18

Un coup d'œil dans le rétro

Nous avons abordé de très nombreuses notions depuis le début de ce livre, et avec elles nous avons touché du doigt certains aspects essentiels du développement informatique. Il est temps de revenir sur ce que nous avons fait et de l'analyser avec un peu de recul. Nous insistons sur ces points, car être un bon développeur c'est beaucoup plus que simplement savoir écrire du C++.

18.1. Au-delà du code

Jusqu'à présent, nous avons abordé différents aspects de la programmation en C++, que ce soit des notions de base (comme les conditions ou les boucles) ou des notions plus avancées (comme l'utilisation d'outils fournis par la bibliothèque standard, ainsi que la création de fonctions et de structures de données personnalisées). Vous avez ainsi déjà acquis un arsenal de techniques assez complet pour faire des programmes relativement complexes. La création du gestionnaire de discographie de la partie précédente est là pour le prouver.

Cependant, nous avons vu dans ces deux parties bien plus que de simples notions de C++. Nous avons découvert une certaine manière de réfléchir qui traduit les qualités d'un bon développeur : le désir d'avoir toujours un code de qualité, à travers un travail de conception poussé et des bonnes pratiques. Citons, parmi celles-ci, les tests unitaires pour vérifier que notre code fait ce qu'on lui demande, les contrats pour définir clairement ce que les différents intervenants attendent comme entrées et sorties et les exceptions pour remonter des problèmes à l'utilisateur.

Ce sont ces qualités qui feront qu'un développeur sera non seulement capable d'utiliser son langage, mais aussi de mener ses projets à bien. Pour être un bon marin, il ne suffit pas de savoir utiliser la voile et le gouvernail, encore faut-il savoir naviguer, s'orienter sur la mer. De même, il ne suffit pas pour un développeur de maîtriser ses outils pour mener sa barque à bon port.

Vous avez commencé à vous rendre compte de ces aspects-là, notamment grâce à la deuxième partie qui insiste fortement sur l'amélioration de la qualité du code. Et c'est l'une des choses les plus importantes à retenir de ces deux premières parties. Dans cet

interlude, nous allons aborder de nouvelles notions qui s'inscrivent dans cette optique, à savoir faire de vous de bons développeurs et pas simplement des personnes qui savent écrire en C++.

18.2. Tout est une question de paradigme

Nous avons souvent dit que tout problème de développement a de nombreuses solutions, l'essentiel étant que les choix que l'on fait soient guidés par la volonté de produire un code propre et robuste à travers une bonne conception.

Cette diversité se manifeste notamment à travers la manière de programmer. Il existe différentes approches, appelées *paradigmes*. Le C++ est un langage *multiparadigme*. Il permet de programmer de manière très libre et nous laisse choisir parmi plusieurs de ces manières de programmer. Nous en avons déjà croisé quelques-unes.

Le paradigme impératif

Le paradigme impératif est le principal paradigme que l'on a utilisé. C'est aussi celui de beaucoup d'autres langages de programmation, en particulier des premiers inventés. Selon cette approche, le programme est une suite d'instructions qui s'exécutent *linéairement* et le font avancer. Il comprend des conditions, boucles, fonctions, etc. Une grande partie des notions que l'on a abordées s'inscrit, au moins partiellement, dans ce paradigme.

Voici un exemple d'un code écrit typiquement avec le paradigme impératif, qui met chaque élément d'un tableau au carré.

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

int main()
{
  std::vector<int> nombres { -8, 7, 48, 366, 1, 4, 3 };
  for (int i = 0; i < nombres.size(); ++i)
  {
    nombres[i] *= nombres[i];
  }

  for (auto nombre : nombres)
  {
    std::cout << nombre << std::endl;
  }

  return 0;
}
```

```
64
49
2304
133956
1
16
9
```

Le paradigme fonctionnel

Le paradigme fonctionnel est aussi en partie couvert par le langage C++. C'est un paradigme dans lequel, comme son nom l'indique, les fonctions jouent un rôle prépondérant. D'ailleurs, dans les langages dit *fonctionnellement purs*, la modification de variable peut ne pas exister et tout est réalisé avec des calculs sur résultats de fonctions ! Nous avons principalement vu ce paradigme à travers les lambdas, qui sont un concept issu tout droit des langages fonctionnels. De manière plus générale, le fait de pouvoir manipuler les fonctions comme des variables est un emprunt à ce paradigme.

Le code suivant est une réécriture de l'exemple précédent, dans un style plus fonctionnel.

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

int main()
{
  std::vector<int> nombres { -8, 7, 48, 366, 1, 4, 3 };
  std::for_each(std::begin(nombres), std::end(nombres),
    [](int & nombre) -> void
  {
    nombre *= nombre;
  });

  for (auto nombre : nombres)
  {
    std::cout << nombre << std::endl;
  }

  return 0;
}
```

```
64
49
2304
133956
1
16
9
```

