PG208
Programmation Orientée Objet / Langage C++
TP2

Y. Bornat A. Valade

April 5, 2024

1 Gestions de fichiers

Comme précisé dans le TP1, la gestion des fichiers se fait sous la forme de Flux d'entrée et de sortie. Ces flux permettent de nombreuses subtilités, mais nous allons surtout nous focaliser sur un usage équivalent au C. Pour accéder aux fonctions liées aux flux d'entrée et de sortie il faut inclure la bibliothèque `<fstream>`. Les flux d'entrée et de sortie correspondent respectivement aux types `std::ifstream` (pour l'entrée, donc la lecture) et `std::ofstream` (pour la sortie donc l'écriture).

Cela fait notre première différence avec le C, où on utilisait le même type de donnée pour les fichiers en lecture et en écriture (un *descripteur de fichier* ou `FILE *`),

1.1 Écrire un fichier

1.1.1 Syntaxe

Avant d'écrire dans un fichier, il faut l'ouvrir, c'est à dire créer le flux en sortie. Pour cela, la syntaxe de l'initialisation uniforme est très pratique : `std::ofstream fichier {"fichier.txt"}` Notez bien la différence entre l'affectation et l'initialisation : le flux `fichier` ne vaut pas la chaîne de caractère `"fichier.txt"`, mais l'utilise pour son initialisation.

Un programme qui écrit dans un fichier va donc s'écrire comme suit :

```
#include <fstream>

int main()
{
    std::ofstream fich {"fichier_test.txt"};
    fich << "hello";
    return 0;
}
```

Vous remarquerez que des concepts du C ont disparu. En effet, on ne parle plus d'ouvrir et de fermer le fichier. Techniquement ces opérations sont toujours présentes, mais elles sont cachées par la nature même du type `std::ofstream`. En effet l'ouverture du fichier se fait au moment de l'initialisation et la fermeture se fait à la fermeture de l'accolade, lorsque la variable `fich` est détruite. Dans certains cas particuliers (ou si vous êtes masochiste) il reste possible d'effectuer ces opérations manuellement. Le code ci-dessous vous montre comment :

```
#include <fstream>

int main()
{
   std::ofstream fich;
   fich.open("fichier_test.txt");
   fich << "hello";
   fich.close();
   return 0;
}
```

1.1.2 Fichiers déjà existants

On ne maitrise pas tout dans la gestion de fichiers. Il se pourrait notamment qu'une erreur se produise lors de l'initialisation du flux. Pour repérer ce genre de chose, le flux se comporte comme un bouléen. Il est donc possible de le tester pour voir si l'ouverture s'est bien passée :

```
#include <iostream>
#include <fstream>

int main()
{
   std::ofstream fich {"fichier_test.txt"};
   if(fich) {
       fich << "hello";
   } else {
       cout << "Oooops ! Erreur de fichier" << std::endl;
   }
   return 0;
}
```

De la même façon, il est possible d'ajouter un paramètre pour indiquer comment le programme doit se comporter si le fichier existe déjà :

- `ios::trunc` : écrase le fichier qui existe déjà
- `ios::ate` : positionne l'écriture en fin de fichier
- `ios::app` : positionne l'écriture en fin de fichier et interdit de revenir en arrière

Ainsi, si vous ouvrez le fichier par `std::ofstream fich {"toto.txt", ios::app}`, il sera impossible de modifier ce qui est déjà écrit dedans.

1.1.3 Se positionner dans le fichier

Comme en C, il est possible de se déplacer dans le fichier pour modifier des données déjà écrites. Pour connaitre la position en cours, on utilise la fonction `.tellp()`. Si on est en fin de fichier, `.tellp()` correspondra donc à la taille de ce dernier. Pour modifier la position, on utilise `.seekp(pos, ref)`. `pos` est un entier (potentiellement négatif), et `ref` définit la référence à utiliser :

- `ios::beg` : Le début du fichier (`pos` est positif ou nul)
- `ios::end` : La fin du fichier (`pos` est négatif ou nul)
- `ios::cur` : La position en cours (`pos` est non nul)

