PG208
Programmation Orientée Objet / Langage C++
TP1

Y. Bornat A. Valade

March 28, 2024

1 Introduction

L'objectif de cette série de *TPs / Cours intégrés* est de découvrir la programmation orientée objet. Selon l'indice TIOBE qui mesure la popularité des langages de programmation, cela fait 10 ans (une éternité en informatique) que les cinq langages les plus utilisés sont (dans l'ordre du classement 2024)

- Python
- C
- C++
- Java
- C#

Sur ces cinq langages, seul le C n'est pas orienté objet. Il est encore très utilisé car très proche du comportement d'un processeur, et donc très efficace (pour être dans le vent, on dira que c'est le langage dont l'empreinte carbone est la plus faible). Pour tout le reste (vitesse d'écriture, lisibilité, facilité de maintenance, portabilité ...), il y a l'*objet*.

1.1 Pourquoi le C++ ?

Parce que !

Python est un langage qui possède des concepts très puissants, mais cela a un coût : c'est un langage interprété. Cela signifie qu'il n'est pas possible d'exécuter un programme Python. C'est un autre programme (l'interpréteur) qui, en lisant le code python, fait ce qui est demandé. L'avantage est que seul l'interpréteur à besoin d'être recompilé quand on change de processeur cible, cela en fait un langage très portable, mais lent.

Le C++ présente le double avantage d'être rapide à prendre en main quand on a l'habitude du C, et d'être très efficace en terme de puissance. Quand on change de processeur cible, il faut tout recompiler, mais vu qu'il est très répandu, le compilateur dont vous avez besoin existe certainement déjà.

Concernant les autres langages, Java est compilé, mais pas en langage machine comme C ou C++. Le code produit n'est même pas exécutable, il est interprété par une *machine virtuelle*. La différence avec Python vient du fait que ce code intermédiaire demande très peu de ressources pour être interprété. Il est donc à la fois efficace et facilement portable d'un processeur à l'autre. C'est pour cela qu'il est utilisé dans les appareils mobiles (contrairement aux ordinateurs type PC, il n'y a pas de jeu d'instructions standard pour les smartphones/tablettes)

Le C# (prononcer C *sharp*) ressemble beaucoup au C++, mais a été conçu pour l'écosystème Microsoft. Il n'a donc pas sa place dans un enseignement qui se veut générique et orienté applications électroniques.

1.2 Organisation de l'enseignement

Nous passerons d'abord deux séances pour utiliser les avantages du C++ sans faire de programmation objet, il y a déjà beaucoup de choses à voir. Les trois séances suivantes seront passées sur les concepts objet. Les dernières se feront sur un petit projet.

2 C'est Parti !

2.1 Généralités

Histoire de passer assez vite sur les fichiers et les outils :

- l'extension d'un fichier C++ est .cpp (par convention, rien ne l'oblige si ce n'est l'envie d'être compris pas les autres)
- l'extension d'un *header* en C++ est .hpp (au lieu de .h en C)
- le compilateur n'est plus *gcc* mais *g++*. Étant donné que les deux compilateurs font partie de la même suite logicielle, les options sont les mêmes. Si vous travaillez sur votre propre machine, il se pourrait que la version du langage par défaut ne soit pas la version 2011 que nous utilisons comme référence[1]. Vous aurez alors besoin d'utiliser l'option de compilation `--std=c++11`.

Ça, c'est fait !

2.2 Bonjour le monde

Eh beh oui ! On commence par dire bonjour au monde. Et rien que là, il y a des choses à dire ! Déjà, on n'utilise plus les bilbiothèques stdio.h ou stdlib.h, non ! Quand on fait du C++, on appelle la bibliothèque iostream avec le code suivant : `#include <iostream>` (sans le *.h* donc).

De même, plus de *printf*. Pour afficher du texte, il faudra utiliser la syntaxe suivante :

```
1  std::cout << "Bonjour\n";
```

Et même là, on est encore trop proche du C. En effet, ceux qui travaillent sous windows ont certainement remarqué que selon le système d'exploitation, le code de saut de ligne n'est pas le même. Sous Windows, il faut écrire `\n\r` alors que sur les autres systèmes, un `\n` suffit. Histoire de ne pas s'embêter, le saut de ligne est défini de façon générique au niveau du langage, et il s'appelle `std::endl` (pour *end line*)

Pour dire bonjour au monde le programme complet ressemblera donc à :