Le paradigme générique

Le paradigme générique s'appuie essentiellement sur l'idée que, parfois, on se fiche du type réel des données que l'on manipule. Ce qui compte, c'est la manière dont elles peuvent être utilisées. Nous l'avons principalement appréhendé sous deux formes :

- les algorithmes standards qui peuvent s'appliquer à tout un tas de collections différentes ;
- les fonctions templates qui permettent d'écrire du code fonctionnant avec plusieurs types, du moment qu'on peut les manipuler de cette manière.

En reprenant les exemples précédents, utilisons le paradigme générique. Voici un résultat qu'on peut obtenir.

```
#include <algorithm>
#include <iostream>
#include <list>
#include <vector>

template <typename Collection>
void au_carre(Collection & collection)
{
  std::for_each(std::begin(collection), std::end(collection),
    [](auto & nombre) -> void
  {
    nombre *= nombre;
  });
}

template <typename Collection>
void affichage(Collection const & collection)
{
  for (auto element : collection)
  {
    std::cout << element << std::endl;
  }
}

int main()
{
  std::vector<int> nombres { -8, 7, 48, 366, 1, 4, 3 };
  au_carre(nombres);
  affichage(nombres);

  std::cout << std::endl;

  std::list<int> autres { 10, 30, 60 };
  au_carre(autres);
  affichage(autres);
```

```
  return 0;
}
```

```
64
49
2304
133956
1
16
9

100
900
3600
```

La programmation par contrat

Bien que ce ne soit pas aussi poussé qu'avec d'autres langages comme Eiffel ou le D, il est possible en C++ de faire de la programmation par contrat, ainsi que nous l'avons vu au chapitre Erreur, erreur, erreur.... Les principes de la programmation par contrat se combinent très bien avec les autres paradigmes, alors n'hésitez pas à en user et en abuser.

Note › *C++23 devrait inclure de nouveaux outils pour faire de la programmation par contrat, directement intégrés dans le langage, ce qui permettra de faire des choses encore plus poussées.*

Comme vous le voyez, à la différence d'autres langages plus restrictifs, le C++ met à notre disposition tout un panel d'outils très variés pour écrire du code. Nous pouvons prendre un peu de chaque paradigme et nous servir de ses points forts. Dans la quatrième partie du livre, nous aborderons un nouveau paradigme, lui aussi très important en C++ : la programmation orientée objet.

Attention › *Ce n'est pas une critique envers d'autres langages que de dire qu'ils sont moins permissifs. Il est parfois préférable d'utiliser un langage qui fait des choix forts. Simplement, l'un des attraits du C++ est précisément cette permissivité. Mais c'est aussi un écueil. Comme on peut faire beaucoup de choses, on peut aussi très facilement faire n'importe quoi ! Vous devez donc être très vigilant et toujours vous efforcer de produire du code propre et robuste. D'où notre insistance sur ce point.*

18.3. L'art de la conception

Nous l'avons compris, le C++ offre de nombreux outils, ainsi que la liberté d'utiliser et mélanger plusieurs paradigmes. Mais ce ne sont là qu'outils et instruments. Il faut les assembler intelligemment et harmonieusement pour produire de bons programmes. Et pour cela, avant de se ruer sur son clavier et de martyriser énergiquement ses pauvres touches, il faut *réfléchir et concevoir*.

On prend un stylo et un papier (ou tout autre support qui vous convient) , et on se pose des questions sur le programme sur lequel on travaille. On réfléchit à ses fonctionnalités, ce qu'il rendra comme services, ce qu'il ne fera pas, ce qu'il sera capable de gérer, etc. On se demande quels sont les cas d'erreurs, les entrées invalides, ce qui peut potentiellement poser problème, etc.