1.2 Lire un fichier

1.2.1 Syntaxe

Sans surprise, pour lire un fichier, on utilisera la syntaxe suivante :

```
#include <fstream>

int main() {
    std::ifstream fich {"data.txt"};
    if(fich) {
        int a;
        std::string s;
        fich >> a;
        fich >> s;
    } else {
        cout << "ERREUR";
    }
}
```

La lecture en utilisant la notation de flux permet d'interpréter les données. C'est à dire que si la variable de destination est un entier, cet entier sera déduit des caractères constituant le fichier. Si cette interprétation échoue, il reste possible d'utiliser `.clear()` et `.ignore()` pour régler le problème.

Il est également possible de lire le flux comme une suite de caractères. La fonction `.get(c)` modifie `c` et lui donne la valeur du caractère suivant. Cette fonction renvoie un bouléen qui permet de savoir si on est arrivé à le fin du fichier. Si c'est le cas, la valeur de `c` n'est pas significative.

La dernière technique est la lecture ligne par ligne grâce à la fonction `std::getline(f, str)` qui place dans `str` une ligne complète du fichier. Cette ligne est affectée sans le saut de ligne final. Comme d'habitude, cette fonction renvoie un bouléen pour identifier les erreurs ou les fins de fichier.

1.2.2 Se positionner dans le fichier en lecture

Pas de grande surprise, la syntaxe est très proche des flux en écriture, juste avec des noms un peu différents `.tellg()` et `.seekg(pos, ref)`.

1.3 Nom de fichier paramétrable

Dans la grosse majorité des cas, les noms de fichier à lire ou écrire ne sont pas codés en dur dans le programme. Et là, c'est le drame ! En effet, l'initialisation des flux a besoin de connaître le nom de fichier au format *C-string*. À partir du moment ou votre nom de fichier est mémorisé dans une variable de type `std::string`, il faudra donc la convertir avec la fonction `.c_str()`.

1.4 Exercice

Écrivez un programme qui prend deux noms de fichiers en ligne de commande. Le premier fichier sera lu (nous l'appellerons *source*), le second fichier sera écrit (et donc écrasé s'il existe déjà) nous l'appellerons *destination*. Votre programme doit copier *source* dans *destination* ligne par ligne, en inversant l'ordre. Ainsi, la première ligne de source doit être la dernière de destination et vice versa. Si vous êtes joueur, vous pouvez tester votre programme sur vos propres fichiers source. Sinon, il y a le fichier `jourdain.txt` sur la page de ressources.

Rappel : pour accéder aux arguments de la ligne de commande, il n'y a pas de syntaxe C++ particulière. On utilisera donc la syntaxe du C qui consiste à déclarer `main` par la ligne suivante : `int main (int argc, char *argv[])`. Où `argc` est le nombre d'éléments dans `argv`, et `argv` un tableau de *C-string*.

2 Les espaces de nom

C'est fatigant de taper tous ces `std::` n'est-ce pas ? À quoi cela correspond-il ? Il s'agit de l'espace de noms. Voilà. C'est tout.

2.1 Pour quoi faire ?

Il ne faut pas oublier que l'esprit du C++ est de permettre la migration en douceur d'un projet en C. Et à quelques exceptions près, c'est réussi. Le souci est que de (très) nombreux mot-clés ont été ajoutés dans la syntaxe du C++, et ces mot-clés étaient potentiellement déjà utilisés dans des programmes en C. Il fallait donc trouver une technique qui permet de préciser, en cas de conflit, quelle est la définition à retenir pour le nom. En demandant de préciser `std::` devant chaque mot-clé, il devient clair que c'est la version du standard C++ qui est demandée puisque cette syntaxe est illégale en C. Cette syntaxe permet notamment de créer une fonction ou un type `vector` sans qu'il y ait de problème de conflit.

Un autre avantage, c'est que pour les gros projets, il est possible de créer son propre espace de nom. Par exemple, pour les bibliothèques système, graphique ou de calcul scientifique, il est fréquent d'avoir énormément de fonctions à gérer. Vu leur grand nombre, il faut soit des noms particulièrement longs, soit un espace de nom différent. Dans ce dernier cas, il faut simplement que l'étiquette de l'espace de nom soit courte, et différente de celle du voisin :-).

