http://en.cppreference.com/w/cpp/language/operator_precedence

Surcharger les opérateurs

La sémantique de valeur autorise la comparaison entre les objets. Cela signifie qu'il peut être licite d'écrire :

```
i == j;
j < k;
// etc.
```

Il n'est pas forcement pertinent de créer tous les opérateurs (une classe peut être comparable par égalité sans être comparable par plus-petit-que par exemple), mais lorsque l'on définit certains opérateurs, il est logique de définir certains autres opérateurs :

- égalité == et différent !=
- < et >, ≤ et ≥
- +a et -a
- + et +=, - et -=, * et *=, / et /=, % et %=
- ++a et a++, –a et a–
- !, && et ||
- ~, &, |, «, » et ^

A voir, pour chaque classe, s'il est pertinent ou non de créer ces opérateurs.

Opérateurs libres et membres

Les opérateurs sont des fonctions, qui ont une signature particulière, définie dans la norme. Le plus simple est de lire la doc : http://en.cppreference.com/w/cpp/language/expressions#Operators

Comme pour n'importe quelle fonction, il est possible d'avoir une version

membre et une version libre. La différence avec les autres fonctions “classiques” est que cela ne change pas la façon dont on appelle l'opérateur.

```
struct A {
    void foo(); // membre
};
void foo(A a);  // libre

A a {};
a.foo(); // membre
foo(a);  // libre


struct A {
    bool operator==(A a);
};

A a {};
A b {};
a == b; // appel de == membre

struct A {
};
bool operator==(A a, b b);

A a {};
A b {};
a == b; // appel de == libre
```

Avec la fonction membre, l'opérateur s'applique sur l'objet appelé (this) et sur l'objet passé en argument. Avec la fonction libre, comparaison des objets passé en argument

Attention sur la syntaxe. Si on sépare la définition et l'implémentation de la fonction membre, on va écrire :

```
struct A {
    bool operator==(A a);
};
```

```
bool A::operator==(A a) { ... }
```

Il faut bien faire attention à ne pas confondre la fonction membre A::operator et la fonction operator (sans le spécificateur de portée A::). La première version (membre) s'applique sur l'objet courant et n'a donc besoin que d'une seul paramètre de fonction `A::operator==(A a)`. Le second doit prendre deux argument `operator==(A a, A b)`.

On peut être tenté de déclarer les 2 versions d'un opérateur :

```
struct A {
    bool operator==(A a);   // membre
};
bool operator==(A a, b b);  // libre

A a {};
A b {};
a == b; // erreur appel de quelle version ?
```

Cependant, comme l'appel de ces 2 fonctions est identique, le compilateur ne sait pas quelle version appelée et produit une erreur :

```
main.cpp:17:21: error: use of overloaded operator '==' is
ambiguous
(with operand types 'A' and 'A')
    std::cout << (a == b) << std::endl;
                  ~ ^  ~
main.cpp:5:10: note: candidate function
    bool operator==(A a) {
         ^
main.cpp:10:6: note: candidate function
bool operator==(A a, A b) {
     ^
1 error generated.
```

Il faut donc donner qu'une seule version.

Quelle version donner ? Ecrivons l'opérateur == pour comparer avec un int :

On peut être tenté de déclarer les 2 versions d'un opérateur :

```
#include <iostream>

struct A {
    int i {};
    bool operator==(int j) {
        return this->i == j;
    }
};
/*
bool operator==(A a, int i) {
    return a.i == i;
}
*/
int main() {
    A a { 123 };
    int i { 456 };
    std::cout << std::boolalpha << (a == i) << std::endl;
    std::cout << std::boolalpha << (i == a) << std::endl;
}
```

Avec l'opérateur libre, le compilateur est capable d'appeler aussi bien a == i que i == a, alors que la version membre nécessite que A soit le premier argument.

Il est donc plus intéressant d'écrire la version libre... sauf qu'il faut que l'opérateur puisse accéder aux variables membres private de la classe (comme ce n'est pas une fonction membre, elle n'a accès qu'en membres publiques).