Note *› Nous avons déjà commencé à faire ça, notamment avec les contrats et la réflexion sur nos entrées et sorties.*

On découpe aussi de grosses fonctionnalités en plus petites afin qu'elles soient plus facilement abordables. Un logiciel de traitement d'images peut ainsi être découpé en sous-tâches comme l'ouverture d'un fichier PNG, la modification d'un unique pixel, la sauvegarde du PNG modifié, etc.

C'est là que le fait de noter ses idées se révèle utile. On peut comparer plusieurs solutions à un même problème, on peut visualiser plus facilement à quoi ressemblera le programme final, on peut écrire du pseudo-code pour s'aider à mieux comprendre un algorithme complexe, etc. Et puis, personne n'ayant une mémoire infaillible, noter ses idées aide à s'en rappeler.

Cependant concevoir, anticiper ne signifie pas que le programme est figé et qu'il n'évoluera pas d'un pouce. Au contraire, réfléchir avant de coder permet de mieux prévoir des aspects qui pourront être amenés à changer. Cela rend le programme plus évolutif.

Attention *› C'est un exercice délicat que de trouver l'équilibre entre le néant et trop de spécifications, entre une trop grande rigidité et une absence de réflexion et de conception. Ne vous inquiétez pas. Comme beaucoup de choses, tout vient en pratiquant.*

18.4. De la persévérance avant tout

Être un bon développeur, c'est être autonome et se débrouiller. Cela ne veut pas dire que les développeurs ne se parlent jamais entre eux et restent chacun dans leur coin, coupés les uns des autres. Au contraire, l'existence même du tutoriel à l'origine de ce livre et de la plateforme Zeste de Savoir sont des preuves que l'échange est important.

Mais il ne faut pas vous attendre à ce que tout vous tombe cuit dans le bec. Un bon développeur ne compte pas sur les autres pour faire son travail. Il n'attend pas de réponse toute faite. Quand il est bloqué, il réfléchit d'abord et demande ensuite. Dans cette partie (chapitre Chasse aux bugs !), nous verrons aussi comment débusquer vos erreurs en exécutant pas à pas votre code. C'est ce qu'on appelle le *débogage*. Avec cette technique, vous pourrez déjà bien avancer seul dans la recherche de la solution.

Si vous êtes vraiment bloqué sur un problème, vous pouvez bien entendu demander un coup de pouce, notamment sur Zeste de Savoir. Mais quand cela arrivera, montrez ce

que vous avez tenté, les erreurs qui vous bloquent, si vous avez déjà fait des recherches dessus, etc. Et vous verrez, vous aurez de l'aide et des réponses ☺ .

En conclusion, laissez-moi vous lister quelques qualités, en plus de l'autonomie, qui feront de vous un meilleur développeur.

- *Patience*. Il en faut pour apprendre, pour comprendre, pour faire et corriger des erreurs, pour chercher des solutions, pour ne pas s'énerver quand ça ne marche pas.
- *Persévérance* et *motivation*. Développer, c'est parfois rencontrer des problèmes qui nous dépassent, qui semblent trop complexes à première vue, ou bien des notions trop abstraites. Ces qualités aideront à ne pas abandonner à la première difficulté.
- *Humilité*. Un bon développeur accepte de ne pas avoir réponse à tout et donc de demander de l'aide quand il n'y arrive vraiment pas. Il accepte les conseils et les corrections qu'on lui donne. Il accepte de ne pas se reposer sur ses acquis, se remet régulièrement à jour et en question. Cela lui évite les réponses méprisantes, arrogantes et hautaines qu'on voit parfois sur certains forums.

18.5. Récapitulons !

- Vous avez déjà beaucoup appris sur le langage C++, mais aussi sur des aspects plus généraux du développement, comme l'écriture de tests unitaires, de contrats ou la réflexion à mener sur les entrées/sorties.
- Le C++ est un langage très riche qui permet de programmer de différentes manières.
- Cette liberté est une force mais aussi un danger, c'est pourquoi une bonne conception et du code de qualité sont primordiaux.
- Être un bon développeur demande aussi plusieurs qualités, dont la persévérance, l'autonomie et la patience.
- La suite de cette partie vous aidera à tendre vers cet idéal en vous apprenant de nouvelles notions pour mieux développer.