Cours de C++ 2020/2021

Xavier Juvigny

2 mars 2021

Table des matières

Chapitre 1

Introduction

1.1 Qu'est ce que le C++

1.1.1 Un peu d'histoire

L'histoire du C++ est très lié à l'histoire du C. Le langage C lui-même est historiquement lié au système UNIX.

En 1969, la première version du système UNIX voit le jour dans les laboratoires Bell, dans le New Jersey, pour une machine appelée coquettement DEC PDP-7. Mais le langage utilisé pour écrire ce système était l'assembleur. Or l'assembleur est un langage trop proche du matériel informatique utilisé et qui permet peu de suivi et d'évolution des logiciels.

Peu après l'apparition d'UNIX, Ken Thompson créa un nouveau langage de programmation, nommé B, lui-même inspiré du langage BCPL (Basic Combined Programming Language, développé par Martin Richards). Le but de B était de créer un langage plus simple que le BCPL, lui-même plus simple déjà que le langage CPL. Mais ce but ne fut que trop bien atteint, et le B s'avéra être un peu trop simple et trop dépendant de l'architecture.

Peu après l'apparition du B, une nouvelle machine fut introduite : le PDP-11. UNIX et B ont tout de suite étés transférés vers cette nouvelle architecture, plus rapide que le PDP-7. Mais réécrire UNIX en assembleur (car les assembleurs ne sont pas les mêmes de machine à machine) était une tâche trop ardue, et on considéra de le réécrire en B. Le problème était alors que le B était relativement lent, du au fait qu'il s'agissait d'un langage interprété (comme le Basic et le Java aujourd'hui).

C'est pour cela qu'en 1971, on commença a travailler sur le successeur de B, nommé dans l'ordre des choses C. On s'accorde pour dire que Dennis Ritchie est le véritable créateur du C. La puissance du C était véritable : ce langage n'avait pas été fait spécifiquement pour un type de machine, ou pour une système particulier. C'est donc un langage hautement portable. De plus, son rayon d'action s'étend du bas niveau de la machine (le C peut générer du code aussi rapide que du code assembleur), au haut niveau des langages orientés problèmes. Virtuellement, on peut donc tout faire avec du C : du pilote de matériel jusqu'au logiciel de gestion, en passant par les jeux.

Le C devint vite tellement populaire que tout le monde voulait en faire sa version. Si bien que les différents compilateurs devinrent incompatible les uns avec les autres, et on avait perdu une grande part de la portabilité du C. L'idée vint alors qu'il fallait un standard pour le langage C. C'est un comité de l'ANSI (American National Standards Institute) qui mit au point ce standard en 1983, et depuis, la plupart des compilateurs (Borland, Microsoft, GNU CC...) se sont pliés à ce standard, si bien qu'un programme écrit en ANSI C est compilable presque partout.

Le C avait donc de nombreux avantages. Mais il lui manquait certaines caractérisrique des compilateurs modernes : la programmation orientée objet, fonctionnelle, …

Le Dr. Bjarne Stroustrup de Bell Laboratories est le créateur du C++, dont le développement date du début des années 1980. Son but était de créer un langage permettant la programmation orientée objet tout en restant hautement performant. Il fallait aussi toucher un large public. Or les deux dernières caractéristiques désignaient directement le C, déjà utilisé pour écrire des milliards de lignes de code dans tous les domaines. Il créa alors le C++ sur la base du C, en prenant soin de préserver la compatibilité : tout programme C peut être compilé en C++. La plupart des ajouts sont inspirés des langages Simula67 et Algol68.

Mais, de la même façon que le C, l'évolution du langage C++ commençait à diverger entre les éditeurs de compilateur. Dans le courant de l'année 1994 les comités de normalisation ISO et ANSI ont publié une version préliminaire officielle du C++ X3 JI 6/WG21. La dernière version préliminaire fut publiée en décembre 1996. A part quelques points de détail, elle comportait l'essentiel de ce qui allait être la version définitive, datée du 27 septembre 1998 notée norme ISO/IEC 14882 :1998(E). On se réfère à ce standard en parlant du C++ 98.

Il fallut ensuite de nombreuses années avant que le standard soit remis à jour. Une première mise à jour, essentiellement pour corriger quelques problèmes liés au standard 98 (en particuliers des omissions) a été effectué en 2003 (standard ISO/IEC 14882 :2003, C++ 03). Une vraie évolution du langage eu lieu avec la sortie d'un nouveau standard le 12 Août 2011 (standard ISO/IEC 14882 :2011, C++ 11) qui apporta un grand nombre de nouveautés. Depuis, le standard est mis à jour tous les trois ans, en rajoutant de nouvelles fonctionnalités permettant de repousser les limites du langage. Il existe donc actuellement un standard datant du 18 Août 2014 (C++ 14) qui apporte quelques nouveautés au C++ 11, une autre de Mars 2017 (C++ 17) qui corrige certains défauts du C++14 (en enlevant certaines fonctionnalités du langage) et en rajoute un grand nombre, et finalement la version actuelle publié par l'ISO (avec un peu de retard dû au COVID…) en Décembre 2020 qui rajoute un grand nombre de nouveautés au langage. La prochaine version du standard apparaîtra donc en 2023, et le travail sur cette future version a déjà commencé…

1.1.2 Caractéristiques du C++

Le C++ est donc un langage compilé dont la performance aujourd'hui est proche de celle du C, qui permet plusieurs paradigmes de programmation, à savoir :

- La programmation structurée : de la même manière qu'en C, on décompose le problème en des fonctions agissant sur des structures de données ;
- La programmation orientée objet : la programmation est centralisé sur les données à gérer et la façon dont on les manipule ;
- La programmation fonctionnelle : On décompose le problème en un ensemble de fonction (proche du sens mathématique) qu'on manipule ensuite pour créer le programme.

Lors d'un projet informatique, il est déconseillé de se fixer un seul paradigme de programmation. Selon les différentes parties du projet, un paradigme sera mieux adapté qu'un autre. Il faut donc savoir rester pragmatique et utiliser le bon paradigme de programmation selon le problème rencontré.

Il faut aussi savoir que le C++ est doté en standard d'une bibliothèque d'objets et de fonction très riche qui permet de gagner du temps lors de l'élaboration d'un projet. En particuliers, on trouvera dans cette bibliothèque :

- une partie utilitaire : des pointeurs intelligents, des fonctions de mesure du temps (calendrier ou chronomètre), des fonctions de hashage, une définition de pair, de tuple (comme en python), de conversion de chaînes de caractère, …

— une partie pour gérer des conteneurs : tableaux statiques (array), tableaux dynamiques (vector), liste simplement ou doublement chaînées, dictionnaires avec ou sans tables de hashage, ensemble assurant l'unicité des éléments, des queues (à gestion prioritaire ou non), …ainsi que des fonctions applicables dessus comme des tris (partiels ou complets), la recherche d'éléments (rapide ou non), etc.
— Une partie pour gérer les chaînes de caractère : transformer des réels ou des entiers en chaîne de caractère, gérer (de façon basique pour l'instant) des chaînes en UTF 8, UTF 16 ou UTF 32 (permettant d'avoir des caractères internationaux : arabes, chinois, japonais, indien, etc. ou encore des symboles mathématiques) ;
— Une partie pour gérer les entrées–sorties et les fichiers : créer des fichiers formatés ou binaires, gérer les répertoires (création, destruction, etc.) ou les liens entre fichiers, gestion des accès aux fichiers (les permissions en lecture, écriture, exécution), etc.
— Une partie mathématique : une bibliothèque permettant de gérer facilement les nombres complexes, une bibliothèque très riche pour les nombres aléatoires, permettant de choisir le générateur aléatoire mais aussi la loi de distribution que les nombres doivent suivre, des fonctions complexes comme les polynômes de laguerre, de legendre, d'Hermite, et., la fonction beta, les intégrales elliptiques, les fonctions de Bessel de premier et second type, la fonction zeta de Riemann, …
— Une bibliothèque permettant de manipuler des expressions régulières
— Une bibliothèque permettant de gérer les threads posix (indépendamment du système d'exploitation)
— Enfin, depuis C++ 17, la plupart des fonctions proposées par ces librairies peuvent être exécutées en parallèle sur les architectures multi-cœurs des ordinateurs modernes.

Nous verrons au fur et à mesure du cours quelques fonctionnalités de cette bibliothèque très riche. Ce qu'il faut cependant comprendre, c'est que le C++ est un langage extrémement riche proposant une très grande étendue de fonctions. Il est tout simplement impossible de maîtriser à 100% ce langage. D'ailleurs, le Dr. Bjarne Stroustrup (créateur du C++) a lui-même estimé qu'il ne connaissait qu'environ 70% du langage. Il serait donc illusoire de totalement maîtriser le langage à la fin de ce cours ! (par ailleurs, votre professeur ne maîtrise pas non plus 100% du C++ !)

1.2 Compiler des sources C++

Plusieurs compilateurs existent sous Linux, Mac ou Windows :

Sous Linux, le compilateur `g++` n'est pas installé par défaut (contrairement au `gcc`). Il faut donc l'installer en se reportant à la documentation d'installation qui dépend de la distribution utilisée. Il existe aussi un autre compilateur `clang` que je recommande chaudement pour diverses raisons, en particuliers son interaction incroyable avec l'éditeur sublime (que je recommande également, mais qui est un shareware parfaitement utilisable sans payer la license) permettant de formater ces sources selon des critères qu'on peut soit même définir ainsi qu'une compilation en temps réel du code en train d'être édité au moment de la sauvegarde pour vérifier les problèmes de compilation éventuels. Il existe également un compilateur C++ de PGI gratuit mais que je ne recommande pas sauf si on veut programmer "facilement" la carte graphique. Enfin, il existe le compilateur C++ d'Intel, payant (et cher !) qui est livré avec des bibliothèques optimisées pour le calcul scientifique.

Sous Mac, pour avoir accès à `g++`, il faut installer `Homebrew` (gratuit) qui de toute façon sera également nécessaire en troisième année pour le calcul parallèle. Vous pouvez alternativement installer `XCode` qui fournit `clang` comme compilateur C++ !

Sous Windows, outre `CodeBlock` que je ne recommande pas forcément, vous pouvez instal-

ler `MSys 2` qui est un environnement à la Unix permettant de compiler des sources en exécutables Windows comme si vous étiez sous Unix ! MSys 2 est basé sur une distribution Linux (ArchLinux) est propose les dernières versions de g++ ou de clang ainsi que d'autres outils familiés pour un utilisateur d'unix (Makefile, python, opengl, etc.) Du point de vue éditeur, `sublime text` est également proposé sous Windows et peut très bien interagir avec la version clang proposée par Msys 2 pour avoir les mêmes fonctionnalités que sous Linux. Enfin, une autre solution est de téléchargé Visual C++ dans sa version Community, qui a l'avantage d'être gratuit même si vous n'aurez pas accès à des options d'optimisation. Par contre, le cours proposé ici se base sur les options proposés par g++ (et clang qui possède les mêmes options que g++) et fera quelques explorations dans le monde de C++20, version du C++ qui n'est pas encore supporté par Visual C++ (mais bien supporté par g++ et clang).

