Copies, déplacements et indirections

Dans le chapitre précédent, vous avez vu comment échanger des informations avec une fonction, en utilisant les paramètres de fonction et le retour de fonction.

Le type d'échange que vous avez vu s'appelle un passage par valeur. Cela consiste a partir de l'objet qui se trouve dans la fonction appelante, et d'en faire une copie qui sera utilisable dans la fonction appelée.

```
void f(int i) {
    // i est une nouvelle "variable", accessible uniquement
dans la fonction f
    // et qui contient la même valeur que la variable j de
la fonction g.
}

void g() {
    const int j { 123 };
    f(j);
}
```

La valeur est copiée lors de l'appel de la fonction f, ce qui implique que les éventuelles modifications du paramètre `i` ne seront pas répercutées sur la variable `j`.

Il existe d'autres façon de transmettre une information dans une fonction, il convient donc de détailler d'abord les concepts de copie, de déplacement et d'indirection.

Copie et déplacement

La copie d'objets

La copie consiste donc a creer un nouvel objet, identique a un objet existant. Ces deux objets seront indépendants, c'est a dire que si l'un des objets est modifié ou détruit, l'autre objet ne sera pas modifie.

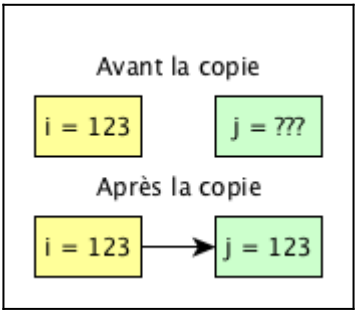
main.cpp

```
#include <iostream>

int main() {
    const int i { 123 };
    int j { i };  // j est une copie de i, elle contient la même valeur
    std::cout << "i=" << i << ", j=" << j << std::endl;
    j = 456;
    std::cout << "i=" << i << ", j=" << j << std::endl;
}
```

affiche :

```
i=123, j=123
i=123, j=456
```



Tous les objets ne sont pas copiables. Les types fondamentaux (`int`, `double`, `float`, etc) sont copiables, ainsi que la très grande majorité des classes de la bibliothèque standard.

Souvenez vous, cela a été abordé dans le chapitre Les

fonctionnalités de base des collections, la majorité des classes de la bibliothèque standard possèdent une sémantique de valeur et sont copiables. Un exemple de classe non copiable est `std::unique_ptr`. Cela sera détaillé dans la partie sur la programmation objet.

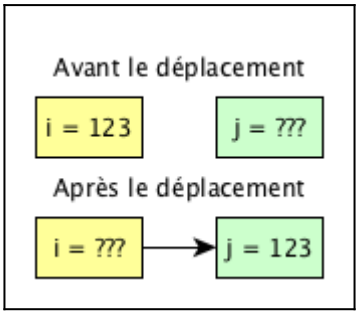
Le deplacement d'objets

Le déplacement d'objets consiste a déplacer (*move*) un objet depuis une variable vers une autre. L'objet n'est pas modifié dans cette opération, il est conservé a l'identique. Cette opération peut être réalisée en utilisant la fonction `std::move`.

main.cpp

```
#include <iostream>

int main() {
    int i { 123 };
    int j { std::move(i) };
    std::cout << "j=" << j << std::endl;
}
```



Cette notion a aussi ete vu rapidement dans Les fonctionnalités de base des collections et sera detaille dans la partie sur la programmation objet.

Contrairement a la copie qui permet d'obtenir deux objets au final, le deplacement ne modifie pas le nombre d'objets, il y a toujours un seul objet valide apres l'operation. La variable qui contenait l'objet initialement contient ensuite un objet invalide qui ne doit jamais etre utilise (cela produirait un comportement indefini). La variable ne peut etre utilisee que pour lui affecter un nouvel objet.

main.cpp

```
#include <iostream>

int main() {
    int i { 123 };
    int j { std::move(i) };
    std::cout << "j=" << j << std::endl; // interdit
d'utiliser i ici
    i = 456;
    std::cout << "i=" << i << ", j=" << j << std::endl; //
ok
}
```

Tous les objets ne sont pas deplacable (*movable*). La copie et le deplacement sont independants, il est donc possible d'avoir des objets copiables et deplacables, des objets deplacables et non copiables, ou des objets non copiables et non deplacables.