2.2 C'est vraiement indispensable ?

Un programmeur étant, par nature, avare de frappe de touches, il existe une ligne permettant de s'en affranchir : `using namespace std;`. Pour le bloc qui contient cette ligne, le compilateur considérera que

tout l'espace de nom `std::` est fusionné avec l'espace de nom courant. Et hop ! tout le code est plus clair. D'autant qu'on peut aussi importer d'autres espaces de noms.

2.3 Pourquoi le garder ?

La technique précédente, bien que très confortable, n'est pas considérée comme une bonne pratique de codage. Dès que le projet devient conséquent (en nombre de lignes, en nombre de participants ...) les espaces de noms rendent beaucoup de services :

- Face à un identifiant inconnu, on sait immédiatement dans quelle doc rechercher. Si c'est `std::`, on cherche dans la doc du C++, s'il n'y a pas d'espace de nom, c'est dans la doc du projet en cours, si c'est autre espace de nom ...
- C++ version 2008 contenait environ 550 mot-clefs dans l'espace de nom standard, pour C++11, c'est 1100. C++14 ? 1220 (petite année). C++23 : 1820. Qu'est-ce qui garantit que les noms de fonction que vous avez choisi aujourd'hui seront encore disponibles l'année prochaine ? Le simple fait que tous ces nouveaux mot-clés sont protégés par un espace de nom différent.
- La remarque précédente s'applique à tout autre sous-projet régulièrement mis à jour. En milieu professionnel, on déteste les surprises, car la loi de Murphy fait qu'elles sont rarement bonnes. Un projet en bonne santé utilise les espaces de nom.

La politique pour cet enseignement est de ne pas favoriser la fusion des espace de nom, mais (pour une fois) de ne pas l'interdire non plus. Faites votre choix, et assumez-le.

3 Tableaux associatifs

Maintenant, on va attaquer des choses un peu plus bizarres. Jusqu'ici, les tableaux que vous avez utilisé en programmation associaient les éléments stockés à un entier. Pour un tableau de taille n, les éléments étaient donc identifiés par une valeur allant de 0 à $n-1$[1]. Et pourquoi d'abord ? Et si on veut faire autrement ? Les tableaux associatifs permettent d'identifier les cases du tableau avec des entiers arbitraires, comme des numéros de téléphone, ou même des types qui ne sont pas des entiers. Juste pour le plaisir.

Les tableaux associatifs peuvent être ordonnés, ou pas. C'est selon les besoin du programmeur. Nous ne passerons du temps que sur les tableaux non ordonnés, car leur implémentation est plus efficace. Dans la suite du sujet, il seront mentionné sous leur nom de déclaration : `unordered_map`.

3.1 Syntaxe

Un peu de terminologie d'abord : pour chaque élément stocké dans le `unordered_map`, son identifiant est appelé *clé*. La clé peut être un entier, un flottant, ou même une chaîne de caractères (c'est ça qui est beau). L'élément stocké sera simplement nommé *element*.

La déclaration se fait en mentionnant le type de la clé d'abord puis le type de l'élément ensuite. Par exemple, pour un `unordered_map` dont les clés sont des `std::string` et dont les éléments sont des `int` on se retrouvera avec la déclaration suivante : `std::unordered_map<std::string, int>`.

Le code suivant donne un exemple d'usage complet :

```
1   #include <iostream>
2   #include <unordered_map>
3
4   int main()
5   {
6       // declaration :
7       //    keys     : string
8       //    elements : float
9       std::unordered_map<std::string, float> eval;
10
11      // element insertion
12      eval["Bornat"]  =  1.5;
```

[1] Qui a parlé de MATLAB ? c'est quoi MATLAB ?

```
        eval["Kolbl"]   =  3.5;
        eval["Renaud"]  =  2.5;
        eval["Taris"]   =  3.0;
        eval["Valade"]  = 17.25;
        eval["Vincent"] =  2.75;

        // element read
        std::cout << eval["Taris"]  << std::endl;
    }
```

