Structures

Déclaration et utilisation

Structures, tableaux et pointeurs

Structures et fonctions

Enumérations

Structures

- Permet de désigner un ensemble de valeurs de types différents
- Déclaration: précise le nom et le type de chacun des champs
  - Mais ne réserve pas de variable correspondant à cette structure

```
struct enreg{
    int num, qte;
    float prix;
};
```

- Il est possible de déclarer une variable du type correspondant
  - L'instruction *enreg article;* réserve un emplacement pour deux *int* et un *float*

Utilisation des champs d'une structure

- Chaque champ peut être utilisé comme une variable du même type
  - L'opérateur « point » (.) est utilisé pour accéder à un champ
    - Exemple: *article.qte = 1;*
- Peut affecter à une structure le contenu d'une structure de même type
  - Exemple: *article_bis = article;* où *article_bis* est une variable de type *enreg*
- Une structure peut être initialisée lors de sa déclaration
  - Exemple: *enreg article_ter = { 100, 285, 200 };*

  - Les expressions fournies devront être des constantes de types compatibles
  - Il est possible d'omettre certaines valeurs

Structure comportant des tableaux

- Chaque champ d'une structure peut être de type quelconque
  - Tableau, pointeur, structure

```
struct personne{
    char nom[30], prenom[20];
    int age;
};
personne employe;
```

- *employe.nom[4]* désigne le 5^{em} caractère du champ *nom* d'*employe*
- Il est possible d'initialiser les champs de type tableau d'une structure

```
personne employeur = { "Dupont", "Jules ", 30 };
```

Tableau de structures

- Il est également possible de faire des tableaux de structures

```
struct point{
      int x, y;
};
point courbe[50];
```

- *courbe[4].x* désigne le champ x du 5^{em} *point* du tableau
- Il est possible d'initialiser les champs des éléments du tableau

```
point droite[2]={ {0, 1}, {1, 0} };
```

Structure comportant des structures

- Les champs peuvent être de type structure quelconque

```
struct cercle{
    point centre;
    int rayon;
};
cercle disque;
```

- *disque.centre*.x désigne le champs x du champ *centre*

Pointeur de structure

- Une structure ne peut pas contenir un champ du type de la structure
  - Mais il peut contenir un champ de type pointeur sur le type de la structure

```
struct case_point{
        point element;
        liste_points *suivant;
};
case_point liste;
liste.suivant = new case_point;
```

Cette structure représente une case d'une liste chainée. Le chainage se fait par l'intermédiaire des pointeurs: le champ *suivant* pointe vers la case suivante dans la liste. Ici, l'espace mémoire associé à la variable liste est réservé lors de sa déclaration, et l'espace mémoire de la case suivante est réservé dynamiquement (*new* crée la case et renvoie un pointeur dessus)

Portée du type structure

- La portée dépend de l'emplacement de sa déclaration
  - Au sein d'une fonction: accessible que depuis cette fonction
  - En dehors d'une fonction: accessible partout à partir de sa déclaration
- Pas possible de faire référence à un type défini dans un autre fichier source
  - Pour partager des types de structures, il faut les définir dans des fichiers d'en-tête
    - Incorporer l'en-tête en utilisant *#include*
- Des types structure différents peuvent contenir des champs de même nom

Transmission d'une structure par valeur

- Les champs de l'argument muet sont recopiés dans l'argument effectif

```
void exchange(point p){
        int c = p.x;
        p.x = p.y;
        p.y = c;
}
main(){
        point p1;
        exchange(p1);
        …
}
```

Ici, lors de l'appel de la fonction *exchange*, la variable locale *p* est créé en copiant les valeurs des champs de *p1* (équivalent à *point p = p1*). Changer la valeur d'un champ de *p* n'a donc aucun impact sur les champs de la variable *p1*

Transmission par référence

- Permet d'avoir accès au champ de l'argument muet

```
void exchange(point &p){
        int c = p.x;
        p.x = p.y;
        p.y = c;
}
main(){
        point p1;
        exchange(p1);
        …
}
```

Ici, la variable locale *p* est une référence vers la variable *p1*. Changer la valeur d'un champ de *p* va donc avoir un impact sur le même champ de la variable *p1*

Transmission par pointeur

- Peut également simuler la transmission par référence avec des pointeurs

```
void exchange(point *p){
        int c = (*p).x;
        (*p).x = (*p).y;
        (*p).y = c;
}
main(){
        point p1;
        exchange(&p1);
        …
}
```

Ici, lors de l'appel de la fonction exchange, la variable locale *p* est créée avec comme valeur l'adresse de *p1* (c'est un pointeur). L'instruction **p* permet d'accéder à la variable *p1*, et l'instruction *(*p).x* permet d'accéder au champ x de la variable *p1*