Enfin, pour ceux qui ne possèdent qu'une tablette ou un téléphone portable, sachez qu'il existe des sites vous proposant de compiler des programmes C++ et de les exécuter. Bien entendu, cela peut s'avérer utile pour des exercices mais bien insuffisant pour mener un programme de la taille d'un projet ! Un lien intéressant proposant du C++ 17 est la page web suivante : `https://www.onlinegdb.com/online_c++_compiler` qui propose un compilateur mais aussi un outil de débogage de votre code (gdb en l'occurence).

1.2.1 Les options de compilation

Lors de la compilation de vos sources, vous aurez selon le contexte de production de votre code plusieurs options à fournir au compilateur.

Si vous êtes en train de développer, il est conseillé d'utiliser un maximum d'options permettant de détecter les bogues et les maladresses d'écriture en amont plutôt qu'à la fin de votre projet ! Il est donc conseillé de compiler votre source en exécutant la commande suivante :

`g++ -std=c++20 -g -Wall -pedantic -D_GLIBCXX_DEBUG -o <nom de l'exécuble> <nom du ou des fichiers sources>`

Si vous utilisez clang, il suffit de remplacer g++ par clang (et garder les mêmes options). Voici ce que font chaque option :

- `-std=c++20` : compile le code selon la norme ISO C++ 20
- `-g` : Permet de conserver les symboles des variables et fonctions pour pouvoir lancer un éventuel débogage et voir la pile d'appel
- `-Wall` : Actionne les messages de Warning pour toutes les options possibles (variables non intialisées, non utilisées, nommant une variable dans un sous bloc avec un nom de variable déjà existant, etc.)
- `-pedantic` : Rend le compilateur "pédant" (sic !) et lui demande de générer des messages de warning si on s'écarte même très légèrement de la norme ISO. Cela permet en particuliers de s'assurer que notre code sera portable avec d'autres compilateurs.
- `-D_GLIBCXX_DEBUG` : Permet de détecter la moindre erreur mémoire faite en utilisant la bibliothèque standard du C++, en particuliers les débordements d'indice en utilisant les tableaux statiques ou dynamiques.

Si vous êtes en train de compiler pour livrer un exécutable, on veut bien sûr avoir le code le plus compact et rapide possible. Il est conseillé dans ce cas de compiler vos sources en exécutant les options suivantes :

`g++ -std=c++20 -O3 -march=native -DNDEBUG -o <nom de l'exécuble> <nom du ou des fichiers sources>`

Voici ce que signifie chaque option :

- `-O3` : permet une optimisation maximale, quitte à modifier l'ordre des opérations et ne pas avoir un résultat identique au binaire près (si il est important que les opérations

soient dans un ordre identique à ce qui a été écrit, remplacer `-O3` par `-O2`)
— `-march=native` : Émet du code machine adapté au processeur se trouvant sur la machine où le code est compilé (et profiter ainsi des architectures modernes des processeurs pour un code plus rapide !)
— `-DNDEBUG` : Supprime les assertions (qu'on verra plus loin dans le cours) qui servent à déboguer le code.

Il faut avoir conscience qu'en mode dévelopement, votre code s'exécutera relativement lentement par rapport à l'exécutable créer avec toutes les optimisations données pour livrer un exécutable. Cependant, cela vous permettra déjà de supprimer un grand nombre d'erreurs de programmation sans trop vous casser la tête.

Enfin, il est conseillé de créer un makefile pour compiler vos codes, permettant :
— De recompiler à chaque fois des fichiers qui n'ont pas été modifiés entre deux compilations ;
— D'éviter de retaper à chaque fois la ligne de commande avec toutes les options nécessaires.

1.3 Des bonnes pratiques de programmation

Lorsqu'on programme, on passe un certain temps à écrire du code, mais on passe beaucoup plus de temps à **lire** du code que soi-même ou d'autres ont écrit. Il est donc important d'avoir un code clair et agréable à lire.

On peut faire l'analogie entre l'écriture d'un code et l'écriture d'un texte. Lorsque le texte est mal écrit et mal présenté, qu'il contient des fautes d'orthographe, que les phrases sont mal structurées et que les idées ne sont pas organisées, ce texte est très difficile à lire et donc à comprendre. Il en va de même pour le code : un code brouillon est très diffile et fatiguant à comprendre…de plus, les bogues s'y cachent beaucoup plus facilement.

Considérons le code (en C) suivant :

```
int k(int i)
{
int rsflkj = 1; if (i==1) return rsflkj; else rsflkj = i;
                          return rsflkj * k(i-1);
}
```

Arrivez-vous du premier coup d'œil à savoir ce que fait cette fonction ? C'est la fonction factorielle. Elle serait bien plus simple à comprendre avec un code bien écrit, non ?

Il est donc très important de respecter certaines bonnes pratiques de programmation qui s'appliquent naturellement à la rédaction d'algorithmes pour rendre le plus agréable possible la lecture de code.

Cette section n'a pas vocation à vous inventorier toutes les bonnes pratiques de programmation, ce n'est pas la vocation première de ce cours, mais vous donne déjà quelques éléments essentiels là-dessus (on reparlera également de bonnes pratiques dans divers chapitres de ce cours).

1.3.1 Contrat–Interface contre mise en œuvre d'un algorithme

Le **contrat** caractèrise l'**interface** d'un algorithme, c'est-à-dire qu'il explique le plus clairement possible ce que l'algorithme est capable de produire comme sorties étant donné ce qu'on lui fournit en entrée. On y spécifiera les conditions en entrée de l'algorithme (les pré-conditions)

qui vont décrire les valeurs que peuvent prendre les données fournies en entrées pour que l'algorithme soit capable de fonctionner, et les conditions en sortie de l'algorithme (post-conditions) qui nous renseignera sur ce que l'on peut attendre à obtenir comme résultats.

Par conséquent, lorsqu'on consulte le contrat d'un algorithme, on est renseigné sur ce à quoi on peut s'attendre, ainsi que sur les limites de l'algorithme que l'on va utiliser, sans pour autant avoir besoin de comprendre comment l'algorithme est mise en œuvre. Il nous appartient de composer avec ces informations.

Le contrat permet donc de savoir très exactement ce que l'algorithme est capable de faire, mais il ne dit rien sur **comment** l'algorithme va s'y prendre pour résoudre le problème.

1.3.2 Précondition et post-condition

Les préconditions et les postconditions sont des conditions faisant partie du contrat relatif à un algorithme.

Les préconditions sont des conditions vérifiant que les données connues à l'entrée de l'algorithme appartiennent bien au domaine de définition de l'algorithme et qu'elles sont cohérentes entre elles.

Les post-conditions sont des conditions vérifiant que les données en sortie appartiennent bien à l'image de l'algorithme ou du moins ont des valeurs cohérentes et attendues par rapport à ce que fait l'algorithme.

Ainsi par exemple, prenons un algorithme calculant la racine carrée d'un nombre réel :

- L'unique pré-condition de cette fonction est que le nombre réel soit positif ou nul ;
- L'unique pré-condition de cette fonction est que la racine carrée retournée soit positive ou nul ;

Si on traduit cela en C ou C++, on utilisera les assertions que l'on trouve dans la bibliothèque `assert.h` (en C) ou `cassert` en C++. Les assertions sont des fonctions demandant en entrée une condition. Si cette condition à l'exécution est vérifiée, le programme continue son exécution, sinon le programme s'arrête sur l'assertion et affiche la ligne et le fichier où l'assertion n'a pas été vérifiée. Les assertions ne sont activées que si l'option `-DNDEBUG` a été omise, sinon elles sont ignorées par le compilateur. Ainsi, pour notre racine carrée, nous traduirons la pré-condition et la post-condition par :

```
#include <assert.h> // assert.h accepté aussi en C++, mais on préfère #include <
    cassert>

double sqrt(double x)
{
  assert(x>=0); // Pré-condition

  sq = ...            // Calcul de la racine qu'on stocke dans sq

  assert(sq >= 0); // Post-condition

  return sq;
}
```

Listing 1.1 – Exemple d'utilisation d'assertion (C ou C++)

La simple lecture du code nous indique donc les valeurs valides pour cette fonction et les conditions sur les valeurs de retour.

Les préconditions et les postconditions peuven être des conditions demandant eux-mêmes un algorithme permettant de vérifier l'état ou les états de valeurs en entrées ou en sortie.

Ainsi, prenons un algorithme de tri sur des valeurs contenus dans un tableau dont on ne connaît pas a priori les types (en Python, cela arrive souvent, en C++ nous verrons que nous

aurons également souvent ce cas de figure avec les templates). Les pré-conditions et les post-conditions sont :

- Pré-conditions :
  - Que les valeurs dans le tableau puisse être comparer deux à deux à l'aide d'un opérateur de comparaison ;
  - Que cet opérateur de comparaison définisse une relation d'ordre (au sens mathématiques du terme) ;
- Post-conditions :
  - Que les valeurs du tableau en sortie soit une permutation des valeurs du tableau en entrée ;
  - Que les valeurs du tableau en sortie vérifie la relation d'ordre en parcourant linéairement le tableau ;

On voit que les préconditions et les post-conditions sont moins triviales ici que pour la racine carrée.

La première pré-condition en C++ sera de toute façon vérifiée automatiquement par le compilateur (en Python, on peut vérifier cette pré-condition pendant le tri).

La deuxième pré-condition est impossible à vérifier par le compilateur ou les assertions. C'est donc au programmeur ici de s'assurer que l'opérateur de comparaison employée définit bien une relation d'ordre.

Pour les post-conditions, la première post-condition est loin d'être triviale à mettre en œuvre : soit on utilise un check-sum qui nous assure partiellement que la post-condition est vérifiée, soit en C++ on peut utiliser une fonction déjà définie dans la bibliothèque standard (Dans la bibliothèque `algorithm`, il existe une fonction `is_permutation` qui vérifie qu'un tableau de valeurs est bien la permutation d'un autre tableau de valeurs).

Quant à la seconde post-condition, il suffit de parcourir linéairement tout le tableau en sortie pour vérifier que les éléments consécutifs vérifient bien la relation d'ordre.

Remarquons que dans cet exemple, les pré et post-conditions sont loin d'être triviales et ont un coup en temps de calcul qui peut être non négligeable. Cependant, en C et C++, rappelons que ce coût ne sera effectif que lorsqu'on sera en phase de développement (`-DDEBUG`) mais ne sera plus présent lors de la production finale du code (où on utilisera l'option `-DNDEBUG`).

Même si le code produit durant la phase de développement sera plus lent, une bonne utilisation des pré et post conditions peut réduire drastiquement le temps de développement en permettant de détecter assez tôt des bogues potentiels. Il ne faut pas oublier que la relecture du code et la phase de "débogage" sont des parties du développement qui prennent la plus grosse partie du temps de mise en œuvre.

1.3.3 Qu'est ce qui caractérise un code "bien écrit"

Un code mettant en œuvre un algorithme est bien écrit si il a les propriétés suivantes :

- Être facile à lire, par soi-même mais aussi par les autres ;
- Avoir une organisation logique et évidente ;
- Être explicite, montrer clairement les intentions du développeur ;
- Être soigné et robuste au temps qui passe.

Regardons en détail chacune de ces caractéristiques :

Le code doit être facile à lire

Pour que le code soit facile à lire, il faut d'une part qu'il soit bien structuré et bien présenté, et d'autre part, que les noms des variables et des fonctions soient choisis avec soin.

