Pointeur this

Le pointeur **this** est un pointeur sur l'objet courant. Toutes les méthodes reçoivent ce pointeur en paramètre de manière implicite. Ainsi, si on prend l'exemple de la méthode setName() de la classe Employee :

```
void Employee::setName(string name) {
    name_ = name;
}
```

Une manière équivalente d'écrire cette fonction serait :

```
void Employee::setName(string name) {
    this->name_ = name;
}
```

Car c'est un pointeur.

Notez qu'on fait appel à l'opérateur d'accès (->) vu précédemment, et non pas à (.), puisqu'il faut déréférencer le pointeur **this** avant d'accéder à l'attribut name_ de l'objet courant. Supposons maintenant que nous voulions appeler plusieurs fois consécutives une méthode d'un objet. Imaginons par exemple une méthode incrementer(), que l'on veut appliquer plusieurs fois :

```
int main() {
    ClassA objet;
    objet.incrementer();
    objet.incrementer();
    objet.incrementer();
}
```

Si la méthode nous retournait une référence au même objet, le même effet pourrait être obtenu en faisant des appels en cascade:

```
int main() {
    ClasseA objet;
    objet.incrementer().incrementer().incrementer();
}
```

Le pointeur **this** est particulièrement intéressant lorsqu'on veut faire ce genre de manipulations puisqu'il nous permet de retourner une référence vers l'objet courant. La définition et l'implémentation de la classe ClasseA serait alors :

```
class ClasseA {
public:
    ClasseA();
    ClasseA& incrementer();
private:
    int att_;
};

ClasseA::ClasseA(): att_(0) {}
```

Reference vers lui même appel en cascade

```
ClasseA& ClasseA::incrementer(){
    ++att_;
    return *this;
}
```

Programmation orientée objet

Introduction à la surcharge des opérateurs et fonctions

Surcharge de fonctions

- Il y a surcharge de fonction lorsqu'on définit et implémente une nouvelle fonction qui a le même nom qu'une fonction existante, mais dont:
  - Le nombre de paramètres est différent **et/ou**
  - Le type d'un ou plusieurs paramètres est différent
- Certains types sont susceptibles à la conversion automatique (« casts »), il faut donc faire bien attention

Exemples

Attention: Le type de retour **n'est pas** un critère qui satisfait la surcharge de fonctions

```
void f(int a) {}

void f(int a, double b) {}

void f(int a, double b, int c) {}

void f(int a, double b, string s) {}

void f(int a, double b, Fraction f) {}

void f(int a, double b, Point p) {}
```

Le nombre de paramètres est différent

Le type du dernier paramètre est différent

Surcharges d'opérateurs

- Consiste **à redéfinir la fonctionnalité d'un opérateur** tel que +, -, ou += pour une classe.
- Étant donné qu'on fait de la surcharge de fonctions, **on ne peut pas créer de nouveaux opérateurs**, on se contente de redéfinir ceux existants
- **L'ordre de priorité des opérateurs est conservé**, il faut donc bien comprendre le fonctionnement d'un opérateur avant de le surcharger

Opérateurs définis en C++

cppreference - surcharge des opérateurs

Pouvant être surchargés	Ne pouvant être surchargés
+ - * / % ^ & | [] () ~ = < > += -= *= /= %= ^= &= ! ++ -- -> , >> << && != <= >= |= || <<= >>= == new delete	. :: ?: .* sizeof

Types d'opérateurs

- Les opérateurs ont un **nombre déterminé de paramètres** et ne peuvent pas avoir de paramètres par défaut
- Les **opérateurs binaires** acceptent deux paramètres qui sont, dans l'ordre, l'opérande de gauche et l'opérande de droite. **Les opérateurs ne sont donc pas commutatifs**. L'opérateur d'addition (+) est binaire 2 objets.
- Les **opérateurs unaires** acceptent un seul paramètre qui est l'objet courant. L'opérateur d'incrémentation (++) est unaire