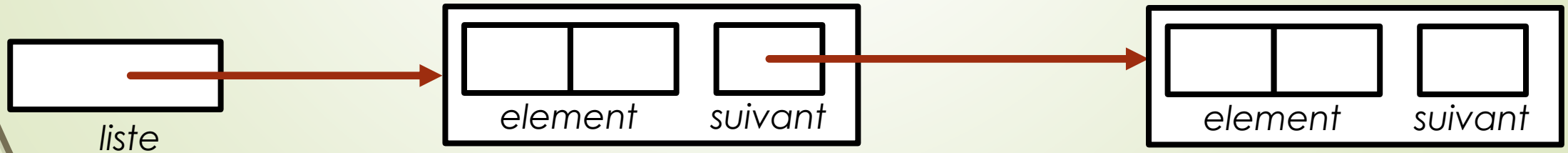
L'opérateur ->

- La notation *(*p).x* désigne le champ x de l'élément pointé par *p*
- L'opérateur -> permet aussi d'accéder au champ à partir du pointeur
  - La notation *p->x* est équivalente à la notation *(*p).x*
- Peut également être utilisé pour des structures allouées dynamiquement

*case_point *liste = new case_point;*

liste->suivant = new case_point;

Ici, *liste* est un pointeur qui contient l'adresse de la case allouée dynamiquement par *new*. La seconde instruction va réserver l'espace mémoire pour une second case et mettre son adresse dans le champ *suivant* de la case pointée par *liste*



Structure en valeur de retour

- Une fonction peut fournir en valeur de retour une structure

```
point milieu(point a, point b){
    point m;
    m.x = (a.x+b.x)/2;
    m.y = (a.y+b.y)/2;
    return m;
}
point a, b, c;
c = milieu(a,b);
```

La variable *m* est locale à la fonction *milieu* et sera supprimée à la fin de la fonction. Une copie de sa valeur (de ses champs) est retournée par la fonction avant la destruction de *m*. Cette copie sera affectée au point c (recopie des champs)

Pointeur en valeur de retour

- Une fonction peut renvoyer un pointeur ou référence sur une structure
  - Attention: la structure pointée ne doit pas être locale à la fonction

```
point* milieu(point a, point b){
    point *m = new point;
    m->x = (a.x+b.x)/2;
    m->y = (a.y+b.y)/2;
    return m;
}
point a, b, *c;
c = milieu(a,b);
```

Ici, un *point* est créé dynamiquement, et son adresse est stockée dans la variable locale *m*. Cette variable locale est détruite à la fin de la fonction (mais pas le *point*). La valeur de cette variable (l'adresse du *point*) est renvoyée par la fonction milieu, et cette adresse est stockée dans c.

TD 3

Enumération

- Cas particulier du type entier

enum couleur { jaune, rouge, bleu, vert};

- L'énumération couleur comporte 4 valeurs possibles
  - *jaune*, *rouge*, *bleu* et *vert* sont les constantes du type *couleur*
- Il est possible de déclarer des variables du type énumération

couleur c1, c2;

c1 = jaune;

c2 = c1;

Ici, *c1* et c2 sont des variables de type *couleur*. *c1* a pour valeur *jaune*, et c2 se voit affecter la valeur de *c1* (c'est-à-dire *jaune*)

Propriétés

- Une énumération associe à chacun de ses constantes une valeur entière
  - Attribue la valeur 0 à la première, la valeur 1 à la seconde, …
- Il est possible d'influer sur ces valeurs
  - Exemple: *enum couleur { jaune = 2, rouge = 5, bleu = -1, vert = 2};*
  - Les constantes ne sont plus modifiables après leurs déclarations

- Ne peut pas porter le même nom qu'une autre constante ou variable
- Règles de portée similaires à celles des structures

Classes et objets

Structures généralisées

Classes

Constructeurs et destructeurs

Champs static

Exploitation d'une classe

Structure généralisée

- Il est possible d'associer des méthodes à une structure
  - Fonction membre dont l'en-tête est déclaré dans la structure
    - La définition d'une fonction membre n'apparait pas dans la structure

```
struct point{
    int x,y;
    void initialise(int, int);
};
point p;
```

Ici, la fonction membre *initialise* est déclaré au sein de la structure *point*. Elle sera définie en dehors de cette structure (slide suivant)

- L'appel à une fonction membre est similaire à l'accès à un champ
  - Exemple: *p.initialise(4, 5);*

Définition d'une fonction membre

- Elle se fait par une définition (presque) classique de fonction

```
void point::initialise(int abs, int ord){
    x = abs;
    y = ord;
}
```

Si la fonction *initialise* est appelée par le point *p* (voir slide précédent) alors la valeur du premier argument sera affectée à *p.x*, et la valeur du second argument sera affectée à *p.y*.

