Ch. 4 - Types structurés

Langage C / C++

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

6 octobre 2021

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Table des matières

Introduction

Les différents types

- En C / C++, « tout » a un type
  - Objets, références, fonctions et expressions
- Il existe deux grandes catégories de types
  1. Types de base
     Valeurs entières, arithmétiques, booléennes
  2. Types structurés
     Références, pointeurs, tableaux, fonctions, énumérations et classes
- On n'a pas abordé les classes (et « variantes ») et énumérations
- Énumération : stocke un ensemble fini de valeurs constantes

Les différents types

- En C / C++, « tout » a un type
  - Objets, références, fonctions et expressions
- Il existe deux grandes catégories de types
  1. Types de base
     Valeurs entières, arithmétiques, booléennes
  2. Types structurés
     Références, pointeurs, tableaux, fonctions, énumérations et classes
- On n'a pas abordé les classes (et « variantes ») et énumérations
- Énumération : stocke un ensemble fini de valeurs constantes

Les différents types

- En C / C++, « tout » a un type
  - Objets, références, fonctions et expressions
- Il existe deux grandes catégories de types
  1. Types de base
     - `void`, `nullptr_t`, arithmétiques, `bool`
  2. Types structurés
     - Références, pointeurs, tableaux, fonctions, énumérations et classes
- On n'a pas abordé les classes (et « variantes ») et énumérations
- Énumération : stocke un ensemble fini de valeurs constantes

Les différents types

- En C / C++, « tout » a un type
  - Objets, références, fonctions et expressions
- Il existe deux grandes catégories de types
  1. Types de base
     - `void`, `nullptr_t`, arithmétiques, `bool`
  2. Types structurés
     - Références, pointeurs, tableaux, fonctions, énumérations et classes
- On n'a pas abordé les classes (et « variantes ») et énumérations
- Énumération : stocke un ensemble fini de valeurs constantes

Les différents types

- En C / C++, « tout » a un type
  - Objets, références, fonctions et expressions
- Il existe deux grandes catégories de types
  1. Types de base
     - `void`, `nullptr_t`, arithmétiques, `bool`
  2. Types structurés
     - Références, pointeurs, tableaux, fonctions, énumérations et classes
- On n'a pas abordé les classes (et « variantes ») et énumérations
- Énumération : stocke un ensemble fini de valeurs constantes

Les différents types

- En C / C++, « tout » a un type
  - Objets, références, fonctions et expressions
- Il existe deux grandes catégories de types
  1. Types de base
     - `void`, `nullptr_t`, arithmétiques, `bool`
  2. Types structurés
     - Références, pointeurs, tableaux, fonctions, énumérations et classes
- On n'a pas abordé les classes (et « variantes ») et énumérations
- Énumération : stocke un ensemble fini de valeurs constantes

Les différents types

- En C / C++, « tout » a un type
  - Objets, références, fonctions et expressions
- Il existe deux grandes catégories de types
  1. Types de base
     - `void`, `nullptr_t`, arithmétiques, `bool`
  2. Types structurés
     - Références, pointeurs, tableaux, fonctions, énumérations et classes
- On n'a pas abordé les classes (et « variantes ») et énumérations
- Énumération : stocke un ensemble fini de valeurs constantes

Les différents types

- En C / C++, « tout » a un type
  - Objets, références, fonctions et expressions
- Il existe deux grandes catégories de types
  1. Types de base
     - `void`, `nullptr_t`, arithmétiques, `bool`
  2. Types structurés
     - Références, pointeurs, tableaux, fonctions, énumérations et classes
- On n'a pas abordé les classes (et « variantes ») et énumérations
- Énumération : stocke un ensemble fini de valeurs constantes

Les différents types

- En C / C++, « tout » a un type
  - Objets, références, fonctions et expressions
- Il existe deux grandes catégories de types
  1. Types de base
     - `void`, `nullptr_t`, arithmétiques, `bool`
  2. Types structurés
     - Références, pointeurs, tableaux, fonctions, énumérations et classes
- On n'a pas abordé les classes (et « variantes ») et énumérations
- Énumération : stocke un ensemble fini de valeurs constantes

Paradigme orienté objet (1/2)

- Modélisation des composants d'un programme sous forme d'objets
- Les objets sont instanciés à partir d'un modèle conceptuel : la classe
- Les objets ont tous des caractéristiques communes

Exemple

- Tous les animaux ont des pattes
- Tous les mammifères ont 4 pattes et des mamelles
- Les humains sont des mammifères avec 2 mamelles et un nom

- Un objet est donc une instance d'une classe aux caractéristiques particulières
  - `abs` est un humain (nom : `"Romain"`)

Paradigme orienté objet (1/2)

- Modélisation des composants d'un programme sous forme d'objets
- Les objets sont instanciés à partir d'un modèle conceptuel : la classe
- Les objets ont tous des caractéristiques communes

Exemple

- Tous les animaux ont des pattes
- Tous les mammifères ont 4 pattes et des mamelles
- Les humains sont des mammifères avec 2 mamelles et un nom

- Un objet est donc une instance d'une classe aux caractéristiques particulières
  - `abs` est un humain (nom : `"Romain"`)

Paradigme orienté objet (1/2)

- Modélisation des composants d'un programme sous forme d'objets
- Les objets sont instanciés à partir d'un modèle conceptuel : la classe
- Les objets ont tous des caractéristiques communes

Exemple

- Tous les animaux ont des pattes
- Tous les mammifères ont 4 pattes et des mamelles
- Les humains sont des mammifères avec 2 mamelles et un nom

- Un objet est donc une instance d'une classe aux caractéristiques particulières
  - `abs` est un humain (nom : `"Romain"`)

Paradigme orienté objet (1/2)

- Modélisation des composants d'un programme sous forme d'objets
- Les objets sont instanciés à partir d'un modèle conceptuel : la classe
- Les objets ont tous des caractéristiques communes

Exemple

- Tous les animaux ont des pattes
- Tous les mammifères ont 4 pattes et des mamelles
- Les humains sont des mammifères avec 2 mamelles et un nom

- Un objet est donc une instance d'une classe aux caractéristiques particulières
  - `abs` est un humain (nom : `"Romain"`)

Paradigme orienté objet (1/2)

- Modélisation des composants d'un programme sous forme d'objets
- Les objets sont instanciés à partir d'un modèle conceptuel : la classe
- Les objets ont tous des caractéristiques communes

Exemple

- Tous les animaux ont des pattes
- Tous les mammifères ont 4 pattes et des mamelles
- Les humains sont des mammifères avec 2 mamelles et un nom

- Un objet est donc une instance d'une classe aux caractéristiques particulières
  - `abs` est un humain (nom : `"Romain"`)

Paradigme orienté objet (1/2)

- Modélisation des composants d'un programme sous forme d'objets
- Les objets sont instanciés à partir d'un modèle conceptuel : la classe
- Les objets ont tous des caractéristiques communes

Exemple

- Tous les animaux ont des pattes
- Tous les mammifères ont 4 pattes et des mamelles
- Les humains sont des mammifères avec 2 mamelles et un nom

- Un objet est donc une instance d'une classe aux caractéristiques particulières
  - `abs` est un humain (nom : `"Romain"`)

Paradigme orienté objet (1/2)

- Modélisation des composants d'un programme sous forme d'objets
- Les objets sont instanciés à partir d'un modèle conceptuel : la classe
- Les objets ont tous des caractéristiques communes

Exemple

- Tous les animaux ont des pattes
- Tous les mammifères ont 4 pattes et des mamelles
- Les humains sont des mammifères avec 2 mamelles et un nom

- Un objet est donc une instance d'une classe aux caractéristiques particulières
  - `abs` est un humain (nom : `"Romain"`)

Paradigme orienté objet (1/2)

- Modélisation des composants d'un programme sous forme d'objets
- Les objets sont instanciés à partir d'un modèle conceptuel : la classe
- Les objets ont tous des caractéristiques communes

Exemple

- Tous les animaux ont des pattes
- Tous les mammifères ont 4 pattes et des mamelles
- Les humains sont des mammifères avec 2 mamelles et un nom

- Un objet est donc une instance d'une classe aux caractéristiques particulières
  - `abs` est un humain (nom : `"Romain"`)

Paradigme orienté objet (1/2)

- Modélisation des composants d'un programme sous forme d'objets
- Les objets sont instanciés à partir d'un modèle conceptuel : la classe
- Les objets ont tous des caractéristiques communes

Exemple

- Tous les animaux ont des pattes
- Tous les mammifères ont 4 pattes et des mamelles
- Les humains sont des mammifères avec 2 mamelles et un nom

- Un objet est donc une instance d'une classe aux caractéristiques particulières
  - `abs` est un humain (nom : `"Romain"`)

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type A par un objet de type B

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type A par un objet de type B

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type A par un objet de type B

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type A par un objet de type B

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type A par un objet de type B

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type A par un objet de type B

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type A par un objet de type B

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type A par un objet de type B

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type A par un objet de type B

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type A par un objet de type B

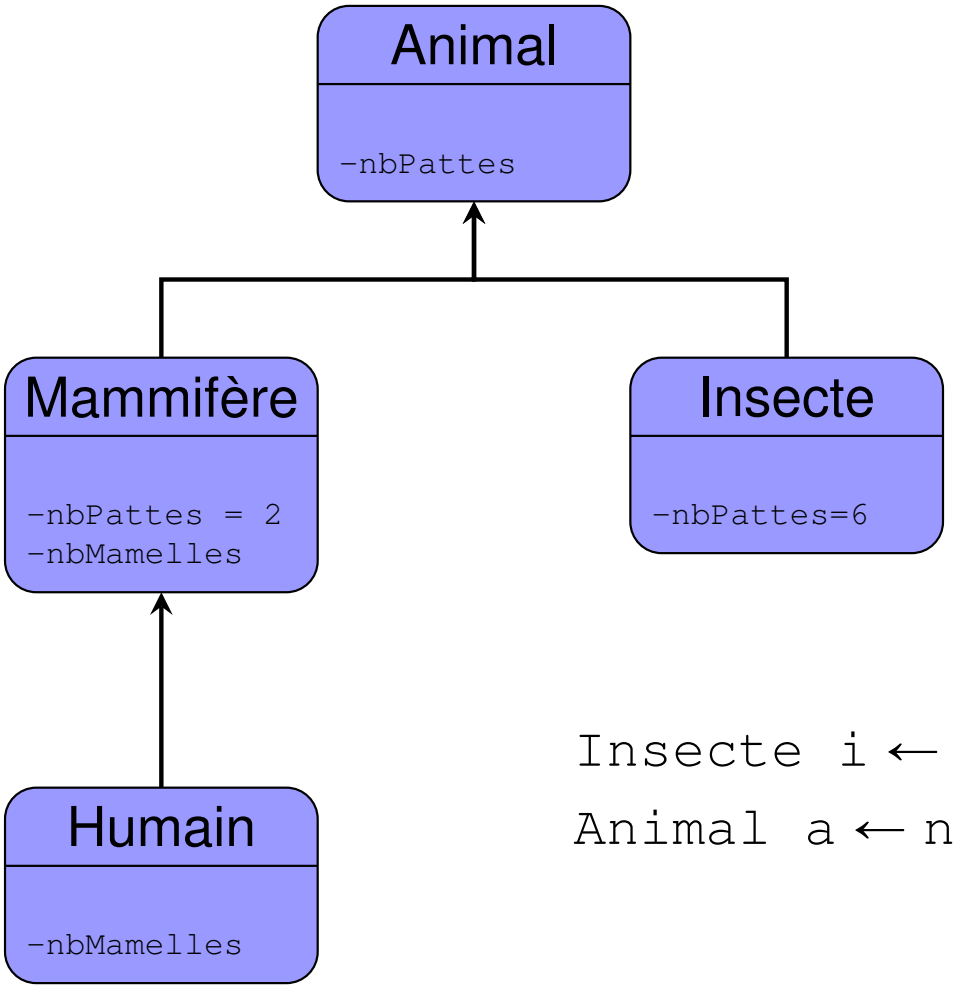
Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type `A` par un objet de type `B`

Paradigme orienté objet (2/2)