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, world!" << std::endl;
5      return 0;
6  }
```

Le préfixe `std::` sert à définir l'espace de nom auquel appartiennent `cout` et `endl`. Nous verrons plus tard en quoi il est très utile.

2.3 Affichages plus complexes

Vous aurez remarqué une syntaxe plus agréable. En effet, il n'est plus nécessaire d'indiquer le type à afficher comme avec *printf*. Il est déduit par le compilateur.

Il est donc également possible d'afficher des entiers ou des flottants aussi simplement :

```
1  #include <iostream>
2
3  int main() {
4      int i {42};
5      std::cout << "le double de " << i << " est " << 2*i << std::endl;
```

[1] Aucune remarque désobligeante envers des ordinateurs qui traînent des pieds pour suivre les normes qu'ils n'ont pas établi eux-même et qui ont une pomme sur le devant. Oups, ah ben si !

```
6         std::cout << "Son carr\'e est " << i*i << std::endl;
7         return 0;
8     }
```

Vous avez remarqué l'initialisation de la variable *i* en ligne 4 ? Cette syntaxe est appelée *initialisation uniforme*. L'ancienne version héritée du C reste valable, mais il est conseillé d'utiliser celle-ci. Pour la suite, par souci de bien acquérir les bonnes habitudes, nous utiliserons toujours les formes conseillées. Il faut donc bien retenir que ce qui est *conseillé* par la norme, est en fait *obligatoire* dans le cadre de cet enseignement.

2.4 C'est à vous

Écrivez le programme *carres.cpp* qui affiche le carré des entiers de 1 à 10 sous la forme suivante :

```
Le carré de 1 est 1
Le carré de 2 est 4
Le carré de 3 est 9
Le carré de 4 est 16
Le carré de 5 est 25
Le carré de 6 est 36
Le carré de 7 est 49
Le carré de 8 est 64
Le carré de 9 est 81
Le carré de 10 est 100
```

3 Entrées

3.1 Syntaxe

La syntaxe pour demander des informations à l'utilisateur est la suivante :

```
1     int value;
2     std::cin >> value;
```

On est assez proche de la syntaxe permettant l'affichage, ça ne devrait gêner personne. Par contre, cela devient plus embarassant si on veut intercepter les erreurs de l'utilisateur. En effet, comment repérer si la valeur entrée par l'utilisateur n'est pas un entier ?

3.2 Gestion des erreurs de saisie

La syntaxe est plus inhabituelle. Il faut savoir que `std::cin >> valeur` renvoie un résultat (vrai ou faux) qui détermine si la conversion s'est bien passée. S'il y a eu un problème, on peut le régler grâce à `clear` et `ignore`. Démonstration :