Pour ce qui est de la structure et de la présentation, il est important, particulièrement en C et C++, de respecter une règle fixe d'indentation en veillant à ce que des blocs d'instructions se

trouvant au même niveau doivent être précédés du même nombre d'espaces, ce qui nous conduit naturellement à bien **indenter** notre code.

Par exemple, si on regarde le code C ci–dessous, il est évident que le manque d'indentation ne facilite par la lecture et la compréhension du code, n'est-ce-pas ?

```
void m(int n, float* A, float* B, float* C) {
int i,j,k;
for (i = 0; i < n; ++i ){
float a = 0.;
for (j = 0; j < n; ++j ){
for (k = 0; k < n; ++k ){
a += A[i+k*n]*B[k+j*n];
}
C[i+j*n] += a;
}
```

Listing 1.2 – Code sans indentation

Pour ce qui est du choix des noms des choses (variables, fonctions, structures, etc.), nous en reparlerons un peu plus loin.

Le code doit avoir une organisation logique et évidente

Ce point est plus délicat car nous avons souvent des solutions différentes pour résoudre le même problème. Il est donc normal qu'un code qui semble logique à quelqu'un, semble "tordu" à son voisin.

Étant conscient de cela, il faut vous efforcer de trouver des solutions logiques et simples aux problèmes que vous devez résoudre et d'éviter d'emprunter des chemins plus compliqués qui ne feraient que semer la confusion.

Par exemple, pour afficher les nombres de un à dix, il suffit de faire une boucle qui fait évoluer un compteur entre un et dix et qui affiche ce compteur à chaque itération. La solution qui consisterait à faire une boucle qui fait évoluer un compteur entre neuf et zéro et qui afficher à chaque itération la valeur $10-\texttt{compteur}$ fonctionne aussi mai est à proscrire car elle est moins "simple".

De même, lors de la déclaration d'une fonction, éviter si possible d'avoir des paramètres redondants ou qui peuvent se déduire d'autres paramètres dans la liste de vos paramètres, cela vous évitera en particuliers d'avoir des incohérences entre les divers paramètres de votre fonction.

Par exemple, pour orthogonaliser un vecteur par rapport à un autre vecteur :

```
void orthonogalise(double u[3], double nrmu, double v[3])
{
    double dotuv = u[0]*v[0]+u[1]*v[1]+u[2]*v[2];
    v[0] = v[0] - dotuv*u[0]/(nrmu*nrmu);
    v[1] = v[1] - dotuv*u[0]/(nrmu*nrmu);
    v[2] = v[2] - dotuv*u[0]/(nrmu*nrmu);
}
```

Listing 1.3 – Exemple de paramètre superflu

Le paramètre donnant la norme de u est redondante avec les cœfficients de u et il y a un risque que la valeurs passée ne soit pas celle de u, ce qui amènera l'algorithme a faire un calcul faux !

Quitte à faire un peu plus de calcul, il est donc préférable de ne passer que les deux vecteurs u et v !

```
void orthonogalise(double u[3], double v[3])
{
```

```
    double sqr_nrm_u = u[0]*u[0]+u[1]*u[1]+u[2]*u[2]; // Calcul ||u||²

    assert(sqr_nrm_u > 1.E-14); // Pre-condition verifiant que le vecteur u n'est
    pas nul

    double dotuv = u[0]*v[0]+u[1]*v[1]+u[2]*v[2];
    v[0] = v[0] - dotuv*u[0]/sqr_nrm_u;
    v[1] = v[1] - dotuv*u[0]/sqr_nrm_u;
    v[2] = v[2] - dotuv*u[0]/sqr_nrm_u;

    assert(std::abs(v[0]*u[0]+v[1]*u[1]+v[2]*u[2]) < 1.E-14);
}
```

Listing 1.4 – Exemple sans paramètre superflu

Le code doit être explicite

Lorsqu'on écrit des algorithmes ou que l'on développe des programmes, on est parfois tenté de prendre des raccourcis car "on sait" que telle ou telle méthode permet de faire telle ou telle chose bien pratique.

Il n'est pas interdit de prendre des raccourcis, mais il faut toujours prendre le soin de l'expliquer, au moins au travers des commentaires, pourquoi on fait cela. C'est important à la fois pour permettre aux autres de comprendre pourquoi votre solution est astucieuse…mais aussi pour vous, au cas où vous ne vous souveniez plus de "pourquoi vous avez fait ça".

Par exemple, si vous devez afficher une matrice de dimension $M \times M$, la procédure usuelle est de faire deux boucles imbriquées permettant d'afficher chacun des éléments de la matrice. Or, si vous savez que votre matrice est triangulaire, vous allez probablement vouloir optimiser votre double boucle d'affichage. C'est naturellement une bonne idée…mais pensez bien à rappeler dans le commentaire pourquoi vous procédez de la sorte.

Le code doit être soigné et robuste au temps qui passe

Lorsqu'on écrit un code, on a la fâcheuse tendance à s'arrêter dès que celui-ci fonctionne. C'est un tort ! Le code doit être entretenu. Cela signifie qu'il faut relire son code après l'avoir terminé, vérifié que l'on a bien supprimé les éléments obsolètes, vérifier que les commentaires sont à jour et cohérents avec le code conservé, etc.

Cette opération de "maintenance" du code est cruciale, mais elle est pourtant souvent négligée par beaucoup, ce qui peut poser des problèmes, notamment lorsque vous rencontrez un bogue.

L'exemple le plus classique est le suivant : Vous mettez un œuvre une fonction `tri` pour trier des éléments d'un tableau. Vous n'êtes pas satisfaits du comportement de cette méthode lorsque vous l'utilisez depuis votre programme principal. Vous mettez en œuvre une nouvelle fonction `tri_rapide` qui utiliser une autre stratégie pour trier les éléments du tableau. Cette méthode marche mieux. Vous l'utilisez donc dans votre programme principal. Votre programme fonctionne et vous passez à autre chose, sans penser à intégrer proprement vos modifications dans votre programme. Quelques jours plus tard, vous reprenez votre code et vous observez un bogue. Vous pensez que cela vient du tri effectué, et vous allez donc observer ce qui se passe dans `tri`. Après quelques heures de recherche, vous êtes furieux contre vous-même car vous réalisez que la méthode `tri` n'est plus utilisée dans votre code…

Croyez-le ou non, mais les problèmes de ce type arrivent beaucoup plus souvent qu'on ne le pense…surtout quand on cherche à faire vite.

Avant de passer au point suivant, un autre exemple de code illustrant ce qui se passe très souvent. Voyez-vous le problème :

```
void une_fonction(bool continuer)
{
```

```
  // La boucle s'arrête si i est négatif ou si continuer prend la valeur false
  int i = 0, j = 4;
  while (continuer)
  {
    std::cout << "Mon code marche" << std::endl;
    // i += 1;
    j += 1;
    if (j>10) continuer = true;
  }
}
```

Comme vous pouvez le voir, le premier commentaire ne correspond plus du tout à ce que fait l'algorithme et obsurcit plutôt la compréhension du code plutôt qu'il l'éclaire ! Ce premier commentaire devait sûrement être pertinent dans les premières versions du code mais au fur et à mesure de l'évolution de ce dernier, ne correspond plus du tout à ce que fait l'algorithme !

1.3.4 C'est bien joli tout ça, mais coder proprement ça prend du temps !

Faux ! Il ne faut pas confondre vitesse et précipitation.

On a souvent tendance à penser que l'on perd énormément de temps à soigner son code, à le structurer correctement, à le reorganiser et à le documenter, mais c'est faux. Au contraire, on gagne du temps à faire tout cela.

Voici quelquels arguments pour en convaincre :

— Si vous adoptez les bonnes pratiques dès le début, vous faites déjà 50% du travail ;

— Si le code est bien écrit, il est plus facile, et donc plus rapide à relire, et n'oubliez pas que vous passez plus de temps à lire votre code qu'à l'écrire...donc quand votre code est propre, vous vous faites gagner du temps ;

— Si le code est logique et bien structuré, il sera plus facile de retrouver les bogues qu'il contient, et donc de l'améliorer...

Ce sont donc autant de raisons qui devraient vous convaincre qu'il est important d'être organisé, clair, méthodique et rigoureux quand vous développez.

1.3.5 De l'importance des commentaires

Les commentaires sont essentiels pour "éclairer" votre code. Un commentaire est un texte qui est ignoré par le compilateur lorsqu'il crée l'exécutable, mais qui peut être lu par le développeur lorsqu'il lit le programme.

Bien que les commentaires soient essentiels, il ne faut pas en abuser.

Un bon commentaire peut :

— Faciliter la lecture du code ;

— Apporter une indication sur un choix de conception ;

— Expliquer une motivation qui ne serait pas évidente ;

— Donner un exemple pour permettre de mieux comprendre ce que fait le code.

Quelques exemples de mauvais commentaires :

— Un commentaire qui décrit un morceau de code qui n'existe plus ;

— Un commentaire qui explique une évidence ;

— Un commentaire sur plusieurs lignes pour expliquer une chose simple ;

— Un commentaire sur l'historique des modifications d'un fichier. C'est parfois utile, mais dans la plupart des cas, il vaut mieux confier cette tâche à votre gestionnaire de versions qui fera le travail pour vous.

Amusez-vous à trouver pourquoi tous les commentaires ci–dessous sont des mauvais commentaires :

```
i = 0; // On initialise la variable i à zéro
i = i + 1; // On incrémente de un la variable i
// On additionne a et b et on stocke le résultat dans c
c = a + b;
// Ci—dessous, on fait une double boucle pour afficher la matrice :
for (i = 0; i < 10; ++i )
  std::cout << "valeur : " << i << " ";
// Fin du for
std::cout << std::endl; // Retour à la ligne
/*
  Et maintenant, on va s'occuper de retourner la valeur de i. On utilise pour cela
  l'instruction return  à laquelle on passe la valeur de i
  */
return i;
```

Remarque : Si vous vous sentez obligé de commenter la fin d'un bloc d'instructions, c'est que probablement votre bloc d'instruction est trop long et que vous pouvez certainement fragmenter votre code en éléments plus simples.

Attention : Les commentaires ne doivent pas palier un manque de clarté de votre code. Si vous avez besoin de commentaires pour cela, c'est probablement que vous pouvez améliorer votre code pour le rendre plus lisible. Essayez donc de le reécrire, au moins partiellement, en l'améliorant.

Comme nous l'avons déjà évoqué, les commentaires, comme le code, doivent être maintenus, c'est à dire qu'ils doivent évoluer avec le code, et disparaître si le code correspondant disparaît. Par conséquent, il faut veiller au bon dosage de vos commentaires, de sorte à ne pas alourdir inutilement votre travail de maintenance. Et puis, pour citer Guido Von Rossum, le créateur du langage Python, "Les interfaces, c'est comme les blagues, si on doit les expliquer, c'est qu'elles sont ratées !". Donc si on doit commenter l'utilisation d'une fonction avec un grand nombre de commentaire, c'est que la fonction fait soit trop de chose, soit est inutilement compliquée !

Enfin, une bonne pratique est d'utilisé des outils tels que `doctest` qui vous permettent d'écrire des petits tests pour vos fonctions tout en les documentant. Les avantages de cette pratique sont d'une part que cela vous "force" à tester votre code, et qu'il permet en même temps d'en avoir une documentation "par l'exemple" qui sera forcément cohérente avec le code et à jour (sinon les tests ne fonctionneraient pas).

