IFT339
Structures de données

Thème 1 : Types abstraits et Structures de données

Aïda Ouangraoua
Département d'informatique

UNIVERSITÉ DE
SHERBROOKE

Définition

- **Type Abstrait de Données (TAD)** : spécification d'un ensemble d'objets de même type et d'un ensemble d'opérations possibles sur ces objets.

Exemples : Entier, Point2D

TAD
Ensemble d'objets de même type
Ensemble d'opérations possibles sur ces objets.

Entier
Ensemble d'entiers
Entier : Autre type → Entier **==** : Entier → Booléen **!=** : Entier → Booléen **<** : Entier → Booléen **>** : Entier → Booléen **+** : Entier → Entier **-** : Entier → Entier **/** : Entier → Entier ***** : Entier → Entier

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

Définition

- **TAD caractérisé par**:
  - Son identité (nom, adresse)
  - Type des données (Domaine)
  - Opérations sur données
  - Propriétés des opérations

Point
Points dans un plan
Point : Réel x Réel → Point Point : Ø → Point == : Point → Booléen Init : Réel x Réel → Ø Distance : Point → Réel

- **Indépendant de:**
  - représentation utilisée pour les données et algorithmes utilisés pour les opérations
    Exemple pour Point : représentation avec coordonnées polaires (rayon, angle) ou coordonnées cartésiennes (abscisse, ordonnée) ?
  - Langage utilisée pour l'implémentation

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

Définition

- **Structure de données** : Implémentation d'un TAD dans un langage donnée (inclut la représentation des objets et peut inclure des opérations non visibles par les utilisateurs)

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

Exemples en C++

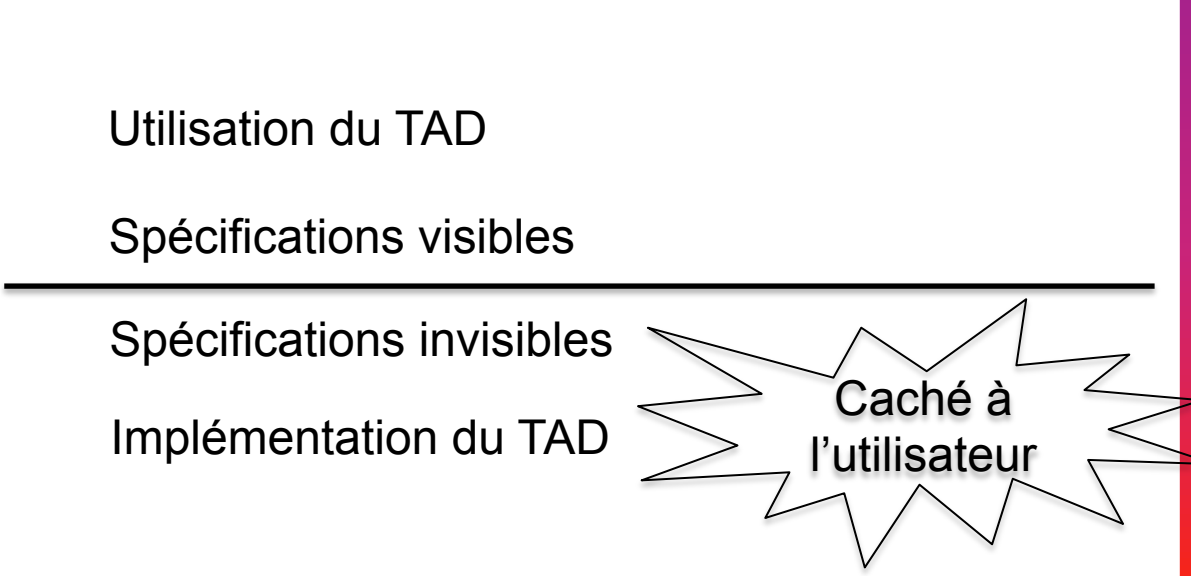
Fichier Point.h:
Définition de la classe: domaine (visible), représentation des objets (invisible), et opérations avec leurs propriétés (visibles et invisibles)

Fichier Point.cpp:
Implémentation des algorithmes des opérations

Définition

- **Deux vues pour les programmeurs**
  - **Vue Utilisateur (TAD)**
    - ne connaît que les opérations visibles du TAD
    - pas d'accès à la représentation et aux détails internes
  - **Vue Concepteur (Structure de données)**
    - Conçois et implémente le TAD
    - accès à la représentation et aux détails d'implémentation

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



Permet une conception modulaire, rigoureuse et l'intégration de nouveaux types à partir des types de base.