```
1     int value;
2     std::cout << "entrez un entier" << std::endl;
3     if (std::cin >> value) {
4         // using value
5         std::cout << value + 1 << std::endl;
6     } else {
7         // problem using value
8         char response[100];
9         std::cin.clear();
10        std::cin.ignore(0);
11        std::cin >> response;
12        std::cout << response << " n'est pas un entier" << std::endl;
13    }
```

Ce programme demande d'abord un entier. Si la saisie pose problème (s'il n'est pas possible de convertir l'entrée en entier) le test est faux et on passe dans la clause `else`. Dans ce cas, `std::cin.clear()` efface l'erreur en cours sur la saisie, et `std::cin.ignore(0)` précise qu'on ne veut supprimer aucun caractère, ce sera utile pour récupérer l'ensemble de la saisie dans `response`, mais cette fois sous forme de chaîne de caractères pour éviter les erreurs.

Si, au contraire vous voulez jeter la saisie erronée, alors utilisez plutôt `std::cin.ignore(100)` pour ignorer *jusqu'à* 100 caractères (s'ils sont présents) ou `std::cin.ignore('\n')` qui va supprimer les caractères de l'entrée jusqu'au saut de ligne. Vous pouvez également utiliser une combinaison des deux, comme `std::cin.ignore(100, '\n')` si le besoin s'en fait sentir, ou définir un autre caractère jusqu'auquel tout supprimer.

3.3 Au Taf !

Histoire de vous faire la main, écrivez un programme (déjà écrit l'année dernière) qui calcule les racines d'un polynôme de degré 2. Ce programme doit être tolérant aux erreurs de saisie. L'objectif est de le faire avec la syntaxe du C++. Par contre, il n'y a pas besoin de pousser le détail au point de sortie les racines complexes. Les racines réelles suffiront. Pour calculer les racines carrées, vous aurez besoin de `std::sqrt()` accessible avec `#include <cmath>`.

3.4 Un petit mot sur les flux d'entrée/sortie

Vous l'aurez remarqué, en C++ les fonctions telles que `printf` et `scanf` sont remplacées[2] par de mystérieux `std::cin` et `std::cout`. On appelle cela des flux d'entrée/sortie. Sans entrer dans le détail, ils agissent comme des interfaces vers les entrées/sorties du programme. Comme vous l'avez vu avec `std::cin` ils embarquent aussi d'autres fonctionnalités utiles comme la gestion d'erreurs.

Les opérateurs `>>` et `<<` permettent respectivement de *lire* dans un flux d'entrée, et d'*écrire* dans un flux (*stream*) de sortie (d'où le nom de la bibliothèque `iostream` : *input/output streams*).

Au cours de votre exploration du langage, vous trouverez d'autres cas d'usages des flux (*manipulation de chaînes de caractères, écriture dans un fichier...*). Comme leur usage est similaire, vous saurez vous en servir grâce à ce que vous aurez vu aujourd'hui.

4 Les *conteneurs*

Dans cette partie, on va commencer à entrevoir la puissance du C++. En effet, en C, lorsqu'il est nécessaire de mémoriser plusieurs éléments de façon indexée, il n'y a que deux choix, les tableaux ou une organisation faite à la main à base de pointeurs (la plus basique étant la liste chainée). Le C++ fournit de nombreux types qui, bien que basés sur des tableaux ou l'usage de pointeurs, sont standardisés et déjà prêts à l'usage.

De façon générale, les types de données qui permettent de mémoriser d'autres types sont appelés *conteneurs* (*container* EN). Il partagent tous des aspects communs que nous allons voir progressivement.

4.1 Les tableaux

Alors, oui, les tableaux existent déjà en C, on les déclare par `int tableau[42]`, mais il existe un type C++ très proche, équivalent mais beaucoup plus pratique. Désormais, à chaque fois que nous mentionerons un *tableau*, il s'agira du type défini par `std::array`. Ce type nécessite la bibliothèque *array* qui n'est pas incluse par défaut.

Un programme minimaliste qui utilise un tableau de 5 *int* appelé *tableau* ressemblera à ça :

```
1  #include <array>
2
3  int main() {
4      std::array<int, 5> tableau;
5      for (int i=0; i<5; i++) {
6          tableau[i] = 3*i + i*i;
7      }
8      return 0;
9  }
```

Et là commence la révolution !!!

- On peut connaitre la taille du tableau en faisant appel à la fonction size. Par exemple, pour un tableau `tab`, on pourra connaitre sa taille en appelant `tab.size()`.

[2] Elles existent toujours, mais on évite de s'en servir, par convention.

- Lorsqu'un tableau est passé à une fonction, on envoie une copie et pas l'original. Il est donc possible de le modifier sans conséquences. De même, il est possible de renvoyer un tableau comme étant le résultat d'une fonction. Il n'y aura pas de problème de déréférencement des données puisque la valeur retournée ne sera pas l'original, mais une copie.

Bon OK, le dernier point peut être gênant quand on manipule des gros tableaux, mais on verra comment contourner ce souci. Le point réellement gênant avec les tableaux, c'est que si une fonction doit en recevoir un en paramètre, elle doit connaitre sa taille à l'avance (ou faire appel à des notions que l'on verra *bien* plus tard)

Le tableau est le plus basique des *conteneur*s, mais on a déjà franchi un pas en terme de facilité d'usage. Tout ce que nous verrons sur les tableaux est valable pour tous les *conteneur*s.

4.2 Élévation au carré des éléments d'un tableau

Écrivez une fonction qui reçoit un tableau d'entiers et qui renvoie un tableau d'entiers de même taille. Chaque case du tableau renvoyé contiendra le carré de la case correspondante dans le tableau reçu au lancement de la fonction. Vous testerez votre fonction dans un programme qui l'applique sur au moins deux tableaux différents.

4.3 Les vecteurs

Le type vecteur est un conteneur très proche du type tableau. À une différence près, on peut changer sa taille ! Et ça change tout ! La déclaration minimale devient donc :

```cpp
#include <vector>

int main() {
    // sans initialisation (donc de taille nulle)
    std::vector<int> vecteur_u;

    // avec initialisation (donc de taille 4)
    std::vector<int> vecteur_i {1, 2, 3, 4};