Comme tous les conteneurs, il est fourni avec les fonctions `.size()`, `.empty()` et `.at()`. En voici d'autres un peu plus spécifiques :

- `.count(key)` : renvoie 1 si la clé `key` est utilisée ou 0 sinon.
- `.insert({key, elt})` : Insère un nouvel élément dans le tableau. La différence avec l'affectation classique est que l'insertion échoue si la clé est déjà utilisée. Notez bien que `.insert()` ne prend pas deux arguments mais un seul constitué des deux parties
- `.erase(key)` : Efface l'élément dont la clé est donnée en argument
- `.clear()` : Efface tout le tableau

A quoi ça sert ? Essentiellement à ranger des structures en fonction d'une valeur. Imaginez que vous avez une structure qui permet d'identifier des étudiants, des livres ou des véhicules. Sur un système bien ordonné, chaque élément sera associé à un entier qui servira d'identifiant, et qui correspondra certainement à la case du tableau dans lequel sont rangées toutes les informations disponibles. Mais comment faire pour retrouver un élément si vous ne connaissez pas ce numéro ? Soit il faut parcourir tous les éléments, soit il faut créer un système de classification qui soit un peu plus efficace. C'est concrètement ce que fait `unordered_map`, donc pas besoin de le refaire vous-même. Les avantages sont que le travail est déjà fait, il n'y a pas de bugs, la documentation est abondante et les autres contributeurs potentiels sont déjà opérationnels.

3.2 Itération

Il peut arriver qu'il soit nécessaire de faire une itération sur l'ensemble des éléments d'un `unordered_map`. Pour cela est-ce que l'on obtient les clés ou les éléments ? Ben les deux. Pour itérer sur un tel conteneur, il faut utiliser le type `std::pair` qui contient essentiellement deux attributs : `.first` et `.second`. `.first` correspond à la clé, et `.second` à la valeur de l'élément. Voici à quoi cela ressemble dans l'exemple précédent :

```
    for (std::pair<std::string, float> item : eval) {
        std::cout << item.first << " : " << item.second  << std::endl;
    }
```

Petite remarque : quand vous testez ce code, l'ordre d'affichage (donc l'ordre d'itération) n'a rien à voir avec l'ordre dans lequel les éléments sont entrés, ou tout autre ordre de classement. Cela est dû à l'organisation interne de la strucure. Si vous voulez qu'un ordre soit conservé, il faut alors utiliser `map`, mais dont l'usage est plus gourmand en ressources.

A propos du type `std::pair`, il s'agit du cas particulier d'un type utilisé assez courant en objet, le *tuple*. Nous y reviendrons en temps voulu, ou pas, selon l'avancée tout au long des séances.

3.3 Exercice

La page de ressources contient plusieurs textes déjà traités pour éviter les soucis. Notamment, il ne contiennent que des lettres, des chiffres et les caractères suivants : `-,.:;'"!?()#`. Vous choisirez celui de votre choix.

- Écrivez un programme qui donne la distribution des lettres de ce texte. C'est à dire, le nombre de fois ou chaque lettre est utilisée (Nous considèrerons qu'une lettre accentuée est différente de la même lettre non accentuée ou avec un autre accent).

- Écrivez un programme qui donne la distribution de mots de ce texte. C'est à dire, pour chaque mot, le nombre de fois qu'il apparait. Nous considèrerons que les mots sont des suites de caractères séparés par des espaces ou des sauts de ligne, toute forme de ponctuation est un séparateur de mot.

3.4 Question philosophique

Pourquoi est-il dangereux d'utiliser des flottants comme clé pour un tableau associatif ?

4 Surcharges

Allez ! On commence les trucs plus rigolos ! Vous avez remarqué qu'une fonction peut avoir des comportements différents selon la nature et le nombre de ses arguments ? C'est parce que, selon le cas, il s'agit tout simplement de fonctions différentes qui ont le même nom.