Programmation par type abstrait

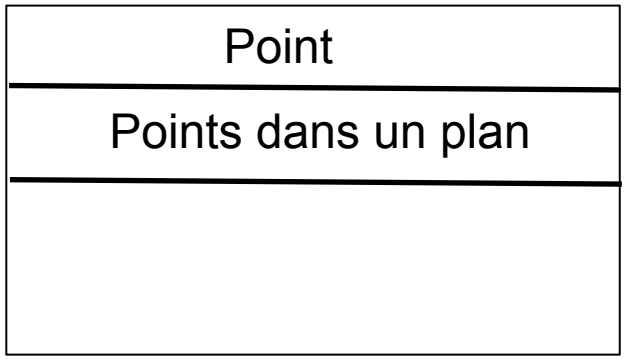
- **Respecte les principes de:**

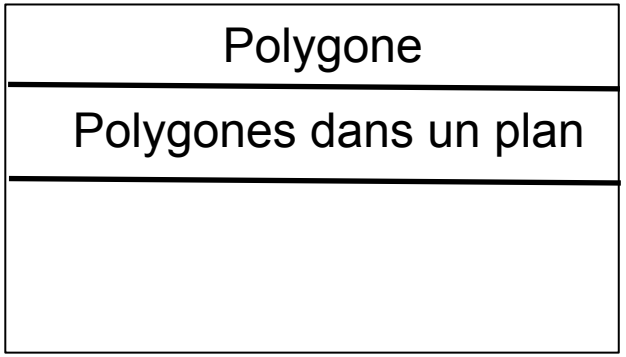
  - **Modularité** : division des tâches en module
    - Exemple pour une application permettant de définir un ensemble de polygones particuliers (rectangle, carré, triangle équilatéral, losange) dans un plan et les manipuler (translation en spécifiant un vecteur (point), rotation autour du centre du polygone 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)

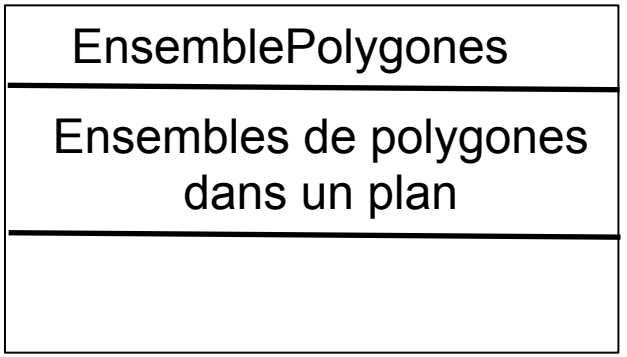
  - **Réutilisabilité** : un même module peut être ré-utilisé à différents endroits
    - Exemple : Le module Point pour les figures

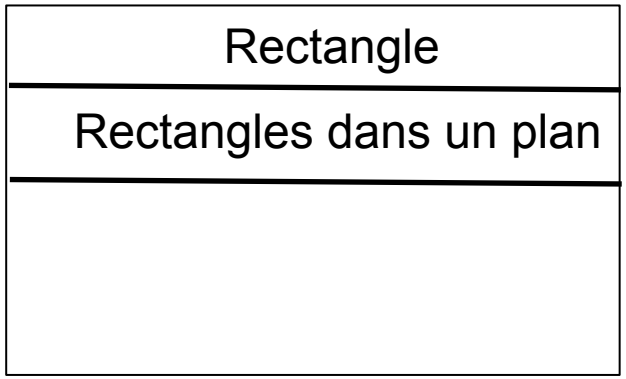
  - **Encapsulation** : regroupement des données et leurs opérations
    - Regroupement de la représentation et des opérations dans chaque module