1.3.6 Comment nommer les choses ?

Les noms que vous choisissez pour vos variables vont grandement contribuer à la lisibilité de votre code.

Par exemple, vous conviendrez que le code suivant est peu clair :

```
gfdjkgldfj = 4;
ezgiofdgfdljkrljl = 1;
gfdjkgldfj = ezgiofdgfdljkrljl + gfdjkgldfj;
```

tandis que le code suivant

```
x = 4;
x += 1;
```

est bien plus clair et fait exactement la même chose !

La première règle est donc de choisir des noms de variables prononçables et faciles à retenir.

Vous devez également choisir des noms de variables explicites pour vous mais aussi pour les autres.

Par exemple, `a` est bien moins explicite que `adresseClient`. De même, `lf` est moins explicite que `largeurFenetre`. Pensez également que vous serez probablement amenés à chercher vos variables dans votre code avec un outil de recherche. À votre avis, combien d'occurences de `a` allez vous trouver ? Et combien d'occurences de `adresseClients` ?

Évitez également de choisir des noms de variables qui induisent un contre-sens. Par exemple, si vous écrivez `matrice = 8`, on pourrait penser que la variable est une matrice, or il s'agit clairement d'un entier. Au moment de l'affectation, il est facile de se rendre compte du type de la variable, mais maintenant, imaginez que vous rencontriez au beau mileu de votre code la ligne suivant :

```
matrice = 4 * matrice;
```

Comment allez vous interpréter cette instruction ?

Évitez également les noms de variables qui n'ont pas de sens, comme `plop`, surtout si vous en utilisez plusieurs dans la même portion de code. Vous risquez de vous perdre dans les noms, et donc d'introduire inutilement des bogues dans votre code.

Ne trichez pas non plus quand vous choisissez des noms de variables. Par exemple, en C, on ne peut pas nommer une variable `case` ou `volatile` car ce sont des mots réservés du langage. Si vous pensez avoir besoin de ces mots pour nommer vos variables, ne trichez pas en écrivant `ccase` ou `vvolatile`, vous risqueriez de vous perdre dans vos propres astuces.

Enfin, essayez d'être cohérents losque vous choisissez vos noms de variables. Par exemple, si vous décidez d'utiliser le français pour nommer vos variables, utilisez le français tout du long, n'alternez pas avec l'anglais. `longueurPath` et `lengthChemin` sont des noms pour le moins curieux !

1.3.7 En conclusion

Pour revenir sur la première fonction (la fonction factorielle) présentée au début de cette section, on pourrait l'écrire proprement de la manière suivante :

```
long fact(long n)
{
  assert(n>=0); // Pré-condition

  if (n == 0) // Cas particuliers : 0! = 1
    return 1;

  long resultat = n * fact(n-1); // n! = n * (n-1)!

  assert(resultat > 0);          // Post-condition
  return resultat;
}
```

La fonction possède un nom évoquant la fonction factorielle. La pré-condition indique que n doit être positif pour que la fonction soit valide. La post-condition indique que le résultat de la factorielle est une valeurs entière supérieure ou égale à un ! Avec peu de commentaires, des assertions et un choix adéquat du nom des choses, le code devient bien plus clair et facile à lire.

Chapitre 2

Initiation au C++

Nous allons attaquer dans ce chapître la programmation en C++. Pour ceux rompu aux arcanes du C, beaucoup de passages de ce chapître seront des rappels du C, mais ce chapître contient également son lot de nouveautés par rapport aux langage C. Aussi n'est-il pas conseillé de sauter ce chapître même si le C n'a plus de secret pour vous. Passez simplement les passages évoquant des spécificités communs au C et C++.

De même, ce chapître ne changera pas fondamentalement votre paradigme de programmation, vous pouvez simple le voir comme un chapître présentant un langage C "amélioré".

Enfin, ce chapître, ainsi que le reste du cours, n'a pas vocation à vous présenter le C++ de A à Z. Ce serait très ambitieux de la part de l'auteur, voire impossible à réaliser. Le créateur du C++, le Dr. Bjarn Stroustrup déclare lui-même ne connaître que 70% du langage C++ ! Un débutant connaît environ 10% du langage et un expert 60% du langage.

Il ne faut donc pas hésiter à se documenter, en particuliers sur internet. Pour avoir une documentation de référence (pas pour apprendre), on peut se référer aux deux sites suivants (en anglais) :

— `https://en.cppreference.com/w/`
— `http://www.cplusplus.com/`

2.1 Les blocs d'instruction

Comme son nom le laisse entendre, le bloc d'instruction est un "bloc" contenant une ou plusieurs instructions. En C et C++, le bloc commence par une accolade ouvrante et se termine par une accolade fermante. L'exemple ci–dessous vous montre un bloc d'instructions demandant votre nom et l'affichant sur la console.

```
{
  printf("Veuillez rentrer votre nom s'il vous plait : ");
  char buffer[256];
  scanf("%255s", buffer);
  printf("Bonjour %s\n", buffer);
}
```

Listing 2.1 – Exemple de bloc d'instructions

Un bloc d'instruction peut lui-même contenir un bloc d'instruction. On parle alors de sous-blocs d'instruction.

2.2 Et pour commencer : un petit programme "Hello world"

Nous allons commencer par disséquer un petit programme affichant "Hello world" sur une console. Voici le programme :

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello World !" << std::endl;
6     return EXIT_SUCCESS;
7 }
```

Listing 2.2 – Un programme Hello World !

1ère ligne : `#include` est une instruction demandant au compilateur de "charger" des instructions ou des nouveaux types (du moins leurs déclarations) permettant d'étendre les possibilités du langage. Ici, nous demandons au compilateur de charger une "bibliothèque" `iostream` (**i**nput **o**utput **stream**) qui permet d'afficher sur l'écran divers messages. Il faut en effet savoir que les instructions de base du C++ (comme pour le C) ne permettent pas d'afficher un quelconque message à l'écran. Il faut pour cela utiliser une "bibliothèque" permettant d'étendre les possibilités du langage de base. Les symboles `<` et `>` autour de `iostream` signifie que cette bibliothèque est une bibliothèque externe, c'est à dire une bibliothèque qui n'est pas définie dans le même espace de travail que le programme. Ici, la bibliothèque `iostream` est une bibliothèque standard du C++ faisant partie du standard ISO (et qu'on retrouve donc pour tous les compilateurs compatibles avec la norme internationale) ;

3e ligne : Elle définit une "fonction spéciale" `int main()` (on verra plus loin ce qu'est une fonction en C++), qui sert de point d'entrée pour l'exécution du programme, c'est à dire qu'à l'éxécution du programme compilé, l'ordinateur commencera à exécuter les instructions se trouvant dans le bloc d'instruction de la fonction `main` (ce qui se trouve dans les accolades juste en dessous de `int main()`. Un programme C++ contiendra toujours une et une seule fonction `main` qui servira d'unique point d'entrée lors de l'exécution du programme compilé. Cette fonction renvoie toujours en entier (`int`) au sysstème pour lui signaler si l'exécution s'est bien déroulée ou non.

4e ligne : On reconnaît ensuite le début d'un bloc d'instruction, qui correspond ici au bloc d'instruction que doit exécuter l'ordinateur lorsqu'il rentre dans la fonction `main` à l'exécution. La fin du bloc d'instruction se trouve à la ligne sept.

5e ligne : On demande à l'ordinateur d'afficher "Hello World !" sur une console (celle à partir de laquelle on a lancé le programme). `std::cout` déclare qu'on veut afficher un message sur une console (`cout` est une contraction de **c**onsole **out**put), le symbole `<<` indique qu'on veut rediriger un flux de messages vers cette console pour les afficher (les messages se trouvant à droite du symbole `<<`). Le premier message à rediriger vers la console est une chaîne de caractères contenant le message "Hello World !", le message étant entouré par des guillemets, puis afin de rediriger un second message sur la console, on reutilise le symbole `<<` qui s'enchaîne avec le premier `<<`. Enfin on définit le second (et dernier) message comme étant, `std::endl`, qui est une instruction d'affichage demandant d'afficher tous les messages redirigés auparavant et de passer à la ligne suivante sur la console (`endl` est une contraction de **end l**ine). Enfin, le symbole `;` signifie qu'on a terminer de définir une instruction (Il faut mettre un `;` à la fin de chaque instruction en C++).

6e ligne : On demande au programme de renvoyer une valeur `EXIT_SUCCESS` définie dans

la bibliothèque `cstdlib` elle-même utilisée par `iostream`, et qui signale au système que l'exécution du programme s'est bien passée (en cas d'erreur ou de problème rencontré dans le programme, on aurait pu retourner `EXIT_FAILURE`).

2.3 Headers et fichiers sources

Un projet C++ contiendra principalement deux types de fichiers : les headers et les fichiers sources.

Par convention, les headers propre au projet auront pour extension `.hpp`. Par exemple

```
polynomes.hpp
rationnel.hpp
matrice.hpp
```

Les headers contiendront la définition de nouveaux types et la déclaration de fonctions, c'est à dire l'inventaire des "instructions" que l'on rajoute au langage avec les paramètres attendus et quel type de valeurs est retourné.

On y reviendra sur le chapître des fonctions.

Les fichiers sources auront pour convention l'extension `.cpp`. Par exemple :

```
polynomes.cpp
rationnel.cpp
matrice.cpp
```

Les fichiers sources contiennent le code décrivant les algorithmes permettant de répondre à ce que sont censés faire les fonctions déclarées dans le header correspondant (sauf pour celui contenant la fonction main qui n'a pas de header correspondant).

Par exemple, supposons que nous voulons construire une bibliothèque qui permet de calculer le carré d'un entier. On crée d'abord le header `carre.hpp` qui nous permettra de spécifier une interface, c'est à dire quel sera le nom de la fonction, les types des valeurs attendues et le type de valeur retournée :

```
#ifndef _CARRE_HPP_
#define _CARRE_HPP_

int carre(int n);

#endif
```

Les deux premières lignes ainsi que la dernière sont utiles pour s'assurer qu'un fichier source, de façon directe ou indirecte, inclut plusieurs fois la déclaration de la fonction `carre`. Outre un gain non négligeable en temps de compilation, il se trouve également que le C++ n'aime pas qu'on déclare plusieurs fois la même fonction. Certains programmes, au lieu de l'utilisation du pré-processeur, utilisent la directive `#pragma once` pour s'assurer de n'inclure qu'une seule fois le fichier d'entête, mais cette directive, bien que fort répandue, n'est pas dans le standard et donc non portable.

Ensuite vient la définition de la fonction, qui comprend : le type de valeur qu'elle retourne, son nom et le type de paramètre qu'elle attend. On n'écrit ici aucun algorithme décrivant comment nous allons calculer le carré de n ! On termine la déclaration par un point virgule.

On créée ensuite le fichier source correspondant, `carre.cpp` qui décrira la fonction avec un algorithme permettant de calculer le carré de n. Bien que cela ne soit pas obligatoire, il

est conseillé d'inclure `carre.h` pour s'assurer qu'on conserve bien la même interface que celle déclarée dans le header !