Le deplacement est parfois considere comme une copie, et donc cela n'a pas de sens d'avoir un objet copiable, mais non deplacable. C'est possible syntaxiquement de faire cela, mais cela sera considere comme une erreur de conception dans ce cours.
En pratique, ce qui se passe reellement lors d'un deplacement est un peu complexe. Dans certains cas (par exemple avec les types fondamentaux), une copie sera realisee. Dans d'autres cas (par exemple avec `std::string` ou `std::vector`), les donnees ne sont reellement pas copiees et le deplacement est plus performant que la copie. Dans tous les cas, vous pouvez retenir que le deplacement ne sera jamais plus couteux que la copie (au pire, il peut etre equivalent a une copie).

La fonction `std::move` ne realise donc pas a proprement parle un deplacement, mais dit au compilateur qu'un objet peut etre deplace. Libre a lui de realiser ou non un deplacement, si c'est possible. Mais dans d'autres cas, le compilateur peut decider par lui meme qu'un objet peut etre deplace et le fera automatiquement. (C'est une optimisation automatique, puisque le deplacement sera potentiellement plus performant que la copie).

```
int f() {
    int i { 123 };
    return i;  // i ne sera plus utilise ensuite dans f et
peut etre deplace
}
```

Dans ce code, la variable `i` ne sera plus utilisable apres le `return` (puisque la fonction sera terminee) et peut donc etre deplacee vers la fonction appelante.

Les indirections

La concept d'indirection

Avec une copie ou un deplacement, vous avez dans les deux cas une variable locale a la fonction qui contient un objet. Mais il peut etre interessant aussi de pouvoir manipuler un objet qui n'est pas dans la fonction, par exemple pour modifier un objet qui se trouve dans un autre fonction ou acceder a un objet depuis plusieurs endroits du code.

```
#include <iostream>

int f(int j) {
    return j + 456;
}

int main() {
    int i { 123 };
```

```
    i = f(i);
    std::cout << i << std::endl;
}
```

affiche :

```
579
```

Dans ce code, l'objet initialement dans la variable `i` est dans un premier temps copie dans la variable `j`de la fonction `f`, puis le resultat est deplacer depuis la variable `j` de la fonction `f` vers la variable `i` de la fonction `main`. Cela fait beaucoup de manipulation d'objets.

Les indirections sont un moyen d'accéder a un objet, sans devoir faire de copie ou de deplacement. Travailler sur une indirection revient a travailler sur l'objet indirectement. Toute modification sur l'indirection sera visible dans la variable d'origine et vice-versa.

Le code precedent peut etre modifie de la facon suivante :

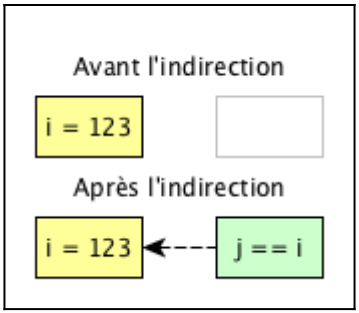
```
void f(int & j) {  // notez bien l'ajout de & ici
    j += 456;
}

int main() {
    int i { 123 };
    f(i);
    std::cout << i << std::endl;
}
```

affiche :

```
579
```

Dans ce code, la variable `j` dans la fonction `f` est une indirection (une reference) vers la variable `i` de la fonction `main`. Utiliser `j` revient a utiliser indirectement `i`, la modification realisee sur `j` est en fait realisee sur `i`.