A voir donc en fonction du design, si cela à un sens d'écrire == dans les 2 sens ou s'il faut accéder aux membres.

Pour garantir que le comportement est cohérent, écrire un opérateur en utilisant l'autre (par exemple définir != à partir de == et !)

La signatures des opérateurs

en annexe ?

Déjà utilisé. Et écrit dans les prédicat

Opérateurs de comparaison

Operator name	Syntax	Overloadable	Prototype examples (for class T)	
			Inside class definition	Outside class definition
equal to	a == b	Yes	bool T::operator ==(const T2 &b) const;	bool operator ==(const T &a, const T2 &b);
not equal to	a != b	Yes	bool T::operator !=(const T2 &b) const;	bool operator !=(const T &a, const T2 &b);
less than	a < b	Yes	bool T::operator <(const T2 &b) const;	bool operator <(const T &a, const T2 &b);
greater than	a > b	Yes	bool T::operator >(const T2 &b) const;	bool operator >(const T &a, const T2 &b);
less than or equal to	a ⇐ b	Yes	bool T::operator ⇐(const T2 &b) const;	bool operator ⇐(const T &a, const T2 &b);
greater than or equal to	a >= b	Yes	bool T::operator >=(const T2 &b) const;	bool operator >=(const T &a, const T2 &b);

```
inline bool operator< (const X& lhs, const X& rhs){ /* do
actual comparison */ }
inline bool operator> (const X& lhs, const X& rhs){return
rhs < lhs;}
inline bool operator<=(const X& lhs, const X& rhs){return !(
lhs > rhs);}
inline bool operator>=(const X& lhs, const X& rhs){return !(
lhs < rhs);}

inline bool operator==(const X& lhs, const X& rhs){ /* do
actual comparison */ }
inline bool operator!=(const X& lhs, const X& rhs){return !(
lhs == rhs);}
```

Opérateurs arithmétiques

```
class X {
 public:
  X& operator+=(const X& rhs) // compound assignment (does
not need to be a member,
  {                           // but often is, to modify the
private members)
    /* addition of rhs to *this takes place here */
    return *this; // return the result by reference
  }

  // friends defined inside class body are inline and are
hidden from non-ADL lookup
  friend X operator+(X lhs,       // passing first arg by
value helps optimize chained a+b+c
                     const X& rhs) // alternatively, both
parameters may be const references.
  {
     return lhs += rhs; // reuse compound assignment and
return the result by value
  }
};
```

Les opérateurs d'incrémentation et décrémentation

++ et –. Deux version, en préfixe et postfixe. Quand on écrit operator++, il faut savoir si on défini ++a ou a++. Pour régler ce problème, on ajoute un paramètre qui ne sert pas dans l'opération :

```
struct X {
    X& operator++() {
        // actual increment takes place here
        return *this;
    }
    X operator++(int) {
        X tmp(*this); // copy
        operator++(); // pre-increment
        return tmp;   // return old value
    }
};
```

On comprend mieux ici pourquoi il est préférable d'utiliser ++a par défaut : cela évite une copie

Idem pour –

Remarque : cela peut avoir un sens de déclarer ++, mais pas –. Voir pas exemple BidirectionalIterator et ForwardIterator

Les opérateurs de flux

Utile pour écrire cout « a « endl; Utilisation de iostream :

```
std::ostream& operator<<(std::ostream& os, const T& obj)
{
  // write obj to stream
  return os;
}
std::istream& operator>>(std::istream& is, T& obj)
{
  // read obj from stream
  if( /* T could not be constructed */ )
```

```
    is.setstate(std::ios::failbit);
  return is;
}
```

Opérateurs particuliers

- Array subscript operator

```
struct T {
    value_t& operator[](std::size_t idx) {
         /* actual access, e.g. return mVector[idx]; */
    };
    const value_t& operator[](std::size_t idx) const {
        // either actual access, or reuse non-const overload
        // for example, as follows:
        return const_cast<T&>(*this)[idx];
    };
};
```


Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++