- :: est un opérateur de résolution de porté
  - Sert à modifier la portée d'un identificateur
  - Signifie ici que l'identificateur *initialise* est celui de la structure *point*
    - La déclaration doit être faite avant la définition (par exemple, dans un fichier d'en-tête)
  - x et y correspondent aux attributs de la structure appelante de la fonction

Classes

- En programmation orientée objet, les données sont encapsulées
  - Leur accès ne peut se faire que par le biais de fonctions membres
- Classe: structure où certains membres peuvent être publics ou privés
  - Remplace mot-clé *struct* par *class*
  - Précise les membres *public* et *private*

```
class Point{
    private:
        int x,y;
    public:
        void initialise(int, int);
};
```

Fonctions membres

- Leur définition se fait de la même manière que pour les structures
- Elles ont accès à l'ensemble des membres (privés/publiques) de la classe

```
void Point::initialise(int abs, int ord){
    x = abs;
    y = ord;
}
```

Utilisation d'une classe

- L'utilisation d'une classe se fait comme une structure
  - Excepté que les membres privés ne sont pas accessibles

Point p;

p.initialise(4, 5);

- On dit que *p* est un objet de type *Point*

Fonctions membres privées

- Les fonction membres peuvent également être privées
  - Les mots-clés *public* et *private* peuvent apparaitre plusieurs fois

```
class Point{
    private:
        int x,y;
    public:
        void initialise(int, int);
    private:
        void deplace(int, int);
};
```

Affectation d'objets

- L'affectation d'un objet se fait de la même manière qu'une structure
  - Recopie de l'ensemble des valeurs des champs
    - Indifféremment, que ceux-ci soient public ou privé

```
Point p1, p2;
p1.initialise(4, 5);
p2 = p1;
```

Ici, les champs de *p1* sont initialisés par la fonction membre *initialise*. L'affectation *p2=p1* va recopier les valeurs des champs de *p1* dans ceux de *p2*. Néanmoins, *p1* et *p2* resteront des objets distincts: si *p1* ou *p2* est modifié par la suite, cela n'aura aucune conséquence sur l'autre variable

En mémoire:

Constructeur et destructeur

- Le constructeur permet d'initialiser les champs d'un objet à sa création
  - Permet d'avoir des valeurs « correctes » de ces champs
  - Effectue également les opérations nécessaires au bon fonctionnement
    - Allocation dynamique de mémoire, ouverture de fichier, connexion à un serveur, …
  - Appelé automatiquement après la réservation en mémoire d'un objet
  - Porte le même nom que la classe
- Le destructeur est appelé automatiquement avant la destruction de l'objet
  - Porte le même nom que la classe, précédé par ~

Constructeur

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
};
Point a(4, 5);
```

```
Point::Point(int abs, int ord){
    x = abs;
    y = ord;
}
```

L'espace mémoire de *a* est réservé en mémoire, et ensuite le constructeur est appelé avec comme paramètres 4 et 5 afin d'initialiser les champs de *a*

- Attention: Lorsque une classe a un constructeur, il n'est plus possible de créer un objet sans fournir ses arguments (sauf si il n'en a pas)

Constructeur sans argument

- Un constructeur peut ne pas avoir d'argument
  - La déclaration d'un d'objet s'écrit comme si il n'y avait pas de constructeur

```
class Point{
    private:
        int x,y;
    public:
        Point();
};
Point a;
```

```
Point::Point(){
    x = 0;
    y = 0;
}
```

L'espace mémoire de *a* est réservé en mémoire, et ensuite le constructeur sans paramètre est appelé afin d'initialiser les champs de *a*

- Si pas de constructeur, tout se passe comme si il en avait un qui ne fait rien

Appel constructeur et destructeur

TD 4
Exercice 1

```
class Test{
        private:
                int id;
        public:
                Test(int);
                ~Test();
};
Test::Test(int i){
        id = i;
        cout << " constr " << id << " \n ";
}
Test::~Test(){
        cout << " destr " << id << " \n ";
}
void fonction(int val){
        Test t2(val);
}
main(){
        Test t1(1);
        fonction(2);
}
```

Affichage:
constr 1
constr 2
destr 2
destr 1

Allocation dynamique de champs

- Une classe peut contenir des éléments alloués dynamiquement
  - L'allocation dynamique doit être effectuée par le constructeur
  - La libération doit être effectuée par le destructeur

