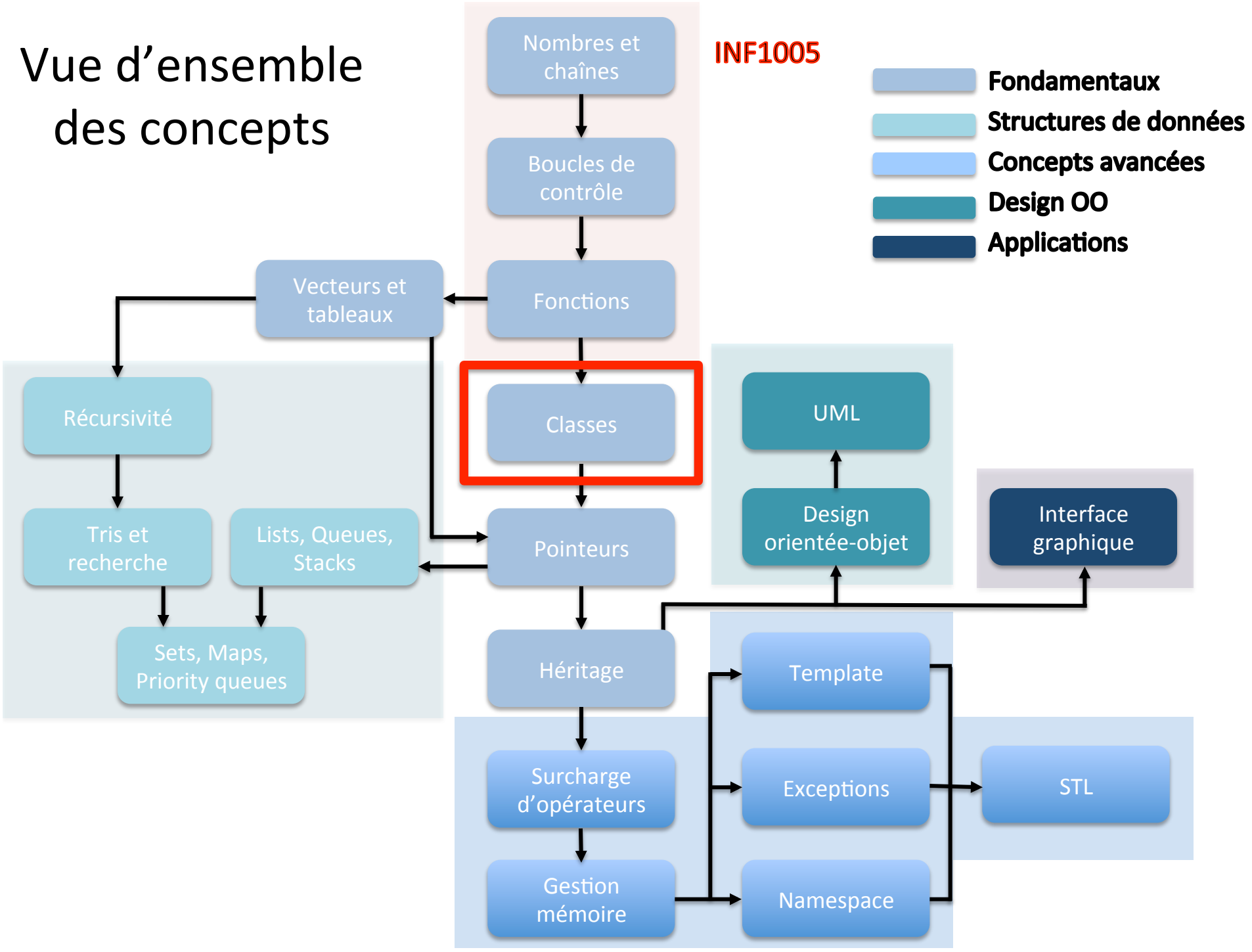
Programmation orientée objet

Le constructeur de copie et l'opérateur =

Michel Gagnon

Vue d'ensemble des concepts



Motivation

- Considérons avec attention la fonction suivante, qui construit une collection de *n* cercles décalés de 1:

```
vector<Cercle> mult(Cercle cercle, int n)
{
    vector<Cercle> v;
    for (int i = 0; i < n; ++i) {
      v.push_back(cercle);
      cercle.move(1,0);
    }
    return v;
}
```

Il faut passer le paramètre par valeur, parce qu'il sera modifié dans la fonction.