```
#include "carre.hpp"

int carre(int n)
{
  return n*n;
}
```

Le bloc d'instruction juste en dessous de la fonction (sans point virgule à la fin de la ligne !), décrit en langage C++ l'algorithme qui calcule le carré de n et le retourne en sortie de la fonction.

Pour appeler cette fonction, il suffit d'écrire une ligne du type

```
int res = carre(5);
```

Listing 2.3 – Exemple d'utilisation de la fonction

qui appelle la fonction avec le paramètre cinq et renvoie la valeur vingt-cinq qu'on rangera dans la variable res.

En C++, on a aussi parfois besoin d'autres types de fichiers, qui essentiellement contiennent des lignes devant se trouver dans un header mais qui pour une raison ou une autre peuvent être mises dans un autre fichier. Pour ces fichiers, soit on garde le protocole de rajouter le suffixe `.hpp` soit on choisit une autre extension, mais dans ce cas, rien de normalisé existe.

2.4 Les variables

Les variables sont des noms (symboles) auxquelles on associe des valeurs. Ces valeurs peuvent changer au cours du temps mais ce n'est pas obligatoire. Par exemple, les variables dans les langages fonctionnels ne peuvent jamais changer de valeurs.

2.4.1 Convention sur le nom des variables

Le nommage des variables (mais également des structures, des classes, des fonctions qu'on verra plus loin) est assez souple mais doit obéir tout de même à certaines règles.

Une variable ne doit pas contenir :

— Un espace ou une tabulation
— Une ponctuation (`, ; . : ! ? # \`) ;
— Des simples ou double quotes (`'` ou `"`) ;
— Des symboles servant à définir des opérateurs (`+ - ^ / * | &`) ;
— Des parenthèses, bracket ou accolades `( [ { } ] )` ;
— Les symboles `@` et ©.

Une variable ne doit pas également commencer par un chiffre.

Par contre, un caractère peut contenir toute lettre de l'alphabet (accents compris), ainsi qu'une grande partie des caractères unicodes (alphabet grecque, japonais, indou, etc.).

Ainsi, les noms suivants sont valides : `a, _a,` π`, ζ, clé, périmètre`,…

Les noms suivants **ne sont pas valides** : `3a, {z, !b, la clef`

Utiliser des lettres autres que les lettres anglosaxonnes ne posent aucun soucis de portabilité, car cela fait partie de la norme C++ depuis 2011, tant que l'éditeur que vous utilisez supporte l'édition en unicode, ce qui est le cas de la grande majorité des éditeurs de texte aujourd'hui.

Les compilateurs Intel, Gnu, Clang et Microsoft supportent les variables nommées avec des caractères unicodes (pour gcc, ce n'est que depuis la version 10 du compilateur !).

Ce qui dit exactement la norme C++ 11 : Un identifieur pour une variable est une séquence arbitrairement longue de lettres et de chiffres. Chaque caractère universel dans un identifieur devra être une lettre qui respecte la norme ISO 10646 et être un caractère comme spécifié dans E.1. L'élément inital ne devra pas être un caractère universel qui appartient aux ensembles désignés par E.2. Les lettres minuscules et majuscules sont différentes. Tout les caractères sont spécifiques.

E.1 liste les caractères permis dont font parti les lettres grecques, les lettres accentuées mais aussi des caractères comme ª ¯ ² ³ ´ µ · ¹ º ¼ ½ ¾ Ø ° Ö
E.2 liste les caractères qui ne sont pas permis en première lettre dans l'identifieur d'une variable, à savoir les chiffres, et ce qui peut ressembler à des quotes.

Pour plus de précision, voir le draft ISO C++ 11 (`https://isocpp.org/files/papers/n4296.pdf` à la page 1265).

2.4.2 Déclaration

Une variable est définie par son **type** (un réel, un entier, une chaîne de caractère, un tableau, etc.) suivi du nom de la variable. Les variables peuvent être déclarées à n'importe quel endroit du code. Une variable n'existe qu'au sein du bloc d'instruction dans lequel elle est déclarée. Elle ne sera en outre visible que dans ce bloc d'instruction ou dans les sous-blocs d'instruction contenus dans le bloc d'instruction.

Il faut savoir que chaque variable déclarée dans un bloc d'instruction est stockée dans une zone mémoire particulière du programme nommée *pile d'exécution*. L'état de cette pile évolue au cours du temps en fonction des variables que l'on a défini.

Ainsi par exemple (`int` déclarant le type de la variable comme un entier) :

```
1  {
2    int a;
3    {
4      int b = 3;
5      a = 2;
6      int c = 4;
7      b = 5;
8    }
9    a= 8;
10 }
```

Listing 2.4 – Exemple de pile d'exécution

aura pour pile d'exécution :

Numéro de ligne	1	2	3	4	5	6	7	8	9	10
Pile										
						c=4	c=4			
				b=3	b=3	b=3	b=5			
		a	a	a	a=2	a=2	a=2	a=2	a=8	

On voit que les variables s'entassent sur la pile et disparaissent dès qu'on quitte le bloc d'instruction où elles ont été déclarées.

Si une variable est déclarée en dehors de tout bloc d'instruction, c'est une variable *globale* qui aura une durée de vie aussi longue que l'exécution du code lui-même. Elle sera alors visible par tous les blocs d'instructions. Il est fort peu recommandé d'utiliser les variables globales car elles peuvent être génératrices de nombreux bogues, en particulier dans un contexte multi-threads (qu'on verra dans le cours de l'année prochaine).

2.4.3 Types scalaires

Les types scalaires sont des types de variables représentant des nombres entiers ou réels, voire complexes.

Le type booléen

Le type booléen est un type de variable ne pouvant prendre que deux valeurs : vrai ou faux. On déclare une variable booléenne en C++ à l'aide du mot clef `bool` et les valeurs vrai ou faux correspondent respectivement aux mots clefs `true` et `false`.

```
bool flag = true; // La variable flag est mise à vraie
flag = false;
```

Il est possible de faire des opérations logiques sur les booléens, à savoir :

— OU logique (symbole `||`) : `f1 || f2` Cet opération renvoie vrai si au moins une des deux valeurs est vraie (voir la table de vérité 2.1)

Valeur de f1	Valeur de f2	Valeur de `f1 || f2`
false	false	false
true	false	true
false	true	true
true	true	true

TABLE 2.1 – Table de vérité de l'opérateur OU logique

— ET logique (symbole `&&`) : `f1 && f2` Cet opération renvoie vrai si et seulement si les deux valeurs sont vraies (voir la tableau de vérité 2.2)

Valeur de f1	Valeur de f2	Valeur de `f1 && f2`
false	false	false
true	false	false
false	true	false
true	true	true

TABLE 2.2 – Table de vérité de l'opérateur ET logique

— OU EXCLUSIF (symbole `^`) : `f1 ^ f2` Cet opération renvoie vrai si une seule des deux valeurs est vraie (et l'autre fausse) (voir la table de vérité 2.3)

Valeur de f1	Valeur de f2	Valeur de `f1 ^ f2`
false	false	false
true	false	true
false	true	true
true	true	false

TABLE 2.3 – Table de vérité de l'opérateur OU exclusif

— NON logique (symbole `!`) : `!f1` Cet opération renvoie faux si la valeur est vraie et vraie si la valeur est fausse (voir la table de vérité 2.4)

Valeur de f	Valeur de `!f`
`false`	`true`
`true`	`false`

TABLE 2.4 – Table de vérité de l'opérateur NON logique

Par ailleurs, les opérateurs de comparaison, c'est à dire
— est égale à (symbole `==`, attention aux deux symboles `=`)
— est différent de symbole `!=`)
— est inférieur à (symbole `<`)
— est supérieur à (symbole `>`)
— est inférieur ou égal à (symbole `<=`)
— est supérieur ou égal à (symbole `>=`)
renvoie des valeurs booléennes. Ainsi, les instructions suivantes sont valides :

```
bool est_plus_grand = 3 > 4; // est_plus_grand vaut false
bool est_plus_petit = 2 < 5; // est_plus_petit vaut true
bool flag = (1<5) && (5>2);  // flag vaut true
bool is_equal = (4==-4+7);   // is_equal vaut false
```

Si on veut afficher un booléen, on va, de la même manière que pour le programme "Hello World !" vu dans la section précédente, utiliser un flux (de message) :

```
std::cout << "est_plus_grand <-- " << est_plus_grand << std::endl
          << "est_plus_petit <-- " << est_plus_petit << std::endl;
```

Listing 2.5 – Exemple d'affichage basique d'un booléen

A l'éxécution, cette ligne d'instruction affichera

```
est_plus_grand <-- 0
est_plus_petit <-- 1
```

ce qui ne semble pas correspondre à la valeur de deux booléens mais à deux entiers. En fait, c'est bien la valeur des deux booléens…

Par convention, dans beaucoup de langages, dont le C++, la valeur entière zéro correspond à la valeur `false` et la valeur `1` (ou d'autres valeurs entières non nulles) correspond à la valeur `true`.

Les deux lignes suivantes sont parfaitement valides (mais non conseillé pour des raisons de lisibilité du code) :

```
bool fi = 0; // fi vaut |false|
bool fj = 1; // fj vaut |true|
```

Il est possible d'afficher sur l'écran une valeur "plus naturelle" pour les booléens. Il faut pour cela utiliser la bibliothèque standard `iomanip` qui contient entre autre une fonctionnalité permettant d'afficher `true` au `false` à l'écran.

Ainsi, le programme suivant affichera bien des `true` et des `false` à l'écran :

```
#include <iostream>
#include <iomanip>
```

```cpp
int main()
{
  bool f1 = 0; // f1 est faux
  bool f2 = 1; // f2 est vrai
  std::cout << std::boolalpha << "f1 : " << f1 << std::endl;
  std::cout << "et f2 : " << f2 << std::endl;
  std::cout << std::noboolalpha << "f1 && f2 : " << f1 && f2 << std::endl;
  std::cout << "f1 || f2 : " << f1 || f2 << std::endl;
  return EXIT_SUCCESS;
}
```

Listing 2.6 – Exemple d'utilisation de std ::boolalpha

L'instruction `std::boolalpha` dans le flux de message demande que les messages qui le suivent dans le flux d'affichage devront afficher `true` ou `false` si ce sont des booléens.

L'instruction `std::noboolalpha` dans le flux de message demande lui, au contraire, que les messages qui le suivent dans le flux d'affichage n'affichent plus les booléens que comme une valeur entière

Ainsi, l'exécution de ce code affichera sur la console :