    // avec initialisation a zero (de taille constante)
    std::vector<int> vecteur_j (10);

    ...
    ...

    return 0;
}
```

En plus de `.size()`, il est possible d'utiliser de nombreuses autres fonctions :

- `.empty()` : renvoie un bouléen, pour savoir si le vecteur est vide (surprise !)
- `.front()` : renvoie le premier élément du vecteur
- `.back()` : renvoie le dernier élément du vecteur
- `.push_back(x)` : ajoute l'élément `x` à la fin du vecteur
- `.pop_back()` : supprime le dernier élément du vecteur et le renvoie
- `.assign(n, x)` : efface l'ancien vecteur, et crée un vecteur contenant `n` fois l'élément `x` à la place.
- `.resize(n)` : modifie la taille du vecteur. Si la nouvelle taille est plus grande, les éléments ajoutés sont mis à *la valeur par défaut*. Il est également possible d'utiliser `.resize(n, x)` où `x` est la valeur à placer dans les nouvelles cases

4.4 Tableaux ou vecteurs ?

Sur un ordinateur *classique*, avec un système d'exploitation *normal*, l'usage des vecteurs est bien plus souple, pour un surcout négligeable, voire nul, en temps de calcul et en empreinte mémoire. On utilisera donc plutôt des vecteurs.

Pour des applications embarquées (microcontrôleurs, systèmes d'exploitations spécifiques, ...) les tableaux peuvent être le seul choix possible, principalement à cause de la gestion mémoire. Nous dirons sobrement que lorsque vous aurez atteint un niveau où cette question est importante, vous aurez de très loin oublié ce TP :-).

4.5 Itération sur un *conteneur*

Histoire de pousser encore plus le niveau d'abstraction, il est possible de faire des itérations sur les *conteneur*s. En effet, si tab est un tableau de flottants, on a très souvent besoin d'écrire des structures ressemblant à :

```
1    for (int i=0; i<tab_size; ++i){
2        float elemt = tab[i];
3        ...
4        ...
5    }
```

Vous aurez déjà remarqué la notation démodée propre au C (que vous n'avez plus le droit d'utiliser). En C++, on considère que tab est un `std::array<float>` et la boucle pourrait s'écrire :

```
1    for (int i {0}; i<tab.size(); ++i){
2        float elemt = tab[i];
3        ...
4        ...
5    }
```

C'est déjà bien mieux, mais pourquoi s'embêter à utiliser un indice si on ne s'en sert pas dans le contenu de la boucle ? Le C++ permet d'écrire directement :

```
1    for (float elemt : tab){
2        ...
3        ...
4    }
```

Cette technique devient malheureusement inutilisable si on a besoin de la valeur de l'élément ET de son indice dans le tableau.

4.6 Un peu de pratique

Écrivez une fonction qui affiche tous les éléments d'un tableau ou d'un vecteur (à vous de choisir lequel, et si l'affichage se fait sur une seule ligne ou avec un élément par ligne).

5 Les références

Un dernier concept, et c'est fini. Le C++ introduit la notion de référence. Cette notion a déjà partiellement été introduite en C, puisqu'un pointeur peut être utilisé comme référence. Cette fois-ci, nous sommes face à un usage un peu restreint, mais qui améliore la prise en main, et permet d'éviter de nombreux bugs. Si on déclare une variable par la ligne suivante: `int& ref {a}`, cela signifie que `ref` est une référence vers `a`. Il n'y aura donc aucune différence entre modifier `ref` et modifier `a`.

5.1 Différence avec les pointeurs

A première vue, la différence est très faible avec l'usage de pointeurs, et ce concept peut sembler superflu, et pourtant...

- Il n'est pas possible de créer/déclarer une référence sans l'initialiser.
- Il n'est pas possible d'initialiser une référence à `NULL`

- Il n'est pas possible de changer la cible d'une référence. Une fois la référence créée, elle est indissociable de l'original. À tel point qu'il n'est même plus légitime de parler d'original et de référence.

Une référence est donc TOUJOURS valide et ne génère aucun risque d'erreur de segmentation.

5.2 Usage

Le réel intérêt des références est de compenser le comportement du C++ qui travaille toujours avec des copies. Par exemple, une fonction qui met tous les éléments d'un vecteur à zero s'écrira comme suit:

```
1    void zero(std::vector<int>& vect){
2        for (int& elt : vect) {
3            elt = 0;
4        }
5    }
```

Dans ce cas de figure, seule la référence sur le vecteur passé en argument est copiée, et non le vecteur entier. Ouf ! par contre, deux références sont nécessaires. En effet, on travaille sur les éléments originaux du tableau original.

6 De la pratique, enfin !

L'objectif des exercices suivants est bien évidemment de vous entrainer. L'objectif n'est donc pas uniquement de produire un résultat correct, mais de le faire de la façon qui vous parait la plus efficace possible.

Bien évidemment, chaque fonction doit être accompagnée du programme qui en teste le bon fonctionnement.

- Somme des éléments d'un *conteneur* : Écrivez la fonction `sum` qui calcule et renvoie la somme des éléments d'un *conteneur*. Le type de *conteneur* et d'éléments à l'intérieur est laissé à votre choix.
- Produit scalaire Écrivez la fonction `scal` qui calcule et renvoie le produit scalaire de deux conteneurs de même dimension.
- Somme membre à membre Écrivez la fonction `add` qui calcule et renvoie la somme membre à membre entre deux tableaux.

7 Chaînes de caractères

Les chaînes de caractères étaient implémentées comme des tableaux de type `unsigned char` en C, il est donc logique qu'en C++, ce soit un conteneur équivalent. C'est effectivement le cas, mais pour en améliorer la lecture, un type spécial a été défini : il s'agit du type `std::string` accessible dans la bibliothèque `<string>`.

7.1 Détails

Ce que nous avons vu sur les conteneurs reste valable. Une variable de type `std::string` peut être utilisée avec toutes les techniques vues pour les vecteurs. Cela simplifie beaucoup de tâches de manipulation. Mais le plus pratique, c'est qu'il est maintenant possible de concaténer des `string`s de façon assez intuitive, avec l'opérateur '+'. Ainsi, pour dire bonjour au monde, il est possible d'écrire le code suivant, ou toute autre variante :

```
1    #include <iostream>
2    #include <string>
3    using namespace std::literals;
4
5    int main()
6    {
```

