Copies, déplacements et indirections

Organisation de la memoire

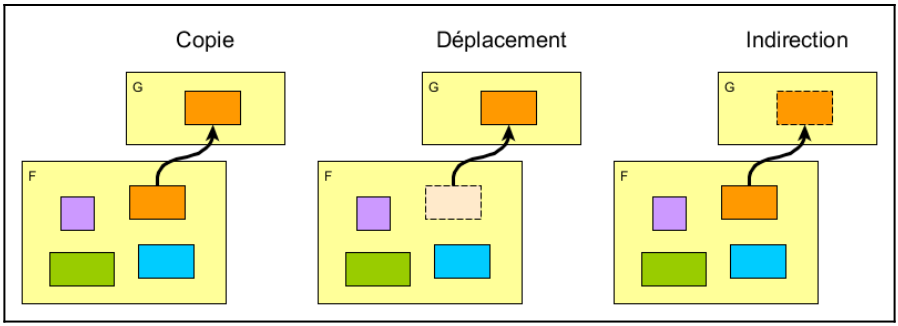
Vue d'ensemble

Avant propos : description simplifiee, pour comprendre comment fonctionne la memoire, pour expliquer les notions expliquees dans ce chapitre. Mais organisation et gestion de memoire beaucoup plus complexe : plusieurs types de memoire, plusieurs niveaux d'organisation logiciel, gestion par le systeme, etc.

Memoire peut etre vue comme un immense tableau d'octets. Dans ce tableau, l'index est l'adresse memoire, chaque octet peut donc etre identifie de maniere unique par son adresse.

figure

Object font 1 a plusieurs octets. Un objet simple est un bloc de plusieurs octets continues. Connaitre la taille d'un objet : sizeof.



Les copies

copie = creation d'un objet identique en memoire. ie les octets sont identiques, mais avec une adresse differente. On obtient 2 objet valides.

figure

deux objets copies sont independants. Si on modifie ou supprime l'un, l'autre n'est pas modifie.

Tous les objets ne sont pas copiables. Types fondamenteux (int, float, etc) sont par defaut copiable. Pour les objets plus complexes, cela dependra de la semantique que l'on veut donner a sa classe (semantique de valeur). La majorite des classes de la STL ont une semantique de valeur. Sera detaille dans la partie POO.

Les déplacements

deplacement = changement de l'adresse d'un objet, l'objet est conserve a l'identique. L'objet initial n'est plus valide.

figure

En pratique, un deplacement peut etre une copie deguisee : copie puis suppression de l'objet initial (dans ce cas, le deplacement aura le meme

cout que la copie). Ca sera la cas des types fondamentaux. Pour des objets plus complexes (par exemple std::string, std::vector, etc), toutes les donnees ne seront pas copiee. (dans ce cas, le deplacement sera beaucoup moins couteux que la copie).

```
void g(int j) {}

void f() {
    int i { 123 };   // non const, le déplacement est une
modification
    g(std::move(i)); // l'objet contenu dans la variable i
est déplacé dans
                     // la variable j de la fonction g, i
devient invalide
}
```

Concerne aussi la semantique de valeur. Mais independant de la copie (ie un objet peut etre copiable et movable, copiable et non movable, ou non copiable et non movable)

indirections

Indirection = permet d'accéder à distance à un objet, comme s'il était dans le contexte.

```
void g(int & j) {
    ++g;
}

void f() {
    int i { 123 };
    g(i);
    std::cout << i << std::endl; // affiche 124
}
```

j est une indirection (référence) sur i, permet d'utiliser l'objet de i dans g. Toute modification sur j sera en fait effectué sur l'objet de i.

Référence constante : indirection, mais non modifiable

Plusieurs types d'indirections : peut etre null ou non (référence doit toujours être valide), managé ou non = responsable de détruire l'objet (référence = non managé), affectable ou non (référence = non affectable)

- référence
- std::reference_wrapper (affectable)
- itérateur = indirection sur collection
- pointeur = peut etre null
- pointeur intelligent = managé (partagé = std::shared_ptr ou non = unique_ptr), pointeur nu = non managé

Indirections en C et C++

Même si le C et le C++ sont deux langages différents, ils partagent une origine commune (d'où leur noms qui sont proches et des syntaxes que l'on peut retrouver dans les deux langages) et surtout, il est courant d'utiliser des bibliothèques écrites en C dans un programme C++.

