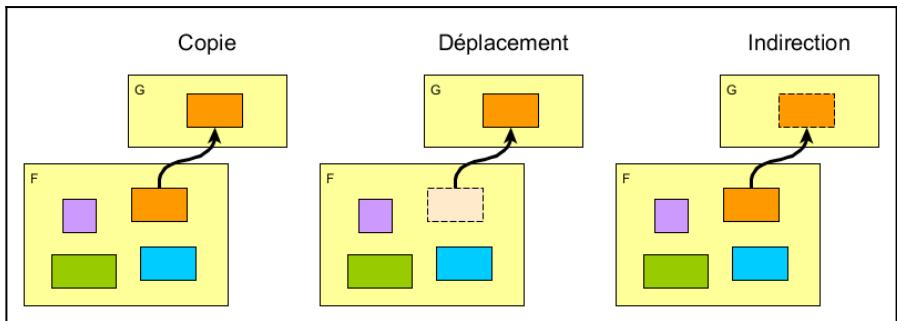


Copies, déplacements et indirections

contexte appelant et contexte de la fonction

schéma global



copie

duplication de l'objet en mémoire, indépendants. si modification de l'un, l'autre n'est pas modifié. A la fin de la fonction, la copie est détruite, toutes les modifications sont perdues.

Uniquement sur objets copiables

déplacement

l'objet est déplacé depuis le contexte de la fonction appelant vers le contexte de la fonction appelée. Idem copie; les modifications sont

perdus à la fin de la fonction. Différence avec la copie : plus light pour les objets complexes, et variable initiale devient invalide.

```
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
}
```

indirections

cas particulier des références.

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 être 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 être 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).

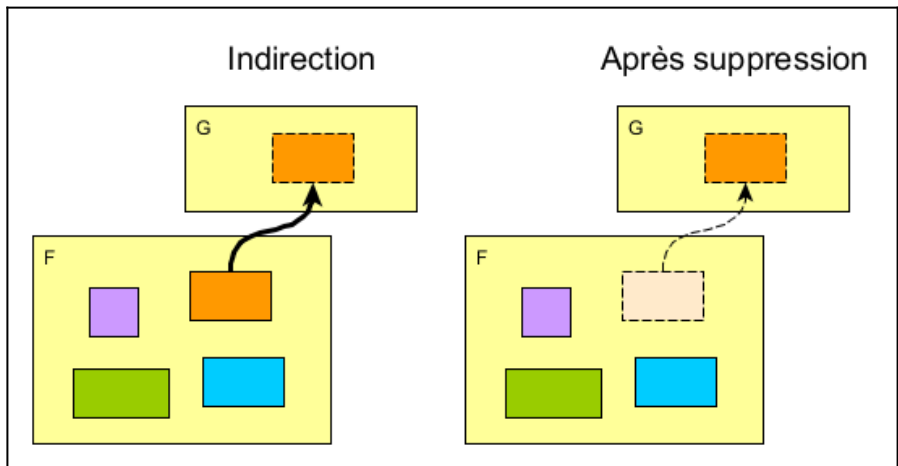
Cela peut sembler être une difficulté du C++ par rapport au C, puisque cela nécessite d'apprendre plusieurs syntaxes et concepts, là où il n'en faut qu'un seul en C. En fait, chaque type d'indirection apporte des fonctionnalités et garanties différentes, ce qui peut simplifier le code. Il est donc important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

Paramètres de retour

Ne pas retourner une indirection sur une 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.

Conclusion

Bien comprendre les concepts de durée de vie et de portée, en particulier pour les indirections invalides. Suivre quand les objets sont créés, sont détruits, sont déplacés. Idem pour les indirections et les objets suivis.

Chapitre précédent	Sommaire principal	Chapitre suivant
------------------------------------	------------------------------------	----------------------------------