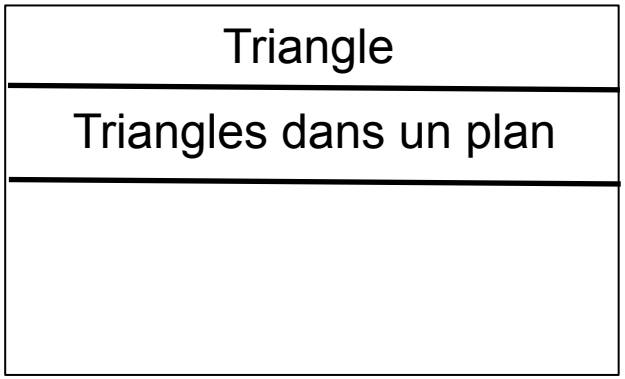
  - **Abstraction** : accès protégé aux détails internes

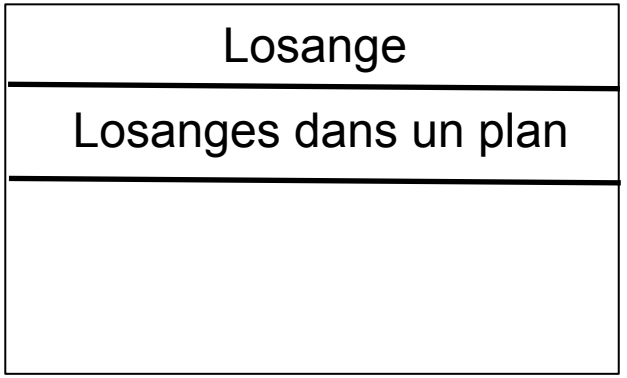

Point
Points dans un plan

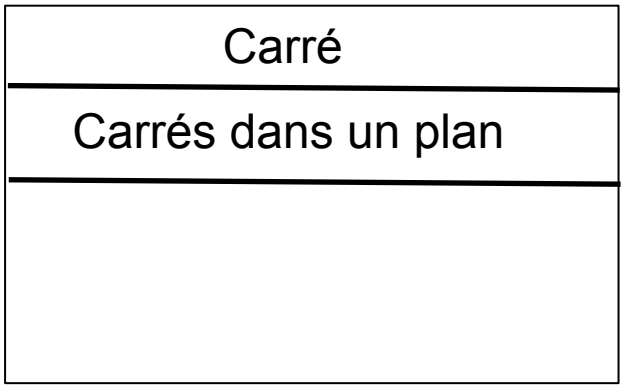

Polygone
Polygones dans un plan


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 Orientée Objet (POO)

- **Programme :** ensemble de objets (modules, types) interagissant entre eux ➔ modularité

- Objets caractérisés par la représentation des données et les opérations encapsulés dans la définition de l'objet ➔ encapsulation

- Conception en mettant l'emphase sur le comportement des objets (propriétés des opérations) et pas sur les opérations qui les manipulent (algorithmes des opérations) ➔ abstraction procédurale