```
class Vecteur{
    private:
        int taille, *vec;
    public:
        Vecteur(int);
        ~Vecteur();
};
```

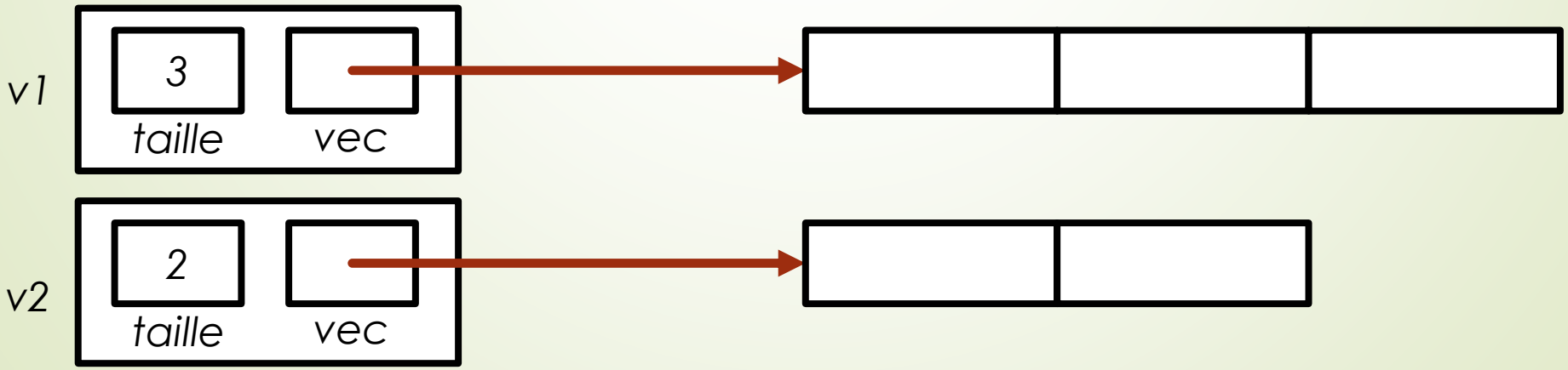
```
Vecteur::Vecteur(int t){
    taille = t;
    vec = new int[taille];
}
Vecteur::~Vecteur(){
    delete [] vec;
}
```

Quelques règles

- Un constructeur peut comporter un nombre quelconque d'arguments
- Un destructeur ne peut pas disposer d'argument
- Constructeur et destructeur ne renvoie pas de valeur
- Attention: lorsqu'une classe a des champs alloués dynamiquement
  - L'affectation entre objet de même type ne concerne pas les parties dynamiques

Vecteur v1(3), v2(2);

v2 = v1;

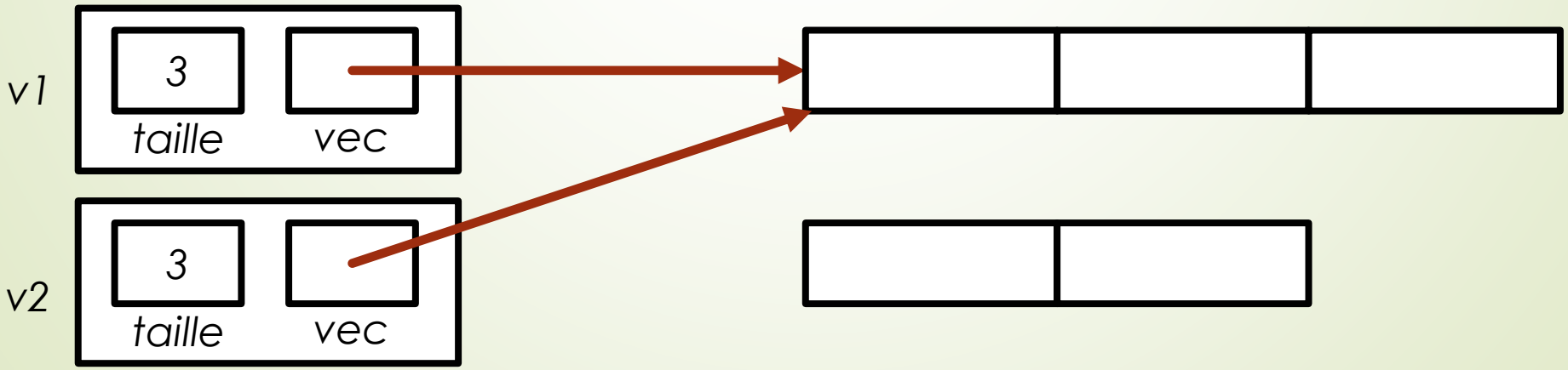


Quelques règles

- Un constructeur peut comporter un nombre quelconque d'arguments
- Un destructeur ne peut pas disposer d'argument
- Constructeur et destructeur ne renvoie pas de valeur
- Attention: lorsqu'une classe a des champs alloués dynamiquement
  - L'affectation entre objet de même type ne concerne pas les parties dynamiques

Vecteur v1(3), v2(2);

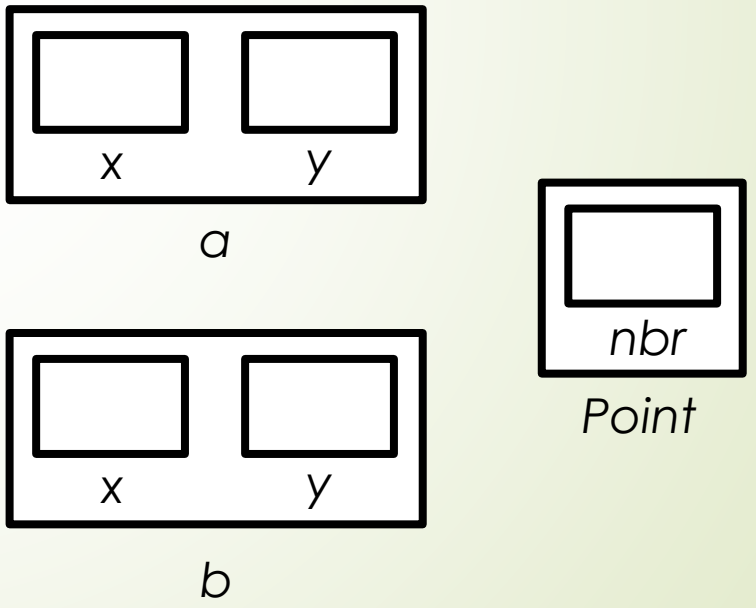
v2 = v1;



Champs static

- Chaque objet possède ses propres champs
  - Déclarer un champ *static* permet de le partager entre les différents objets
    - Existe en un seul exemplaire pour tous les objets de la classe