Motivation

```
void qc(Point p){}

int main(void){
   Point p1;
   Point p2(1,2);

   Point p3(p2);

   Point p4=p2;

   qc(p4);

   p3=p1;
}
```

Motivation

```
void qc(Point p){}

int main(void){
   Point p1;        //constructeur par défaut
   Point p2(1,2);

   Point p3(p2);

   Point p4=p2;

   qc(p4);

   p3=p1;
}
```

Motivation

```
void qc(Point p){}

int main(void){
   Point p1;       //constructeur par défaut
   Point p2(1,2); //constructeur par param.

   Point p3(p2);

   Point p4=p2;

   qc(p4);

   p3=p1;
}
```

Motivation

```
void qc(Point p){}

int main(void){
   Point p1;       //constructeur par défaut
   Point p2(1,2); //constructeur par param.

   Point p3(p2);  //???

   Point p4=p2;

   qc(p4);

   p3=p1;
}
```

Motivation

```
void qc(Point p){}

int main(void){
   Point p1;       //constructeur par défaut
   Point p2(1,2); //constructeur par param.

   Point p3(p2);  //???

   Point p4=p2;   //???

   qc(p4);

   p3=p1;
}
```

Motivation

```
void qc(Point p){}

int main(void){
   Point p1;        //constructeur par défaut
   Point p2(1,2); //constructeur par param.

   Point p3(p2);   //???

   Point p4=p2;    //???

   qc(p4);

   p3=p1;
}
```

nouvel objet =>
constructeur **de copie**

Motivation

```
void qc(Point p){}//???

int main(void){
   Point p1;        //constructeur par défaut
   Point p2(1,2); //constructeur par param.

   Point p3(p2);   //???

   Point p4=p2;    //???

   qc(p4);

   p3=p1;
}
```

nouvel objet =>
constructeur **de copie**

Motivation

```
void qc(Point p){}//???

int main(void){
   Point p1;        //constructeur par défaut
   Point p2(1,2); //constructeur par param.

   Point p3(p2);   //???

   Point p4=p2;    //???

   qc(p4);

   p3=p1;
}
```

copie sur pile =>
constructeur **de copie**

nouvel objet =>
constructeur **de copie**

Motivation

```
void qc(Point p){}//???

int main(void){
   Point p1;        //constructeur par défaut
   Point p2(1,2); //constructeur par param.

   Point p3(p2);   //???

   Point p4=p2;    //???

   qc(p4);

   p3=p1;          //???
}
```

copie sur pile =>
constructeur **de copie**

nouvel objet =>
constructeur **de copie**

Motivation

```
void qc(Point p){}//???

int main(void){
   Point p1;        //constructeur par défaut
   Point p2(1,2); //constructeur par param.

   Point p3(p2);   //???

   Point p4=p2;    //???

   qc(p4);

   p3=p1;          //???
}
```

copie sur pile =>
constructeur **de copie**

nouvel objet =>
constructeur **de copie**

objet existant =>
opérateur **d'affectation**

Constructeur de copie

- Lorsqu'on fait une **copie** d'un objet, il faut créer un **nouvel objet** qui sera utilisé dans la fonction
- On utilisera donc un constructeur lors de la création de ce nouvel objet
- Ce constructeur recevra comme paramètre un autre objet de la même classe, soit celui qu'on doit copier

Constructeur de copie (suite)

- Si on ne définit pas ce constructeur de copie, C++ utilisera un constructeur par défaut, qui copie tout simplement les attributs ("**shallow copy**")
- Or, cela peut être problématique lorsqu'un attribut est un pointeur: on se retrouve alors avec deux pointeurs qui pointent au même endroit
- Si on ne veut pas que cela se produise, il faut alors **définir un constructeur de copie** qui copiera non pas le pointeur, mais plutôt l'entité pointée par celui-ci ("**deep copy**")