```
f1 : false
et f2 : true
f1 && f2 : 0
f1 || f2 : 1
```

Les entiers

Il est possible en C++ (et en C) de définir des variables de type entier (pouvant contenir des valeurs entières). Cependant, ces entiers ne peuvent contenir qu'une certaine étendue de valeurs, du fait que leur représentation dans la mémoire de l'ordinateur ne prend que peu de place.

Le C++ définit plusieurs types d'entiers, selon l'étendue des valeurs que l'on désire mais aussi si on souhaite des entiers pouvant être négatifs ou non (signé ou non signé selon la terminologie C).

Voici un tableau récapitulant les divers types d'entiers proposés de base par le C/C++ :

Type C	Commentaire	étendue
`char`	Entier représenté sur 8 bits signé ou non signé	$[0; 255]$ ou $[-127; 127]$
`signed char`	Entier signé représenté sur 8 bits	$[-127; 127]$
`unsigned char`	Entier non signé représenté sur 8 bits	$[0; 255]$

`short`
`short int`
`signed short`
`signed short int`
	Entier signé représenté sur 16 bits	$[-32767; 32767]$

`unsigned short` `unsigned short int`	Entier non signé représenté sur 16 bits	$[0; 65535]$

`int`
`signed`
`signed int`
	Entier signé représenté sur au moins 16 bits	Contient
		$[-32767; 32767]$

`unsigned` `unsigned int`	Entier non signé représenté sur au moins 16 bits	Contient $[0; 65535]$

`long`
`long int`
`signed long`
`signed long int`
	Entier signé représenté sur au moins 32 bits	Contient
		$[-2^{31} + 1; 2^{31} - 1]$

`unsigned long` `unsigned long int`	Entier non signé représenté sur au moins 32 bits	Contient $[0; 2^{32} - 1]$

`long long`
`long long int`
`signed long long`
`signed long long int`
	Entier signé représenté sur 64 bits	$[-2^{63} + 1; 2^{63} - 1]$

`unsigned long long` `unsigned long long int`	Entier non signé représenté sur 64 bits	$[0; 2^{64} - 1]$

Il est très important de s'assurer lorsqu'on programme que les valeurs prises par nos entiers ne débordent pas, c'est à dire n'est pas contenu dans l'intervalle des valeurs que peut prendre le type d'entier. On parle alors de débordement de la représentation d'entier. Dépasser ces valeurs mènent à des bogues très dur à détecter et pouvant avoir des conséquences dramatiques. L'exemple le plus tristement célèbre d'un tel bogue est l'explosion d'ariane V lors de son premier vol, à cause d'une variable entière ayant dépassé les valeurs autorisées par son type. Pour illustrer le problème, considérons le code suivant :

```
short s = 32769;
signed char  t = 130;
std::cout << "s = " << s << " et t = " << int(t) << std::endl;
```

Listing 2.7 – Débordement d'entier

Si on exécute le code correspondant, le programme affichera

```
s = -32767 et t = -126
```

Ce ne sont pas les valeurs attendues !

L'explication vient du fait que le type `short` qui représente les entiers signés sur deux octets ne peut prendre que des valeurs entre $-2^{15} + 1 = -32767$ et $2^{15} - 1 = 32767$ et que le type `char` qui représente les entiers signés sur un octet ne peut prendre que des valeurs entre $1 - 2^7 = -127$ et $2^7 - 1 = +127$.

Un autre problème des entiers en C/C++ est que certains entiers ont une taille et une étendue dépendant du système d'exploitation. En particuliers :

— Le type `char` considéré comme signé sous Linux, Windows et Mac OS mais non signé sur Androïd !
— Les entiers longs codés sur trente-deux bits sous Windows mais sur soixante-quatre bits sur Linux ou Mac-OS

Heureusement, il existe un header `cstdint` qui permet de pouvoir spécifier sans confusion possible le type d'entier qu'on souhaite manipuler :

```
#include <cstdint>

int main()
{
  std::uint8_t byte; // byte est un entier non signé (u) représenté sur 8 bits (un
    octet)
  std::int8_t sbyte; // sbyte est un entier signé représenté sur 8 bits (un octet)
  std::uin16_t ush;  // ush est un entier non signé représenté sur 16  bits (deux
    octets)
  std::int16_t  sh;  // sh est un entier signé représenté sur 16 bits (deux octets)
  std::uint32_t uent;// uent est un entier non signé représenté sur 32 bits (quatre
     octets)
  std::int32_t ent;  // ent est un entier signé représenté sur 32 bits (quatre
    octets)
  std::uint64_t ulg; // ulg est un entier non signé représenté sur 64 bits (huit
    octets)
  std::int64_t   lg; //  lg est un entier signé représenté sur 64 bits (huit octets
    )
}
```

Listing 2.8 – Déclaration des entiers sans ambigüité

Enfin, il est vivement conseillé d'éviter le plus possible l'emploi des entiers non signés qui sont sources de nombreux bogues ! Prenons l'exemple suivant :

```
std::uint32_t i,j;
for ( i = 1; i < 99; ++i )
{
  for ( j = i+1; j >= i-1; --j)
  {
     // On fait des calculs ici en utilisant i et j
  }
}
```

Listing 2.9 – Exemple de bogue généré par l'emploi d'un entier non signé

L'exécution de ce code ne nous rendra jamais la main ! En effet, dès la première itération sur `i`, le test de continuation de la boucle sur `j` demande à ce que `j` soit supérieure ou égal à zéro, ce qui est toujours le cas puisque `j` est de type non signé !

Ce problème n'aurait pas eu lieu si on avait utilisé des entiers signés (ce qui est généralement suffisant).

De plus, si dans un code, un entier doit être positif pour que l'algorithme reste dans son domaine d'application, il est préférable d'expliciter cette contrainte plutôt que de l'impliciter en prenant un entier non signé pour représenter l'entier (voir le chapître précédent).

Prenons l'exemple suivant :

```
// Recherche racine carrée d'un entier de la forme n² par dichotomie
std::uint32_t sqn = 3249; // n² = 57²
std::uint32_t a = sqn;
std::uint32_t b = 0;
std::uint32_t c = (a+b)/2;
while (c*c-sqn != 0)// Tant que c n'est pas la racine carrée de a
```

```
{
  if (c*c-sqn<0)// Si c est plus petit que la racine carrée de a
  {
    b = c;
    c = (b+a)/2;
  }
  else
  {
    a = c;
    c = (b+a)/2;
  }
}
assert(c*c == sqn); // Post-condition
std::cout << "racine de " << sqn << " est egal a : " << c << std::endl;
```

Listing 2.10 – Problèmes rencontrés avec les entiers non signés

Ce programme pose plusieurs problème :

— Il n'est pas clair que `sqn` doit être un entier positif (l'algorithme pourrait chercher une racine complexe) si on ne fait pas attention à son type (qui varie du type signé que d'une lettre !). Initialisé `sqn` avec un nombre négatif risque de poser de sérieux problèmes…

— Plus grave, le programme ne marche pas et la boucle `while` (qu'on verra un peu plus loin) bouclera à l'infini. En effet, le test `c*c-sqn<0` n'est jamais vérifié car `c` et `sqn` sont des entiers non signés, et le résultat en C++ (comme en C) de la différence de deux entiers non signés est un entier non signé !

En remplaçant les entiers non signés par des entiers signés, on résoud les deux points mentionnés ci-dessus :

```
// Recherche racine carrée d'un entier de la forme n² par dichotomie
std::int32_t sqn = 3249; // n² = 57² (on peut remplacer)
assert(sqn>=0); // Pré-condition explicitant bien que sqn doit être positive !
std::int32_t a = sqn;
std::int32_t b = 0;
std::int32_t c = (a+b)/2;
while (c*c-sqn != 0)// Tant que c n'est pas la racine carrée de a
{
  if (c*c-sqn<0)// Si c est plus petit que la racine carrée de a
  {
    b = c;
    c = (b+a)/2;
  }
  else
  {
    a = c;
    c = (b+a)/2;
  }
}
assert(c*c == sqn); // Post-condition
std::cout << "racine de " << sqn << " est egal a : " << c << std::endl;
```

Listing 2.11 – Résolution problème entiers non signés avec des entiers signés

Ainsi, on évitera à la fois de nombreux bogues parfois difficiles à trouver et une documentation plus claire pour vous ou un autre lecteur éventuel du code.

En ce qui concerne l'affichage des entiers, comme on le voit sur l'exemple ci-dessus, il suffit de rajouter notre entier dans le flot de sortie. Ainsi, dans l'exemple ci–dessus,

```
std::cout << "racine de " << sqn << " est egal a : " << c << std::endl;
```

affichera bien

```
racine de 3249 est egal a : 57
```

Si on veut formater la sortie des entiers, on peut utiliser (en incluant `iomanip`) les fonctions `std::setw(n)` qui réserve `n` caractères pour le prochain affichage et `std::setfill(c)` qui complètera par le caractère `c` les `n` caractères réservés. Ainsi :

```
std::int32_t value1 = -32;
std::int32_t value2 =   3;

std::cout << "value1 = "  << value1 << std::endl;
std::cout << " et value2 = " << value2 << std::endl;

std::cout << "123456789ABCDEF" << std::endl;

std::cout << std::setw(15) << "value1 = " << std::setw(4) << value1 << std::endl;
std::cout << std::setw(15) << " et value2 = " << std::setw(4)  << std::setfill('0')
    << value2 << std::endl;
```

affichera

```
value1 = -32
 et value2 = 3
123456789ABCDEF
      value1 =  -32
   et value2 = 0003
```

Les lettres "123456789ABCDEF" permettant de numéroter les colonnes en hexadécimales de 1 à F, cela permet de voir qu'on affiche bien les entiers après la quinzième colonne sur la console.

Les réels

Comme les entiers, les réels font partie du C++ natif (c'est à dire qu'on n'a pas besoin d'inclure des bibliothèques). Les réels sont représentés sur les ordinateurs en virgule flottante, c'est à dire qu'un réel x est représenté sur un ordinateur sous la forme

$$x = \text{signe} \times \text{mantisse} \times 2^{\text{exposant}}$$

où

- signe est un bit valant zéro pour les nombres positifs et un pour les nombres négatifs ;
- mantisse est un entier **non signé** représenté sur m bits (n dépendant du type de réel choisit) ;
- exposant est un entier **signé** représenté sur e bits (e dépendant du type de réel choisit).

Il existe trois types de réels :

- Les réels simple précision : Codés sur trente-deux bits, leur mantisse est représenté sur vingt-trois bits et leur exposant sur huit bits. La valeur d'un réel simple précision se note de chiffres avec la virgule symbolisée par un point, avec un `f` à la fin du nombre. Par exemple : `float pi = 3.1415f;`
- Les réels double précision : Codés sur soixante-quatre bits, leur mantisse est représenté sur cinquante-deux bits et leur exposant sur onze bits. La valeur d'un réel double précision se note de chiffres avec la virgule symbolisée par un point. Par exemple : `double pi = 3.1415;`

— Les réels longs double précision : Censés être sur cent-ving-huit bits, ils sont sur la grande majorité des compilateurs représentés seulement sur quatre-vingt bits sur les processeurs de type Intel du fait qu'aucun processeur ne supporte actuellement en natif des réels cent-vingt-huit bits. La valeur d'un réel long double précision se note de chiffres avec la virgule symbolisée par un point et avec le caractère `l` ou `L` à la fin du nombre. Par exemple : `long double pi = 3.1415L`.

Il existe deux valeurs spéciales pour les réels : NaN et la valeur infinity qu'on trouve dans la bibliothèque `limits`.

La première valeur est une valeur indiquant que le nombre représenté ne correspond pas à la représentation d'un nombre réel (NaN est la contraction de "Not a Number"). Puisqu'obtenir un Nan peut être un comportement anormale pour un code, le C++ offre deux "valeurs" différentes pour les Nans : `quiet_NaN` pour calculer sans broncher même si le résultat n'a pas de signification, et `signaling_NaN` qui si utilisé lors d'un calcul signale une erreur.

La seconde valeur comme son nom l'indique représente une valeur infinie.

La syntaxe pour ces valeurs est de la forme

```
std::numeric_limits<Type>::quiet_NaN;
std::numeric_limits<Type>::signaling_NaN;
std::numeric_limits<Type>::infinity;
```

où `Type` est le type de réel pour lequel on veut une de ces trois valeurs (`float` ou `double`)

Les valeurs de type `NaN` ont la particularité de ne jamais être égal. Ainsi,

