Les enjeux de la conception logicielle

A ce niveau du cours, vous avez toutes les connaissances nécessaires pour créer n'importe quelle application. En effet, pour réaliser un calcul, un programme a besoin de :

- manipuler les données (avec les variables) ;
- réaliser des calculs dessus ;
- pouvoir écrire des algorithmes (avec les tests et les boucles).

Ce sont les fonctionnalités de base, que de nombreux langages de programmation fournissent. Les autres fonctionnalités d'un langage servent en fait à faciliter la conception d'application de plus en plus complexes.

Il est important de bien comprendre les enjeux de la conception d'applications complexe, pour utiliser correctement les outils qu'un langage met a disposition.

Vous avez déjà vu un outil visant à faciliter la conception des applications : les fonctions. Ce chapitre utilisera donc les fonctions comme exemple, mais comprenez bien que cela est applicable aux outils que vous verrez dans l'avenir, en particulier la programmation orientée objet.

Quel est le problème avec la conception d'applications complexes ? La reponse est evidente : elles sont complexes ! A cause de leur taille. A cause des besoins ou des contraintes. A cause de leurs objectifs.

De nos jours, une application sera facilement développée en équipe, sera maintenu pendant plusieurs années, devra fonctionner sur plusieurs systèmes, repondre a des contraintes de coûts de développement et de maintenant, des contraintes de fiabilité. Il faut maîtriser tous ces aspects pour obtenir une application respectant la qualité logicielle :

- la capacité fonctionnelle : le programme fait ce qui est attendu ;
- la fiabilité : le programme fournit le bon résultat ;
- la maintenabilité : il est facile de corriger les erreurs, et d'ajouter de nouvelles fonctionnalités (évolutivité) ;
- la performance : le programme travaille efficacement ;
- la portabilité : le programme est utilisable sur différentes plateformes ;
- la facilité d'utilisation : les programme est facile à utiliser.

Décomposer un problème complexe

Les capacités humaines sont limitées. (Oui, même vous qui pensez être plus intelligents que les autres). Une approche classique pour résoudre un problème complexe est de le décomposer en sous-problèmes un peu moins complexes. Puis diviser ces sous-problèmes en problèmes encors plus simples, et ainsi de suite, jusqu'a obtenir des problemes que vous savez résoudre.

La décomposition en sous-problèmes peut s'exprimer directement avec des fonctions : chaque sous-probleme donné une fonction. Une décomposition en sous-probleme donné une liste d'appels de fonctions.

```
void problem_1() {
    // resolution du probleme 1
}

void problem_2() {
    // resolution du probleme 2
}

void problem_3() {
    // resolution du probleme 3
}

void complex_problem() {
    problem_1();
    problem_2();
    problem_3();
```

```
}
```

C'est aussi simple que cela. Une problématique bien comprise sera naturellement décomposable en fonctions. Chaque fonction devrait être relativement simple, en general au maximum quelques dizaines de lignes. (Plusieurs guides de codage donne le chiffre - arbitraire - de 25 lignes. Ce n'est pas une contrainte absolue, mais c'est une bonne base d'essayer de respecter une telle limite).

Il faut bien comprendre un point : le découpage en fonctions doit découler naturellement de la décomposition du problème. Découper une code en plusieurs fonctions, sans organisation logique, juste parce que “en C++, on utilise des fonctions” n'a aucun sens.

Lisibilité du code

Il faut bien garder en mémoire qu'un code est plus souvent lu qu'il n'est écrit. Il faut donc qu'il soit le plus lisible possible. Quand une autre personne lit votre code, elle devrait pouvoir comprendre comment votre probleme se décompose, rien qu'en lisant l'organisation de vos fonctions. (Les noms que vous donnez à vos fonctions et variables sont donc primordiaux).

Un facteur important pour la lisibilité est la *localité des informations*. Lorsque vous lisez un code, le cerveau conserve les informations lues dans une mémoire temporaire et ces informations sont utilisées pour comprendre la suite du code. Cette mémoire temporaire a une capacité limitée, ce qui fait que lorsque de nouvelles informations arrivent, les anciennes informations sont oubliées. Si vous avez besoin ensuite de ces informations, il faudra donc aller relire le code déjà lu (perte de temps) ou prendre le risque de ne pas comprendre totalement.

Pour comprendre correctement une ligne de code, il faut que les informations nécessaire à sa compréhension soient en mémoire. Plus les informations sont locales (c'est-à-dire proche de la ligne de code à comprendre), plus un code sera facile a lire.

Une règle (que vous avez déjà vue) est de déclarer une variable au

moment ou vous en avez besoin. (En plus, cela permet souvent d'ajouter `const` dans les déclarations).

Le code suivant :

```
void foo() {
    int i { 1 };
    int j { 2 };
    int k { 3 };

    f1(i);
    f2(j);
    f3(k);

    g1(i);
    g2(j);
    g3(k);

    h1(i);
    h2(j);
    h3(k);

    bar(i, j, k);
}
```

sera moins lisible que le suivant :

```
void foo() {
    int i { 1 };
    f1(i);
    g1(i);
    h1(i);

    int j { 2 };
    f2(j);
    g2(j);
    h2(j);

    int k { 3 };
    f3(k);
    g3(k);
```

```
    h3(k);

    bar(i, j, k);
}
```

(Bien sur, sur un code aussi simple et aussi court, vous ne verrez pas de différence de lisibilité. Cette règle aura un impact lorsque le code devient un peu important).