- Chaque objet possède des données qui lui sont propres
  - Attributs
- Chaque objet peut réaliser plusieurs fonctionnalités, dépendant de ses attributs
  - Fonctions membres
- Les attributs ne sont pas « visibles » en dehors de la classe
  - Encapsulation
- Un objet peut posséder plusieurs types « hiérarchiques »
  - Héritage
- Conversion implicite entre types « hiérarchiques » compatibles
  - Polymorphisme
- Chaque objet est typé
  - En l'absence des relations ci-dessus, impossible de substituer un objet de type `A` par un objet de type `B`

Exemple




```
Insecte i ← nouvel Insecte
Animal a ← nouvel Insecte
```

Types structurés en C

Limitations

- Pas de classes, pas d'héritage
  - Mais on a quand même une manière de structurer le code
- Les `struct` et `union` permettent de déclarer des « paquets » de données
- On peut définir des fonctions *indépendantes* prenant en paramètres des `struct` ou `union`
  - Pas de fonctions membres
  - On ne peut pas écrire `maFraction.add(f2)`
- On peut effectuer une certaine généricité avec la conversion implicite des pointeurs vers `void*`
- Définir une `struct` ou `union` ne définit pas d'alias de type associé
  - Il faut explicitement faire un `typedef`

Limitations

- Pas de classes, pas d'héritage
  - Mais on a quand même une manière de structurer le code
- Les `struct` et `union` permettent de déclarer des « paquets » de données
- On peut définir des fonctions *indépendantes* prenant en paramètres des `struct` ou `union`
  - Pas de fonctions membres
  - On ne peut pas écrire `maFraction.add(f2)`
- On peut effectuer une certaine généricité avec la conversion implicite des pointeurs vers `void*`
- Définir une `struct` ou `union` ne définit pas d'alias de type associé
  - Il faut explicitement faire un `typedef`

Limitations

- Pas de classes, pas d'héritage
  - Mais on a quand même une manière de structurer le code
- Les `struct` et `union` permettent de déclarer des « paquets » de données
- On peut définir des fonctions *indépendantes* prenant en paramètres des `struct` ou `union`
  - Pas de fonctions membres
  - On ne peut pas écrire `maFraction.add(f2)`
- On peut effectuer une certaine généricité avec la conversion implicite des pointeurs vers `void*`
- Définir une `struct` ou `union` ne définit pas d'alias de type associé
  - Il faut explicitement faire un `typedef`

Limitations

- Pas de classes, pas d'héritage
  - Mais on a quand même une manière de structurer le code
- Les `struct` et `union` permettent de déclarer des « paquets » de données
- On peut définir des fonctions *indépendantes* prenant en paramètres des `struct` ou `union`
  - Pas de fonctions membres
  - On ne peut pas écrire `maFraction.add(f2)`
- On peut effectuer une certaine généricité avec la conversion implicite des pointeurs vers `void*`
- Définir une `struct` ou `union` ne définit pas d'alias de type associé
  - Il faut explicitement faire un `typedef`

Limitations

- Pas de classes, pas d'héritage
  - Mais on a quand même une manière de structurer le code
- Les `struct` et `union` permettent de déclarer des « paquets » de données
- On peut définir des fonctions *indépendantes* prenant en paramètres des `struct` ou `union`
  - Pas de fonctions membres
  - On ne peut pas écrire `maFraction.add(f2)`
- On peut effectuer une certaine généricité avec la conversion implicite des pointeurs vers `void*`
- Définir une `struct` ou `union` ne définit pas d'alias de type associé
  - Il faut explicitement faire un `typedef`

Limitations

- Pas de classes, pas d'héritage
  - Mais on a quand même une manière de structurer le code
- Les `struct` et `union` permettent de déclarer des « paquets » de données
- On peut définir des fonctions *indépendantes* prenant en paramètres des `struct` ou `union`
  - Pas de fonctions membres
  - On ne peut pas écrire `maFraction.add(f2)`
- On peut effectuer une certaine généricité avec la conversion implicite des pointeurs vers `void*`
- Définir une `struct` ou `union` ne définit pas d'alias de type associé
  - Il faut explicitement faire un `typedef`

Limitations

- Pas de classes, pas d'héritage
  - Mais on a quand même une manière de structurer le code
- Les `struct` et `union` permettent de déclarer des « paquets » de données
- On peut définir des fonctions *indépendantes* prenant en paramètres des `struct` ou `union`
  - Pas de fonctions membres
  - On ne peut pas écrire `maFraction.add(f2)`
- On peut effectuer une certaine généricité avec la conversion implicite des pointeurs vers `void*`
- Définir une `struct` ou `union` ne définit pas d'alias de type associé
  - Il faut explicitement faire un `typedef`

Limitations

- Pas de classes, pas d'héritage
  - Mais on a quand même une manière de structurer le code
- Les `struct` et `union` permettent de déclarer des « paquets » de données
- On peut définir des fonctions *indépendantes* prenant en paramètres des `struct` ou `union`
  - Pas de fonctions membres
  - On ne peut pas écrire `maFraction.add(f2)`
- On peut effectuer une certaine généricité avec la conversion implicite des pointeurs vers `void*`
- Définir une `struct` ou `union` ne définit pas d'alias de type associé
  - Il faut explicitement faire un `typedef`

Limitations

- Pas de classes, pas d'héritage
  - Mais on a quand même une manière de structurer le code
- Les `struct` et `union` permettent de déclarer des « paquets » de données
- On peut définir des fonctions *indépendantes* prenant en paramètres des `struct` ou `union`
  - Pas de fonctions membres
  - On ne peut pas écrire `maFraction.add(f2)`
- On peut effectuer une certaine généricité avec la conversion implicite des pointeurs vers `void*`
- Définir une `struct` ou `union` ne définit pas d'alias de type associé
  - Il faut explicitement faire un `typedef`

Illustration

- Fichier `typedef.c`

```c
struct A {};
typedef struct A A; //comment

union B {};
typedef union B B; //comment

//void print_addr1(struct A* a)
void print_addr1(A * a)
{
    printf("%p\n", a);
}

//void print_addr2(union B* b)
void print_addr2(B * b)
{
    printf("%p\n", b);
}

int main()
{
    A a; B b;
    //struct A a; union B b;
    print_addr1(&a);
    print_addr2(&b);
}
```

Table des matières

Structure

- Structure de donnée de champs non contigus.
  - Pas de possibilité de parcours mémoire.
- Déclaration via le mot-clé `struct`.
- Pas d'initialisation de champs à la déclaration d'une variable.
  - Pas de constructeur : valeurs indéterminées.
- Transmission par valeur : pas de constructeur de recopie.
  - Attention aux pointeurs !
- Affectation possible entre structures de même type uniquement.
  - Même si les deux structures ont les mêmes champs, pas de cast possible.

Structure

- Structure de donnée de champs non contigus.
  - Pas de possibilité de parcours mémoire.
- Déclaration via le mot-clé `struct`.
- Pas d'initialisation de champs à la déclaration d'une variable.
  - Pas de constructeur : valeurs indéterminées.
- Transmission par valeur : pas de constructeur de recopie.
  - Attention aux pointeurs !
- Affectation possible entre structures de même type uniquement.
  - Même si les deux structures ont les mêmes champs, pas de cast possible.

Structure

- Structure de donnée de champs non contigus.
  - Pas de possibilité de parcours mémoire.
- Déclaration via le mot-clé `struct`.
- Pas d'initialisation de champs à la déclaration d'une variable.
  - Pas de constructeur : valeurs indéterminées.
- Transmission par valeur : pas de constructeur de recopie.
  - Attention aux pointeurs !
- Affectation possible entre structures de même type uniquement.
  - Même si les deux structures ont les mêmes champs, pas de cast possible.

Structure

- Structure de donnée de champs non contigus.
  - Pas de possibilité de parcours mémoire.
- Déclaration via le mot-clé `struct`.
- Pas d'initialisation de champs à la déclaration d'une variable.
  - Pas de constructeur : valeurs indéterminées.
- Transmission par valeur : pas de constructeur de recopie.
  - Attention aux pointeurs !
- Affectation possible entre structures de même type uniquement.
  - Même si les deux structures ont les mêmes champs, pas de cast possible.

Structure

- Structure de donnée de champs non contigus.
  - Pas de possibilité de parcours mémoire.
- Déclaration via le mot-clé `struct`.
- Pas d'initialisation de champs à la déclaration d'une variable.
  - Pas de constructeur : valeurs indéterminées.
- Transmission par valeur : pas de constructeur de recopie.
  - Attention aux pointeurs !
- Affectation possible entre structures de même type uniquement.
  - Même si les deux structures ont les mêmes champs, pas de cast possible.

Structure

- Structure de donnée de champs non contigus.
  - Pas de possibilité de parcours mémoire.
- Déclaration via le mot-clé `struct`.
- Pas d'initialisation de champs à la déclaration d'une variable.
  - Pas de constructeur : valeurs indéterminées.
- Transmission par valeur : pas de constructeur de recopie.
  - Attention aux pointeurs !
- Affectation possible entre structures de même type uniquement.
  - Même si les deux structures ont les mêmes champs, pas de cast possible.

Structure

- Structure de donnée de champs non contigus.
  - Pas de possibilité de parcours mémoire.
- Déclaration via le mot-clé `struct`.
- Pas d'initialisation de champs à la déclaration d'une variable.
  - Pas de constructeur : valeurs indéterminées.
- Transmission par valeur : pas de constructeur de recopie.
  - Attention aux pointeurs !
- Affectation possible entre structures de même type uniquement.
  - Même si les deux structures ont les mêmes champs, pas de cast possible.

Structure

- Structure de donnée de champs non contigus.
  - Pas de possibilité de parcours mémoire.
- Déclaration via le mot-clé `struct`.
- Pas d'initialisation de champs à la déclaration d'une variable.
  - Pas de constructeur : valeurs indéterminées.
- Transmission par valeur : pas de constructeur de recopie.
  - Attention aux pointeurs !
- Affectation possible entre structures de même type uniquement.
  - Même si les deux structures ont les mêmes champs, pas de cast possible.

Structure

- Structure de donnée de champs non contigus.
  - Pas de possibilité de parcours mémoire.
- Déclaration via le mot-clé `struct`.
- Pas d'initialisation de champs à la déclaration d'une variable.
  - Pas de constructeur : valeurs indéterminées.
- Transmission par valeur : pas de constructeur de recopie.
  - Attention aux pointeurs !
- Affectation possible entre structures de même type uniquement.
  - Même si les deux structures ont les mêmes champs, pas de cast possible.