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

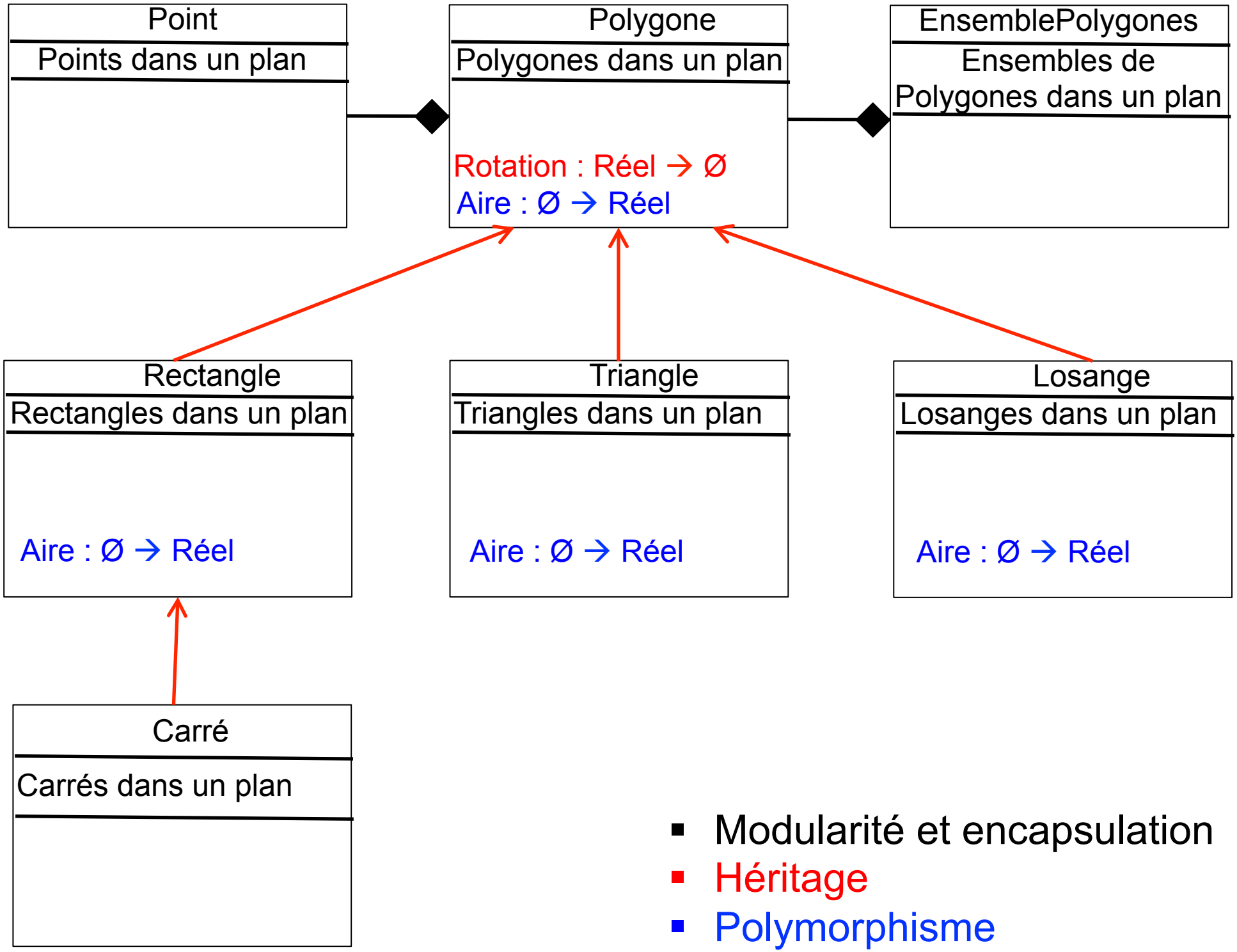
Programmation Orientée Objet (POO)

- **Programme :** ensemble de objets (modules, types) interagissant entre eux ➔ modularité

- Objets caractérisés par la représentation des données et les opérations encapsulés dans la définition de l'objet ➔ encapsulation

- Conception en mettant l'emphase sur le comportement des objets (propriétés des opérations) et pas sur les opérations qui les manipulent (algorithmes des opérations) ➔ abstraction procédurale

