Copies, déplacements et indirections

Organisation de la memoire

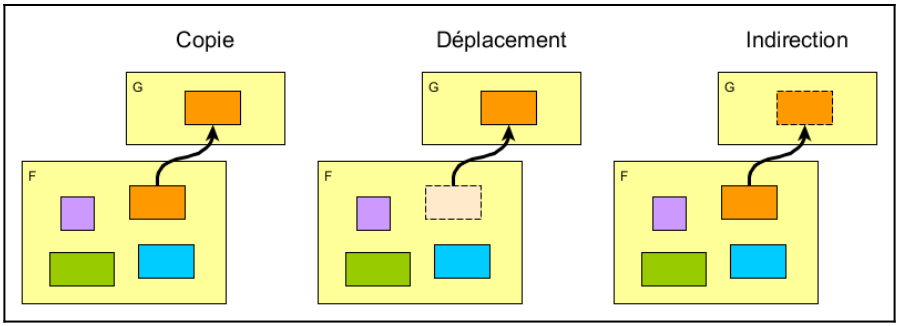
Vue d'ensemble

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 = “objet” qui permet d'accéder à un autre objet à distance. Utiliser une indirection revient à utilisé l'objet qui est lié à l'indirection.

image

Si A est un objet et si B est une indirection sur A, alors utiliser B revient à utiliser A.

Exemple d'indirection déjà vu : les itérateurs. Un itérateur est une indirections sur un élément d'une collection. Accéder à l'itérateur revient à accéder à l'élément.

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
```

```
    auto it = std::begin(v);
    std::cout << (*it) << std::endl;
    (*it) = 5;
    for (auto i: v) std::cout << i << ' '; std::cout << std
::endl;
}
```

affiche :

```
1
5 2 3 4
```

`it` permet d'accéder au premier élément du tableau.

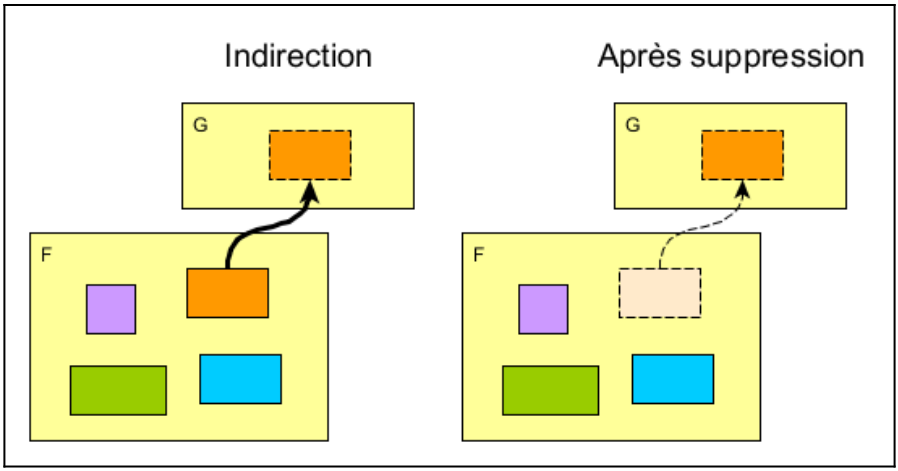
Validité d'une indirection

Le problème de la validité d'une indirection a déjà été abordé pour les itérateurs. Pour rappel :

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

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    auto it = std::begin(v);
    std::cout << (*it) << std::endl;
    v.clear();
    std::cout << (*it) << std::endl;
}
```

Après l'appel à `clear`, l'élément correspondant à l'itérateur n'est plus valide et utiliser `it` produite donc un comportement indéterminé.



Et retenez surtout qu'un UB ne produit pas forcement un crash (c'est le cas de ce code, il semblera valide à l'exécution, mais pourra faire n'importe quoi) et ne produira JAMAIS une erreur de compilation.

Ce qui est valide pour l'itérateur est valide pour toutes les indirections : il faut que l'objet correspond à l'indirection soit valide, sinon l'indirection est invalide et l'utiliser produit un UB.

Types d'indirection

classification :

- 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)
- cas particulier des itérateurs (indirection sur élément d'une collection)

Pas voir en détail toutes les indirections, certaines sont moins utiliées et seront vues dans les chapitres complémentaires.

- 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

Note sur les 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.

Au contraire, en C, les pointeurs nus seront utilisés comme indirection sur un objet, comme indirection sur un élément d'un tableau ou pourront même représenter un tableau. Il n'est pas possible de savoir ce que l'on peut faire ou non avec un pointeur uniquement en regardant son type (idem pour le compilateur, ce qui veut dire qu'il ne peut rien vérifier), il faut regarder le contexte d'utilisation du pointeur pour savoir comment

l'utiliser.

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.

Fonctions

Pourquoi présenter les indirections dans la partie sur les fonctions ? Les indirections sont utilisables en dehors des fonctions (en particulier les itérateurs), mais elles sont particulièrement intéressante avec les fonctions.

On va se limiter à un type d'indirection : les références.

Pile d'appel de fonction

Explications simplifié, pour comprendre la mécanisme. *Stack* en anglais.

Lorsqu'une fonction est appelée (cas particuler pour la fonction main qui est appelée par le système), cela créé en mémoire un contexte pour cette fonction. Ce contexte contient les informations nécessaires pour l'exécution de la fonction, en particulier la mémoire correspondant aux variables locales de la fonction (ainsi que les paramètres d'appel, que l'on peut considérer comme des variables locales).

```
void f(int i) {
    double d;
```

```
    bool b;
}
```

image (contexte d'une fonction avec ses variables locale)

A chaque fois qu'une fonction est appelé, le nouveau contexte “s'empile” sur les précédents (on parle de “Pile d'appel des fonctions”).

```
void h() {}

void g() {
    h();
}

void f() {
    g();
}
```

image (état de la pile après chaque appel et retour)

Lorsque f est exécuté, la pile contient le contexte d'exécution de f. Après l'appel de g, la pile contient f et g, puis après l'appel de h, la pile contient f, g et h. Après le retour de h, la pile contient à nouveau f et g, et après le retour de g, la pile contient f.