La décomposition d'un code en fonctions aide également à améliorer la lisibilité du code, en isolant chaque partie du code qui est indépendante des autres. Dans le code précédent, vous voyez qu'il y a trois parties de code indépendantes, puis une partie qui dépendent des autres lignes. Il est donc possible de découper le code en plusieurs fonctions pour montrer cette indépendance dans le code.

```
int foo_1() {
    int i { 1 };
    f1(i);
    g1(i);
    h1(i);
    return i;
}

int foo_2() {
    int j { 2 };
    f2(j);
    g2(j);
    h2(j);
    return j;
}

int foo_3() {
    int k { 3 };
    f3(k);
    g3(k);
    h3(k);
    return k;
}

void foo() {
```

```
    const int i { foo_1() };
    const int j { foo_2() };
    const int k { foo_3() };

    bar(i, j, k);
}
```

(Encore une fois, sur un code aussi simple, la différence de lisibilité n'est pas significative. Cela est important dans des codes plus complexes.)

Dans le premier code, pour comprendre l'appel de la fonction `bar`, il faut lire les déclarations des variables `i`, `j` et `k`, qui sont éloignées de l'appel de `bar`. (Imaginez si la fonction `foo` faisait plusieurs centaines de lignes de code, il faudrait faire défiler le code pour aller chercher la déclaration des fonctions).

Dans le dernier code, la déclaration des variables utilisées dans la fonction `bar` sont plus proches, ce qui a un impact sur la lisibilité. (Dans ce cas, même si la taille du code est plus importante, la déclaration des variables reste proche de leur utilisation).

Les fonctions sont des abstractions

Prenez un code d'exemple simple :

```
std::string s { "hello, world" };
std::cout << s.size() << std::endl;
```

Vous devriez comprendre (normalement, si ce cours n'est pas trop mal conçu) que ce code affiche la longueur de la chaine de caracteres “hello, world”. Vous le savez parce que vous savez que `std::string` représente une chaîne de caractères, que celle-ci est initialisée avec la chaîne “hello, world”, que `std::cout` permet d'afficher quelque chose, et que la fonction membre `size()` retourne la taille de la chaîne.

Pourtant, si quelqu'un vous demande ce que fait exactement chacune des étapes de ce code, vous seriez incapable de répondre. Rien que le fonctionnement interne de `std::cout` est relativement complexe. Et pourtant, cela ne vous empêche pas de comprendre ce code, et même

d'écrire vos propres code utilisant ces fonctionnalités.

La raison est que vous avez juste besoin de savoir comment utiliser ces fonctionnalités, pas le détail de ce qu'elles font en interne. Ce qui vous intéresse, c'est abstraction qu'elle représente : `std::string` représente une chaîne, `std::cout` permet d'afficher, etc.

Le découpage du code en fonction permet donc de créer des abstractions qui facilite la lecture du code. En écrivant une fonction qui peut se comprendre sans lire son code, vous renforcer la localité des informations et donc la lisibilité.

La partie publique d'une fonctionnalité est son **interface**. Pour une fonction, c'est donc sa signature (son nom et la liste de ses paramètres). Pour une classe, c'est l'ensemble de ses fonctions membres (par exemple, la classe `std::vector` contient les fonctions membres `size`, `push_back`, `front`, etc.).

Les détails interne d'une fonctionnalité est son implémentation. Pour une fonction, il s'agit du code de cette fonction. Pour une classe, c'est l'ensemble des variables membres et l'implémentation des fonctions membres. Il y a quelques subtilités sur l'implémentation des classes en C++, mais vous verrez cela dans les chapitres sur la programmation orientée objet.

Vous verrez dans les prochains chapitres qu'il est possible (et même recommandé quand c'est possible) de séparer l'interface et l'implémentation des fonctionnalités dans des fichiers différents, ce qui permet bien distinguer les deux. Cela est détaillé dans le prochain chapitre.

La création d'abstractions simples à utiliser (mais remplissant des tâches parfois très complexes) est l'un des buts principaux de la programmation orientée objet et de la méta-programmation. Vous verrez également cela par la suite.

La reutilisabilité

La réutilisabilité du code est quelque chose que vous utiliser depuis le début de ce cours, sans forcément le savoir. Ce terme signifie simplement que le même code est réutiliser plusieurs fois. Par exemple, quand vous utilisez des fonctionnalités de la bibliothèque standard, vous utilisez au final le même code (écrit par quelqu'un d'autre que vous) plusieurs fois.

En pratique, cela signifie qu'un code qui réalise plusieurs fois la même tâche (mais qui n'est pas possible d'écrire une boucle).

...

Plus généralement, créer une bibliothèque logicielle permet de faciliter la réutilisation du code. C'est pour cela qu'une partie importante de la conception se focalise sur la création de bibliothèques logicielles, une application n'étant au final qu'un ensemble de bibliothèques qui interagissent entre elles.

La testabilite

libs et modules

la conception d'application vue comme la conception de libs

Chapitre précédent	Sommaire principal	Chapitre suivant