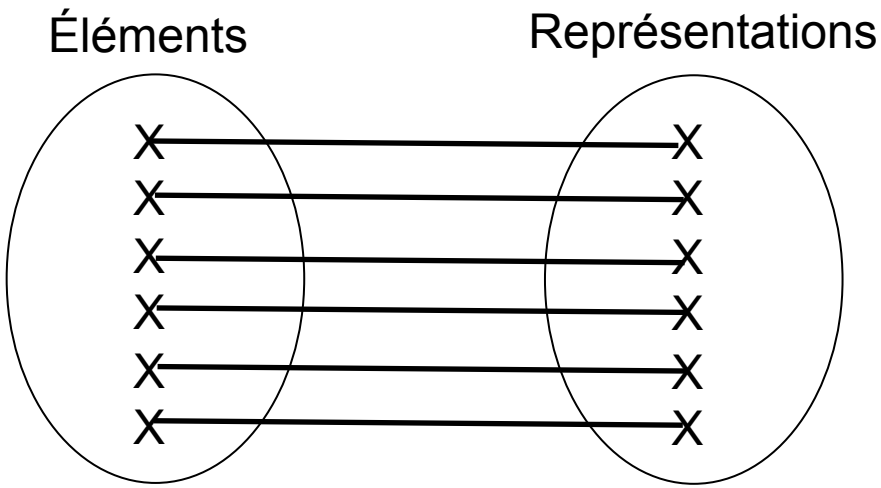
- **Caractéristiques de la POO :**
  - Exemple pour l' application permettant de définir des polygones et les manipuler
    - Modularité et encapsulation
    - Héritage
    - Polymorphisme



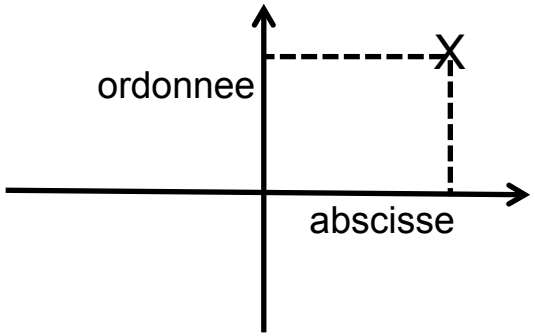
- Modularité et encapsulation
- Héritage
- Polymorphisme

Choix d'une bonne représentation

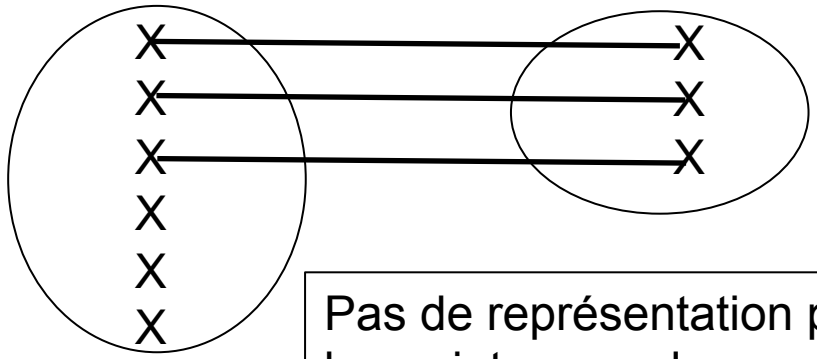
- Choix en fonction du problème à traiter et des opérations les plus utilisées
- Représentation avec laquelle les opérations seront les plus efficaces
- Bijection entre l'ensemble des éléments du type et l'ensemble des représentations



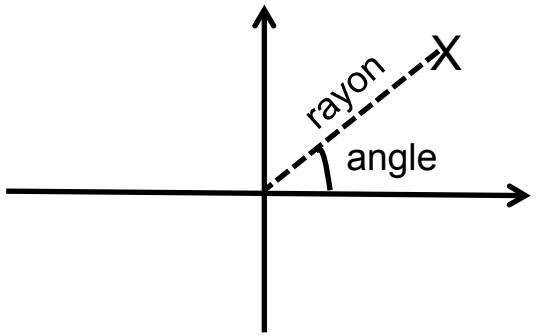
Exemple de mauvaise représentation



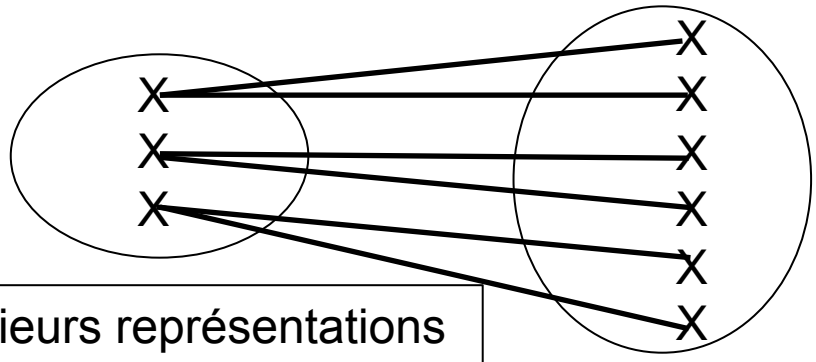
Point
Points dans un plan
abscisse (Entier) **ordonnée** (Entier) **Point** : Réel x Réel → Point **Point** : Ø → Point **==** : Point → Booléen **Init** : Réel x Réel → Ø **Distance** : Point → Réel