Il n'y a pas de copie ou de deplacement dans ce code. Les indirections seront donc souvent utilisees pour eviter la copie d'objets complexes ou pour modifier un objet dans une fonction. Lorsque la fonction ne doit pas modifier l'objet, la reference sera mis en constante avec le mot-cle `const`.

```
void f(int & i) {
    i += 456;  // ok, la reference n'est pas constante
}

void g(int const& i) {
    i += 456;  // erreur, la reference est constante
}
```

affiche l'erreur suivante :

```
main.cpp:6:7: error: cannot assign to variable 'i' with
const-qualified type 'const int &'
    i += 456;
    ~ ^
main.cpp:5:19: note: variable 'i' declared const here
void g(int const& i) {
       ~~~~~~~~~~~^
```

Pour pouvez egalement trouver les syntaxes equivalentes suivantes pour ecrire une reference constante :

```
TYPE const & NOM
const TYPE & NOM
```

N'oubliez pas aussi que l'utilisation des espaces est libre en C++, vous pouvez donc trouver ces syntaxes avec ou sans espace avant et apres la

reference `&`.

La regle est que le mot-cle `const` s'applique au type qui se trouve a gauche. Et s'il n'y a rien a gauche, il s'applique a droite.

Pour comprendre une syntaxe avec reference, il est souvent plus simple de lire de la droite vers la gauche. Ainsi, la ligne `TYPE const & NOM` peut se lire : “NOM est une reference sur une constante de type TYPE”.

Classification des indirections

Les references vues precedent ne sont qu'un des nombreux types d'indirections qui existent. C'est le type d'indirection qui est le plus utilise et celui que vous devrez utiliser par defaut.

Il est meme possible de creer des classes qui representent une indirection, il y a donc potentiellement un nombre infini de types differents d'indirections. Le but de ce chapitre n'est donc pas de presenter toutes les indirections, mais uniquement de donner les caracteristiques des indirections du langage C++ et de la bibliotheque standard.

Semantiques de reference et de pointeurs

Pour commencer, il y a deux types principaux d'indirections : les indirections a semantiques de reference et les indirection a semantiques de pointeurs.

Semantiques

Une sémantique est le sens qui est donne a un concept, c'est a dire l'ensemble des operations que ce concept permet de faire.

Une “indirection a semantique de reference” est donc une indirection qui n'est pas forcement une reference, mais qui se manipulera comme une

reference. Et de meme, une “indirection a semantique de pointeur” est une indirection qui se manipule comme un pointeur, sans forcement etre un pointeur. (Les pointeurs seront vu dans la partie sur la programmation orientée objet).

Par exemple, les itérateurs que vous avez etudie avec les collections sont des indirections a semantique de pointeur, mais qui ne sont pas des pointeurs.

Les références sont des indirections qui permettent d'acceder directement a un objet. La syntaxe pour utiliser l'indirection est exactement la meme que la syntaxe pour utiliser l'objet. Dans certains cas, le compilateur peut même remplacer l'indirection par un *alias de variable*, c'est a dire utiliser directement la variable référencée, mais avec un nom different. (Le code généré par le compilateur supprimera complètement l'indirection).

```
int i { 123 };
int & j { i };

j += 456; // strictement equivalent a i += 456;
```

Les pointeurs sont des indirections qui ne permettent pas d'utiliser directement un objet. Pour accéder a l'objet, il faut d'abord appliquer une opération particulière, appellee “déréférencement”. “Déréférencer un pointeur” consiste donc a accéder a l'objet pointe par un pointeur.

Par exemple, pour utiliser un iterateur, vous avez vu dans le chapitre Les itérateurs qu'il fallait utiliser l'opérateur `*`, place devant la variable.

main.cpp

```
#include <iostream>
#include <vector>

int main() {
    const std::vector<int> v { 12, 23, 34 };
    const auto it = cbegin(v);
    std::cout << (*it) << std::endl;
}
```

affiche :

```
12
```

Dans ce code, `it` est un itérateur (une indirection) sur le premier élément de la collection et `*it` (déréférencement de l'itérateur) correspond a ce premier élément (la valeur entière 12).