Initialisation

- Pas de constructeur

Initialisation implicite

- Les valeurs des attributs dépendent de la classe d'allocation (cf. Ch. 5)
- `point p;`

Initialisation explicite

- Affectation avec `= { ... }`
- Les attributs manquants sont mis à zéro

- On peut assigner des valeurs par affectation des attributs
  - `p.x = 2; p.y = 3;`

Initialisation

- Pas de constructeur

Initialisation implicite

- Les valeurs des attributs dépendent de la classe d'allocation (cf. Ch. 5)
- `point p;`

Initialisation explicite

- Affectation avec `= { ... }`
- Les attributs manquants sont mis à zéro

- On peut assigner des valeurs par affectation des attributs
  - `p.x = 2; p.y = 3;`

Initialisation

- Pas de constructeur

Initialisation implicite

- Les valeurs des attributs dépendent de la classe d'allocation (cf. Ch. 5)
  - `point p;`

Initialisation explicite

- Affectation avec `= { ... }`
- Les attributs manquants sont mis à zéro

- On peut assigner des valeurs par affectation des attributs
  - `p.x = 2; p.y = 3;`

Initialisation

- Pas de constructeur

Initialisation implicite

- Les valeurs des attributs dépendent de la classe d'allocation (cf. Ch. 5)
- `point p;`

Initialisation explicite

- Affectation avec `= { ... }`
- Les attributs manquants sont mis à zéro

- On peut assigner des valeurs par affectation des attributs
  - `p.x = 2; p.y = 3;`

Initialisation

- Pas de constructeur

Initialisation implicite

- Les valeurs des attributs dépendent de la classe d'allocation (cf. Ch. 5)
- `point p;`

Initialisation explicite

- Affectation avec = { ... }
- Les attributs manquants sont mis à zéro

- On peut assigner des valeurs par affectation des attributs
  - `p.x = 2; p.y = 3;`

Initialisation

- Pas de constructeur

Initialisation implicite

- Les valeurs des attributs dépendent de la classe d'allocation (cf. Ch. 5)
- `point p;`

Initialisation explicite

- Affectation avec = `{ ... }`
- Les attributs manquants sont mis à zéro

- On peut assigner des valeurs par affectation des attributs
  - `p.x = 2; p.y = 3;`

Initialisation

- Pas de constructeur

Initialisation implicite

- Les valeurs des attributs dépendent de la classe d'allocation (cf. Ch. 5)
- `point p;`

Initialisation explicite

- Affectation avec = `{ ... }`
- Les attributs manquants sont mis à zéro

- On peut assigner des valeurs par affectation des attributs
  - `p.x = 2; p.y = 3;`

Initialisation

- Pas de constructeur

Initialisation implicite

- Les valeurs des attributs dépendent de la classe d'allocation (cf. Ch. 5)
- `point p;`

Initialisation explicite

- Affectation avec = `{ ... }`
- Les attributs manquants sont mis à zéro

- On peut assigner des valeurs par affectation des attributs
  - `p.x = 2; p.y = 3;`

Initialisation

- Pas de constructeur

Initialisation implicite

- Les valeurs des attributs dépendent de la classe d'allocation (cf. Ch. 5)
- `point p;`

Initialisation explicite

- Affectation avec `= { ... }`
- Les attributs manquants sont mis à zéro

- On peut assigner des valeurs par affectation des attributs
  - `p.x = 2; p.y = 3;`

Exemple

- Fichier `point.c`

```
1  struct point
2  {
3    double x, y;
4  };
5
6  struct point2
7  {
8    double x, y;
9  };
10
11 typedef struct point point;
12 typedef struct point2 point2;
13
14 double dist(point p1, point p2)
15 {
16   return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
17 }
```

Exemple

- Fichier `point.c`

```
1  int main()
2  {
3    point p;
4
5    printf("%f %f\n", p.x, p.y);//undefined
6
7    p.x = p.y = 0;
8
9    point p2 = {1, 1}; //try with = {1}
10   printf("%f %f\n", p2.x, p2.y);
11
12   printf("%f\n", dist(p, p2));
13
14   point2 brol;
15   //point p3 = (point)brol;
16   //dist(p1, brol); //no conversion
17 }
```

Table des matières

Union

- Déclaration et utilisation similaire aux structures
- Différence majeure : la mémoire est partagée entre *tous* les champs
- Déclaration via le mot-clé `union`.
- Idée : si une union possède un champ X et un champ Y, elle stocke *soit* un X, *soit* un Y.
- La taille d'une union est d'au moins la taille du plus grand champ.
- Si on affecte une valeur à un champ, la zone de mémoire partagée est modifiée

Hygiène de programmation

- Éviter

Union

- Déclaration et utilisation similaire aux structures
- Différence majeure : la mémoire est partagée entre *tous* les champs
- Déclaration via le mot-clé `union`.
- Idée : si une union possède un champ X et un champ Y, elle stocke *soit* un X, *soit* un Y.
- La taille d'une union est d'au moins la taille du plus grand champ.
- Si on affecte une valeur à un champ, la zone de mémoire partagée est modifiée

Hygiène de programmation

- Éviter

Union

- Déclaration et utilisation similaire aux structures
- Différence majeure : la mémoire est partagée entre *tous* les champs
- Déclaration via le mot-clé `union`.
- Idée : si une union possède un champ X et un champ Y, elle stocke *soit* un X, *soit* un Y.
- La taille d'une union est d'au moins la taille du plus grand champ.
- Si on affecte une valeur à un champ, la zone de mémoire partagée est modifiée

Hygiène de programmation

- Éviter

Union

- Déclaration et utilisation similaire aux structures
- Différence majeure : la mémoire est partagée entre *tous* les champs
- Déclaration via le mot-clé `union`.
- Idée : si une union possède un champ X et un champ Y, elle stocke *soit* un X, *soit* un Y.
- La taille d'une union est d'au moins la taille du plus grand champ.
- Si on affecte une valeur à un champ, la zone de mémoire partagée est modifiée

Hygiène de programmation

- Éviter

Union

- Déclaration et utilisation similaire aux structures
- Différence majeure : la mémoire est partagée entre *tous* les champs
- Déclaration via le mot-clé `union`.
- Idée : si une union possède un champ X et un champ Y, elle stocke *soit* un X, *soit* un Y.
- La taille d'une union est d'au moins la taille du plus grand champ.
- Si on affecte une valeur à un champ, la zone de mémoire partagée est modifiée

Hygiène de programmation

- Éviter

Union

- Déclaration et utilisation similaire aux structures
- Différence majeure : la mémoire est partagée entre *tous* les champs
- Déclaration via le mot-clé `union`.
- Idée : si une union possède un champ X et un champ Y, elle stocke *soit* un X, *soit* un Y.
- La taille d'une union est d'au moins la taille du plus grand champ.
- Si on affecte une valeur à un champ, la zone de mémoire partagée est modifiée

Hygiène de programmation

- Éviter

Union

- Déclaration et utilisation similaire aux structures
- Différence majeure : la mémoire est partagée entre *tous* les champs
- Déclaration via le mot-clé `union`.
- Idée : si une union possède un champ X et un champ Y, elle stocke *soit* un X, *soit* un Y.
- La taille d'une union est d'au moins la taille du plus grand champ.
- Si on affecte une valeur à un champ, la zone de mémoire partagée est modifiée

Hygiène de programmation

- Éviter

Union

- Déclaration et utilisation similaire aux structures
- Différence majeure : la mémoire est partagée entre *tous* les champs
- Déclaration via le mot-clé `union`.
- Idée : si une union possède un champ X et un champ Y, elle stocke *soit* un X, *soit* un Y.
- La taille d'une union est d'au moins la taille du plus grand champ.
- Si on affecte une valeur à un champ, la zone de mémoire partagée est modifiée

Hygiène de programmation

- Éviter

Exemple (1/2)

- Fichier `union.c`

```
1  union Data
2  {
3      int i;
4      float f;
5      char str[20];
6  };
7
8  int main( )
9  {
10     union Data data;
11
12     data.i = 10;
13     printf( "data.i : %d\n", data.i);
14     printf( "data.f : %f\n", data.f);
15     printf( "data.str : %s\n", data.str);
16
17     data.f = 220.5;
18     printf( "data.i : %d\n", data.i);
19     printf( "data.f : %f\n", data.f);
20     printf( "data.str : %s\n", data.str);
21
22     strcpy( data.str, "C Programming");
23     printf( "data.i : %d\n", data.i);
24     printf( "data.f : %f\n", data.f);
25     printf( "data.str : %s\n", data.str);
26 }
```

Table des matières

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe des structures
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe des structures
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe des structures
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe des structures
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe des structures
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe des structures
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe des structures
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe des structures
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe des structures
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe des structures
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Champs de bits

- Le C possède des opérateurs permettant de travailler sur les motifs binaires (C++aussi).
  - `|`, `&`, «, »
- Le langage permet de définir, au sein des structures, des variables occupant un nombre défini de bits.
  - Les champs de bits
- Seul cas d'un type de taille non multiple d'un byte
- Utiles pour :
  - Compacter l'information : $0 \leqslant i \leqslant 15$ tient sur 4 bits au lieu de 16
  - Parcourir un motif binaire en mémoire
- Syntaxe des structures
  - `T t : s;` : le champ `t` de type `T` occupe `s` bits.
  - `T : s;` : On saute `s` bits.

Types autorisés

- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Types autorisés

- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Types autorisés

- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Types autorisés

- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Types autorisés

- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Types autorisés

- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Types autorisés

- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Types autorisés

- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Types autorisés

- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Types autorisés

- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Types autorisés

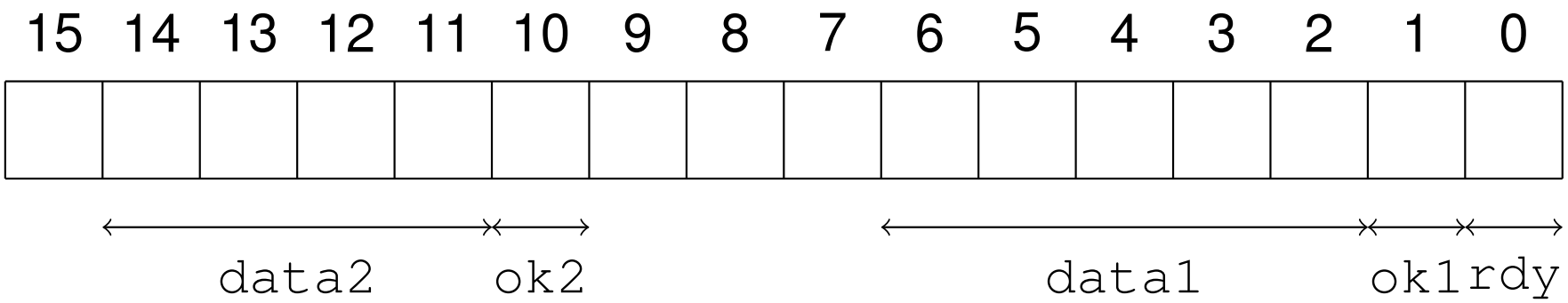
- Trois uniques types d'attributs autorisés
  1. Avec `unsigned int`, `unsigned int b:3;` $\in [0, 7]$
  2. Avec `signed int`, `signed int b:3;` $\in [-4, 3]$
  3. Avec `int`, `int b:3;` $\in [0, 7]$ ou $\in [-4, 3]$
  4. Avec `bool`, `int b:1;` $\in [0, 1]$ (`true` et `false`)
- Unique cas où `int` $\neq$ `signed int`
- On ne peut pas
  - créer de pointeurs de champs de bits
  - utiliser `sizeof` avec les champs de bits

Hygiène de programmation

- Éviter

Exemple

```
1  struct state
2  {
3    unsigned rdy : 1;
4    unsigned ok1 : 1;
5    int data1    : 5;
6    int          : 3;
7    unsigned ok2 : 1;
8    int data2    : 4;
9  };
```



Énumérations

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - Les énumérateurs ne sont pas des `lvalue`
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - Les énumérateurs ne sont pas des `lvalue`
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - Les énumérateurs ne sont pas des `lvalue`
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - Les énumérateurs ne sont pas des `lvalue`
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - Les énumérateurs ne sont pas des `lvalue`
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - Les énumérateurs ne sont pas des `lvalue`
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - Les énumérateurs ne sont pas des `lvalue`
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - Les énumérateurs ne sont pas des `lvalue`
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - Les énumérateurs ne sont pas des `lvalue`
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - Les énumérateurs ne sont pas des `lvalue`
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Énumération

- Définit un type initialisable sur un nombre fini non vide de valeurs
  - Les énumérateurs ne sont pas des `lvalue`
  - On ne peut pas les réaffecter
- Les énumérateurs sont définis globalement
- Associe une valeur `int` à chacuns des énumérateurs
- Possibilité de spécifier explicitement ces valeurs.
  - Possibilité d'affecter une même valeur à plusieurs énumérateurs.
- Possibilité d'affecter une valeur entière hors des valeurs possibles.
  - Résultat dépendant du compilateur.
- Possibilité de tests avec `switch`
- Maintenez un code *lisible*

Exemple

- Fichier `enum.c`

```c
enum couleur {  rouge, vert, bleu   };
enum boolean {vrai = 1, faux = 0};

main()
{
  enum couleur c1 = rouge;
  enum couleur c2 = c1;

  printf("%i\n", c1);
  printf("%i\n", c2);

  int n = bleu;
  int p = vert * n + 2;

  printf("%i\n",n);
  printf("%i\n",p);
  //vert = n;

  enum couleur c3 = faux;
  enum boolean brol = 3 * c2 + 4;

  printf("%i\n",c3);
  printf("%i\n",brol);
}
```