Note : mémoire = très gros tableau d'octets. “empilement” est conceptuel, il n'y a pas de notion de “dessus” et “dessous” dans une mémoire.

Contexte d'une fonction

Le contexte d'une fonction est l'ensemble des variables qu'une fonction peut utiliser.

```
void g() {
    i = 123; // erreur
}

void f() {
    int i {};
    g();
```

```
}
```

Exception :

- variables globales, accessible partout. A éviter (complexifie la lecture du code)
- les littérales (il faut bien qu'elle soit quelque par dans la mémoire. Elle sont dans un espace dédié, le data segment. Et il y a un segment spécifique pour le texte. Mais tout cela est géré en interne par le compilateur, pas besoin de s'en préoccuper).

Taille limite de la Pile

Il existe une limite à la taille de la pile (par exemple 1 Mo avec MSVC). Dépend des configuration et système. Mais si vous essayer de créer 1 milliard de int dans une fonction, cela va produire très certainement un dépassement de pile. (erreur de type ” Segmentation fault”).

2 conséquences :

Appeler une fonction consomme un peu de mémoire sur la pile (même si la fonction n'a pas de variable locale). Appelle récursif illimité finira donc pas saturé la pile et produire une crash.

Parfois (souvent) besoin d'utiliser plus que 1 mo de mémoire (penser à une simple image en mémoire, cela peut prendre plusieurs Mo). Il existe un second espace mémoire, le Tas (ou mémoire dynamique), qui contient tout ce qui n'est pas sur la Pile.

Différences Pile/Tas

- Pile = taille très limité, Tas = beaucoup plus de mémoire (en première approximation = infini… en vrai = taille limité, espace libre généré par le système et peut changer avec le temps = besoin de gérer si la mémoire est dispo ou non)
- Pile = généré automatiquement lors des appels et retour de fonction, Tas = à gérer par le code

Cas particulier des objets complexes comme std::vector et std::string.

Sont en fait constitué de plusieurs parties :

- une petite partie, de taille fixe, qui sera sur la Pile par défaut
- les données proprement dites (les éléments ou le texte), qui sont sur le Tas
- des indirections depuis la partie sur la Pile vers la Partie sur le Tas

image d'un vector

Des collections plus complexes (std::list par exemple) peuvent contenir plusieurs objets différents en mémoire, liés entre eux par des indirections.

En pratique, pas de limite au nombre de sous objets que peut contenir un objet complexe, et avoir des organisations très complexe en mémoire.

Note sur std::array : toutes les données sur la Pile, donc problème si tableau trop grand. Utiliser std::vector (ou dynarray) si c'est le cas.

Remarque sur vector et copie/deplacement. Quand on copie, cela copie toutes les données. Quand on move, cela copie simplement la petite partie sur la Pile. Plus performant. (Mais le plus performant reste d'utiliser une indirection).

Déplacer une partie des explications dans un chapitre complémentaire

Passage de valeurs

Contextes de fonctions indépendants = pas possible d'accéder directement à une variable. Comme faire ?

```
void g() {
    i = 123; // erreur
}

void f() {
```

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

On copie l'objet depuis le contexte de f dans le contexte de g. Il est alors possible d'utiliser cet objet dans g.

La syntaxe est celle déjà vue :

```
// déclaration
void f(TYPE NOM)

// appel
f(VALEUR)  // littérale, expression, variable
```

Par exemple :

```
void g(int i) {
    i = 123;  // ok
}

void f() {
    int i {};
    g(i);
}
```

L'objet est perdu à la fin de la fonction (sauf si retourné), donc les modifications faites sont perdues.

```
void g(int i) {
    i = 123;  // ok
}

void f() {
    int i {};
    g(i);
    std::cout << i << std::endl; // affiche 0
}
```

différence entre copie et déplacement ? (selon le type de value, rvalue ou lvalue, std::move)

retour de valeur

Passage par référence

Indirection sur l'objet de f dans g. Chaque indirection à une syntaxe spécifique. Pour la référence, 2 types : const ou non.

```
// déclaration
void f(TYPE const& NOM)  // const
void f(TYPE      & NOM)  // non const

// appel
f(VALEUR)  // littérale, expression, variable
```

const = on ne peut pas modifier le paramètre, non const = on peut.

Dans tous les cas, ce n'est pas la référence que l'on manipule, mais c'est en fait l'objet dans f. En particulier, si on fait une modification dans g, elle sera visible dans f.

```
void g(int & i) {  // on modifie i donc non const
    i = 123;  // ok
}

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

Note : une indirection prend un peu de place dans la Pile. Environ le même cout qu'un type fondamental (int, float, etc)

Conclusion : quoi utiliser ?

- pour éviter la copie = indirection. Si la copie n'est pas couteuse (types fondamenteaux) ou si on a besoin d'une copie (si on veut faire des modifications sans les conserver)

- pour modifier un objet = référence non const
- par défaut (si on ne sait pas si la copie est couteuse) = référence constante.

Valeur par défaut

selon le type de passage ?

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.

+ cas particulier ds fonctions pure

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

// alternative à

void g(int & i) {
    ...
```

```
}
```


Chapitre précédent	Sommaire principal	Chapitre suivant