Lorsque le nom d'une variable n'a pas d'importance (par exemple dans un code d'exemple ou pour des explications), il est classique de nommer une référence par `ref` et un pointeur par `ptr`, `p` ou `q`.

```
// indirection a sémantique de référence
ref = 123;
std::cout << ref << std::endl;

// indirection a sémantique de pointeur
(*ptr) = 123;
std::cout << (*it) << std::endl;
```

Les parenthèses ne sont pas forcément indispensables, l'opérateur de déréférencement `*` est prioritaire par rapport a l'opérateur d'affectation `=` et l'opérateur de flux `«`. C'est a dire que le code `*ptr = 123;` sera interprété comme équivalent a `(*ptr) = 123;` (le pointeur est déréférencé PUIS une valeur est affectée a l'objet pointé) et non comme `*(ptr = 123);` (le pointeur est modifié PUIS il est déréférencé).

La dernière syntaxe est valide, mais produira très probablement un crash. Manipuler les pointeurs de cette façon s'appelle l'arithmétique des pointeurs, c'est quelque chose de très complexe et ne donc être réalisé que dans des cas très particuliers (interfaçage avec le système, le matériel ou d'autres langages).

Dans ce cours, pour éviter les ambiguïtés sur l'ordre d'évaluation des opérateurs ou la confusion avec l'opérateur de multiplication `*`, les parenthèses seront toujours utilisées.

Validité des indirections

Vous avez vu dans le chapitre sur les itérateurs que ceux-ci pouvait être invalide. Quand c'est le cas, ils prennent une valeur particulière, correspondant a `std::end()`, il faut donc toujours tester un itérateur avant utilisation.

```
assert(it != std::end(v));
(*it) = 123;
```

Vous avez également vu qu'un itérateur pouvait être invalide, mais ne pas être testable, par exemple après que la collection soit modifiée.

```
std::vector<int> v { 1, 2, 3 };
auto it = std::begin(v);
v.clear();
assert(it != std::end(v));
(*it) = 123;  // erreur
```

Certaines opérations peuvent invalider les itérateurs, et seule la documentation permet de savoir cela. En cas de doute, considérez que les itérateurs sont invalides.

Mais en fait, ce problème de validité n'est pas spécifique aux itérateurs, mais a toutes les indirections. Certains types d'indirection (comme les références ou les pointeurs intelligents) permettent d'avoir des garanties plus fortes si elles sont correctement utilisées, mais il faut toujours rester vigilants.

L'équivalent de `std::end()` pour les pointeurs est `nullptr` ("pointeur nul"), il est donc possible de tester un pointeur avec `assert`. Mais, comme pour les itérateurs, un pointeur peut être invalide, mais ne pas être testable ("dangling pointer").

```
assert(ptr != nullptr);
assert(ptr);  // équivalent
```

Apres avoir déréférencé une indirection a sémantique de pointeur, vous obtenez encore une indirection, mais a sémantique de référence cette

fois ci. Cette nouvelle indirection peut être conservée dans une variable, comme n'importe quelle indirection.

```
std::vector<int> v { 1, 2, 3 };
auto it = std::begin(v);
assert(it != std::end(v));
int & ref = (*it);
ref = 123;  // ok, equivalent a (*it) = 123
```

Mais vous voyez dans ce code qu'il est facile d'invalider une référence. Si vous appelez `clear` après avoir créé la référence, celle-ci sera invalide, tout comme l'itérateur.

Dans ces conditions, pourquoi une référence est plus sûre qu'un pointeur ? La raison est qu'elle n'est pas destinée a être utilisée dans les mêmes conditions qu'un pointeur.

- un pointeur est modifiable, il peut recevoir une nouvelle valeur (affectation), être nul, faire des opérateurs arithmétiques dessus. Une référence est définie a l'initialisation et ne sera plus modifiable ensuite.
- un pointeur pourra être transmis entre differentes fonctions et classes et avoir une duree de vie assez longue. Une reference ne sera pas conservee lorsque la duree de vie des objets n'est pas garantie.

La difference de securite entre pointeur et reference tient donc uniquement a leur utilisation. Les references imposent des contraintes plus fortes, mais offrent en retour des garanties plus fortes. Pour ameliorer la securite du code, il faut donc priviliger l'utilisateur des references, c'est a dire de concevoir au maximum son code pour rester dans les contraintes imposees par les references. (C'est pour cette raison que les pointeurs sont vu tres tardivement dans ce cours).