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 les langages
- **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 types), string (chaîne de caractères), enum (type enuméré)
  - **Adresse :** pointeur (adresse d'un objet en mémoire)

Représentation liée à l'architecture

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

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

- 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

    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 d'un entier positif ou nul x

  Pour i de n-1 à 0, faire:

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

  Si $b_i == 1$

  $x \leftarrow x - 2^i$

- 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) = 001011100

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

Entiers (sur n bits)
Entiers compris entre -2^{n-1} et 2^{n-1}
chaine_bin (Chaine_base2) **Entier** : Chaine_base10 → Entier **+** : Entier → Entier **- unaire** : Ø → Entier **-** : Entier → Entier * : Entier → Entier / : Entier → Entier

Opérations sur les types entiers

Entiers (sur n bits)
Entiers compris entre -2^{n-1} et 2^{n-1}
chaine_bin (Chaine_base2) **Entier** : Chaine_base10 → Entier **+** : Entier → Entier **- unaire** : Ø → Entier **-** : Entier → Entier * : Entier → Entier */* : Entier → Entier

Chaine_base2
Chaines binaires de longueur n
chaine (chaine de 0 ou 1) **Chaine_base2** : Chaine_base10 → Chaine_base2 **complement** : Ø → Chaine_base2 **decaler** : Entier → Chaine_base2 **+** : Chaine_base2 → Chaine_base2 **- unaire** : Ø → Chaine_base2 **-** : Chaine_base2 → Chaine_base2 * : Chaine_base2 → Chaine_base2 */* : Chaine_base2 → Chaine_base2

Opérations sur les types entiers

- **Chaine_base2** : Chaine_base10 → Chaine_base2

  Entree : $d_m d_{m-1} \ldots d_0$

  $x \leftarrow \Sigma_{i=0..m}\ d_m \times 10^m$

  Pour i de n-1 à 0, faire:

  $b_i \leftarrow x / 2^i$

  Si $b_i == 1$

  $x \leftarrow x - 2^i$

  cet_element.chaine = $b_{n-1} b_{n-2} \ldots b_1 b_0$

  Sortie : cet_element

- Exemple sur 8 bits [-128,127]:

  Entree : 46

  $x = 4 \times 10^1 + 6 \times 10^0$

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

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

  .

  .

  Cet_element.chaine : 00101110

Chaine_base2
Chaines binaires de longueur n
chaine (chaine de 0 ou 1) **Chaine_base2** : Chaine_base10 → Chaine_base2 **complement** : Ø → Chaine_base2 **decaler** : Entier → Chaine_base2 **+** : Chaine_base2 → Chaine_base2 **- unaire** : Ø → Chaine_base2 **-** : Chaine_base2 → Chaine_base2 * : Chaine_base2 → Chaine_base2 / : Chaine_base2 → Chaine_base2

Opérations sur les types entiers

- **complement** : Ø → Chaine_base2

  Entree : Ø

  nouveau ← Chaine_base2(0)

  $b_{n-1}b_{n-2} \ldots b_1b_0$ ← cet_element.chaine

  Pour i de n-1 à 0, faire:

  c_i ← 1- b_i

  nouveau.chaine = $c_{n-1}c_{n-2} \ldots c_1c_0$

  Sortie : nouveau

- Exemple sur 8 bits [-128,127]:

cet_element.chaine : 00101110

cet_element.complement().chaine : 11010001

Chaine_base2
Chaines binaires de longueur n
chaine (chaine de 0 ou 1) **Chaine_base2** : Chaine_base10 → Chaine_base2 **complement** : Ø → Chaine_base2 **decaler** : Entier → Chaine_base2 **+** : Chaine_base2 → Chaine_base2 **- unaire** : Ø → Chaine_base2 **-** : Chaine_base2 → Chaine_base2 * : Chaine_base2 → Chaine_base2 / : Chaine_base2 → Chaine_base2