Exemple

- Fichier `enum2.c`

```
1  typedef enum color { RED, GREEN, BLUE} color;
2
3  enum Foo { A, B, C=10, D, E=1, F, G=F+C}; //don't try this at home
4  //A=0, B=1, C=10, D=11, E=1, F=2, G=12
5
6  int main()
7  {
8      color c = RED;
9
10     switch(c)
11     {
12         case RED   : printf("red\n"); break;
13         case GREEN : printf("green\n"); break;
14         case BLUE  : printf("blue\n"); break;
15     }
16 }
```

En C++

- Possibilité de définir des énumérations fortement typées
- Déclaration avec `enum class NAME ... ;`
- Possibilité de choisir le type sous-jacent des énumérateurs avec `enum class NAME : TYPE ... ;`
  - Doit posséder une conversion implicite vers `int`
- Dans tous les cas, pas de conversion implicites vers les entiers
  - On peut obtenir la valeur de l'énumérateur avec `static_cast`
- Accès aux énumérateurs avec l'opérateur de résolution de portée
- Possibilité de surcharge d'opérateur (cf. Ch. 8)

Hygiène de programmation

- Éviter les énumérations non fortement typées en C++

En C++

- Possibilité de définir des énumérations fortement typées
- Déclaration avec `enum class NAME ... ;`
- Possibilité de choisir le type sous-jacent des énumérateurs avec `enum class NAME : TYPE ... ;`
  - Doit posséder une conversion implicite vers `int`
- Dans tous les cas, pas de conversion implicites vers les entiers
  - On peut obtenir la valeur de l'énumérateur avec `static_cast`
- Accès aux énumérateurs avec l'opérateur de résolution de portée
- Possibilité de surcharge d'opérateur (cf. Ch. 8)

Hygiène de programmation

- Éviter les énumérations non fortement typées en C++

En C++

- Possibilité de définir des énumérations fortement typées
- Déclaration avec `enum class NAME ... ;`
- Possibilité de choisir le type sous-jacent des énumérateurs avec `enum class NAME : TYPE ... ;`
  - Doit posséder une conversion implicite vers `int`
- Dans tous les cas, pas de conversion implicites vers les entiers
  - On peut obtenir la valeur de l'énumérateur avec `static_cast`
- Accès aux énumérateurs avec l'opérateur de résolution de portée
- Possibilité de surcharge d'opérateur (cf. Ch. 8)

Hygiène de programmation

- Éviter les énumérations non fortement typées en C++

En C++

- Possibilité de définir des énumérations fortement typées
- Déclaration avec `enum class NAME ... ;`
- Possibilité de choisir le type sous-jacent des énumérateurs avec `enum class NAME : TYPE ... ;`
  - Doit posséder une conversion implicite vers `int`
- Dans tous les cas, pas de conversion implicites vers les entiers
  - On peut obtenir la valeur de l'énumérateur avec `static_cast`
- Accès aux énumérateurs avec l'opérateur de résolution de portée
- Possibilité de surcharge d'opérateur (cf. Ch. 8)

Hygiène de programmation

- Éviter les énumérations non fortement typées en C++

En C++

- Possibilité de définir des énumérations fortement typées
- Déclaration avec `enum class NAME ... ;`
- Possibilité de choisir le type sous-jacent des énumérateurs avec `enum class NAME : TYPE ... ;`
  - Doit posséder une conversion implicite vers `int`
- Dans tous les cas, pas de conversion implicites vers les entiers
  - On peut obtenir la valeur de l'énumérateur avec `static_cast`
- Accès aux énumérateurs avec l'opérateur de résolution de portée
- Possibilité de surcharge d'opérateur (cf. Ch. 8)

Hygiène de programmation

- Éviter les énumérations non fortement typées en C++

En C++

- Possibilité de définir des énumérations fortement typées
- Déclaration avec `enum class NAME ... ;`
- Possibilité de choisir le type sous-jacent des énumérateurs avec `enum class NAME : TYPE ... ;`
  - Doit posséder une conversion implicite vers `int`
- Dans tous les cas, pas de conversion implicites vers les entiers
  - On peut obtenir la valeur de l'énumérateur avec `static_cast`
- Accès aux énumérateurs avec l'opérateur de résolution de portée
- Possibilité de surcharge d'opérateur (cf. Ch. 8)

Hygiène de programmation

- Éviter les énumérations non fortement typées en C++

En C++

- Possibilité de définir des énumérations fortement typées
- Déclaration avec `enum class NAME ... ;`
- Possibilité de choisir le type sous-jacent des énumérateurs avec `enum class NAME : TYPE ... ;`
  - Doit posséder une conversion implicite vers `int`
- Dans tous les cas, pas de conversion implicites vers les entiers
  - On peut obtenir la valeur de l'énumérateur avec `static_cast`
- Accès aux énumérateurs avec l'opérateur de résolution de portée
- Possibilité de surcharge d'opérateur (cf. Ch. 8)

Hygiène de programmation

- Éviter les énumérations non fortement typées en C++

En C++

- Possibilité de définir des énumérations fortement typées
- Déclaration avec `enum class NAME ... ;`
- Possibilité de choisir le type sous-jacent des énumérateurs avec `enum class NAME : TYPE ... ;`
  - Doit posséder une conversion implicite vers `int`
- Dans tous les cas, pas de conversion implicites vers les entiers
  - On peut obtenir la valeur de l'énumérateur avec `static_cast`
- Accès aux énumérateurs avec l'opérateur de résolution de portée
- Possibilité de surcharge d'opérateur (cf. Ch. 8)

Hygiène de programmation

- Éviter les énumérations non fortement typées en C++

En C++

- Possibilité de définir des énumérations fortement typées
- Déclaration avec `enum class NAME ... ;`
- Possibilité de choisir le type sous-jacent des énumérateurs avec `enum class NAME : TYPE ... ;`
  - Doit posséder une conversion implicite vers `int`
- Dans tous les cas, pas de conversion implicites vers les entiers
  - On peut obtenir la valeur de l'énumérateur avec `static_cast`
- Accès aux énumérateurs avec l'opérateur de résolution de portée
- Possibilité de surcharge d'opérateur (cf. Ch. 8)

Hygiène de programmation

- Éviter les énumérations non fortement typées en C++

En C++

- Possibilité de définir des énumérations fortement typées
- Déclaration avec `enum class NAME ... ;`
- Possibilité de choisir le type sous-jacent des énumérateurs avec `enum class NAME : TYPE ... ;`
  - Doit posséder une conversion implicite vers `int`
- Dans tous les cas, pas de conversion implicites vers les entiers
  - On peut obtenir la valeur de l'énumérateur avec `static_cast`
- Accès aux énumérateurs avec l'opérateur de résolution de portée
- Possibilité de surcharge d'opérateur (cf. Ch. 8)

Hygiène de programmation

- Éviter les énumérations non fortement typées en C++

Exemple

- Fichier `enum.cpp`

```
1  enum class Color { red, green = 20, blue };
2
3  enum class altitude : char
4  {
5       high='h',
6       low='l'
7  };
8
9  int main()
10 {
11     Color r = Color::blue;
12     switch(r)
13     {
14         case Color::red   : cout << "red" << endl;   break;
15         case Color::green: cout << "green" << endl; break;
16         case Color::blue  : cout << "blue" << endl;  break;
17     }
18
19     // int n = r;
20     int n = static_cast<int>(r); // OK, n = 21
21 }
```

Classes

C++ : les classes

- Les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++, on peut définir des fonctions membres dans des classes
  - Méthodes en Java
- Accès aux membres / attributs avec `.`
  - Avec `->` via un pointeur (p. ex., `this`)

Différences entre classes et structures

- En l'absence de spécificateur d'accès,

C++ : les classes

- Les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++, on peut définir des fonctions membres dans des classes
  - Méthodes en Java
- Accès aux membres / attributs avec `.`
  - Avec `->` via un pointeur (p. ex., `this`)

Différences entre classes et structures

- En l'absence de spécificateur d'accès,

C++ : les classes

- Les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++, on peut définir des fonctions membres dans des classes
  - Méthodes en Java
- Accès aux membres / attributs avec `.`
  - Avec `->` via un pointeur (p. ex., `this`)

Différences entre classes et structures

- En l'absence de spécificateur d'accès,

C++ : les classes

- Les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++, on peut définir des fonctions membres dans des classes
  - Méthodes en Java
- Accès aux membres / attributs avec `.`
  - Avec `->` via un pointeur (p. ex., `this`)

Différences entre classes et structures

- En l'absence de spécificateur d'accès,

C++ : les classes

- Les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++, on peut définir des fonctions membres dans des classes
  - Méthodes en Java
- Accès aux membres / attributs avec `.`
  - Avec `->` via un pointeur (p. ex., `this`)

Différences entre classes et structures

- En l'absence de spécificateur d'accès,

C++ : les classes

- Les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++, on peut définir des fonctions membres dans des classes
  - Méthodes en Java
- Accès aux membres / attributs avec `.`
  - Avec `->` via un pointeur (p. ex., `this`)

Différences entre classes et structures

- En l'absence de spécificateur d'accès,

C++ : les classes

- Les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++, on peut définir des fonctions membres dans des classes
  - Méthodes en Java
- Accès aux membres / attributs avec `.`
  - Avec `->` via un pointeur (p. ex., `this`)

Différences entre classes et structures

- En l'absence de spécificateur d'accès,
  1. les membres d'une `class` sont privés, ceux d'une `struct` sont publics
  2. les membres d'une mère au sein d'une fille sont privés dans une `class`, publics dans une `struct`

C++ : les classes

- Les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++, on peut définir des fonctions membres dans des classes
  - Méthodes en Java
- Accès aux membres / attributs avec `.`
  - Avec `->` via un pointeur (p. ex., `this`)

Différences entre classes et structures

- En l'absence de spécificateur d'accès,
  1. les membres d'une `class` sont privés, ceux d'une `struct` sont publics
  2. les membres d'une mère au sein d'une fille sont privés dans une `class`, publics dans une `struct`

C++ : les classes

- Les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++, on peut définir des fonctions membres dans des classes
  - Méthodes en Java
- Accès aux membres / attributs avec `.`
  - Avec `->` via un pointeur (p. ex., `this`)

Différences entre classes et structures

- En l'absence de spécificateur d'accès,
  1. les membres d'une `class` sont privés, ceux d'une `struct` sont publics
  2. les membres d'une mère au sein d'une fille sont privés dans une `class`, publics dans une `struct`

C++ : les classes

- Les classes permettent de définir un ensemble variables de différents types regroupées sous un même nom
  - Mot-clé `struct` et `class`
- En C++, on peut définir des fonctions membres dans des classes
  - Méthodes en Java
- Accès aux membres / attributs avec `.`
  - Avec `->` via un pointeur (p. ex., `this`)

Différences entre classes et structures

- En l'absence de spécificateur d'accès,
  1. les membres d'une `class` sont privés, ceux d'une `struct` sont publics
  2. les membres d'une mère au sein d'une fille sont privés dans une `class`, publics dans une `struct`

Exemple classe

- Fichier `point-class.cpp`

```
1  class point
2  {
3    double x, y;
4
5    public:
6      point(int x, int y)
7      {
8        this->x = x;
9        this->y = y;
10     }
11
12     inline double getX()
13     {
14       return x;
15     }
16
17     inline double getY()
18     {
19       return y;
20     }
21
22     double dist(point p)
23     {
24       return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
25     }
26 };
```

Exemple classe

- Fichier `point-struct.cpp`

```
1  struct point
2  {
3    point(int x, int y)
4    {
5      this->x = x;
6      this->y = y;
7    }
8
9    inline double getX()
10   {
11     return x;
12   }
13
14   inline double getY()
15   {
16     return y;
17   }
18
19   double dist(point p)
20   {
21     return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
22   }
23
24   private:
25     double x, y;
26 };
```

Utilisation

- Fichiers `point-class.cpp` et `point-struct.cpp`

```
1  int main()
2  {
3    point p1(1,1);
4    //cout << p1.x << " " << p1.y << endl; //ko
5    cout << p1.getX() << " " << p1.getY() << endl;
6    point p2(2,2);
7    cout << p2.getX() << " " << p2.getY() << endl;
8    cout << "dist = " << p1.dist(p2) << endl;
9  }
```