4.1 La triche : Valeurs par défaut

La première possibilité, qui dans les fait n'est *pas* une surcharge, c'est de fournir une valeur par défaut aux arguments. En effet, à partir du moment où un argument a une valeur par défaut, il devient optionnel. Ainsi, la fonction `add` suivante peut être utilisée avec deux, trois ou quatre arguments :

```
1  float add(float x1, float x2, float x3 = 0.0, float x4 = 0.0) {
2      return x1 + x2 + x3 + x4;
3  }
```

Si utilisée avec deux arguments (pour `x1` et `x2`), la fonction utilisera 0.0 pour `x3` et `x4`. Bien qu'il soit techniquement possible d'utiliser `add` avec un ou même aucun argument, il faut reconnaitre que c'est un peu inutile dans notre cas.

4.2 La vraie surcharge

Le principe de la surcharge est que les fonctions ne sont plus distinguées *que* par leur nom, mais aussi par leurs arguments. Ainsi, une fonction `void toto(int)` peut cohabiter avec une fonction `void toto(float)` sans qu'il y ait la moindre chose en commun si ce n'est le nom. Par contre, il n'est pas possible de surcharger deux fonctions si elles ne diffèrent que par la valeur qu'elles renvoient. En effet, dans le cas `foo(bar(x))`, la valeur de retour de `bar(x)` est utilisée pour déduire quelle est la bonne fonction `foo` à utiliser. Elle ne peut donc pas servir à déduire la bonne fonction `bar`.

4.3 Exercice

Comme exemple, nous allons reprendre différentes versions de la fonction du TP1 qui permettait d'additionner deux conteneurs (bon OK, deux vecteurs). Nous l'appellerons `add`. Écrivez les différentes versions suivantes :

- `std::vector<double> add(std::vector<double> v1, std::vector<double> v2)` : La version de base, qui renvoie la somme éléments par éléments des deux vecteurs.
- `std::vector<double> add(std::vector<double> v, double d)` : Pour cette version, on additionne `d` à chaque élement de `v`
- `double add(double d, std::vector<double> v)` : Cette fois, c'est `d` qui est en premier, donc on va additionner `d` et la somme des éléments de `v`. OK, cette version est un peu inutile, mais c'est pour l'exercice. Elle prend tout son sens sur des données ou l'addition n'est pas commutative.

Bien évidemment, l'intérêt est que *toutes* ces fonctions soient déclarées et utilisées dans le même programme[2]. Là où en C, il aurait fallu plusieurs fonctions avec des noms différents et donc tout un ecosystème à connaitre, avec la surcharge, il suffit de connaitre une fonction et, si les surcharges sont bien pensées et intuitives, l'usage d'un nouveau type de données devient presque instinctif. L'étape suivante, c'est que même si c'est instinctif, ça reste lourd à lire.

Comme si on allait s'arrêter en si bon chemin.

[2] Je vous vois, celles et ceux qui commentent les fonctions qui ne leur servent plus. Pour cet exercice, c'est interdit. ;)

5 Opérateurs en ligne

Maintenant que nous avons vu comment créer une fonction d'addition qui soit capable d'accepter toute une variété d'arguments, ce qui serait vraiment bien, c'est de l'utiliser avec l'opérateur `+` comme une vraie addition. Ça tombe bien, c'est prévu. Pour ça, il suffit que votre fonction s'appelle `operator+` et accepte deux arguments. Reprenez les exemples de la partie précédente si vous ne me croyez pas.

5.1 Liste non exhaustive des opérateurs surchargeables

Il y a assez peu de limites aux opérateurs qui peuvent être surchargés. Il suffit de déclarer la fonction `operator`XX en remplaçant *XX* par l'opérateur de votre choix.

Il est notamment possible de surcharger (au moins) les opérateurs binaires suivant : `+`, `-`, `*`, `/`, `==`, `!=`, `>=`, `<=`, `>`, `<`, `&`, `&&`, `|`, `||`, `^`, `>>`, `<<`, `+=`, `-=`, `*=`, `/=`, `&=`, `&&=`, `|=`, `||=`, `^=`, `>>=` et `<<=`.