Opérations sur les types entiers

- **decaler** : Entier → Chaine_base2

  Entree : k

  nouveau ← Chaine_base2(0)

  $b_{n-1}b_{n-2} \ldots b_1b_0$ ← cet_element.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

  nouveau.chaine = $c_{n-1}c_{n-2} \ldots c_1c_0$

  Sortie : nouveau

- Exemple sur 8 bits [-128,127]:

  Entree : k

cet_element.chaine : 00101110

cet_element.decaler(3).chaine : 00101110000

Chaine_base2
Chaines binaires de longueur n
chaine (chaine de 0 ou 1) **Chaine_base2** : Chaine_base10 → Chaine_base2 **complement** : Ø → Chaine_base2 **decaler** : Entier → Chaine_base2 **+** : Chaine_base2 → Chaine_base2 **- unaire** : Ø → Chaine_base2 **-** : Chaine_base2 → Chaine_base2 * : Chaine_base2 → Chaine_base2 / : Chaine_base2 → Chaine_base2

Opérations sur les types entiers

- **+** : Chaine_base2 → Chaine_base2

  Entree : autre

  nouveau ← Chaine_base2(0)

  $b_{n-1}b_{n-2} \ldots b_1b_0$ ← cet_element.chaine

  $c_{n-1}c_{n-2} \ldots c_1c_0$ ← autre.chaine

  retenue = 0

  Pour i de 0 à n-1, faire:

  somme ← $d_i + b_i$ + retenue

  d_i ← somme % 2

  retenue ← 1 si somme ≥ 2, 0 sinon

  nouveau.chaine = $d_{n-1}d_{n-2} \ldots d_1d_0$

  Sortie: nouveau

- Exemple sur 8 bits [-128,127]:

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

Chaine_base2
Chaines binaires de longueur n
chaine (chaine de 0 ou 1) **Chaine_base2** : Chaine_base10 → Chaine_base2 **complement** : Ø → Chaine_base2 **decaler** : Entier → Chaine_base2 **+** : Chaine_base2 → Chaine_base2 **- unaire** : Ø → Chaine_base2 **-** : Chaine_base2 → Chaine_base2 * : Chaine_base2 → Chaine_base2 / : Chaine_base2 → Chaine_base2

Opérations sur les types entiers

- **+** : Chaine_base2 → Chaine_base2

  Entree : autre

  nouveau ← Chaine_base2(0)

  $b_{n-1}b_{n-2} \ldots b_1b_0$ ← cet_element.chaine

  $c_{n-1}c_{n-2} \ldots c_1c_0$ ← autre.chaine

  retenue = 0

  Pour i de 0 à n-1, faire:

  somme ← $d_i + b_i$ + retenue

  d_i ← somme % 2

  retenue ← 1 si somme ≥ 2, 0 sinon

  nouveau.chaine = $d_{n-1}d_{n-2} \ldots d_1d_0$

  Sortie: nouveau

- Exemple sur 8 bits [-128,127]:

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

Chaine_base2
Chaines binaires de longueur n
chaine (chaine de 0 ou 1) **Chaine_base2** : Chaine_base10 → Chaine_base2 **complement** : Ø → Chaine_base2 **decaler** : Entier → Chaine_base2 **+** : Chaine_base2 → Chaine_base2 **- unaire** : Ø → Chaine_base2 **-** : Chaine_base2 → Chaine_base2 * : Chaine_base2 → Chaine_base2 / : Chaine_base2 → Chaine_base2

Dépassement de domaine se traduit par un signe faux

Opérations sur les types entiers

- **+** : Chaine_base2 → Chaine_base2

  Entree : autre

  nouveau ← Chaine_base2(0)