Cette approche est assez classique en C++. Pour résoudre un probleme, il est souvent possible d'utiliser plusieurs approches : des approches plus génériques (moins de contraintes, mais

également moins de garanties) ou des approches plus spécialisées (plus de contraintes, donc plus de garanties).
En programmation moderne, la taille des projets est de plus en plus grande (ainsi que la complexité), la qualité du code est donc de plus en plus prioritaire. C'est pour cela que les langages modernes ont des bibliothèques standards de plus en plus importantes et le C++ n'échappe pas a cette règle : le langage C ne propose qu'un seul type d'indirection (les pointeurs), alors que le C++ en propose beaucoup plus (pointeurs du C, références, itérateurs, pointeurs intelligents, etc.).

Cela pourrait laisser penser que les langages modernes sont plus complexes et plus long a apprendre (la norme C tient en 400 pages, la norme C++ en 1500 pages et la documentation du Java doit faire plusieurs milliers de pages), mais ça serait une vision fausse. Si vous souhaitez réaliser une tâche en particulier, vous aurez dans le premier cas peu de fonctionnalités utilisables (donc facile a apprendre), mais vous devrez tout faire vous même, et donc avoir un code plus important et plus complexe (et plus de risque d'erreur). Dans le second cas, vous aurez plus d'outils a apprendre, mais ils seront plus puissants pour chaque tâche. Le code sera donc plus simple et avec moins d'erreur.

Et en C++, vous devrez étendre cette approche a votre code. Si un outil n'est pas disponible dans le langage ou une bibliothèque, plutôt que d'écrire directement le code pour résoudre cette tâche, il faudra faire en sorte de créer ces outils, de les rendre réutilisable et sécurisé, puis de les utiliser pour résoudre votre problème.

Les références comme parametre de fonction

Quelles sont les conditions pour que les références restent valides lors des appels de fonctions ?

Le cas le plus simple est lorsque la référence est utilisée comme paramètre de fonction.

main.cpp

```cpp
#include <iostream>

void f(int & ref) {
    ref = 456;
}

int main() {
    int i { 123 };
    f(i);
    std::cout << i << std::endl;
}
```

affiche :

```
456
```

Dans ce cas, vous avez la garantie que la fonction appelée `f` se terminera avant la fonction appelante `main`. L'objet provenant de `main` sera donc forcément valide pendant toute la duree de l'exécution de la fonction `f` et il n'y a aucun risque d'avoir une référence invalide. Et cela serait encore valide si la fonction `f` appelait une autre fonction, puis une autre fonction et ainsi de suite.

Pour le retour de fonction, la situation est différente.

main.cpp

```cpp
#include <iostream>

int & f() {
    int i { 123 };
    return i;
}

int main() {
    int & i { f() };  // probleme
    std::cout << i << std::endl;
}
```

Dans ce code, la fonction `f` retourne une référence sur une variable locale a la fonction. Lorsque la fonction se termine (c'est a dire tout de

suite après que la référence soit créée, puisque la référence est le paramètre de retour), la variable locale `i` est détruite et la référence devient invalide.

Ne JAMAIS retourner une référence sur une variable locale !

Notez bien que ce code ne produit pas d'erreur (et affichera peut être la valeur correcte “123”). Une référence étant considérée comme toujours valide, il n'y a pas de vérification effectuée. Cela va produire un comportement indéterminée (*undefined behavior*), c'est de la responsabilité du développeur de vérifier son utilisation des références.

Une référence ne peut être retournée par une fonction uniquement si elle correspond a un objet dont la durée de vie n'est pas limité par la fonction. Par exemple une référence qui serait passé en paramètre.

```
int & f(int & ref) {
    ref += 123;
    return ref;  //ok
}
```

Dans tous les cas, faites bien attention quand vos objets sont créées et détruits et quand une indirection est valide ou non.

Rvalue-reference

Depuis le début de ce cours, le terme de “valeur” (*value*) est utilisé indifféremment pour désigner une littérale, une variable (constante ou non) ou une expression (un calcul, un retour de fonction, etc).