Malgré cette origine commune, ces deux langages ont des mécanismes très différents, en particulier concernant la gestion de la mémoire. Mais dans ce chapitre, c'est un autre point qui va être abordé : les indirections.

Le langage C propose qu'un seul type d'indirection, les pointeurs nus, alors que le C++ en proposent différents types (voire virtuellement une infinité, puisqu'il est possible de définir des classes implémentant des indirections, comme c'est le cas des itérateurs par exemple).

Cela peut sembler être une difficulté du C++ par rapport au C, puisque cela nécessite d'apprendre plusieurs syntaxes et concepts, là où il n'en faut qu'un seul en C. En fait, chaque type d'indirection apporte des fonctionnalités et garanties différentes, ce qui peut simplifier le code. Il est donc important de bien comprendre les spécificités de chaque type d'indirections et de les utiliser correctement.

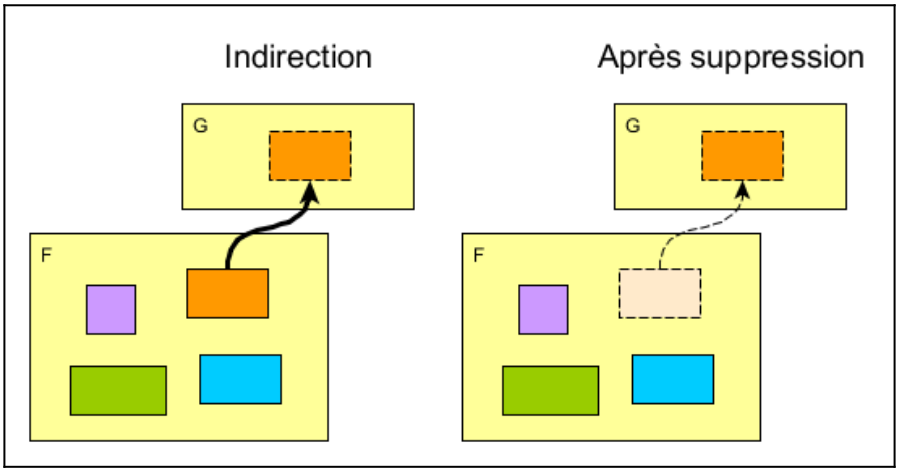
Un dernier point sur les indirections : il est courant de trouver dans des codes C++ (sur internet ou dans de vrais projets) qui sont en fait des

codes C (généralement du C++ écrit en utilisant les pratiques du C). Dans ces codes, vous verrez régulièrement des pointeurs nus, puisque c'est le seul type d'indirection disponible en C. Cependant, en C++, les pointeurs nus sont le type d'indirection qui apportent le moins de garanties et leur utilisation doit donc être limité aux profits des autres types d'indirection (en premier lieu les références).

Faites bien attention d'identifier les codes C++ problématiques et corrigez les si nécessaire.

Validité

Déjà vu pour les itérateurs : l'objet d'origine ne doit pas être déplacé ou supprimé (par exemple après vector::push_back)



Après suppression (n'importe quoi qui invalide l'objet), indirection correspond à un objet invalide, comportement indéterminé. Utilisation de l'indirection = fait n'importe quoi, ne jamais utiliser une indirection invalide.

Dans la cas particulier d'un appel de fonction : f appelle g, f est en “pause” et g s'exécute. Reviens a f quand g est fini et indirection est

détruite dans g. Donc pas de risque qu'une ligne de code dans f invalide l'objet = pas de problème.

Problème → lorsque conserve une indirection dans une variable, contexte multithread, etc.

Fonctions

Pile d'appel de fonction

contexte appelant et contexte de la fonction

lifetime portee

Passage de valeurs

copie/deplacement/indirection dans fonction

Paramètres de retour

Ne pas retourner une indirection sur un variable locale, qui sera détruite à la fin de la fonction (donc indirection sur variable invalide).

(N)RVO : optimisation pour retour, le plus efficace. + conditions d'utilisation.

```
int g() {
    int i;
    return i;
}
```

Par valeur, mais pas de copie ici, optimisé. Note : utilisation de

déplacement = désactive optimisation, moins performant.

Chapitre précédent	Sommaire principal	Chapitre suivant