Remarque

- Ne pas oublier le ’;’ après la déclaration d’une classe ou d’une structure

Implémentation et déclarations

- Souvent, on sépare la déclaration et l'implémentation d'une classe
- Déclaration dans les headers `.h`, implémentation dans les sources `.cpp`
  - Fichiers `.hpp` : headers et sources
- On définit l'implémentation à l'aide de `::`
  - `double point::dist(point p) { ... }`

Fonctions `inline`

- Doivent être implémentés dans la même unité de traduction
- Si défini complètement au sein d'une classe, union ou structure : implicitement inline

Implémentation et déclarations

- Souvent, on sépare la déclaration et l'implémentation d'une classe
- Déclaration dans les headers `.h`, implémentation dans les sources `.cpp`
  - Fichiers `.hpp` : headers et sources
- On définit l'implémentation à l'aide de `::`
  - `double point::dist(point p) { ... }`

Fonctions `inline`

- Doivent être implémentés dans la même unité de traduction
- Si défini complètement au sein d'une classe, union ou structure : implicitement inline

Implémentation et déclarations

- Souvent, on sépare la déclaration et l'implémentation d'une classe
- Déclaration dans les headers `.h`, implémentation dans les sources `.cpp`
  - Fichiers `.hpp` : headers et sources
- On définit l'implémentation à l'aide de `::`
  - `double point::dist(point p) { ... }`

Fonctions `inline`

- Doivent être implémentés dans la même unité de traduction
- Si défini complètement au sein d'une classe, union ou structure : implicitement inline

Implémentation et déclarations

- Souvent, on sépare la déclaration et l'implémentation d'une classe
- Déclaration dans les headers `.h`, implémentation dans les sources `.cpp`
  - Fichiers `.hpp` : headers et sources
- On définit l'implémentation à l'aide de `::`
  - `double point::dist(point p) { ... }`

Fonctions `inline`

- Doivent être implémentés dans la même unité de traduction
- Si défini complètement au sein d'une classe, union ou structure : implicitement inline

Implémentation et déclarations

- Souvent, on sépare la déclaration et l'implémentation d'une classe
- Déclaration dans les headers `.h`, implémentation dans les sources `.cpp`
  - Fichiers `.hpp` : headers et sources
- On définit l'implémentation à l'aide de `::`
  - `double point::dist(point p) { ... }`

Fonctions `inline`

- Doivent être implémentés dans la même unité de traduction
- Si défini complètement au sein d'une classe, union ou structure : implicitement inline

Implémentation et déclarations

- Souvent, on sépare la déclaration et l'implémentation d'une classe
- Déclaration dans les headers `.h`, implémentation dans les sources `.cpp`
  - Fichiers `.hpp` : headers et sources
- On définit l'implémentation à l'aide de `::`
  - `double point::dist(point p) { ... }`

Fonctions `inline`

- Doivent être implémentés dans la même unité de traduction
  - « Même fichier »
- Si défini complètement au sein d'une classe, union ou structure : implicitement inline

Implémentation et déclarations

- Souvent, on sépare la déclaration et l'implémentation d'une classe
- Déclaration dans les headers `.h`, implémentation dans les sources `.cpp`
  - Fichiers `.hpp` : headers et sources
- On définit l'implémentation à l'aide de `::`
  - `double point::dist(point p) { ... }`

Fonctions `inline`

- Doivent être implémentés dans la même unité de traduction
  - « Même fichier »
- Si défini complètement au sein d'une classe, union ou structure : implicitement inline

Implémentation et déclarations

- Souvent, on sépare la déclaration et l'implémentation d'une classe
- Déclaration dans les headers `.h`, implémentation dans les sources `.cpp`
  - Fichiers `.hpp` : headers et sources
- On définit l'implémentation à l'aide de `::`
  - `double point::dist(point p) { ... }`

Fonctions `inline`

- Doivent être implémentés dans la même unité de traduction
  - « Même fichier »
- Si défini complètement au sein d'une classe, union ou structure : implicitement inline

Implémentation et déclarations

- Souvent, on sépare la déclaration et l'implémentation d'une classe
- Déclaration dans les headers `.h`, implémentation dans les sources `.cpp`
  - Fichiers `.hpp` : headers et sources
- On définit l'implémentation à l'aide de `::`
  - `double point::dist(point p) { ... }`

Fonctions `inline`

- Doivent être implémentés dans la même unité de traduction
  - « Même fichier »
- Si défini complètement au sein d'une classe, union ou structure : implicitement inline

Exemple (1/2)

- Fichier `point-decl-impl.h`

```
1  //no include, no using namespace std;
2
3  class point
4  {
5    double x, y;
6
7    public:
8      point(int x, int y);
9      inline double getX();
10     inline double getY();
11     double dist(point p);
12 };
13
14 double point::getX()
15 {
16     return x;
17 }
18
19 double point::getY()
20 {
21     return y;
22 }
```

Exemple (2/2)

- Fichier `point-decl-impl.cpp`

```
1  #include "point-decl-impl.h"
2
3  point::point(int x, int y) {
4      this->x = x;
5      this->y = y;
6  }
7
8  double point::dist(point p) {
9      return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
10 }
```

- Fichier `point-decl-impl-main.cpp`

```
1  #include <iostream>
2  #include "point-decl-impl.h"
3
4  using namespace std;
5
6  int main() {
7    point p1(1,1); point p2(2,2);
8    cout << p1.getX() << " " << p1.getY() << endl;
9    cout << p2.getX() << " " << p2.getY() << endl;
10   cout << "dist = " << p1.dist(p2) << endl;
11 }
```

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Fonctions membres constantes

- Rend une fonction *membre* constante
- Ajoute le CV-qualifier `const` sur la fonction concernée
- Ne modifie pas `this`
  - Impossible de modifier un attribut
  - Impossible d'appeler une fonction non constante
- Usage du mot-clé `const` à la fin du prototype
- Souvent utilisé pour les getters
- Si définition de deux prototypes (un `const` et pas l'autre),
  - les objets `const` appellent le prototype `const`
  - les autres appellent l'autre
- Compilatoirement, offre certaines optimisations
- Surtout utile pour le programmeur

Exemple

- Fichier `const-class.cpp`

```
1  class A
2  {
3      int i;
4
5      public:
6          A(int i) { this->i = i; }
7          void set(int i) { this->i = i; }
8          int& get() { cout << "g " ; return i; }
9          const int& get() const { cout << "cg " ; return i; }
10 };
11
12 int main()
13 {
14     A a(2);
15
16     a.set(3);
17     cout << a.get() << endl;
18     a.get() = 5;
19     cout << a.get() << endl;
20
21     const A ca(42);
22     //ca.set(5);
23     cout << ca.get() << endl;
24 }
```

Constructeurs et destructeurs

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Introduction

- Constructeur : appelé à l'instanciation d'un objet
  - Allocation de mémoire
  - Assignation des attributs, pré-traitement, etc.
- Destructeur : appelé à la destruction de l'objet
  - Désallocation de mémoire
  - Post-traitement, désallocations explicites

Exemple : écriture dans un fichier

- Création : initialisation avec le chemin vers le fichier, test d'existence, ouverture du fichier
- Destruction : vidage des tampons, fermeture du fichier

- Pas de type de retour, même nom que la classe
  - Destructeur préfixé de ~

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 10)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 12)

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 10)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 12)

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 10)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 12)

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 10)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 12)

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 10)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 12)

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 10)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 12)

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 10)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 12)

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 10)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 12)

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 10)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 12)

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 10)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 12)

Constructeur particuliers

- On utilise un constructeur à
  - l'initialisation(implicite ou non),
  - la copie (implicite ou non),
  - la réallocation
- Pas à l'affectation
- Possibilité de plusieurs constructeurs
  - Constructeur par défaut
  - Constructeur de recopie
  - Constructeur de déplacement (cf. Ch. 10)
  - Constructeur « personnalisé »
- Règles d'appel particulières en cas d'héritage (cf. Ch. 12)

Exemple

- Fichier `point-cstr.cpp`

```
1  class point {
2    double x, y;
3    bool copie;
4
5    public:
6      point(int x, int y) {
7        this->x = x; this->y = y;
8        copie = false;
9
10       cout << "Construction de " << x << " " << y << endl;
11     }
12
13     point(const point& p) {
14       this-> x = p.x; this-> y = p.y;
15       copie = true;
16
17       cout << "Copie de " << x << " " << y << endl;
18     }
19
20     ~point() {
21       cout << "Destruction de " << x << " " << y;
22       if(copie)
23         cout << " (copie)";
24       cout << endl;
25     }
26 }
```

Exemple (2/2)

- Fichier `point-cstr.cpp`

```
1  void sayHello(point p) { //fct indep, try with &, *
2    cout << "Hello␣Mr␣point␣" << p.getX() << "␣" << p.getY() << endl;
3  }
4
5  int main() {
6    point p1(0,0); point p2(1,1);
7    cout << p1.getX() << "␣" << p1.getY() << endl;
8    sayHello(p1);
9    cout << p2.getX() << "␣" << p2.getY() << endl;
10   cout << "dist␣=␣" << p1.dist(p2) << endl;
11
12   point p3(p1);//explicit copy
13   p3 = p2;
14 }
```

Remarques

- Copies implicites effectuées
- Destructions implicites effectuées
- Affectation muette

Table des matières

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Appelé à l'instanciation (cf. Ch. 5)
  - Implicitement à la déclaration sans paramètres
- Si aucun constructeur (quel que soit son « type ») n'est défini par l'utilisateur, un constructeur par défaut est ajouté à la compilation
  - Public, et inline
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur par défaut avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Appelé à l'instanciation (cf. Ch. 5)
  - Implicitement à la déclaration sans paramètres
- Si aucun constructeur (quel que soit son « type ») n'est défini par l'utilisateur, un constructeur par défaut est ajouté à la compilation
  - Public, et inline
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur par défaut avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Appelé à l'instanciation (cf. Ch. 5)
  - Implicitement à la déclaration sans paramètres
- Si aucun constructeur (quel que soit son « type ») n'est défini par l'utilisateur, un constructeur par défaut est ajouté à la compilation
  - Public, et inline
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur par défaut avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Appelé à l'instanciation (cf. Ch. 5)
  - Implicitement à la déclaration sans paramètres
- Si aucun constructeur (quel que soit son « type ») n'est défini par l'utilisateur, un constructeur par défaut est ajouté à la compilation
  - Public, et inline
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur par défaut avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Appelé à l'instanciation (cf. Ch. 5)
  - Implicitement à la déclaration sans paramètres
- Si aucun constructeur (quel que soit son « type ») n'est défini par l'utilisateur, un constructeur par défaut est ajouté à la compilation
  - Public, et inline
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur par défaut avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Appelé à l'instanciation (cf. Ch. 5)
  - Implicitement à la déclaration sans paramètres
- Si aucun constructeur (quel que soit son « type ») n'est défini par l'utilisateur, un constructeur par défaut est ajouté à la compilation
  - Public, et inline
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur par défaut avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Appelé à l'instanciation (cf. Ch. 5)
  - Implicitement à la déclaration sans paramètres
- Si aucun constructeur (quel que soit son « type ») n'est défini par l'utilisateur, un constructeur par défaut est ajouté à la compilation
  - Public, et inline
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur par défaut avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Appelé à l'instanciation (cf. Ch. 5)
  - Implicitement à la déclaration sans paramètres
- Si aucun constructeur (quel que soit son « type ») n'est défini par l'utilisateur, un constructeur par défaut est ajouté à la compilation
  - Public, et inline
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur par défaut avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Appelé à l'instanciation (cf. Ch. 5)
  - Implicitement à la déclaration sans paramètres
- Si aucun constructeur (quel que soit son « type ») n'est défini par l'utilisateur, un constructeur par défaut est ajouté à la compilation
  - Public, et inline
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur par défaut avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur par défaut

- Constructeur sans paramètres
  - Possibilité avec valeurs par défaut
- Appelé à l'instanciation (cf. Ch. 5)
  - Implicitement à la déclaration sans paramètres
- Si aucun constructeur (quel que soit son « type ») n'est défini par l'utilisateur, un constructeur par défaut est ajouté à la compilation
  - Public, et inline
  - Instanciation toujours possible
- Si un constructeur avec paramètres est présent et pas de constructeur par défaut, appeler le constructeur par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur par défaut avec `= default;`
  - Même effet qu'un constructeur vide