```
class Point{
    private:
        int x,y;
        static int nbr;
    public:
        Point(int, int);
        ~Point();
};
Point a, b;
```



Initialisation d'un champ *static*

- L'initialisation ne peut pas être faite dans un constructeur
- Elle ne peut également pas être faite lors de sa déclaration
- Elle doit être faite explicitement à l'extérieur de la déclaration de la classe
  - Exemple: *int Point::nbr = 0;*
  - Aussi bien pour les membres statiques privés que publiques

Les membres statiques ont un comportement similaire aux variables globales, excepté que des restrictions d'accès sont possible grâce au mot clé *private*

Exemple

Le membre statique *nbr* de la classe *Point* est initialisé à 0 en début de programme. Cette valeur est ensuite modifiée par le constructeur et le destructeur

```
class Point{
    private:
        int x,y;
        static int nbr;
    public:
        Point(int, int);
        ~Point();
};
```

```
int Point::nbr = 0;
Point::Point(int abs, int ord){
    x = abs;
    y = ord;
    nbr++;
}
Point::~Point(){
    nbr--;
}
```

Exploitation d'une classe

- En pratique, on aura intérêt à découpler la classe de son utilisation
  - La classe pourra être utilisée séparément dans différentes applications
- Isoler les instructions de déclaration de la classe dans un fichier en-tête (.h)

  - Inclut par la commande *#include*
- Fabriquer un module objet en compilant les définitions des classes

  - Prévoir l'initialisation des membres statiques dans ce fichier
- Pour faire appel à la classe dans un programme

  - Inclure la déclaration de la classe (fichier .h)
  - Incorporer le module objet correspondant au moment de l'édition des liens

Exemple

- **Fichier *point.h***

```
class Point{
        private:
                int x,y;
                static int nbr;
        public:
                Point(int, int);
                ~Point();
};
```

- **Module objet *point.o* de *point.cpp***
  - Compilation: *g++ -c point.cpp*

```
#include" point.h "
int Point::nbr = 0;
Point::Point(int abs, int ord){
        x = abs;
        y = ord;
        nbr++;
}
Point::~Point(){
        nbr--;
}
```

Exemple

- **Utilisation de la classe (fichier *test.cpp*)**
  - Compilation: *g++ -c test.cpp* suivi de *g++ –o test point.o test.o*