Programmation orientée objet

Surcharge interne des opérateurs

La surcharge interne

- Il est possible de surcharger un opérateur en tant que **fonction membre**
- Dans ce cas, **l'opérande de gauche est toujours l'objet courant**
- La surcharge de l'opérateur prendra alors comme paramètre l'opérande de droite s'il y a lieu
- Ce type de surcharge doit être **priorisé lorsqu'il est possible de le faire** puisqu'elle respecte l'encapsulation de la classe

Exemple - Surcharge des opérateurs de Fraction

- On veut pouvoir faire:

```
int main() {
    Fraction f1(1, 2), f2(4, 5), f3;
    f3 = f1 + f2;
}
```

↳ Operande

- Un appel équivalent serait:

```
int main() {
    Fraction f1(1, 2), f2(4, 5), f3;
    f3 = f1.operator+(f2);
}
```

Exemple - Surcharge de l'opérateur + pour Fraction

```
class Fraction{
public:
    Fraction();
    Fraction(const double& num, const double& denum);

    Fraction operator+ (const Fraction& fract) const;

private:
    long numerateur_;
    long denominateur_;
    void simplifier();
};
```

Const parce qu'on modifie pas le parametre (operande) mais plutôt l'operateur.

Exemple - Surcharge de l'opérateur + pour Fraction

```
Fraction Fraction::operator+(const Fraction& fract) const {
    Fraction somme;

    somme.numerateur_ = numerateur_ * fract.denominateur_ +
                fract.numerateur_ * denominateur_;

    somme.denominateur_ = denominateur_ * fract.denominateur_;

    somme.simplifier();

    return somme;
}
```

Exemple - Surcharge de l'opérateur + de Fraction

- On veut pouvoir faire:

```
int main() {
    Fraction f1(1, 2), f2(4, 5), f3;
    f3 = f1 + 1;
}
```

- Un appel équivalent serait:

```
int main() {
    Fraction f1(1, 2), f2(4, 5), f3;
    f3 = f1.operator+(1);
}
```

Exemple - Surcharge de l'opérateur + pour Fraction

```
class Fraction{
public:
    Fraction();
    Fraction(const double& num, const double& denum);

    Fraction operator+ (const Fraction& fract) const;
    Fraction operator+ (const long& entier) const;

private:
    long numerateur_;
    long denominateur_;
    void simplifier();
};
```

La signature est la même, sauf pour le type de l'opérande de droite

Exemple - Surcharge de l'opérateur ++ de Fraction

- On veut pouvoir faire:

```
int main() {
   Fraction f1(1, 2), f2(4, 5), f3;
   ++f1;
}
```

- Un appel équivalent serait:

```
int main() {
   Fraction f1(1, 2), f2(4, 5), f3;
   f1.operator++();
}
```

Exemple - Surcharge des opérateurs de Fraction

```
class Fraction{
public:
    Fraction();
    Fraction(const double& num, const double& denum);

    Fraction operator+(const Fraction& fract) const;
    Fraction operator+(const long& entier) const;
    Fraction& operator++();
private:
    long numerateur_;
    long denominateur_;
    void simplifier();
};
```

Référence vers le même objet ce qui permet un appel en cascade

Parce qu'on modifie l'état de l'objet et qu'on retourne une référence vers celui-ci

Exemple - Surcharge des opérateurs de Fraction

```
Fraction& Fraction::operator++() {
   *this = *this + 1;
   return *this;
}
```

Cette façon est plus facile à comprendre et donc préferable à l'autre implémentation

```
Fraction& Fraction::operator++() {
   *this = operator+(1);
   return *this;
}
```

Exemple - Surcharge de l'opérateur + de Fraction

- On veut pouvoir faire:

```
int main() {
    Fraction f1(1, 2), f2(4, 5), f3;
    f3 = 1 + f1;
}
```