Suppression de constructeur par défaut

- On peut empêcher la génération d'un constructeur par défaut avec `= delete;`
  - Permet de s'assurer que des objets sans constructeurs sont dans des états cohérents
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre de type référence ou constant non initialisé
  - `T` possède un membre non initialisé avec un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un destructeur `delete`

Suppression de constructeur par défaut

- On peut empêcher la génération d'un constructeur par défaut avec `= delete;`
  - Permet de s'assurer que des objets sans constructeurs sont dans des états cohérents
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre de type référence ou constant non initialisé
  - `T` possède un membre non initialisé avec un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un destructeur `delete`

Suppression de constructeur par défaut

- On peut empêcher la génération d'un constructeur par défaut avec `= delete;`
  - Permet de s'assurer que des objets sans constructeurs sont dans des états cohérents
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre de type référence ou constant non initialisé
  - `T` possède un membre non initialisé avec un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un destructeur `delete`

Suppression de constructeur par défaut

- On peut empêcher la génération d'un constructeur par défaut avec `= delete;`
  - Permet de s'assurer que des objets sans constructeurs sont dans des états cohérents
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre de type référence ou constant non initialisé
  - `T` possède un membre non initialisé avec un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un destructeur `delete`

Suppression de constructeur par défaut

- On peut empêcher la génération d'un constructeur par défaut avec `= delete;`
  - Permet de s'assurer que des objets sans constructeurs sont dans des états cohérents
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre de type référence ou constant non initialisé
  - `T` possède un membre non initialisé avec un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un destructeur `delete`

Suppression de constructeur par défaut

- On peut empêcher la génération d'un constructeur par défaut avec `= delete;`
  - Permet de s'assurer que des objets sans constructeurs sont dans des états cohérents
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre de type référence ou constant non initialisé
  - `T` possède un membre non initialisé avec un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un destructeur `delete`

Suppression de constructeur par défaut

- On peut empêcher la génération d'un constructeur par défaut avec `= delete;`
  - Permet de s'assurer que des objets sans constructeurs sont dans des états cohérents
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre de type référence ou constant non initialisé
  - `T` possède un membre non initialisé avec un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un constructeur par défaut `delete`
  - `T` hérite d'une classe ayant un destructeur `delete`

Utilité

- Fichier `delete-cstr.cpp`

```
1  struct A { int i; };
2  struct AD
3  {
4      int i;
5      AD() = delete;
6  };
7
8  int main()
9  {
10     A a1;      //i not init
11     A a2{};    //i = 0
12     A a3{42}; //i = 42
13
14     //AD ad;
15     AD ad{};
16     AD ad2{42};
17 }
```

Appels (1/2)

- Fichier `call-cstr.cpp`

```
1  struct A {
2      int x;
3      A(int x = 1): x(x) {} // user-defined default constructor
4  };
5
6  struct B : A {}; // B::B() implicitly-defined, calls A::A()
7
8  struct C {
9      A a;
10 }; // C::C() implicitly-defined, calls A::A()
11
12 struct D: A {
13     D(int y): A(y) {}
14 }; // D::D() not declared
15
16 struct E: A {
17     E(int y): A(y) {}
18     E() = default; // explicitly defaulted, calls A::A()
19 };
20
21 struct F {
22     int& ref;
23     const int c;
24 }; // F::F() is implicitly defined as deleted
```

Appels (2/2)

- Fichier `call-cstr.cpp`

```
1  int main()
2  {
3      A a;
4      B b;
5      C c;
6  //  D d;
7      E e;
8  //  F f;
9  }
```

Table des matières

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Constructeur de recopie

- Appelé quand un paramètre est passé par valeur
  - `T a(b);` (appel explicite)
  - `f(a);`, où `f` est `B f(A a)`
  - `return a;` dans une fonction `A f(B b)`
- Constructeur avec un paramètre constant passé par référence
  - `MaClasse::MaClasse(const MaClasse& c)`
- Si aucun constructeur de recopie n'est présent, un constructeur de recopie par défaut est ajouté à la compilation
  - Public et inline
- Si un constructeur de recopie avec paramètres est présent et pas de constructeur de recopie par défaut, appeler le constructeur de recopie par défaut provoque une erreur de compilation
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Suppression de constructeur de recopie

- On peut empêcher la génération d'un constructeur de recopie avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être copiés
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non copiable
  - `T` possède un constructeur de déplacement ou opérateur d'assignation-déplacement (cf. Ch. 10)
  - `T` hérite d'une classe ayant un constructeur de recopie `delete`

Suppression de constructeur de recopie

- On peut empêcher la génération d'un constructeur de recopie avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être copiés
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non copiable
  - `T` possède un constructeur de déplacement ou opérateur d'assignation-déplacement (cf. Ch. 10)
  - `T` hérite d'une classe ayant un constructeur de recopie `delete`

Suppression de constructeur de recopie

- On peut empêcher la génération d'un constructeur de recopie avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être copiés
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non copiable
  - `T` possède un constructeur de déplacement ou opérateur d'assignation-déplacement (cf. Ch. 10)
  - `T` hérite d'une classe ayant un constructeur de recopie `delete`

Suppression de constructeur de recopie

- On peut empêcher la génération d'un constructeur de recopie avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être copiés
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non copiable
  - `T` possède un constructeur de déplacement ou opérateur d'assignation-déplacement (cf. Ch. 10)
  - `T` hérite d'une classe ayant un constructeur de recopie `delete`

Suppression de constructeur de recopie

- On peut empêcher la génération d'un constructeur de recopie avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être copiés
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non copiable
  - `T` possède un constructeur de déplacement ou opérateur d'assignation-déplacement (cf. Ch. 10)
  - `T` hérite d'une classe ayant un constructeur de recopie `delete`

Suppression de constructeur de recopie

- On peut empêcher la génération d'un constructeur de recopie avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être copiés
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non copiable
  - `T` possède un constructeur de déplacement ou opérateur d'assignation-déplacement (cf. Ch. 10)
  - `T` hérite d'une classe ayant un constructeur de recopie `delete`

Suppression de constructeur de recopie

- On peut empêcher la génération d'un constructeur de recopie avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être copiés
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non copiable
  - `T` possède un constructeur de déplacement ou opérateur d'assignation-déplacement (cf. Ch. 10)
  - `T` hérite d'une classe ayant un constructeur de recopie `delete`

Exemple (1/2)

- Fichier `call-copy.cpp`

```
1  struct A {
2      int n;
3      A(int n = 1) : n(n) { }
4      A(const A& a) : n(a.n) { }
5  };
6
7  void f1(A a) {}
8  void f2(A& a) {}
9
10 A f3()
11 {
12     A a;
13     return a;
14 }
15
16 int main()
17 {
18     A a1(7);
19     A a2(a1);   // copy
20     A a3 = a2; //copy
21
22     f1(a1); //copy
23     f2(a1);
24     A a4 = f3();
25 }
```

Exemple (2/2)

- Fichier `call-copy.cpp`

```
1  struct B
2  {
3      B();
4      B(const B&) = delete;
5  };
6
7  void f4(B b) {}
8  void f5(B& b) {}
9
10 /*
11 B f6()
12 {
13     B b;
14     return b;
15 }*/
16
17 int main()
18 {
19     B b;
20     //f4(b);
21     f5(b);
22
23     //B b2 = f6();
24 }
```

Table des matières

Principe

- Permet d'initialiser « à la volée » les attributs dans un constructeur
- Plus efficace (moins de copies temporaires) que dans les accolades

Remarque importante

- Indispensable pour
  - initialiser les attributs constants
  - initialiser des attributs non initialisables par défaut
  - initialiser les références
  - effectuer de la délégation de constructeurs
- Initialisation avec `:`, sous la forme d'une liste séparée par des `,`

Principe

- Permet d'initialiser « à la volée » les attributs dans un constructeur
- Plus efficace (moins de copies temporaires) que dans les accolades

Remarque importante

- Indispensable pour
  - initialiser les attributs constants
  - initialiser des attributs non initialisables par défaut
  - initialiser les références
  - effectuer de la délégation de constructeurs
- Initialisation avec `:`, sous la forme d'une liste séparée par des `,`

Principe

- Permet d'initialiser « à la volée » les attributs dans un constructeur
- Plus efficace (moins de copies temporaires) que dans les accolades

Remarque importante

- Indispensable pour
  1. initialiser les attributs constants
  2. initialiser des attributs non initialisables par défaut
  3. initialiser les références
  4. effectuer de la délégation de constructeurs

- Initialisation avec `:`, sous la forme d'une liste séparée par des `,`

Principe

- Permet d'initialiser « à la volée » les attributs dans un constructeur
- Plus efficace (moins de copies temporaires) que dans les accolades

Remarque importante

- Indispensable pour
  1. initialiser les attributs constants
  2. initialiser des attributs non initialisables par défaut
  3. initialiser les références
  4. effectuer de la délégation de constructeurs

- Initialisation avec `:`, sous la forme d'une liste séparée par des `,`

Principe

- Permet d'initialiser « à la volée » les attributs dans un constructeur
- Plus efficace (moins de copies temporaires) que dans les accolades

Remarque importante

- Indispensable pour
  1. initialiser les attributs constants
  2. initialiser des attributs non initialisables par défaut
  3. initialiser les références
  4. effectuer de la délégation de constructeurs

- Initialisation avec `:`, sous la forme d'une liste séparée par des `,`

Principe

- Permet d'initialiser « à la volée » les attributs dans un constructeur
- Plus efficace (moins de copies temporaires) que dans les accolades

Remarque importante

- Indispensable pour
  1. initialiser les attributs constants
  2. initialiser des attributs non initialisables par défaut
  3. initialiser les références
  4. effectuer de la délégation de constructeurs

- Initialisation avec `:`, sous la forme d'une liste séparée par des `,`

Principe

- Permet d'initialiser « à la volée » les attributs dans un constructeur
- Plus efficace (moins de copies temporaires) que dans les accolades

Remarque importante

- Indispensable pour
  1. initialiser les attributs constants
  2. initialiser des attributs non initialisables par défaut
  3. initialiser les références
  4. effectuer de la délégation de constructeurs

- Initialisation avec `:`, sous la forme d'une liste séparée par des `,`

Principe

- Permet d'initialiser « à la volée » les attributs dans un constructeur
- Plus efficace (moins de copies temporaires) que dans les accolades

Remarque importante

- Indispensable pour
  1. initialiser les attributs constants
  2. initialiser des attributs non initialisables par défaut
  3. initialiser les références
  4. effectuer de la délégation de constructeurs

- Initialisation avec `:`, sous la forme d'une liste séparée par des `,`

Principe

- Permet d'initialiser « à la volée » les attributs dans un constructeur
- Plus efficace (moins de copies temporaires) que dans les accolades

Remarque importante

- Indispensable pour
  1. initialiser les attributs constants
  2. initialiser des attributs non initialisables par défaut
  3. initialiser les références
  4. effectuer de la délégation de constructeurs

- Initialisation avec `:`, sous la forme d'une liste séparée par des `,`

Exemple

- Fichier `point_init.cpp`

```
1  class point
2  {
3    double x, y;
4
5    public:
6      point(int x = 0, int y = 0) : x(x), y(y) {}
7
8      double getX() const
9      {
10       return x;
11     }
12
13     double getY() const
14     {
15       return y;
16     }
17
18     double dist(point p)
19     {
20       return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
21     }
22 };
```

Délégation

- Fichier `deleg.cpp`

```
1  class A
2  {
3    int i;
4    const int k;
5
6    private:
7      A() : k(5)
8      {
9        cout << "Init " << endl;
10       //k = 5;
11     }
12
13   public:
14     A(int x) : A()/*, i(x)*/
15     {
16       i = x;
17     }
18
19     void print() { cout << "A : " << i << endl;}
20 };
```

- Utilisé aussi pour l'appel de superconstructeurs

Absence de constructeur par défaut

- Fichier `no-cstr.cpp`

```
1  struct A
2  {
3    int i;
4    A(int i) : i(i) {}
5  };
6
7  struct B
8  {
9    A a;
10   B(A a) : a(a) {};//ok
11
12   //B(A a)//ko
13   //{
14   //  this -> a = a;
15   //}
16 };
17
18 int main()
19 {
20   A a(2);
21   B b(a);
22 }
```

Header `initializer_list.h`

- Permet d'avoir des arguments « variables en nombre » dans les constructeurs
  - Aussi dans les fonctions (membres ou non)
- Instanciation avec les accolades
  - `vector<int> v = {1, 2, 3};`
  - `brol.append({1,2,3});`
- Se comporte comme une liste
  - Itérateurs, `size()`, etc.