Mais il y a aussi les opérateurs unaires `+`, `-`, `~` et `!`.

Une restriction tout de même : Il n'est pas possible de surcharger les opérateurs pour des types élémentaires `int`, `char`, `float` ou `double` par exemple. Cette restriction s'applique même si l'opérateur n'est pas défini (par exemple, il n'est pas possible de définir `operator<<(float, int)` pour faire `3.14<<2`).

Petite remarque pour finir : le compilateur ne comprend et ne restreint pas les opérations effectuées par les pointeurs. Mais il faut rester cohérent avec les règles mathématiques. Notamment pour les règles de précédence (priorité), de commutativité ou autres. À ce titre, la possibilité de concaténer deux `std::string` grâce à l'opérateur `+` est décriée car elle n'est pas commutative, contrairement à l'addition.

5.2 Exercices

- Étant donné qu'il est possible de concaténer des `std::string` avec l'opérateur `+`, il semble logique que pour une `std::string s`, on définisse `2*s` comme étant `s+s`, ou encore `3*s == s+s+s`. Surchargez l'opérateur `*` pour cela.
- Toujours en surchargeant l'opérateur `*`, proposez le calcul du produit scalaire de deux `std::vector`.
- Cette fois-ci, nous allons surcharger l'opérateur `<<` pour faire une rotation des éléments d'un vecteur. Ainsi, si `v` vaut `{1,2,3,4}`, `v << 1` donnera `{2,3,4,1}`. Testez votre programme en utilisant une ligne de la forme `v1 = v2 << 2`, et une autre version en utilisant une ligne de la forme `v <<= 3`.

6 Templates

Pour revenir sur le problème de l'exercice d'addition élément par éléments. Il n'était bien sûr pas envisageable de faire des surcharges en écrivant une fonction différente pour chaque longueur. La rigidité de la déclaration qu'impose de donner la taille du tableau dans le type est un peu bloquante d'un certain abord, mais peut être très utile au final. En effet, si on définit un type temporaire uniquement dans le cadre de la fonction, on peut forcer ce même type sur les deux arguments de l'addition, et sur le résultat. Au final, on s'est un peu embêtés à définir un type, mais on est sûrs que les deux tableaux en entrée auront la même longueur. Pour cela, le mot-clef est `template`.

6.1 Syntaxe

Pour créer un *template* (ou *modèle* en bon français), la syntaxe est tout simplement `template <std::size_t SIZE>`, ou `SIZE` est le nom que l'on veut lui donner. Ainsi, une fonction d'addition élément par élément de tableaux ressemblera à :

```
1  template <std::size_t SIZE>
2  std::array<int, SIZE> operator+(const std::array<int, SIZE>& t1, const std::array<int, SIZE>& t2){
3      std::array<int, SIZE> res;
4      for (int i {0}; i<t1.size(); i++) {
5          res[i] = t1[i] + t2[i];
6      }
7      return res;
8  }
```

Remarques :

- Dans la boucle, on fait appel à `t1.size()`, on aurait aussi pu faire appel à `t2.size()` car la vérification des type de données à la compilation garantit que les deux tableaux ont la même taille. À l'usage, pour être encore plus efficace, on utilise directement `SIZE`, car le compilateur peut mieux optimiser les choses.

- Le type de SIZE est `std::size_t`. Techniquement, il s'agit d'un entier, mais le fait de préciser `size_t` permet d'être plus précis.

  – il détermine la taille de quelque chose.

  – son occupation mémoire est directement liée à la capacité de la machine.

  Techniquement, toutes les fonctions de longueur, d'accès aux éléments d'un tableau, ou même `std::sizeof()` ne renvoient pas un `int` ou un `long` mais un `std::size_t`[3] (c'est vrai aussi pour le C). C'est juste que le compilateur fait habituellement la conversion implicitement car les types sont compatibles. Ici, ce n'est pas le cas. Il faut être rigoureux sur les types de données pour éviter les problèmes lors des surcharges

- Les deux arguments sont passés comme des références constantes. WTF ? Si on déclare les entrées avec `const`, cela signifie qu'on ne peut pas les modifier. mais si on ne peut pas les modifier, pourquoi utiliser des références ??? Tout simplement pour éviter une copie. Si les tableaux ont une grande taille, éviter la copie réduit l'occupation mémoire et le temps de calcul, c'est bon pour la planète[4].