```
#include" point.h "
main(){
        Point p1, p2;
}
```

Lors de la compilation, la ligne *#include" point.h "* est remplacée par le contenu du fichier *point.h*. Toutes les déclarations concernant la classe *Point* sont donc présente, et la compilation est possible. A la suite de cela, l'édition des liens va lier la déclaration des fonctions membres de la classe Point avec leurs définitions présente dans le fichier objet point.o

Propriétés des fonctions membres

Surdéfinition

Argument par défaut

Objet transmis en argument

Objet en retour d'une fonction

Fonction membre statique

Surdéfinition de fonction membre

- Surdéfinition: lorsqu'un symbole possède plusieurs significations différentes
  - Le choix de l'une des significations se fait en fonction du contexte
    - Par exemple, la signification de *a*/*b* dépend du type de *a* et *b*
- C++ permet la surdéfinition de fonction
  - S'applique aussi aux fonctions membres et aux constructeurs d'une classe
  - Nécessite un critère pour choisir la bonne fonction lors de l'appel
    - Suivant le nombre d'arguments
    - Suivant le type des arguments

Exemple

Ici, le constructeur et la fonction membre sont initialise sont surdéfinis. Dans les deux cas, le nombre d'argument permet de les différentier

```
class Point{
    private:
        int x,y;
    public:
        Point();
        Point(int, int);
        void initialise(int, int);
        void initialise();
};
Point::Point(){
    x = y = 0;
}
```

```
Point::Point(int abs, int ord){
    x = abs;
    y = ord;
}
void Point::initialise(int abs, int ord){
    x = abs;
    y = ord;
}
void Point::initialise(){
    initialise(0, 0);
}
```

Argument par défaut

- Il est possible d'attribuer des valeurs par défaut à des arguments

  - Se fait lors la déclaration de la fonction
    - Exemple: *void initialise(int = 0, int = 0);*
  - Si un argument est absent, l'appel est fait comme si c'était la valeur par défaut
    - Exemple: *initialise(0);* où le second argument sera 0 (valeur par défaut)
  - Les arguments par défaut doivent concerner les derniers arguments de la liste
  - Ces valeurs par défaut ne sont pas nécessairement des constantes
    - Ne doivent pas faire intervenir de variables locales

Exemple

```
class Point{
    private:
        int x,y;
    public:
        Point(int = 0, int = 0);
        void initialise(int = 0, int = 0);
};
```

```
Point::Point(int abs,int ord){
    x = abs;
    y = ord;
}
void Point::initialise(int abs, int ord){
    x = abs;
    y = ord;
```

Ici, le fait d'avoir des valeurs par défaut pour les deux arguments fait que le constructeur et la fonction membre initialise sont définis pour 0, 1 ou 2 arguments (mais dans les trois cas la définition est la même)

Fonctions membres en ligne

- Il est possible de rendre une fonction membre en ligne
  - En donnant la définition de la fonction dans la déclaration de la classe

```
class Point{
    private:
        int x,y;
    public:
        Point(int abs,int ord){
            x = abs;
            y = ord;
        }
};
```

Dans le cas des fonctions en ligne, l'appel de la fonction est directement remplacé par le compilateur par le code de cette fonction. L'avantage est d'éviter un appel de fonction qui peut prendre du temps. L'inconvénient est que le code machine peut être rallongé (prend plus de place en mémoire).

Objets transmis en argument

- Une fonction membre peut recevoir des arguments du type de sa classe
- Les fonctions membres ont accès à tous les champs des objets de la classe

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
        bool coincide(Point);
        void exchange(Point *);
};
```

```
Point::Point(int abs, int ord){
    x = abs;
    y = ord;
}
bool Point::coincide(Point p){
    return (x == p.x) && (y == p.y);
}
```

Objets transmis par pointeur

- La transmission d'objets par pointeurs permet de les modifier

```
void Point::exchange(Point *p){
    int temp_x = x, temp_y = y;
    x = p->x;
    y = p->y;
    p->x = temp_x;
    p->y = temp_y;
}
Point a(4, 5), b(2, 3);
a.exchange(&b);
```

Dans le cas de la transmission d'objet par valeur (fonction *coincide*), l'argument est transmis comme une copie de l'objet donné à la fonction appelante. L'objet donné par la fonction appelante ne peut donc pas être modifié. Ici, la transmission est faite pour pointeur et l'adresse du *Point b* est fournie. La fonction *exchange* peut donc avoir accès à la variable *b*

Objets transmis par référence

TD 4
Exercice 2

- La transmission par référence permet aussi de modifier les objets
  - Simplifie l'écriture des fonctions et de leurs appels

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
        bool coincide(Point);
        void exchange(Point &);
};
```