En fait, il existe différents types de valeurs, qui respectent des règles sémantiques relativement complexes. Il n'est pas nécessaire de toutes les connaître dans un premier temps, seules deux grandes catégories sont intéressantes au début. (Et les explications vont être simplifiées).

- les *lvalues* (*left-value*) : se sont les variables (constante ou non) ;
- les *rvalues* (*right-value*) : ce sont tout le reste, c'est a dire les

valeurs temporaires (les littérales ou les expressions).

Cette distinction va être importante pour le passages de valeurs dans les fonctions. Sans référence, c'est a dire lors d'un passage par valeur (copie), vous avez vu qu'il est possible d'appeler une fonction avec n'importe quel type de valeur.

```
void f(int) {}

int main() {
    int i { 123 };
    f(i);    // ok avec une variable (lvalue)
    f(123);  // ok avec une littérale (rvalue)
    f(12+34) // ok avec une expression (rvalue)
}
```

Avec les références constantes, la situation est identique, vous pouvez passer n'importe quel type de valeur.

```
void f(int const&) {}

int main() {
    int i { 123 };
    f(i);    // ok avec une variable (lvalue)
    f(123);  // ok avec une littérale (rvalue)
    f(12+34) // ok avec une expression (rvalue)
}
```

La situation change pour les références non constante. Celle-ci ne peuvent accepter qu'un seul type de valeur : les *lvalue*.

```
void f(int &) {}

int main() {
    int i { 123 };
    f(i);    // ok avec une variable (lvalue)
    f(123);  // erreur
    f(12+34) // erreur
}
```

D'ailleurs, ce type de référence est parfois appelle *lvalue-reference*, pour

indiquer qu'elles n'acceptent que des *lvalue*.

Il existe en fait un second type de référence, qui n'acceptent que des *rvalue* et qui s'appelle donc *rvalue-reference*. (Lorsque le type de référence n'est pas précisé, il s'agit de *lvalue-reference*). Les *rvalue-reference* s'écrivent avec l'opérateur `&&` et ne sont jamais constantes.

```
void f(int &&) {}

int main() {
    int i { 123 };
    f(i);    // erreur
    f(123);  // ok avec une littérale (rvalue)
    f(12+34) // ok avec une littérale (rvalue)
}
```

Pour résumer :

Passage	lvalue	rvalue
Par valeur	oui	oui
Référence constante	oui	oui
Référence	oui	**non**
Rvalue-reference	**non**	oui

Pourquoi deux types de référence ?
Les *lvalue* et les *rvalue* ont un comportement très différents lorsqu'elles sont utilisées dans une fonction. La première continuera d'exister après l'appel de la fonction, il est donc possible de modifier la variable dans la fonction.

```
void f(int & i) {
    i += 12;
}

int main() {
    int i { 123 };
    f(i);
```

```
    std::cout << i << std::endl;
}
```

Au contraire, une *rvalue* (le résultat d'un calcul ou la valeur retournée par une fonction) existe que temporairement, elle n'est donc plus accessible après l'appel de la fonction. L'utilisation d'une *rvalue-reference* permet de dire au compilateur : “cette valeur ne sera plus utilisée dans la fonction appelante par la suite, tu peux donc optimiser comme tu veux l'utilisation de cette valeur”.

C'est donc avant tout une question d'optimisation, qui sera surtout intéressant pour les classes complexes. Cela sera vu plus en détail dans la partie programmation orientée objet. Le plus souvent, vous pourrez utiliser les références (*lvalue*) non constante lorsque vous voulez modifier une variable dans une fonction et une (*lvalue*) référence constante dans les autres cas (ou un passage par valeur lorsque la copie est peu coûteuse, par exemple pour les types fondamentaux `int`, `double`, etc.)

Ces règles sont valides aussi pour les paramètres par défaut :

```
void f(int i = 0);         // ok
void f(int const& i = 0)   // ok
void f(int & i = 0);       // erreur (littérale vers
lvalue-reference)
```

std::move

Pour être plus précis sur le rôle de la fonction `std::move`, celle-ci ne “déplace” pas les objets ou n'autorise pas le compilateur a faire un déplacement. Elle convertie simplement une valeur en *rvalue-reference*. Le compilateur va donc interpréter cette valeur comme si c’était une valeur temporaire et pourra appliquer les optimisations compatibles avec un temporaire (déplacement ou copie le cas échéant).