- Un appel équivalent serait:

```
int main() {
    Fraction f1(1, 2), f2(4, 5), f3;
    f3 = 1.operator+(f1);
}
```

On peut seulement surchargé les opérateurs pour les classes que l'on définit

Exemple - Surcharge de l'opérateur + de Fraction

- On veut pouvoir faire:

```
int main() {
    Fraction f1(1, 2), f2(4, 5), f3;
    f3 = 1 + f1;
}
```

- Un appel équivalent serait:

```
int main() {
    Fraction f1(1, 2), f2(4, 5), f3;
    f3 = operator+(1, f1);
}
```

On doit surcharger l'opérateur + de manière externe

Programmation orientée objet

Surcharge externe des opérateurs

Surcharge externe

- Il est possible de surcharger un opérateur en tant que **fonction globale**
- On surcharge les opérateurs de manière externe **lorsque l'opérande de gauche n'est pas un objet de la classe courante**
- Lorsque l'opérateur est surchargé en fonction globale standard, toutes les fonctions nécessaires pour manipuler l'objet doivent être définies dans l'interface de sa classe puisque la fonction n'est pas membre

Exemple - Surcharge de l'opérateur + de Fraction

- On veut pouvoir faire:

```
int main() {
   Fraction f1(1, 2), f2(4, 5), f3;
   f3 = 1 + f1;
}
```

- Un appel équivalent serait:

```
int main() {
   Fraction f1(1, 2), f2(4, 5), f3;
   f3 = operator+(1, f1);
}
```

Exemple – Surcharge de l'opérateur + de Fraction

```
class Fraction{
public:
    Fraction operator+(const Fraction& fract) const;
    Fraction operator+(const long& entier) const;

private:
    long numerateur_;
    long denominateur_;
    void simplifier();
};
```

fonction globale qui se fait à l'extérieur de la classe.

```
Fraction operator+(const long&, const Fraction&);

Fraction operator+(const long& e, const Fraction& f) {
        return (f + e);
}
```

Parce qu'une fonction globale ne peut pas être constante

L'opérateur + fait partie de l'interface de Fraction

Exemple - Surcharge de l'opérateur << de Fraction

- On veut pouvoir faire:

```
int main() {
   Fraction f1(1, 2), f2(4, 5), f3;
   cout << f1 << endl;
}
```

- Un appel équivalent serait:

```
int main() {
   Fraction f1(1, 2), f2(4, 5), f3;
   operator<<(cout, f1) << endl;
}
```

Exemple - Surcharge de l'opérateur << de Fraction

```
class Fraction{
public:
    Fraction();
    Fraction(const double& num, const double& denum);

private:
    long numerateur_;
    long denominateur_;
    void simplifier();
};

ostream& operator<<(ostream& o, const Fraction& f) {
        return o << f.getNumerateur() << "/" <<
                f.getDenominateur();
}
```

Non, on surcharge l'opérateur << en fonction friend

Fonction friend

- Une fonction friend est une **fonction globale** qui a un **accès privilégié aux membres privés** d'une classe
- Ce concept ne détruit pas l'encapsulation puisque:
  - une **classe contrôle** quelle fonctions deviennent friend
  - les opérateurs font partie de l'interface et alors doivent être inclus dans la définition de la classe
  - c'est plus fiable qu'ajouter des accesseurs publiques juste pour l'opérateur

Exemple - Surcharge de l'opérateur << de Fraction

```
class Fraction{
public:
    Fraction();
    Fraction(const double& num, const double& denum);

    friend ostream& operator<<(ostream& o, const Fraction& f);
private:
    long numerateur_;
    long denominateur_;
    void simplifier();
};

ostream& operator<<(ostream& o, const Fraction& f) {
        return o << f.numerateur_ << "/" << f.denominateur_;
}
```

Parce qu'une fonction globale ne peut pas être constante

friend permet alors d'accéder de façon privilégiée aux attributs de l'objet