Indirections dans les fonctions

Plusieurs types d'indirections :

- iterateurs → pour les collections
- references → vu dans les parametres de fonctions
  - lvalue ref
  - rvalue ref
  - reference_wrapper
- pointeurs → polymorphisme + peut etre nullptr
  - intelligent
    - unique_ptr (+ observer_ptr?)
    - shared_ptr + weak_ptr
    - autres... (boost, Qt, etc)
  - non manages
    - raw ptr
    - not_null<T>

Et bien d'autres... (old C++, future C++, libs, etc)

Comment ecrire une fonction qui prend une indirection en parametre ?

```
void f(...);  // quoi mettre ici ?

int* pi;
f(~i~);

std::unique_ptr<int> upi;
f(~upi~);


std::shared_ptr<int> spi;
f(~spi~);
```

Cas particulier des tableaux

Type d'element different a gerer, mais egalement type de collection.

```
void f(...);  // quoi mettre ici ?

std::vector<int> vi;
std::vector<string> vd;
std::list<int> li;
```

Dans certains cas, cela a un sens de pouvoir utiliser des collection d'elements de types differents (par exemple std::find fonctionne sur int ou string), dans d'autres cas non.

Idem pour les types de collections, parfois cela a un sens (par exemple std::find fonctionne sur vector et list) et d'autres cas non (std::sort fonctionne que sur std::vector)

A choisir selon l'API que l'on veut fournir.

Passer par valeur ou reference la collection

Passage dans une fonction :

```
void f(TYPE value);
void f(TYPE & ref);
void f(TYPE const& cref);
void f(TYPE && rref);
```

Avec TYPE = n'importe quel type compatible (ie copiable si par valeur). En particulier, possible que le type soit une collection. Par exemple si TYPE = std::vector<int>

```
void f(std::vector<int> const& cref);

std::vector<int> v;
f(v); // ok
```

Code moins reutilisable → specifique d'un type de collection et d'un type d'element.

```
void f(std::vector<int> const& cref);

std::list<int> l;
f(l); // erreur
```

Version generique avec template

Rappel fonction generique :

```
template<class T>
void f(T value);

template<class T>
void f(T & ref);

template<class T>
void f(T const& cref);

template<class T>
void f(T && rref);
```

Meme chose que precedent, mais le type est deduit pas le compilateur = fonctionne avec n'importe quel type de collection.

```
template<class T>
void f(T const& cref);

std::vector<int> v;
f(v); // ok

std::list<double> l;
f(l); // ok
```

Limitation : uniquement sur totalite de collection, pas une sous partie.

```
template<class T>
```

```
void f(T const& cref);

std::vector<int> v;
const it { std::find(begin(v), end(v), 123) };
f(...); // comment appliquer f que entre it et end(v) ?
```

Paire d'iterateurs

Utilise par les algos stl. Passe en parametre une paire d'iterateurs, deduit par template (genericite).

```
template<Iterator>
void f(Iterator first, Iterator last);

std::vector<int> v;
f(begin(v), end(v)); // ok

std::list<double> l;
f(begin(l), end(l)); // ok

const it { std::find(begin(v), end(v), 123) };
f(it, end(v));       // ok
```

Le plus generique !

Inconvenient = possible d'utiliser des iterateurs provenant de collection differentes + ecriture lourde.

```
template<Iterator>
void f(Iterator first, Iterator last);

std::vector<int> v;
std::list<double> l;
f(begin(v), end(l)); // erreur
```

Les views

Fondamentalement, paire d'iterateur sur une collection. Garantie que les 2 iterateurs viennent d'une meme collection. + ecriture plus simple.

Conceptuellement : semantique d'une collection, mais pas l'ownership. le sera manipule et vu par le code comme etant une collection. Mais ne gere pas les donnees elle meme, utilise une autre collection qui gere les donnees.

La vues peut correspondre a sous partie d'une collection, peut caster les types, et peut proposer d'autres fonctionnalites (par exemple voir un tableau 1D comme un tableau 2D).

Sous collection :

```
const std::vector<int> v { 1, 2, 3, 4, 5 };
const array_view<int> view (begin(v) + 1, end(v) - 1);
std::find(begin(view), end(view), 3); // view s'utilise
comme un tableau contenant { 2, 3, 4 }
```

Cast : (danger...)

```
static_assert(sizeof(uint32_t) == sizeof(float));
const std::vector<uint32_t> v { 1, 2, 3, 4, 5 };
const array_view<float> view (begin(v), end(v));
```

Tableau 2D

```
array_view<int, 2> view ({2, 5}, v1}); // 2D
view[{1, 2}] = 5;                      // accès 2D
```

Note : specialisation string_view pour les chaines.