Remarque

- `objet.fonction(2);` ≠ `objet.fonction({2});`

Header `initializer_list.h`

- Permet d'avoir des arguments « variables en nombre » dans les constructeurs
  - Aussi dans les fonctions (membres ou non)
- Instanciation avec les accolades
  - `vector<int> v = {1, 2, 3};`
  - `brol.append({1,2,3});`
- Se comporte comme une liste
  - Itérateurs, `size()`, etc.

Remarque

- `objet.fonction(2);` ≠ `objet.fonction({2});`

Header `initializer_list.h`

- Permet d'avoir des arguments « variables en nombre » dans les constructeurs
  - Aussi dans les fonctions (membres ou non)
- Instanciation avec les accolades
  - `vector<int> v = {1, 2, 3};`
  - `brol.append({1,2,3});`
- Se comporte comme une liste
  - Itérateurs, `size()`, etc.

Remarque

- `objet.fonction(2);` ≠ `objet.fonction({2});`

Header `initializer_list.h`

- Permet d'avoir des arguments « variables en nombre » dans les constructeurs
  - Aussi dans les fonctions (membres ou non)
- Instanciation avec les accolades
  - `vector<int> v = {1, 2, 3};`
  - `brol.append({1,2,3});`
- Se comporte comme une liste
  - Itérateurs, `size()`, etc.

Remarque

- `objet.fonction(2);` ≠ `objet.fonction({2});`

Header `initializer_list.h`

- Permet d'avoir des arguments « variables en nombre » dans les constructeurs
  - Aussi dans les fonctions (membres ou non)
- Instanciation avec les accolades
  - `vector<int> v = {1, 2, 3};`
  - `brol.append({1,2,3});`
- Se comporte comme une liste
  - Itérateurs, `size()`, etc.

Remarque

- `objet.fonction(2);` ≠ `objet.fonction({2});`

Header `initializer_list.h`

- Permet d'avoir des arguments « variables en nombre » dans les constructeurs
  - Aussi dans les fonctions (membres ou non)
- Instanciation avec les accolades
  - `vector<int> v = {1, 2, 3};`
  - `brol.append({1,2,3});`
- Se comporte comme une liste
  - Itérateurs, `size()`, etc.

Remarque

- `objet.fonction(2);` ≠ `objet.fonction({2});`

Header `initializer_list.h`

- Permet d'avoir des arguments « variables en nombre » dans les constructeurs
  - Aussi dans les fonctions (membres ou non)
- Instanciation avec les accolades
  - `vector<int> v = {1, 2, 3};`
  - `brol.append({1,2,3});`
- Se comporte comme une liste
  - Itérateurs, `size()`, etc.

Remarque

- `objet.fonction(2);` ≠ `objet.fonction({2});`

Header `initializer_list.h`

- Permet d'avoir des arguments « variables en nombre » dans les constructeurs
  - Aussi dans les fonctions (membres ou non)
- Instanciation avec les accolades
  - `vector<int> v = {1, 2, 3};`
  - `brol.append({1,2,3});`
- Se comporte comme une liste
  - Itérateurs, `size()`, etc.

Remarque

- `objet.fonction(2);` ≠ `objet.fonction({2});`

Header `initializer_list.h`

- Permet d'avoir des arguments « variables en nombre » dans les constructeurs
  - Aussi dans les fonctions (membres ou non)
- Instanciation avec les accolades
  - `vector<int> v = {1, 2, 3};`
  - `brol.append({1,2,3});`
- Se comporte comme une liste
  - Itérateurs, `size()`, etc.

Remarque

- `objet.fonction(2);` ≠ `objet.fonction({2});`

Exemple

- Fichier `dataset.cpp`

```
1  class DataSet
2  {
3    double sum;
4    int count;
5
6    public:
7      DataSet() : sum(0), count(0) {}
8      DataSet(const initializer_list<double>& data) : DataSet() { update(data); }
9
10     void update(const initializer_list<double>& data)
11     {
12       for(double d : data)
13       {
14         update(d);
15       }
16     }
17
18     inline void update(double d)
19     {
20       sum += d;
21       count++;
22     }
23
24     inline double mean() const { return sum / count; }
25 };
```

Initialisation explicite : résumé

- `A a; B b;`
- `A a(b);` : appel explicite au constructeur
- `A a = b;` : « conversion »
  1. Opérateur d'affectation surchargé (cf. Ch. 8)
  2. Appel constructeur avec conversion implicite autorisée (cf. Ch. 9)
- `A a {b};` : appel explicite au constructeur, sans conversion implicite
- `A a = {b};`
  - Si pas de constructeur `std::initializer_list`, équivalent à `A a b;`
  - Sinon, appelle le constructeur `std::initializer_list`

Initialisation explicite : résumé

- `A a; B b;`
- `A a(b);` : appel explicite au constructeur
- `A a = b;` : « conversion »
  1. Opérateur d'affectation surchargé (cf. Ch. 8)
  2. Appel constructeur avec conversion implicite autorisée (cf. Ch. 9)
- `A a {b};` : appel explicite au constructeur, sans conversion implicite
- `A a = {b};`
  - Si pas de constructeur `std::initializer_list`, équivalent à `A a b;`
  - Sinon, appelle le constructeur `std::initializer_list`

Initialisation explicite : résumé

- `A a; B b;`
- `A a(b);` : appel explicite au constructeur
- `A a = b;` : « conversion »
  1. Opérateur d'affectation surchargé (cf. Ch. 8)
  2. Appel constructeur avec conversion implicite autorisée (cf. Ch. 9)
- `A a {b};` : appel explicite au constructeur, sans conversion implicite
- `A a = {b};`
  - Si pas de constructeur `std::initializer_list`, équivalent à `A a b;`
  - Sinon, appelle le constructeur `std::initializer_list`

Initialisation explicite : résumé

- `A a; B b;`
- `A a(b);` : appel explicite au constructeur
- `A a = b;` : « conversion »
  1. Opérateur d'affectation surchargé (cf. Ch. 8)
  2. Appel constructeur avec conversion implicite autorisée (cf. Ch. 9)
- `A a {b};` : appel explicite au constructeur, sans conversion implicite
- `A a = {b};`
  - Si pas de constructeur `std::initializer_list`, équivalent à `A a b;`
  - Sinon, appelle le constructeur `std::initializer_list`

Initialisation explicite : résumé

- `A a; B b;`
- `A a(b);` : appel explicite au constructeur
- `A a = b;` : « conversion »
  1. Opérateur d'affectation surchargé (cf. Ch. 8)
  2. Appel constructeur avec conversion implicite autorisée (cf. Ch. 9)
- `A a {b};` : appel explicite au constructeur, sans conversion implicite
- `A a = {b};`
  - Si pas de constructeur `std::initializer_list`, équivalent à `A a b;`
  - Sinon, appelle le constructeur `std::initializer_list`

Initialisation explicite : résumé

- `A a; B b;`
- `A a(b);` : appel explicite au constructeur
- `A a = b;` : « conversion »
  1. Opérateur d'affectation surchargé (cf. Ch. 8)
  2. Appel constructeur avec conversion implicite autorisée (cf. Ch. 9)
- `A a {b};` : appel explicite au constructeur, sans conversion implicite
- `A a = {b};`
  - Si pas de constructeur `std::initializer_list`, équivalent à `A a b;`
  - Sinon, appelle le constructeur `std::initializer_list`

Initialisation explicite : résumé

- `A a; B b;`
- `A a(b);` : appel explicite au constructeur
- `A a = b;` : « conversion »
  1. Opérateur d'affectation surchargé (cf. Ch. 8)
  2. Appel constructeur avec conversion implicite autorisée (cf. Ch. 9)
- `A a {b};` : appel explicite au constructeur, sans conversion implicite
- `A a = {b};`
  - Si pas de constructeur `std::initializer_list`, équivalent à `A a b;`
  - Sinon, appelle le constructeur `std::initializer_list`

Initialisation explicite : résumé

- `A a; B b;`
- `A a(b);` : appel explicite au constructeur
- `A a = b;` : « conversion »
  1. Opérateur d'affectation surchargé (cf. Ch. 8)
  2. Appel constructeur avec conversion implicite autorisée (cf. Ch. 9)
- `A a {b};` : appel explicite au constructeur, sans conversion implicite
- `A a = {b};`
  - Si pas de constructeur `std::initializer_list`, équivalent à `A a b;`
  - Sinon, appelle le constructeur `std::initializer_list`

Initialisation explicite : résumé

- `A a; B b;`
- `A a(b);` : appel explicite au constructeur
- `A a = b;` : « conversion »
  1. Opérateur d'affectation surchargé (cf. Ch. 8)
  2. Appel constructeur avec conversion implicite autorisée (cf. Ch. 9)
- `A a {b};` : appel explicite au constructeur, sans conversion implicite
- `A a = {b};`
  - Si pas de constructeur `std::initializer_list`, équivalent à `A a b;`
  - Sinon, appelle le constructeur `std::initializer_list`

Table des matières

Destructeur

- Fonction particulière appelée à la désallocation de l'objet
  - Désallocation implicite ou explicite (cf. Ch. 5)
  - Règles d'appel particulières en cas d'héritage (cf. Ch. 12)
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de `~`
- Si aucun destructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Public et inline
- Unique (si plusieurs : erreur)
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Destructeur

- Fonction particulière appelée à la désallocation de l'objet
  - Désallocation implicite ou explicite (cf. Ch. 5)
  - Règles d'appel particulières en cas d'héritage (cf. Ch. 12)
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de `~`
- Si aucun destructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Public et inline
- Unique (si plusieurs : erreur)
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Destructeur

- Fonction particulière appelée à la désallocation de l'objet
  - Désallocation implicite ou explicite (cf. Ch. 5)
  - Règles d'appel particulières en cas d'héritage (cf. Ch. 12)
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de `~`
- Si aucun destructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Public et inline
- Unique (si plusieurs : erreur)
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Destructeur

- Fonction particulière appelée à la désallocation de l'objet
  - Désallocation implicite ou explicite (cf. Ch. 5)
  - Règles d'appel particulières en cas d'héritage (cf. Ch. 12)
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de `~`
- Si aucun destructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Public et inline
- Unique (si plusieurs : erreur)
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Destructeur

- Fonction particulière appelée à la désallocation de l'objet
  - Désallocation implicite ou explicite (cf. Ch. 5)
  - Règles d'appel particulières en cas d'héritage (cf. Ch. 12)
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de `~`
- Si aucun destructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Public et inline
- Unique (si plusieurs : erreur)
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Destructeur

- Fonction particulière appelée à la désallocation de l'objet
  - Désallocation implicite ou explicite (cf. Ch. 5)
  - Règles d'appel particulières en cas d'héritage (cf. Ch. 12)
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de `~`
- Si aucun destructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Public et inline
- Unique (si plusieurs : erreur)
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Destructeur

- Fonction particulière appelée à la désallocation de l'objet
  - Désallocation implicite ou explicite (cf. Ch. 5)
  - Règles d'appel particulières en cas d'héritage (cf. Ch. 12)
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de `~`
- Si aucun destructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Public et inline
- Unique (si plusieurs : erreur)
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Destructeur

- Fonction particulière appelée à la désallocation de l'objet
  - Désallocation implicite ou explicite (cf. Ch. 5)
  - Règles d'appel particulières en cas d'héritage (cf. Ch. 12)
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de `~`
- Si aucun destructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Public et inline
- Unique (si plusieurs : erreur)
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Destructeur

- Fonction particulière appelée à la désallocation de l'objet
  - Désallocation implicite ou explicite (cf. Ch. 5)
  - Règles d'appel particulières en cas d'héritage (cf. Ch. 12)
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de `~`
- Si aucun destructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Public et inline
- Unique (si plusieurs : erreur)
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Destructeur