```
std::numeric_limits<float>::quiet_NaN == std::numeric_limits<float>::quiet_NaN;
```

renverra toujours `false`.

Pour tester si la valeur d'un réel est un NaN, il faut utiliser la fonction `isnan` présente dans la bibliothèque `cmath`. Par exemple :

```
double x = 0./0.;
std::cout << std::boolalpha << " x est un nan ? " << std::isnan(x) << std::endl;
```

affichera `true` sur la console.

Quant à la valeur `infinity`, elle a la particularité d'être toujours supérieure à n'importe quelle valeur réelle :

```
#include <limits>

int main()
{
  float fx = std::numeric_limits<float>::max; // Prend la valeur maximale que peut
    prendre un réel simple précision
  float finf = std::numeric_limits<float>::infinity; // Prends la représentation de
    l'infini en réel simple précision
  std::cout << std::boolalpha << fx << " < infini" << " ?  : " << (fx < finf) <<
    std::endl;
  return EXIT_SUCCESS;
}
```

affichera

```
3.40282e+38 < infini ? : true
```

Puisque les réels sur les ordinateurs sont contenus dans un volume de mémoire fixe, leur représentation est entâchée d'erreurs (pour représenter π, il faudrait par exemple une mémoire infinie…)

Introduisons la notion d'erreur relative :

Définition 2.1. On appelle erreur relative d'un réel x, l'erreur faite sur la valeur de sa représentation $\tilde{x}$ dans la mémoire de l'ordinateur :

$$E_r(x, \tilde{x}) = \frac{|x - \tilde{x}|}{|\tilde{x}|}$$

Cette précision relative va dépendre du type de réel choisi :
— Pour les réels simple précision, elle sera d'environ 10^{-6} ;
— Pour les réels double précision, elle sera d'environ 10^{-15} ;
— Pour les réels long double précision sur Intel/AMD, elle sera d'environ 10^{-18}.

Bien sûr, le cumul d'erreur lors des calculs qui se rajouteront ou se compenserons fera que cette erreur sera plus ou moins grande par rapport à l'erreur indiquée ci-dessus. C'est tout l'art du numéricien de choisir, à l'aide de ses outils mathématiques, les bons algorithmes qui permettent d'avoir une solution proche de la solution exacte.

Pour employer les fonctions usuelles utilisées en mathématiques sur les réels, il faut utiliser la bibliothèque cmath qui contient la majorité des fonctions réelles usuelles. Par exemple :

```cpp
#include <cmath>
#include <iostream>

int main()
{
  float fpi = std::acos(-1.f);
  float fx  = std::cos(fpi/4.f);

  double pi = std::acos(1.);
  double x  = std::cos(pi/4.);
```

Pour voir les fonctions mathématiques usuelles existantes dans la bibliothèque cmath, allez voir le lien suivant : `https://en.cppreference.com/w/cpp/numeric/math`

À partir de C++ 17, il existe également d'autres fonctions réelles plus spécialisées que l'on peut trouver dans la bibliothèque cmath, comme des polynômes de Laguerre, Legendre, la fonction zeta de Riemann, etc…

À partir de C++ 20, il existe également une bibliothèque de constantes mathématiques numbers qui définit π, l'exponentielle, la racine de deux ou de trois et leurs inverses, le nombre d'or, la constante d'Euler γ, etc. (voir le lien `https://en.cppreference.com/w/cpp/numeric/constants`).

```cpp
#include <cmath>
#include <numbers>
#include <iostream>

int main()
{
  float π_f = std::numbers::pi_v<float>;
  double π  = std::numbers::pi;
  long double π_lf = std::numbers::pi_v<long double>;
  double π⁻¹ = std::numbers::inv_pi;
```

```
    std::cout << "π_f  = " << std::setprecision(std::numeric_limits<float>::digits10
      +1) << π_f << std::endl;
    std::cout << "π    = " << std::setprecision(std::numeric_limits<double>::digits10
      +1) << π << std::endl;
    std::cout << "π_lf = " << std::setprecision(std::numeric_limits<long double>::
      digits10+1) << π_lf << std::endl;
    std::cout << "π^-1 = " << std::setprecision(std::numeric_limits<double>::digits10
      +1) << π^-1 << std::endl;

    return EXIT_SUCCESS;
}
```

Listing 2.12 – Exemple d'utilisation des constantes en C++ 20

Par défaut, la valeur π est en double précision, mais on peut utiliser `std::numbers::pi_v<Type>` pour avoir une valeur de π en simple précision ou en double long.

Notez l'utilisation de `std::numeric_limits<double>::digits10` avec `std::setprecision` pour afficher les chiffres significatifs de π selon le type de réel employé.

Les complexes

Les nombres complexes ne sont pas des types natifs du C++, mais la bibliothèque standard (défini par le standard ISO) propose des types complexes.

On peut donc trouver la gestion des complexes dans la bibliothèque `complex`, qui permet de définir plusieurs types de complexes :

```
#include <complex>

int main()
{
    ...
    std::complex<float> fcomplex;
    std::complex<double> i(0.,1.);
    std::complex<std::int32_t>   gauss(2,-3); // 2 -3i dans Z + i Z
    ...
}
```

Listing 2.13 – Exemple de déclaration et de définition de nombres complexes

Le type donné entre les chevrons < et > est le type de scalaire qui représente la partie réelle et la partie imaginaire du complexe. Ainsi, dans le code ci–dessus, `fcomplex` est un complexe dont la partie réelle et la partie imaginaire sont de type `float`, `i` quant-à lui aura sa partie réelle et sa partie complexe représentées par des réels double précision `double`, tandis que `gauss` aura sa partie réelle et sa partie imaginaire représentées par des entiers de type `std::int32_t`.

Remarquons aussi que l'initialisation (qu'on verra un peu plus bas) des nombres complexes se fait généralement en donnant la partie réelle et la partie imaginaire entre parenthèses lors de la déclaration.

Néanmoins, si on veut une écriture plus naturelle des nombres complexes dans le code, on le peut depuis le C++ 2014 de la manière suivante :

```
#include <complex>
#include <iostream>

int main()
{
    using namespace std::complex_literals;
    std::complex<double> c = 1.0 + 1i;
```

```
    std::cout << "abs" << c << " = " << std::abs(c) << '\n';
    std::complex<float> z = 3.0f + 4.0if;
    std::cout << "abs" << z << " = " << std::abs(z) << '\n';
}
```

Listing 2.14 – Exemple de déclaration des complexes à l'aide de littéraux

La ligne `using namespace std::complex_literals;` est essentielle pour pouvoir ensuite écrire les complexes d'une manière naturelle. Notons enfin que pour cette écriture naturelle ne fonctionne que si la partie réelle et la partie imaginaire d'un complexe sont réelles : simple (suffixe `if`), double (suffixe `i`) ou longue double (suffixe `ilf`).

Notons que la fonction `std::abs` dans l'exemple ci-dessus calcul la norme du complexe.

Il est possible également d'avoir l'argument du complexe ou bien encore sa norme au carré (ce qui évite de calculer une racine carrée). De plus, les fonctions usuelles sont définies dans `complex` et il est possible de calculer des exponentielles, des racines, des sinus ou encore des cosinus complexes (en prenant la construction de Poincarré pour ces fonctions).

Attention : Contrairement à ce que l'on peut penser, le code ci–dessous :

```
std::complex<double> c = 1. + 1i;
std::cout << c.norm() << std::endl;
```

n'affiche pas la norme de `c` mais le carré de sa norme (sic).

2.4.4 Les chaînes de caractère

Une chaîne de caractère basiquement est un tableau de "caractères", où les caractères sont des chiffres, des lettres, des symboles.

Ces chiffres, lettres, symboles peuvent être représentés de diverses manières en mémoire selon l'encodage choisi. On distingue les encodages suivant :

- ASCII (**A**merican **S**tandard **C**ode for **I**nformation **E**change) : C'est un encodage des caractères paru aux alentours de 1960 qui permettait de transmettre des suites de caractères entre différents appareils électroniques. Codé sur sept bits (plus un bit de contrôle), cet encodage est prévu que pour gérer les caractères anglosaxons. Rien n'est donc prévu dans la norme pour gérer les accents français, ou les lettres allemandes, le cyrillic, le grecque, etc... C'est néanmoins l'encodage le plus universel actuellement, accepté par tous les systèmes et machines actuelles.
- UTF8 (**U**niversal character set **T**ransformation **F**ormat – **8** bits) : est un encodage des caractères conçu pour coder l'ensemble des caractères connus proposé par l'ISO (**I**nternational **S**tandardization **O**rganization). Cet encodage reste totalement compatible avec le codage ASCII pour ce qui concerne les caractères anglo-saxons. Cet encodage a la particularité que chaque caractère n'est pas encodé sur un nombre fixe d'octet mais encodé sur un à quatre octets !
- UTF16 (**U**niversal character set **T**ransformation **F**ormat – **16** bits) : Proche de l'UTF-8, les caractères sont encodés cette fois ci sur deux ou quatre octets. Les caractères occidentaux sont tous encodés sur deux octets ce qui fait que l'UTF16 est compatible en occident avec l'encode UCS-2 (dont les caractères sont encodés avec une taille fixe de deux octets) ;
- UTF32 (**U**niversal character set **T**ransformation **F**ormat – **32** bits) : Proche de l'UTF-8, les caractères ont désormais une taille fixe de quatre octets par caractère.

Si il est possible de définir un caractère par un de ces quatre encodages, le C++, mise à part l'encodage ASCII, ne propose que peu de chose pour gérer efficacement les autres encodages, que ce soit pour les caractères que pour les chaînes de caractères.

Plusieurs types de chaînes de caractères sont proposés en C++ : les chaînes de caractères "basiques" et les `std::string` qui offrent des services permettant de gérer plus facilement les chaînes de caractères que les chaînes "basiques".

2.4.5 Les caractères

La valeur d'un caractère est encadrée par deux simples quotes `'`.

Puisqu'un caractère peut-être codé en ASCII (un octet), en UTF8 (un à quatre octets), en UTF16 (deux ou quatre octets), etc., il faut parfois utiliser un type spécial pour déclaré le type d'un caractère :

- Pour l'ASCII : Le type `char` suffit et on peut l'afficher normalement avec `std::cout` ;
- Pour l'UTF8 : le type `char` est utilisé, mais ne permet pas d'obtenir les caractères codés sur plus de un octet ! On affiche un caractère UTF8 avec `std::cout` ;
- Pour l'UTF16 : le type `wchar_t` est utilisé pour afficher toutes les lettres occidentales possibles à l'aide de `std::wcout` (Wide Cout). On peut également stocker un caractère UTF16 dans le type `char16_t`, mais il n'est plus possible dans ce cas de l'afficher (du moins, rien n'est proposé en C++ pour cela)
- Pour l'UTF32 : On peut stocker le caractère dans un `char32_t` mais rien n'est proposé dans le C++ pour l'afficher (du moins à ma connaissance).

Notons que `wchar_t`, `char16_t` et `char32_t` sont des types basiques à partir de C++ 11.

Pour définir l'encode du caractère, ormis pour l'ASCII où deux simples quotes suffisent, on rajoute un préfixe devant le premier simple quote, à savoir

- Le préfixe 'u8' pour un encodage utf8
- Le préfixe 'L' pour un encodage utf16
- Le préfixe 'u' pour un encodage utf32

