Créer et copier des objets

Fonctions membres qui ont une signature définie par la norme. Constructeur, porte le même nom que la classe, destructeur avec ~ en plus, sans retour de fonction. Opérateur : avec le mot clé `operator`.

```
class A {
    A();                        // exemple de constructeur
    ~A();                       // exemple de destructeur
    A& operator= (A const&); // exemple d'opérateur
};
```

Par défaut, génération de plusieurs fonctions automatiquement par le compilateur. Si on écrit un constructeur, désactivation des versions par défaut.

On peut expliciter que l'on souhaite utiliser la version générée automatiquement par le compilateur, avec `default`. Ou au contraire interdire avec `delete`.

```
class A {
    A() = default;

    A(A const&) = delete;
    A& operator= (A const&) = delete;
};
```

Le constructeur par défaut

Constructeur qui peut être appelé sans argument. Donc soit sans paramètre, soit avec que des paramètres par défaut. Correspond au concept Default Constructible.

```
struct A {
    A();
```

```
};

struct B {
    B(int i = 0);
};

A a {}; // ok
B b {}; // ok
```

Désactiveation :

```
struct A {
    A(int i);
};

A a {}; // erreur, A() est désactivé
```

Pour forcer la création automatique par le compilateur, utilisation de `default`. On peut également supprimer avec `delete`.

```
struct A {
    A() = default; // on utilise la version générée
automatiquement par le compilateur
    A(int i);
    ~A() = delete; // suppression du destructeur
};
```

Par défaut :

- constructeur par défaut `A()`
- copie `A(A const&)` et `operator=(A const&)`
- destructeur `~A()`

explicit et constexpr

delagating constructor

big 3, 5 et 0 rules : les déclarations vont ensemble

liste d'initialisation, initialisation de variables

pointeur this

Copie

Permet de créer un objet par copie d'un autre objet de même type

```
A a {};
A b { a }; // b est une copie de a
b = a;     // b est une copie de a
```

Remarque, quand on écrit :

```
A b = a;
```

Le compilateur comprend qu'il s'agit d'une initialisation et non d'une affectation, le code est remplacé par une appel de constructeur de copie plutôt que d'appeler l'affectation.

```
A b = a;

// est équivalent à
A b { a };

// pas équivalent à
A b {};
b = a;
```

Copie explicite

```
class A {
public:
    A() = default; // défaut
    A(A const&);
    A& operator=(A const&);
};
```

Par exemple :

```
class A {
    int i {};
```

```
    int j {};
public:
    A() = default; // défaut
    A(A const&);
    A& operator=(A const&);
};

A::A(A const& a) : i(a.i), j(a.j) {}

A& A::operator=(A const& a) {
    i = a.i;
    j = a.j;
    return *this;
}
```

portée, namelookup et utilisation de this→

copy and swap idiom : http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Copy-and-swap. Utilisation de copie plutôt que référence constante :

```
A& A::operator=(A a) {
    swap(*this, a);
    return *this;
}

T& T::operator=(T arg) { // copy/move constructor is called
to construct arg
    swap(arg);    // resources exchanged between *this and
arg
    return *this;
}  // destructor is called to release the resources formerly
held by *this

T& operator=(const T& other) // copy assignment
{
    if (this != &other) { // self-assignment check expected
        if (/* storage cannot be reused (e.g. different
sizes) */)
        {
            delete[] mArray;            // destroy storage
```

```
in this
            /* reset size to zero and mArray to null, in
case allocation throws */
            mArray = new int[/*size*/]; // create storage in
this
        }
        /* copy data from other's storage to this storage */
    }
    return *this;
}
T& operator=(T&& other) // move assignment
{
    assert(this != &other); // self-assignment check not
required
    delete[] mArray;        // delete this storage
    mArray = other.mArray;  // move
    other.mArray = nullptr; // leave moved-from in valid
state
    return *this;
}
```

comment écrire swap ?

passage par référence, référence const ou copie ?

Déplacement

Similaire à copie, avec rvalue référence. Quand sont-ils utilisés ?

```
class A {
    A(A&&);
    A& operator=(A&&);
};
```

(remarque : contrairement à &, cela n'a pas de sens de mettre const à &&)

utilisation :

```
A f();
```

```
A a { f() };
a = f();
```

Exemple classique : implémentation pour savoir qui est appelé

```
A::A(A&&) {
    std::cout << "A::A(A&&) id=" << ++id << std::endl;
}

A& A::operator=(A&&) {
    std::cout << "A& A::operator=(A&&) id=" << id << std::
endl;
    return *this;
}
```

Liste de valeurs

Initialisation à partir d'une liste du type { 1, 2, 3, 4 }. Utile en particulier si sémantique de conteneur

```
class A {
    vector<int> v {};
public:
    A(std::initializer_list<T> init);
    A& operator=(std::initializer_list<T> init);
};

A::A(std::initializer_list<T> init) : v(init) {
    // facile avec vector
}

A& A::operator=(std::initializer_list<T> init) {
    v = init;
    return *this;
}
// ou copy ou assign

A a {{ 1, 2, 3, 4, 5 }};
```

```
A b = { 1, 2, 3, 4, 5 };
```

Résumé : liste des constructeurs, destructeur et opérateurs

Implémentation “pour suivre ce qui se passe”.

```
class A {
    // constructeur
    A(); // par défaut
    A(int i, int j); // avec paramètres

    // destructeur
    ~A();

    // copie
    A(A const&);
    A& operator= (A const&);

    // move
    A(A &&);
    A& operator= (A &&);
};
```


Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++