Programmation orientée objet

Le constructeur de copie et l'opérateur =

Samuel Kadoury

Motivation

```
void qc(Employee p){}

int main(void){
   Employee e1;
   Employee e2("Marc",15000);


   Employee e3(e2);

   Employee e4=e2;

   qc(e4);
   e3=e1;
}
```

Motivation

```
void qc(Employee p){}

int main(void){
   Employee e1;   //constructeur par défaut
   Employee e2("Marc",15000);


   Employee e3(e2);

   Employee e4=e2;

   qc(e4);
   e3=e1;
}
```

Motivation

```
void qc(Employee p){}

int main(void){
   Employee e1;   //constructeur par défaut
   Employee e2("Marc",15000); //constructeur
    param.

   Employee e3(e2);

   Employee e4=e2;

   qc(e4);
   e3=e1;
}
```

Motivation

```
void qc(Employee p){}

int main(void){
   Employee e1;   //constructeur par défaut
   Employee e2("Marc",15000); //constructeur
    param.

   Employee e3(e2);  //???

   Employee e4=e2;

   qc(e4);
   e3=e1;
}
```

Motivation

```
void qc(Employee p){}

int main(void){
   Employee e1;   //constructeur par défaut
   Employee e2("Marc",15000); //constructeur
    param.

   Employee e3(e2);  //???

   Employee e4=e2;   //???

   qc(e4);
   e3=e1;
}
```

Motivation

```
void qc(Employee p){}

int main(void){
   Employee e1;    //constructeur par défaut
   Employee e2("Marc",15000); //constructeur
     param.

   Employee e3(e2);   //???

   Employee e4=e2;    //???

   qc(e4);
   e3=e1;
}
```

nouvel objet => constructeur **de copie**

Motivation

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

int main(void){
   Employee e1;   //constructeur par défaut
   Employee e2("Marc",15000); //constructeur
     param.

   Employee e3(e2);   //???

   Employee e4=e2;    //???

   qc(e4);
   e3=e1;
}
```

nouvel objet => constructeur **de copie**

Motivation

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

int main(void){
   Employee e1;   //constructeur par défaut
   Employee e2("Marc",15000); //constructeur
     param.

   Employee e3(e2);   //???

   Employee e4=e2;    //???

   qc(e4);
   e3=e1;
}
```

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

nouvel objet => constructeur **de copie**

Motivation

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

int main(void){
   Employee e1;    //constructeur par défaut
   Employee e2("Marc",15000); //constructeur
     param.

   Employee e3(e2);   //???

   Employee e4=e2;    //???

   qc(e4);
   e3=e1;             //???
}
```

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

nouvel objet => constructeur **de copie**

Motivation

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

int main(void){
   Employee e1;    //constructeur par défaut
   Employee e2("Marc",15000); //constructeur
     param.

   Employee e3(e2);   //???

   Employee e4=e2;    //???

   qc(e4);
   e3=e1;             //???
}
```

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 Company
{
  public:
     ...
     Company (const Company& c);
  private:
     ...
     vector<Employee*> employees_;
};


Company::Company (const Company& c):
   employees_(c.employees_){…}
```

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

Constructeur de copie - deep

```
class Company
{
public:
   Company ();
   Company (const Company& objetCopie);
   ...
private:
   ...
   vector<Employee*> employees_;
   Employee* president_;
};
```

L'attribut est un pointeur.

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.

```
Company::Company(const Company& company)
   : name_(company.name_), employees_(company.employees_),
     president_(nullptr)
{
   president_ = new Employee(company.president_->getName(),
                             company.president_->getSalary());
}
```

On alloue un nouvel espace sur le tas.

Constructeur de copie – deep (alternative)

```
class Company
{
public:
   Company ();
   Company (const Company& objetCopie);
   ...
private:
   ...
   vector<Employee*> employees_;
   Employee* president_;
};

Company::Company(const Company& company)
   : name_(company.name_), employees_(company.employees_),
     president_(nullptr)
{
   president_ = new Employee(*(company.president_));
}
```

Si la classe Employee 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 =:

```
Company objet1;
Company 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 Company, comme on a dû le faire pour le constructeur de copie.

Exemple de définition de l'opérateur =

```
Company& Company::operator=(const Company& company)
{

    delete president_;

    president_ = new Employee(*(company.president_));
    name_ = company.name_;
    employees_ = company.employees_;

    return *this;
}
```

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

On retourne une référence à l'objet parce que l'opérateur = peut être appelé en cascade:
c1 = c2 = c3 (qui est équivalent à **c1 = (c2 = c3)**)

Exemple de définition de l'opérateur =

```
Company& Company::operator=(const Company& company)
{
   if (this != &company) {
        delete president_;

    president_ = new Employee(*(company.president_));
    name_ = company.name_;
    employees_ = company.employees_;
   }
   return *this;
}
```

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

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