```
void Point::exchange(Point &p){
    int temp_x = x, temp_y = y;
    x = p.x;
    y = p.y;
    p.x = temp_x;
    p.y = temp_y;
}
Point a(4, 5), b(2, 3);
a.exchange(b);
```

L'avantage également de la transmission d'objet par référence (ou par pointeur) est d'éviter de faire des recopies d'objets pris en argument (comme c'est le cas pour la transmission par valeur). Il faut par contre faire attention aux modifications faites sur les arguments

Qualificatif const

- Pour le cas de la transmission par pointeur ou par référence, l'argument effectif peut être modifié par la fonction
  - Le qualificatif *const* assure que l'argument effectif ne peut pas être modifié

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
        bool coincide(const Point&);
};
```

```
Point::Point(int abs, int ord){
    x = abs;
    y = ord;
}
bool Point::coincide(const Point &p){
    return (x == p.x) && (y == p.y);
}
```

Fonction renvoie un objet

- Une fonction membre peut renvoyer un objet
  - Par valeur, par adresse ou par référence
- Le type de l'objet ne sera pas forcement le même que celui de la classe

  - Si ce n'est pas le cas, la fonction n'aura pas accès à ses champs privés
- Pour la transmission par valeur, chaque champ de l'objet est recopié

  - Attention aux champs alloués dynamiquement
    - Prévoir un constructeur particulier
- Dans le cas de transmission par référence ou par pointeur

  - Pas d'objet local à la fonction qui sera détruit à sa sortie

Exemple

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
        bool coincide(Point);
        void exchange(Point &);
        Point oppose();
};
```

```
Point Point::oppose(){
    Point p(-x, -y);
    return p;
}
Point a(4, 5), b(2, 3);
b = a.oppose();
```

Ici, la variable locale *p* de la fonction *oppose* est renvoyée par valeur. Cela veut dire que avant d'être supprimée à la sortie de la fonction, une copie de cette variable est renvoyée Si la fonction *oppose* devait renvoyer une référence (ou un pointeur) sur un *Point*, alors cela poserait problème car la variable locale *p* n'existe plus à la fin de la fonction

Le mot-clé *this*

- Le mot-clé *this* est utilisable uniquement dans un fonction membre
  - Désigne un pointeur sur l'objet l'ayant appelé

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
        bool coincide(Point);
        void exchange(Point *);
        Point oppose();
        bool same(Point *);
};
```

```
bool Point::same(Point *p){
    return this == p;
}
Point a(4,5), b(4, 5);
a.same(&b);
```

Ici, *this*, qui est un pointeur sur *a* (l'objet qui a appelé la fonction membre *same*), est comparé à *p*, qui est un pointeur contenant l'adresse de *b*. Les deux objets ayant des adresses différentes, la comparaison renvoie *false* (même si les valeurs d'attributs sont les mêmes)

Fonctions membres *static*

- Ont un rôle totalement indépendant d'un quelconque objet
  - Son appel nécessite le nom de la classe accompagné de l'opérateur de porté ::
    - Peut même être appelé lorsqu'il n'existe aucun objet de sa classe

```
class Point{
    private:
        int x,y;
        static int nbr;
    public:
        Point(int, int);
        ~Point();
        static int decompte();
};
```

```
int Point::nbr=0;
Point::Point(int abs, int ord){
    x = abs; y = ord; nbr++;
}
Point::~Point(){
    nbr--;
}
int Point::decompte(){
    return nbr;
}
Point::decompte();
```

Construction, destruction et initialisation d'objets

Objets dynamiques

Constructeur de recopie

Objets membres

Fonctions amies

Les objets dynamiques

- Il est possible de créer dynamiquement un objet avec l'opérateur *new*
  - L'opérateur *new* appelle un constructeur de l'objet
    - Exemple: *Point *p = new Point(4, 5);*
  - Si il n'y pas de constructeur ou s'il existe un constructeur sans argument
    - La syntaxe **p = new Point;* est acceptée
  - L'opérateur *delete* appellera le destructeur avant libération de la mémoire
    - Exemple: *delete p;*

Initialisation par recopie

- Initialisation par recopie: créer un objet par recopie d'un objet existant
  - Valeur d'un objet transmise en argument d'une fonction
  - Un objet est renvoyé par valeur comme résultat d'une fonction
  - Un objet est initialisé lors de sa déclaration avec un autre objet
-  Attention: l'affectation n'est pas une situation d'initialisation par recopie

Constructeur de recopie

- Un constructeur de recopie est utilisé pour l'initialisation par recopie
  -  Un constructeur de recopie par défaut fait une copie de chacun des champs
    - Problème pour des objets contenant des pointeurs sur des emplacement dynamiques
  - Un constructeur de recopie peut être explicitement fournis dans la classe
    - Constructeur public dont l'argument est une référence vers un objet de la classe
      - Exemple: *Point(const Point &);*
    -  Indispensable lorsque des champs sont alloués dynamiquement