  $b_{n-1}b_{n-2} \ldots b_1b_0$ ← cet_element.chaine

  $c_{n-1}c_{n-2} \ldots c_1c_0$ ← autre.chaine

  retenue = 0

  Pour i de 0 à n-1, faire:

  somme ← $d_i + b_i$ + retenue

  d_i ← somme % 2

  retenue ← 1 si somme ≥ 2, 0 sinon

  nouveau.chaine = $d_{n-1}d_{n-2} \ldots d_1d_0$

  Sortie: nouveau

Chaine_base2
Chaines binaires de longueur n
chaine (chaine de 0 ou 1) **Chaine_base2** : Chaine_base10 → Chaine_base2 **complement** : Ø → Chaine_base2 **decaler** : Entier → Chaine_base2 **+** : Chaine_base2 → Chaine_base2 **- unaire** : Ø → Chaine_base2 **-** : Chaine_base2 → Chaine_base2 * : Chaine_base2 → Chaine_base2 */* : Chaine_base2 → Chaine_base2

- Exemple sur 8 bits [-128,127]:

```
  01101100   retenue                    00101110   retenue
  00101110   (46)                       00101110   (46)
+ 01101101   (109)                    +  00100011   (35)
----------                            ------------
  10011011   (-101) au lieu de (155)    01010001   (81)
```

Opérations sur les types entiers

- **- unaire** : Ø → Chaine_base2

  Entree : Ø

  nouveau ← cet_element.complement()
  +
  Chaine_base2(1)

  Sortie: nouveau

Chaine_base2
Chaines binaires de longueur n
chaine (chaine de 0 ou 1) **Chaine_base2** : Chaine_base10 → Chaine_base2 **complement** : Ø → Chaine_base2 **decaler** : Entier → Chaine_base2 **+** : Chaine_base2 → Chaine_base2 **- unaire** : Ø → Chaine_base2 **-** : Chaine_base2 → Chaine_base2 * : Chaine_base2 → Chaine_base2 */* : Chaine_base2 → Chaine_base2

Opérations sur les types entiers

- **-** : Chaine_base2 → Chaine_base2

  Entree : autre

  nouveau ← cet_element

  \+

  -(autre)

  Sortie: nouveau

Chaine_base2
Chaines binaires de longueur n
chaine (chaine de 0 ou 1) **Chaine_base2** : Chaine_base10 → Chaine_base2 **complement** : Ø → Chaine_base2 **decaler** : Entier → Chaine_base2 **+** : Chaine_base2 → Chaine_base2 **- unaire** : Ø → Chaine_base2 **-** : Chaine_base2 → Chaine_base2 * : Chaine_base2 → Chaine_base2 */* : Chaine_base2 → Chaine_base2

Opérations sur les types entiers

- * : Chaine_base2 → Chaine_base2

  Entree : autre

  Donner un algorithme

  Indice: utiliser les opérations : decaler et +

  Sortie: nouveau

Chaine_base2
Chaines binaires de longueur n
chaine (chaine de 0 ou 1) **Chaine_base2** : Chaine_base10 → Chaine_base2 **complement** : Ø → Chaine_base2 **decaler** : Entier → Chaine_base2 **+** : Chaine_base2 → Chaine_base2 **- unaire** : Ø → Chaine_base2 **-** : Chaine_base2 → Chaine_base2 * : Chaine_base2 → Chaine_base2 */* : Chaine_base2 → Chaine_base2

Opérations sur les types entiers

- **/** : Chaine_base2 → Chaine_base2

  Entree : autre

  Donner un algorithme

  Sortie: nouveau

Chaine_base2
Chaines binaires de longueur n
chaine (chaine de 0 ou 1) **Chaine_base2** : Chaine_base10 → Chaine_base2 **complement** : Ø → Chaine_base2 **decaler** : Entier → Chaine_base2 **+** : Chaine_base2 → Chaine_base2 **- unaire** : Ø → Chaine_base2 **-** : Chaine_base2 → Chaine_base2 * : Chaine_base2 → Chaine_base2 **/** : Chaine_base2 → Chaine_base2

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 terme de représentation, mais ordre modifié :

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