- Fonction particulière appelée à la désallocation de l'objet
  - Désallocation implicite ou explicite (cf. Ch. 5)
  - Règles d'appel particulières en cas d'héritage (cf. Ch. 12)
- Pas de type de retour, pas de paramètres
- Même nom que la classe, précédé de `~`
- Si aucun destructeur n'est présent, un constructeur par défaut est ajouté à la compilation
  - Public et inline
- Unique (si plusieurs : erreur)
- On peut forcer la génération d'un constructeur de recopie avec `= default;`
  - Même effet qu'un constructeur vide

Suppression de destructeur

- On peut empêcher la génération d'un destructeur avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être détruits
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non destructible
  - `T` hérite d'une classe ayant un destructeur `delete`

Suppression de destructeur

- On peut empêcher la génération d'un destructeur avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être détruits
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non destructible
  - `T` hérite d'une classe ayant un destructeur `delete`

Suppression de destructeur

- On peut empêcher la génération d'un destructeur avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être détruits
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non destructible
  - `T` hérite d'une classe ayant un destructeur `delete`

Suppression de destructeur

- On peut empêcher la génération d'un destructeur avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être détruits
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non destructible
  - `T` hérite d'une classe ayant un destructeur `delete`

Suppression de destructeur

- On peut empêcher la génération d'un destructeur avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être détruits
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non destructible
  - `T` hérite d'une classe ayant un destructeur `delete`

Suppression de destructeur

- On peut empêcher la génération d'un destructeur avec `= delete;`
  - Permet de s'assurer que des objets ne peuvent être détruits
  - Performance
- On peut également empêcher implicitement la génération dans une classe `T` si
  - `T` possède un membre non destructible
  - `T` hérite d'une classe ayant un destructeur `delete`

Exemple

- Fichier `destr.cpp`

```
1  void open_f(const string& path) { cout << "Opening " << path << endl;   }
2
3  void flush_f(const string& path) {  cout << "Flushing " << path << endl;   }
4
5  void close_f(const string& path) {  cout << "Closing " << path << endl;   }
6
7  class InputFileStream {
8      const string& path;
9        public:
10           InputFileStream(const string& path) : path(path) {
11               open_f(path);
12           }
13
14           ~InputFileStream()  {
15               flush_f(path);
16               close_f(path);
17           }
18 };
19
20 int main()  {
21     InputFileStream("brol.txt");
22 }//désallocation implicite
```

- Autres exemples au chapitre 5

Déclaration, définition et inclusion

Séparation déclaration / définition

Rappel

- Souvent, les déclarations sont séparées des définitions
  - Déclarations dans des fichiers `.h`
  - Définitions dans des fichiers `.c` / `.cpp`
  - Pas les fonctions `inline`
- Permet, entre autres, d'éviter les problèmes
  - liés à l'ordre des déclarations
  - liés aux inclusions multiples de fichiers

- Implémentation des fonctions membres à l'aide de l'opérateur de résolution de portée (C++)

Séparation déclaration / définition

Rappel

- Souvent, les déclarations sont séparées des définitions
  - Déclarations dans des fichiers `.h`
  - Définitions dans des fichiers `.c` / `.cpp`
  - Pas les fonctions `inline`
- Permet, entre autres, d'éviter les problèmes
  - liés à l'ordre des déclarations
  - liés aux inclusions multiples de fichiers

- Implémentation des fonctions membres à l'aide de l'opérateur de résolution de portée (C++)

Séparation déclaration / définition

Rappel

- Souvent, les déclarations sont séparées des définitions
  - Déclarations dans des fichiers `.h`
  - Définitions dans des fichiers `.c` / `.cpp`
  - Pas les fonctions `inline`
- Permet, entre autres, d'éviter les problèmes
  - liés à l'ordre des déclarations
  - liés aux inclusions multiples de fichiers

- Implémentation des fonctions membres à l'aide de l'opérateur de résolution de portée (C++)

Séparation déclaration / définition

Rappel

- Souvent, les déclarations sont séparées des définitions
  - Déclarations dans des fichiers `.h`
  - Définitions dans des fichiers `.c` / `.cpp`
  - Pas les fonctions `inline`
- Permet, entre autres, d'éviter les problèmes
  - liés à l'ordre des déclarations
  - liés aux inclusions multiples de fichiers

- Implémentation des fonctions membres à l'aide de l'opérateur de résolution de portée (C++)

Séparation déclaration / définition

Rappel

- Souvent, les déclarations sont séparées des définitions
  - Déclarations dans des fichiers `.h`
  - Définitions dans des fichiers `.c` / `.cpp`
  - Pas les fonctions `inline`
- Permet, entre autres, d'éviter les problèmes
  - liés à l'ordre des déclarations
  - liés aux inclusions multiples de fichiers

- Implémentation des fonctions membres à l'aide de l'opérateur de résolution de portée (C++)

Séparation déclaration / définition

Rappel

- Souvent, les déclarations sont séparées des définitions
  - Déclarations dans des fichiers `.h`
  - Définitions dans des fichiers `.c` / `.cpp`
  - Pas les fonctions `inline`
- Permet, entre autres, d'éviter les problèmes
  - liés à l'ordre des déclarations
  - liés aux inclusions multiples de fichiers

- Implémentation des fonctions membres à l'aide de l'opérateur de résolution de portée (C++)

Séparation déclaration / définition

Rappel

- Souvent, les déclarations sont séparées des définitions
  - Déclarations dans des fichiers `.h`
  - Définitions dans des fichiers `.c` / `.cpp`
  - Pas les fonctions `inline`
- Permet, entre autres, d'éviter les problèmes
  - liés à l'ordre des déclarations
  - liés aux inclusions multiples de fichiers

- Implémentation des fonctions membres à l'aide de l'opérateur de résolution de portée (C++)

Séparation déclaration / définition

Rappel

- Souvent, les déclarations sont séparées des définitions
  - Déclarations dans des fichiers `.h`
  - Définitions dans des fichiers `.c` / `.cpp`
  - Pas les fonctions `inline`
- Permet, entre autres, d'éviter les problèmes
  - liés à l'ordre des déclarations
  - liés aux inclusions multiples de fichiers

- Implémentation des fonctions membres à l'aide de l'opérateur de résolution de portée (C++)

Séparation déclaration / définition

Rappel

- Souvent, les déclarations sont séparées des définitions
  - Déclarations dans des fichiers `.h`
  - Définitions dans des fichiers `.c` / `.cpp`
  - Pas les fonctions `inline`
- Permet, entre autres, d'éviter les problèmes
  - liés à l'ordre des déclarations
  - liés aux inclusions multiples de fichiers

- Implémentation des fonctions membres à l'aide de l'opérateur de résolution de portée (C++)

Exemple avec fonctions indépendantes

- Fichiers `fct-decl.h`

```
1  void print_str(const char*);
```

- Fichier `fct-decl.c`

```
1  #include "stdio.h"
2  #include "fct-decl.h"
3
4  void print_str(const char* s)
5  {
6    printf("%s\n", s);
7  }
```

- Fichier `fct-decl-main.c`

```
1  #include "fct-decl.h"
2
3  int main()
4  {
5    print_str("Hello World!");
6  }
```

Exemple C++

- Fichiers `point_decl.h`, `point_decl.cpp` et `point_decl-main.cpp`

```
1  class point
2  {
3    double x, y;
4
5    public:
6      point(double x, double y);
7      inline double getX() const;
8      inline double getY() const;
9      double dist(point p) const;
10 };
```

```
1  int main()
2  {
3    point p1(1,1);
4    //cout << p1.x << " " << p1.y << endl; //ko
5    cout << p1.getX() << " " << p1.getY() << endl;
6    point p2(2,2);
7    cout << p2.getX() << " " << p2.getY() << endl;
8    cout << "dist = " << p1.dist(p2) << endl;
9  }
```

Exemple

- Fichiers `point_decl.h`, `point_decl.cpp` et `point_decl-main.cpp`

```
1  double point::getX() const
2  {
3    return x;
4  }
5
6  double point::getY() const
7  {
8    return y;
9  }
```

```
1  #include "point_decl.h"
2
3  point::point(double x, double y)
4  {
5    this->x = x;
6    this->y = y;
7  }
8
9  double point::dist(point p) const
10 {
11   return sqrt((x - p.x)*(x - p.x)+(y - p.y)*(y - p.y));
12 }
```

Inclusions multiples

- Parfois, des inclusions multiples de fichiers sont nécessaires
  - Un maillon de liste chaînée a un attribut maillon (élément suivant de la liste)
  - Un département est dirigé par un manager, un manager dirige un département

Un problème de taille

- Si un cycle d'attributs apparaît, les objets sont de taille infinie

- Solution : utiliser une adresse et `#ifndef` / `#define`

Inclusions multiples

- Parfois, des inclusions multiples de fichiers sont nécessaires
  - Un maillon de liste chaînée a un attribut maillon (élément suivant de la liste)
  - Un département est dirigé par un manager, un manager dirige un département

Un problème de taille

- Si un cycle d'attributs apparaît, les objets sont de taille infinie

- Solution : utiliser une adresse et `#ifndef` / `#define`

Inclusions multiples

- Parfois, des inclusions multiples de fichiers sont nécessaires
  - Un maillon de liste chaînée a un attribut maillon (élément suivant de la liste)
  - Un département est dirigé par un manager, un manager dirige un département

Un problème de taille

- Si un cycle d'attributs apparaît, les objets sont de taille infinie

- Solution : utiliser une adresse et `#ifndef` / `#define`

Inclusions multiples

- Parfois, des inclusions multiples de fichiers sont nécessaires
  - Un maillon de liste chaînée a un attribut maillon (élément suivant de la liste)
  - Un département est dirigé par un manager, un manager dirige un département

Un problème de taille

- Si un cycle d'attributs apparaît, les objets sont de taille infinie

- Solution : utiliser une adresse et `#ifndef` / `#define`

Inclusions multiples

- Parfois, des inclusions multiples de fichiers sont nécessaires
  - Un maillon de liste chaînée a un attribut maillon (élément suivant de la liste)
  - Un département est dirigé par un manager, un manager dirige un département

Un problème de taille

- Si un cycle d'attributs apparaît, les objets sont de taille infinie

- Solution : utiliser une adresse et `#ifndef` / `#define`

Inclusions multiples

- Parfois, des inclusions multiples de fichiers sont nécessaires
  - Un maillon de liste chaînée a un attribut maillon (élément suivant de la liste)
  - Un département est dirigé par un manager, un manager dirige un département

Un problème de taille

- Si un cycle d'attributs apparaît, les objets sont de taille infinie

- Solution : utiliser une adresse et `#ifndef` / `#define`

Exemple

- Fichier `mag-dep-pourri.cpp`

```
1  struct Manager
2  {
3    Departement& dpt;
4    string nom;
5
6    Manager(string nom, Departement& dpt) : nom(nom), dpt(dpt) {}
7  };
8
9  struct Departement
10 {
11   Manager mgr;
12   string nom;
13
14   Departement(string nom) : nom(nom) {}
15 };
16
17 int main()
18 {
19   Departement esi("ESI");
20   Manager mwi("Willemse", esi);
21   esi.mgr = mwi;
22 }
```

Exemple

- Fichiers `manager.*`, `departement.*` et `mag-dep-main.cpp`

```
1  #ifndef DEP
2  #define DEP
3
4  #include <string>
5
6  struct Manager;
7
8  struct Departement
9  {
10   Manager* mgr;//not allocated here
11   std::string nom;
12
13   Departement(std::string nom, Manager* mgr = nullptr);
14 };
15
16 #endif
```

```
1  #include "departement.h"
2
3  Departement::Departement(std::string nom, Manager* mgr) : nom(nom), mgr(mgr) {}
```

Exemple

- Fichiers `manager.*`, `departement.*` et `mag-dep-main.cpp`

```
1  #ifndef MAG
2  #define MAG
3
4  #include <string>
5  #include "departement.h"
6
7  struct Manager
8  {
9    Departement& dpt;
10   std::string nom;
11
12   Manager(std::string nom, Departement& dpt);
13 };
14
15 #endif
```

```
1  #include "manager.h"
2
3  Manager::Manager(std::string nom, Departement& dpt) : nom(nom), dpt(dpt) {}
```