Exemple

```
class Vect{
    private:
        int taille, *vec;
    public:
        Vect(int);
        ~Vect();
        Vect(const Vect &);
};
Vect::~Vect(){
    delete [] vec;
}
```

```
Vect::Vect(int t){
    taille = t;
    vec = new int[taille];
}
Vect::Vect(const Vect &v){
    taille = v.taille;
    vec = new int[taille];
    for(int i = 0; i < taille; i++)
        vec[i] = v.vec[i];
}
Vect v1(10), v2 = v1;
```

Ici, le constructeur de recopie va créer un nouvel objet de classe *Vect* comme la copie du vecteur pris en paramètre (par exemple, *v2* est la copie de *v1*). Les attributs dynamiques (comme vec) sont alloués dans cette fonction (pour éviter que deux objets de classe *Vect* est un attribut *vec* qui pointe vers un même tableau en mémoire).

Objets membres

```
class Point{
    private:
        int x,y;
    public:
        Point(int, int);
};
```

```
class Cercle{
    private:
        Point centre;
        int rayon;
    public:
        Cercle(int, int, int);
};
```

- Lorsqu'un membre d'une classe nécessite l'appel d'un constructeur
  - La classe doit contenir un ou plusieurs constructeurs
  - Tout constructeur doit spécifier les arguments des constructeurs de ses membres

Appel constructeur

Cercle::Cercle(int abs, int ord, int ray) : ***centre(abs, ord)****{*

rayon = ray;

}

- Lorsque plusieurs membres objets nécessitent un appel à un constructeur
  - Les appels sont séparés par une virgule

- Les constructeurs des membres sont appelés avant celui de l'objet
- Pas nécessaire si un membre possède un constructeur sans argument

Fonction amie

- La programmation orientée objet impose l'encapsulation des données
  - Les membres privés ne sont accessibles qu'aux fonctions membres
- Lors de la définition d'une classe on peut déclarer des fonctions amies
  - Elles peuvent accéder aux membres privés de la classe
  - Le mot-clé *friend* est utilisé dans la déclaration d'amitié
  - Une fonction peut faire l'objet de déclarations d'amitié dans différentes classes

Exemple

```
class Vect{
    private:
        int taille, *vec;
    public:
        Vect(int);
        ~Vect();
        friend int sum(const Vect &);
};
Vect::~Vect(){
    delete [] vec;
}
```

```
Vect::Vect(int t){
    taille = t;
    vec = new int[taille];
}
int sum(const Vect &v){
    int res = 0;
    for(int i = 0; i < v.taille; i++)
        res += v.vec[i];
    return res;
}
```

Ici, la fonction *sum*, qui n'est pas une fonction membre de la classe *Vect*, a accès aux attributs *taille* et *vec* grâce à la déclaration d'amitié faite dans la classe *Vect*

Fonctions membres amies

- Préciser dans la déclaration d'amitié la classe de la fonction membre

```
class Point;
class Vect{
    private:
        int taille;
        Point *vec;
    public:
        Vect(int);
        ~Vect();
        Point sum();
};
```

```
class Point{
    private:
        int x,y;
    public:
        Point(int = 0, int = 0);
        ~Point();
        friend Point Vect::sum();
};
```

Fonctions membres amies

```
Vect::Vect(int t){
        taille = t;
        vec = new Point[taille];
}
Vect::~Vect(){
        delete [] vec;
}
```

```
Point Vect::sum(){
        Point p(0, 0);
        for(int i = 0; i < taille; i++){
                p.x += vec[i].x;
                p.y += vec[i].y;
        }
        return p;
}
```

Ordre déclaration

- La déclaration d'une classe doit précéder celle dont la fonction est amie
  - Ici, la déclaration de *Vect* doit être faite avant celle de *Point*
- Le compilateur doit connaitre l'existence d'une classe pour qu'elle apparaisse dans une déclaration
  - Pas besoin de connaitre précisément les caractéristiques de cette classe
  - Ici, la classe *Point* est déclarée par l'instruction *class Point;*

Classe amie

TD 4
Exercice 3

- Une classe peut être amie d'une autre
  - Toutes les fonctions membres de la classe seront amies

```
class Point;
class Vect{
    private:
        int taille;
        Point *vec;
    public:
        Vect(int);
        ~Vect();
        Point sum();
};
```

```
class Point{
    private:
        int x,y;
    public:
        Point(int = 0, int = 0);
        ~Point();
        friend class Vect;
};
```

Ici, l'instruction *class Point;* permet de déclarer l'existence d'une classe *Point*. Elle est nécessaire car la fonction membre *sum* de la classe *Vect* retourne un objet de type *Point*. La classe *Point* contient une déclaration d'amitié avec la classe *Vect*, ce qui permet au fonctions membres de la classe *Vect* d'avoir accès aux membres privés de la classe *Point*.