Constructeur de copie - shallow

```
class Cours
{
   public:
      ...
      Cours(const Cours& c);
   private:
      ...
      vector<Etudiant*> inscrits_;
};
```

Ici, on se contente de faire une simple copie du vecteur de pointeurs, puisque les étudiants inscrits n'appartiennent pas à un cours (en fait, on pourrait omettre la définition du constructeur de copie).

```
Cours::Cours(const Cours& c):inscrits_(c.inscrits_){}
```

Constructeur de copie - deep

```
class Ligne
{
public:
   Ligne();
   Ligne(const Ligne& objetCopie);
   ...
private:
   Point* premier_;
   Point* dernier_;
};
```

Les deux attributs sont des pointeurs.

Ici, il faut définir un constructeur de copie. En effet, on ne veut pas que l'objet copié pointe sur les mêmes noeuds que la classe originale.

```
Ligne::Ligne(const Ligne& objetCopie):premier_(0),dernier_(0)
{
  premier_ = new Point(objetCopie.premier_->getX(),
                       objetCopie.premier_->getY());
  dernier_ = new Point(objetCopie.dernier_->getX(),
                       objetCopie.dernier_->getY());
}
```

Pour chaque attribut, on alloue un nouvel espace sur le tas.

Constructeur de copie – deep (alternative)

```
class Ligne
{
public:
   Ligne();
    Ligne(const Ligne & objetCopie);
   ...
private:
   Point* premier_;
   Point* dernier_;
}
```

```
Ligne :: Ligne(const Ligne & objetCopie):premier_(0),dernier_(0)
{
  premier_ = new Point(*objetCopie.premier_);
  dernier_ = new Point(*objetCopie.dernier_);
}
```

Si la classe Point a un constructeur de copie, notre tâche est simplifiée, puisqu'on peut l'appeler directement.

Opérateur =

- Ce que nous venons de dire pour la copie d'un objet vaut aussi pour l'opérateur =:

```
MaClasse objet1;
MaClasse objet2;
...
objet1 = objet2;
```

Ici aussi une copie attribut par attribut sera effectuée, à moins qu'on ne redéfinisse l'opérateur =, ce qui, évidemment, doit être fait pour la classe MaClasse, comme on a dû le faire pour le constructeur de copie.

Exemple de définition de l'opérateur =

```
Ligne& Ligne::operator=(const Ligne& autreObjet)
{


     if (premier_ != 0) delete premier_;
     if (dernier_ != 0) delete dernier_;

     premier_ = new Point(*autreObjet.premier_);
     dernier_ = new Point(*autreObjet.dernier_);


     return *this;
}
```

On retourne une référence à l'objet parce que l'opérateur = peut être appelé en cascade:
I1 = I2 = I3 (qui est équivalent à **I1 = (I2 = I3)**)

Exemple de définition de l'opérateur =

```
Ligne& Ligne::operator=(const Ligne& autreObjet)
{
     if (premier_ != 0) delete premier_;
     if (dernier_ != 0) delete dernier_;

     premier_ = new Point(*autreObjet.premier_);
     dernier_ = new Point(*autreObjet.dernier_);

     return *this;
}
```

Qu'est-ce qui se passe si le client fait:
ligne1 = ligne1

On retourne une référence à l'objet parce que l'opérateur = peut être appelé en cascade:
l1 = l2 = l3 (qui est équivalent à **l1 = (l2 = l3)**)

Exemple de définition de l'opérateur =

```
Ligne& Ligne::operator=(const Ligne& autreObjet)
{

   if (this != &autreObjet) {
      if (premier_ != 0) delete premier_;
      if (dernier_ != 0) delete dernier_;

      premier_ = new Point(*autreObjet.premier_);
      dernier_ = new Point(*autreObjet.dernier_);
   }
   return *this;
}
```

Il faut **TOUJOURS** vérifier que l'on n'est pas en train d'affecter un objet à lui-même: **ligne1 = ligne1**

Résumé

- Lorsqu'on définit une classe en C++, il faut toujours penser à définir **au minimum** les items suivants:
  - Le constructeur par défaut
  - Le constructeur de copie
  - L'opérateur =
  - Le destructeur