Copies, déplacements et indirections

Dans le chapitre precedent, vous avez vu comment echanger des informations avec une fonction, en utilisant les parametres de fonction et le retour de fonction.

Le type d'echange 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 appelee.

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

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

La valeur est copiee lors de l'appel de la fonction f, ce qui implique que les eventuelles modifications du parametre `i` ne seront pas repercutees sur la variable `j`.

Il existe d'autres facon de transmettre une information dans une fonction, il convient donc de detailler d'abord les concepts de copie, de deplacement et d'indirection.

Copie et deplacement

La copie d'objets

La copie consiste donc a creer un nouvel objet, identique a un objet existant. Ces deux objets seront independants, c'est a dire que si l'un des objets est modifie ou detruit, 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
meme 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 tres grande majorite des classes de la bibliotheque standard.

Souvenez vous, cela a ete aborde dans le chapitre Les fonctionnalités de base des collections, la majorite des classes de la bibliotheque standard possedent une semantique de valeur et sont copiables. Un exemple de classe non copiable est `std::unique_ptr`. Cela sera detaille dans la partie sur la programmation objet.

Le deplacement d'objets

Le deplacement d'objets consiste a deplacer (*move*) un objet depuis une variable vers une autre. L'objet n'est pas modifie dans cette operation, il est conserve a l'identique. Cette operation peut etre realisee 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 reelement 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 et l'objet peut etre considere “dans la fonction”. 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 i) {
    return i + 456;
}

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

affiche :

```
579
```

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

Les indirections sont un moyen d'acceder a une variable a distance, sans devoir faire de copie ou de deplacement. Utiliser une indirection revient a utiliser indirectement une autre variable.

Le code precedent peut etre modifie de la facon suivante :

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

```
}

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

affiche :

```
579
```

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

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) {
       ~~~~~~~~~~~^
```

Syntaxe alternative
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 droite a gauche. Ainsi, le code `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 celui qui est le plus utilisee 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 majeurs d'indirections : les indirections a semantiques de reference et les indirection a semantiques

de pointeurs.

Semantiques
Une semantique 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.

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

Les references 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 meme remplacer l'indirection par un *alias de variable*, c'est a dire utiliser directement la variable referencee, mais avec un nom different. (Le code genere par le compilateur supprimera completement 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 acceder a l'objet, il faut d'abord appliquer une operation particuliere, appellee “dereferencement”. “Dereferencer un pointeur” consiste donc a acceder 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 `*` unaire préfixé.

main.cpp

```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
```

Initialisation et affectation

non affectable: reference

affectable: pointeur, iterateru, reference_wrapper

Les indirections sont un type d'objet et peuvent donc etre manipule comme un objet : copie, deplacmeent.

Proprietes des objets

ownership: shared/unique

sans ownership: reference, iterteur

Compatiblite

par defaut, arguement accepte

Chapitre précédent	Sommaire principal	Chapitre suivant