Les differents types de références permet d'écrire des surcharges de fonctions. Cela sera vu dans le prochain chapitre.

Copie et déplacement d'indirections

Les indirections sont des types comme les autres et peuvent donc être utilisés comme n'importe quel type. Par exemple, il est possible de créer des variables (comme déjà vu) :

```
int i { 123 };
int & j { i };
```

La seconde ligne de code peut se lire : `j` est une variable, de type “référence sur un entier”, qui contient comme valeur une indirection sur la variable `i`.

Une référence (obtenue directement ou après déréférencement d'un pointeur) peut se convertir implicitement en valeur, par copie. (Il faut donc que le type soit copiable).

```
int i { 123 };
int & j { i };
int & k { j };
int l { j };
```

Dans ce code, la variable `j` est une référence sur un entier (la variable `i`, qui contient la valeur 123). La variable `k` est également une référence sur la variable `i`. (Comme une référence peut être vue comme un alias de variable, elle pourra être supprimée par le compilateur. Et celui-ci sait très bien que `j` est aussi une référence, il n'y a pas de raison que `k` passe par `j` pour référencer `i`).

Note : une conséquence directe a cela est qu'il est possible d'avoir autant d'indirections que vous souhaitez sur un même objet, il n'y a pas de limitation.

Au contraire, la variable `l` n'est pas une référence, mais contient un entier. C'est donc une copie de la valeur contenu dans la variable `i`, et comme toujours lorsqu'il y a une copie, les objets sont indépendants et la modification d'une des deux variables ne sera pas répercutée sur l'autre.

```
j++;
```

```
std::cout << i << std::endl;
l++;
std::cout << i << std::endl;
```

affiche :

```
124
123
```

A noter qu'un objet complexe peut tout a fait contenir des indirections en interne. Dans ce cas, la copie de cet objet (si elle est autorisée) peut produire différents comportement, selon comment cette classe est conçue. Par exemple, les classes `std::string` et `std::shared_ptr` (un type de pointeur intelligent) utilisent des pointeurs sur des objets internes. La copie d'un objet de type `std::string` copie l'intégralité du tableau interne et l'objet copié est indépendant du premier objet (*deep copy*, copie en profondeur). Au contraire, la copie d'un objet de type `std::shared_ptr` ne copie pas l'objet interne et la copie pointe sur le même objet que le pointeur original (*shallow copy*, copie superficielle).

Inférence de type

Les indirections sont également utilisable avec l'inférence de type. Dans le chapitre sur l'inférence, vous avez vu une différence importante entre `auto` et `decltype` : le premier ne conserve pas les modificateurs de types, le second oui. Les références et la constance sont des exemples de modificateurs de type, il faudra donc faire attention a ajouter explicitement une référence si vous utilisez `auto` si nécessaire.

```
int i { 123 };
int & ref { i };

auto a { ref };           // auto sera déduit en int
decltype(auto) b { ref }; // decltype(auto) sera déduit en
int&
decltype(ref) c { i };    // decltype(ref) sera déduit en
int&
```

```
auto & d { ref };          // auto sera déduit en int, donc
le type sera int &
```

Cela est également applicable aux types de retour de fonction.

Il peut être perturbant d'avoir plusieurs mot-clés différents pour l'inférence de type, mais cela permet d'avoir plus de liberté dans les codes. Il faut retenir :

- `auto` lorsque vous souhaitez explicitement une valeur (par copie) ;
- `auto &` (constante ou non) lorsque vous souhaites explicitement une référence ;
- `decltype` lorsque vous souhaitez le type exacte

```
int f();
int & g();

// force une valeur
auto value1 = f();
auto value2 = g();

// force une référence
auto & ref1 = f();
auto & ref2 = g();

// ne force pas
decltype(auto) rv1 = f();  // valeur
decltype(auto) rv2 = g();  // référence
```

Chapitre précédent	Sommaire principal	Chapitre suivant