Note 2 : dans le C++17. A implementer soi meme.

Les indirections sur un objet

Solution 1 : faire le bourrin

Ecrire des surcharges de fonction pour chaque type de parametre possible...

```
f(int);
f(std::unique_ptr<int>);
f(std::shared_ptr<int>);
```

Pas du tout evolutif et maintainable.

Fonction generique

Utilisation de template pour passer les indirections.

```
template<class T>
void f(T const& value);

int* pi;
f(pi);

std::unique_ptr<int> upi;
f(upi);


std::shared_ptr<int> spi;
f(spi);
```

Probleme 1 : acces aux objets. Selon le type d'indirection (ref vs pointeur), acces via l'operateur `.` ou `->`. Besoin de surcharger selon les 2 cas.

```
template<class T>
void f_ref(T const& value) {  // T est une ref
    value.do_something();
}

template<class T>
```

```
void f_ptr(T const& value) {  // T est un pointeur
    value->do_something();
}
```

Probleme 2 : semantique des pointeurs = peut etre nullptr. Donc necessite de verifier les acces avant utilisation. Pas necessaire avec ref.

```
template<class T>
void f_ref(T const& value) {  // T est une ref
    value.do_something();
}

template<class T>
void f_ptr(T const& value) {  // T est un pointeur
    assert(value);            // check ptr
    value->do_something();
}
```

Probleme 3 : gerer l'ownership. Certains n'ont pas l'ownership (ref, raw ptr, weak_ptr, etc) et d'autre oui (unique_ptr, shared_ptr, etc). Difficile de gerer cela avec les template.

Par exemple, si on passe un weak_ptr, il faut locker avant d'utiliser.

```
template<class T>
void f_ptr(T const& sp) {  // T est un weak_ptr
    auto sp { wp.lock() }; // lock en shared_ptr
    assert(sp);            // check ptr
    sp->do_something();
}
```

Autre exemple, si on veut transmettre l'ownership a une fonction :

```
std::unique_ptr<int> up;
f(std::move(up));  // on n'a plus besoin de up ensuite
```

Conclusion : generique, mais trop de cas particulier a gerer. (+ tous les libs possible)

Les references comme parametre universel

Conserver qu'une seule forme : passage par reference.

```
// T est un objet, pas une indirection. Template ou non

void f(T const& ref);  // objet non mutable
void g(T& ref);        // objet mutable
```

Reference apporte les garanties : que l'objet est valide et que la fonction n'a pas l'ownership. Rien a tester dans la fonction et 1 syntaxe a ecrire.

Ensuite, creer des adaptateurs pour chaque type d'indirection, via lambda.

Pour une valeur :

```
int i;
f(i); // appel direct
```

Pour shared_ptr :

```
std::shared_ptr<int> spi;

f(*spi);

[spi](){ f(*spi); }; // asynchrone
```

Pour weak_ptr :

```
std::weak_ptr<int> wpi;

[spi = wpi.lock()](){ f(*spi); }; // lock lors de la capture

[wpi](){ auto spi { wpi.lock(); if (spi) { f(*spi); } }; //
lock lors de l'appel a f
```

Pour unique_ptr :

```
std::unique_ptr<int> upi;
```

```
f(*upi);  // appel direct

[p = std::move(upi)](){ f(*p); }; // asynchrone
```

Pour raw ptr :

```
int* pi;

if (pi) { f(*pi); }  // appel direct... dangling possible

[pi](){ if (pi) { f(*pi); } }; // asynchrone... dangling
possible
```

Not safe, risque de dangling.

etc.

Cas particulier des pointeurs nus

Raw ptr ont semantique “peut etre nullptr”. Generalement, le traitement effectue par f pourra se decomposer en 2 partie strictement distincte, selon si nullptr ou non. Il est donc dans ce cas possible de gerer le cas de nullptr en dehors de f et d'utiliser les references (ne peut pas etre nullptr).

```
void f_nullptr();
void f_notnull(T const& ref);

int* pi;
if (pi) {
    f_notnull(pi);
} else {
    f_nullptr();
}
```

Pas besoin de passer directement un pointeur, ref est “universel”. (les cas ou on doit passer un pointeur doivent rester rare).