → Chaine_bin **- unaire** : Ø → Chaine_bin **-** : Chaine_bin → Chaine_bin * : Chaine_bin → Chaine_bin ***/*** : Chaine_bin → Chaine_bin

Exercice 5

❑ ***/*** : Chaine_bin → Chaine_bin

Entree : m

Donner un algorithme

Sortie: nouveau

Chaine_bin
Chaines binaires de longueur n
chaine (Chaine de 0 ou 1) **Chaine_bin** : Chaine_dec → Chaine_bin **complement** : Ø → Chaine_bin **decaler** : Entier → Chaine_bin **+** : Chaine_bin → Chaine_bin **- unaire** : Ø → Chaine_bin **-** : Chaine_bin → Chaine_bin * : Chaine_bin → Chaine_bin ***/*** : Chaine_bin → Chaine_bin

Représentation d'autres types primitifs

- Type booléen :

Utilise le type int

- Deux valeurs : 0 (faux) et 1 (vrai)
- Dépendamment du compilateur : booléen représenté sur 1 octet ou 1 bit

- Type caractère

Représentation sur 1 octet ($b_7b_6b_5b_4b_3b_2b_1b_0$) : 256 représentations

- 128 pour le code ASCII ($b_7 = 0$)
- 128 pour le code ASCII étendu ($b_7 = 1$)

Caractères signés ou non signés: pas de différence en termes de représentation, mais ordre modifié :

- Unsigned char : ASCII < ASCII étendu
- Signed char : ASCII étendu < ASCII