```
7         std::string mystring;
8         mystring = "Hello "s;
9         mystring += "World"s;
10      std::cout << mystring << std::endl;
11        return 0
12    }
```

7.2 Usage

En plus de toutes celles liées aux vecteurs, les chaînes de caractères sont associées aux fonctions suivantes :

- `.length()` : comme `.size()`, mais c'est plus cohérent de parler de longueur que de taille.
- `.find(str)` : si `str1` et `str2` sont deux chaînes de caractères, `str1.find(str2)` renvoie la position de `str2` dans `str1`. Éventuellement, il est possible d'ajouter la position à partir de où il faut chercher. La fonction est alors `.find(str, pos)`.
- `.find_first_of(str)` : `str1.find(str2)` cherche dans `str1` la première fois où on rencontre un caractère de `str2`. Comme pour `.find()`, on peut préciser une position de départ.
- `.find_last_of(str)` : `str1.find(str2)` cherche dans `str1` la dernière fois où on rencontre un caractère de `str2`. Tout pareil, on peut préciser une position de départ, mais cette fois-ci, on part de la fin de la chaîne.
- `.substr(pos, len)` : renvoie un extrait de la chaîne qui commence à la position `pos` et dont la longueur est `len`.
- `.at(n)` : si `str` est une chaîne de caractères, `str.at(n)` renvoie une référence vers `str[n]` (alors que `str[n]` renvoie une copie)

Malgré le grand intérêt que présente le type `std::string`, il arrive qu'il soit encore nécessaire d'utiliser la représentation du C (un pointeur vers une zone de données qui contient des caractères, puis le caractères nul poru marquer la fin). Pour différencier les représentations on utilise le terme *C-string* pour préciser qu'on manipule un `char *`. La conversion d'une *C-string* en `std::string` se fait de façon transparente (par exemple : `std::string Snew {C_string}` où la variable `C_string`) est de type `char *`. Mais pour l'inverse, on utilise les fonctions `.C_str()` et `.data()` qui sont équivalentes. Elles renvoient un copie de la chaîne de caractère au format *C-string*.

D'ailleurs, lorsqu'une chaïne de caractère est entrée sous forme de litéral (écrite en dur), elle est toujours implémentée sous forme de *C-string* pour assurer la compatibilité avec le C. Pour qu'un litéral soit implémenté sous forme de `std::string`, il faut lui ajouter le suffixe `s`, comme aux lignes 7 et 8 de l'exemple de code précédent. Cette notation nécessite d'inclure la ligne `using namespace std::literals;`.

Il existe également des fonctions plus classiques pour aider les conversions. Elles s'utilisent plus simplement avec une chaîne de caractère en argument.

- `std::stoi(str)` : convertit une chaine de caractère en entier (`int`)
- `std::stol(str)` : convertit une chaine de caractère en long entier (`long`)
- `std::stoll(str)` : convertit une chaine de caractère en `long long`
- `std::stoul(str)` : convertit une chaine de caractère en `unsigned long`
- `std::stoull(str)` : convertit une chaine de caractère en `unsigned long long`
- `std::stof(str)` : convertit une chaine de caractère en flottant (`float`)
- `std::stod(str)` : convertit une chaine de caractère en `double`
- `std::stold(str)` : convertit une chaine de caractère en `long double`

Pour chacune de ces fonctions, il est possible d'ajouter un pointeur vers un entier `std::stoi(str, &pos)`. Cet entier est alors mis à jour avec la position à laquelle s'arrête la valeur dan sla chaîne. Si ce pointeur est null (0 ou `NULL`), il est ignoré.

Pour les conversions vers ds types entiers, il est possible de spécifier un troisième argument de type entier. Il s'agit alors de la base de conversion. Il est ainsi possible de convertir directement des valeurs binaires, octales ou hexadécimales.

7.3 Exercice

Écrivez une fonction calculatrice qui convertit en flottant un chaine de caractères contenant des flottants reliés par les opérations +,-,*,/. Il n'est pas nécessaire de gérer les parenthèses, par contre, le nombre d'opérations est quelconque (pas juste une opération donc) et les priorités doivent être respectées.

Pour cela, le plus simple est d'utiliser la récursivité :

- On commence par chercher le caractère '*' ou '/'
- Si il a été trouvé
  - On extrait la sous-chaîne de caractères avant l'opération
  - on calcule de résultat de la sous-chaîne récursivement
  - On extrait la sous-chaîne de carcatères après l'opération
  - on calcule de résultat de la deuxième sous-chaîne récursivement
  - il n'y a plus qu'à calculer l'opération avec les deux résultats
  - on retourne le résultat
- On cherche le caractère '+' ou '-'
- Si il a été trouvé, on effectue le même genre d'opération que précédemment
  - extraction / calcul sur les sous-chaînes
  - ...
  - calcul de l'opération
  - retour du résultat
- A ce niveau, on devrait avoir un litéral
- retirer les espaces au début et à la fin de la chaîne
- si la chaîne est vide : renvoyer zero
- appeler `std::stof` sur la chaîne restante

8 Conclusion (partielle)

8.1 Pourquoi avons-nous commencé par le C ?

Parce que même si vous ne vous rendez pas encore compte, vous avez acquis beaucoup de réflexes avec le C, qui vous font croire que le C++ est *facile*. Sans ces réflexes imposés par le C, l'approche du C++ serait moins rigoureuse. Il ne faut pas non plus oublier que le C reste plus répandu que le C++.

8.2 Pourquoi le C est-il encore si tenace ?

Parce qu'on ne s'amuse pas à migrer des milliers de lignes de code d'un langage vers un autre juste parce qu'il est meilleur. Il y a aussi les compétences de toute l'équipe à mettre à jour, les outils, les procédures de vérification et de validation, ... Tant qu'il sera facile de trouver des gens compétents en C, la migration ne sera pas économiquement viable.

Parce que, surtout en electronique, il y a de nombreuses machines dont l'architecture ne permet pas de supporter le C++. Le simple type `std::vector` n'est pas implémentable sur un PIC16F84, sur des PIC16F plus gros, il consommerait l'essentiel des ressources mémoire.

Le dernier point est que tout gain a sa contrepartie : on gagne en souplesse, mais on perd en maîtrise. C'est parfois important selon les applications.

8.3 La flexibilité du C++ a-t-elle un coût ?

En terme de temps d'éxécution, oui, mais il est assez faible, car on fait bien plus souvent appel à des bibliothèques qui, elles, sont très optimisées. Pour un mauvais programmeur, on pourrait même se risquer à dire que le C++ sera plus rapide.

En terme d'empreinte mémoire, un programme en C++ sera plus gros, et demandera plus de mémoire pour s'éxécuter, mais encore une fois, pour réduire l'empreinte mémoire, il est beaucoup plus rentable de bien choisir son algorithme et ses structures de données plutôt que son langage de programmation.