- Mais alors pourquoi ne pas retourner une référence ? Pour les mêmes raison qu'on ne renvoie jamais un pointeur vers une variable locale. La variable va être détruite et la référence cassée. Il est alors inévitable de consommer beaucoup de mémoire et de temps de copie pour implémenter cette fonction comme une surcharge de `+`, c'est une des raisons qui explique qu'elle ne soit pas présente nativement.

- Vous avez remarqué qu'il n'y a pas de ; à la fin de ligne template ? c'est tout simplement parce que cette ligne fait partie de la déclaration de la fonction mais par convention, on place le template sur la ligne au-dessus pour améliorer la lisibilité. Cela signifie aussi que le template est local (spécifique à la fonction).

6.2 La généricité

Il est possible de pousser le concept encore plus loin. En effet, le type de donnée que contient le tableau a peu d'importance. Il suffit que l'opérateur `+` soit définit pour ce type. Il est donc possible d'écrire une fonction unique, indifférente du type contenu dans le tableau.

```
1  template <typename CONTENT, std::size_t SIZE>
2  std::array<CONTENT, SIZE> operator+(const std::array<CONTENT, SIZE>& t1, const std::array<CONTENT,
   SIZE>& t2){
3      std::array<CONTENT, SIZE> res;
4      for (int i {0}; i<SIZE; i++) {
5          res[i] = t1[i] + t2[i];
6      }
7      return res;
8  }
```

Remarques :

- Cette surcharge va fonctionner avec tous les tableaux contenant tout type de donnée pour lequel l'opérateur `+` est défini. C'est à dire (à la louche), les `int`, `char`, `short`, `long`, `float`, `double` (classique), mais aussi `std::string` (eh oui !) et surtout `std::array`, puisque qu'on vient de le définir. En définissant l'addition sur les tableaux, on a, presque par accident, aussi défini l'addition sur les tableaux de tableaux, ou les tableaux de tableaux de tableaux ... .

[3] Pour celles et ceux qui activent le flag `-Wall` à la compilation, vous avez du vous en rendre compte en essayant de compiler `for (int i  0 ; i < v.size() ; i++)`. Un `std::size_t` comptant comme un entier non-signé, `g++` va vous prévenir que vous comparez un entier signé et un entier non-signé.

[4] Techniquement, la planète survivra, pour être plus précis, c'est bon pour la biodiversité et donc pour l'espèce humaine. Vous pouvez également être égoïste à plus court terme en disant que c'est bon pour mieux vendre son produit

- Si pour un besoin quelconque il est nécessaire de créer une variable ayant le type contenu dans le tableau, il suffira de la déclarer par `CONTENT variable { ...};`, puisque le template prévient le compilateur que `CONTENT` est un type de donnée. Ici, l'initialisation uniforme prend tout son sens, car on ne sait pas si `CONTENT` est un type de base (`int`, `double` ...) un conteneur, ou même un `struct` quelconque ...

6.3 Exercices

- Écrivez la fonction `total` qui calcule la somme de tous les éléments d'un vecteur, sans restriction sur le type stocké dans ce vecteur, à condition que ce type soit supporté par `operator+`.
- Écrivez la fonction `display`, qui, pour tout conteneur, affiche son contenu sous la forme : `indice : element`, à raison d'un élément par ligne (pour celle-ci il y a un piège :-)).

7 Conclusion

Cette séance a permis d'expérimenter des comportements de niveau plus élevé que ce que vous connaissiez en C *tout court*. Nous n'avons malheureusement qu'un aperçu des possibilités du langage, mais l'essentiel transparait déjà : dans la programmation orientée objet, l'essentiel est de bien décrire le problème dans des structures de données, et, si besoin, créer les structures sur mesure pour obtenir une bonne description. Le reste se règle uniquement en définissant les opérateurs utiles à ces structures.