```
char ascii = 'p';
char utf8  = u8'p';
// Remarquer que utf8 = u8'é' va générer une erreur car le caractère 'é' est codé
   en UTF 8 sur plus de un octet !
wchar_t utf16 = L'é';
// On aurait pu aussi écrire char16_t utf16 = L'é' mais on n'aurait pas pu l'
   afficher dans ce cas.
char32_t utf32 = u'é'; // Mais comment l'afficher ensuite ????
```

Enfin, remarquons, et ce sera le même problème pour les chaînes de caractère, que l'affichage d'un caractère autre qu'un caractère "ASCII" peut poser problème car rien ne vous assure que l'encodage des caractères attendus par votre console est compatible avec l'encodage choisi dans votre code (et c'est ce qui arrive 99.99% du temps !).

Il existe également des caractères de contrôle. Les plus courants sont :

- Le retour à la ligne `'\n'` qui permet d'aller à la ligne suivante
- La tabulation `'\t'` qui permet de tabuler et d'aligner l'affichage
- Le retour en début de ligne `'\r'` qui permet de revenir à la première colonne de la ligne courante

Les chaînes de caractères basiques

La valeur d'une chaîne de caractère est encadrée par deux doubles quotes `"`. Par exemple :

```
"Ceci est une chaîne de caractère !"
```

Notons qu'il est interdit, normalement, de sauter une ligne dans la définition d'une chaîne de caractère, car le saut à la ligne signifiera pour le C++ qu'on change d'instruction. Cela résultera alors par une erreur de compilation.

Notons que contrairement aux caractères, les chaînes de caractères acceptent les accents et autres symboles. Elles utilisent pour cela les caractères étendus (codés sur seize bits).

Si on veut afficher un caractère double quote dans une chaîne de caractère, il faut alors utiliser \". Par exemple (pour citer Desproges : "Dictionnaire superflu à l'usage de l'élite et des bien nantis.") :

```
"\"Alea Jacta Est\" : Ils sont bavards à la gare de l'Est !"
```

Pour avoir un encodage spécifique de la chaîne de caractère, on rajoute les mêmes préfixes que pour un caractère. Ainsi :

```
u8"π est un caractère très spécial !"
```

va coder la chaîne en UTF8.

Une chaîne de caractère étant considéré comme un tableau de caractères, on utilise comme type basique pour les chaînes de caractère le type `char []` qui n'est autre que la déclaration d'un tableau statique de caractère (les tableaux statiques sont présentés plus loin). Ainsi, on peut stocker notre chaîne de caractère UTF 8 comme :

```
char texte[] = u8"π est un caractère très spécial !";
```

Enfin, depuis C++ 11 qui s'est beaucoup inspiré de Python, il est possible de définir des chaînes de caractère "brutes", c'est à dire des chaînes de caractères où le C++ n'interprétera pas les caractères de contrôles, en particuliers le retour à la ligne ou les doubles quotes. Pour former de telles chaînes de caractères, il faut rajouter le préfixe R à gauche du premier double quote, puis à l'intérieur des doubles quotes, on définit un délimiteur constitué de une ou plusieurs caractères suivi d'une parenthèse ouvrante. On écrit ensuite ce que l'on veut jusqu'à ce qu'on écrit une parenthèse fermante suivie du même délimiteur que le début de la chaîne. On fermet ensuite les doubles quotes pour former la chaîne de caractère. Par exemple :

```
char raw_texte[] =
R"DEL(Si nous prenons encadrons une formule par une ( et  par une ) ("parenthèse")
    alors :
      1. Cela permet de s'assurer de l'ordre des opérations;
      2. D'être plus clair dans le code;
      3. Modifier l'ordre des opérations
)DEL";
```

nous permet de définir un texte multi-ligne en UTF 8 qu'on peut ensuite afficher sur console qui affichera :

```
Si nous prenons encadrons une formule par une ( et  par une ) ("parenthèse") alors :
     1. Cela permet de s'assurer de l'ordre des opérations;
     2. D'être plus clair dans le code;
     3. Modifier l'ordre des opérations
```

Remarquer que les retours chariot sont bien pris en compte lors de l'affichage. Pour ceux d'entre vous rompus au langage C, vous serez peut-être surpris que je n'ai pas employé un pointeur sur un tableau de caractère mais un tableau statique de caractère. En fait, à partir de C++ 11, il n'est plus possible de définir une chaîne de caractère constant via un pointeur sur cette chaîne.

Les chaînes de caractères std ::string

Si les chaînes basiques du C++ sont utiles pour afficher des messages à l'écran, elles ne sont guères pratiques à manipuler. Le C++ possède dans sa bibliothèque standard un type chaîne de caractère bien plus facile à manipuler : `std::string`. Pour l'utiliser, il faut inclure la bibliothèque `string` puis déclarer une variable comme étant de type `std::string`.

Ainsi :

```
std::string une_chaîne;
std::string une_deuxième_chaîne = "Ceci est une chaîne de caractère";
```

Listing 2.15 – Déclaration d'une std ::string

Si les deux lignes ci–dessus sont valides, la seconde ligne n'est pas optimale et effectue des copies mémoires inutiles et relativement gourmandes en ressources. En effet, la chaîne de caractère à droite du signe = est une chaîne de caractère basique, qui doit être convertie en chaîne de caractère `std::string`. Pour éviter cette copie, à partir de C++ 14, il est possible de directement déclarer la chaîne de caractère à droite du signe égal comme étant une `std::string`. Pour cela, il faut utiliser le suffixe littéral `s` après le dernier double quote après avoir utilisé un `using namespace` comme dans le code ci–dessous :

```
using namespace std::string_literals;// Indispensable pour utiliser le suffixe s

int main()
{
  std::string une_troisième_chaîne = "Ceci est une std::string"s;
}
```

Listing 2.16 – Utilisation d'un litéral pour définir un std ::string

Attention : En ce qui concerne l'encodage UTF 8, il était facile avant le C++ 20 d'afficher et stocker de l'UTF 8. Depuis le C++ 20, la chaîne de caractère utilisant l'UTF 8 n'utilise plus le type `char` mais le type `char8_t` ce qui a pour conséquence qu'on ne peut plus ni stocker une chaîne de caractère en UTF 8, ni l'afficher avec `std::cout` ! Bref, il vaut mieux éviter si possible de manipuler explicitement des chaînes de caractères avec un encodage spécifique !

Remarque : Si il est préférable actuellement d'éviter en C++ de manipuler des chaînes de caractères explicitement en u8, il est tout à fait possible d'écrire des chaînes de caractères avec des caractères appartenant à l'unicode en UTF 8, qui seront pris comme des caractères ordinaires lors de la manipulation, mais pas à l'affichage. En outre, éviter de manipuler des chaînes de caractères en unicode ne veut pas dire que de déclarer des variables en unicode pose problème...

Les `std::string` on l'avantage de pouvoir se manier facilement pour les traiter efficacement :

```
std::string     nom    = "Dupont"s;
std::string prénom     = "Albert"s;
std::string     âge    = std::to_string(20);
std::string identité = nom + " "s + prénom + "  : âge "s << âge << " ans.\n"s;
std::cout << identité;
```

Listing 2.17 – Exemple de manipulation de std ::string

Ici, on définit un nom et un prénom en `std::string`. On convertit ensuite un entier (ici 20) en chaîne de caractère à l'aide d'une fonction `std::to_string` puis on concatène les diverses variables pour créer une chaîne de caractère donnant l'identité de la personne.

On peut également tester si on trouve une sous-chaîne dans une chaîne (à partir du début de la chaîne ou à partir d'une position spécifiée comme dans l'exemple ci–dessous) :

```
std::string grosse_chaîne = "Ceci est un essai intéressant. Peut-on trouver une
    sous-chaîne dans une grosse chaîne ?";
// On cherche une première occurence de chaîne dans la grosse_chaîne :
std::size_t pos = grosse_chaîne.find("chaîne"s);
if (pos != std::string::npos)
    std::cout << "J'ai trouvé une première occurence de chaîne à la position : " <<
     pos << std::endl;
// Puis on recommence une nouvelle recherche après la première position de chaîne :
pos = grosse_chaîne.find("chaîne"s, pos+1);
if (pos != std::string::npos)
    std::cout << "J'ai trouvé une seconde occurence de chaîne à la position : " <<
    pos << std::endl;
```

Listing 2.18 – Exemple de recherche d'une sous-chaîne dans une chaîne

dont la sortie obtenue est :
```
J'ai trouvé une première occurence de chaîne à la position : 57
J'ai trouvé une seconde occurence de chaîne à la position : 81
```

Enfin, on peut tester la longueur d'une chaîne (en octet) :

```
std::cout << "Longueur de chaîne : " << "chaîne"s.length() << std::endl;
```

qui nous donne en sortie :
```
Longueur de chaîne : 7
```

Remarquons que le résultat affiché n'est pas celui *a priori* attendu, puisque "chaîne" ne contient que six caractère. Cela vient du fait que le caractère est stocké sur deux octets en UTF 8 (qui est ici implicite).

Cela nous permet également de conclure qu'il est préférable d'éviter de manipuler, hors affichage (et encore faut-il s'assurer que notre console est capable d'afficher de l'UTF 8, ce qui n'est pas toujours le cas, surtout sous Windows !), des chaînes de caractères qui possèdent autre chose que de l'ASCII sous peine d'avoir de nombreuses surprises

2.4.6 Déclarations automatiques implicites et explicites

Le compilateur peut également déduire le type d'une variable lorsque cela est possible. On utilise pour cela le mot clef `auto`. On parle alors de *déclaration implicite* d'une variable :

```
void main()
{
    auto i = 1; // OK, i est un entier
    auto x = 3.14; // OK, x est un double
    auto y = 2.5f; // OK, y est un float
    auto z = x + y; // z est un double par promotion de y en double pour le calcul
    auto w = x + i; // w est un double par promotion de i en double pour le calcul

    auto j; // Erreur de compilation ! Le compilateur ne peut pas déduire le type
    de j
}
```

Listing 2.19 – Exemple d'utilisation d'une déclaration implicite

Nous verrons dans le cours de nombreuses utilisations des déclarations implicites, en particuliers lors du cours sur les templates, où ce type de déclaration permet de simplifier fortement le code !

Il existe également un mot clef, `decltype`, permettant au compilateur de déduire *explicitement* un type de variable à partir d'une expression. Par exemple :

```
void main()
{
    int i = 1;
    decltype(i) j; // j est du même type que i et non initialisé
    double x = 3.14;
    decltype(x+i) y; // y est du même type que x+i, c'est à dire un double (
    remarquons que y est non initialisé)

    decltype(i/j+1) k  = 1+i/j; // k est un entier.
}
```

Listing 2.20 – déduction explicite de type

Là encore, l'emploi de ce mot clef est particulièrement utile lors de la programmation à l'aide des templates.

Enfin, si en C ou C++ 98, il n'existait qu'une façon de faire un alias de type en utilisant le mot clef `typedef`,

```
typedef double reel; // Alias de type à la façon C

reel x; // x est en fait un double
```

il existe en C++ 11 et supérieur une deuxième façon utilisant le mot clef `using` et permettant une syntaxe plus claire et plus souple !

```
using reel = double;

reel x; // x est en fait un double
```

De plus, lors du chapître sur les templates, nous verrons que c'est le seul moyen de faire un alias de type sur un template !

2.4.7 Initialisation des variables

L'initialisation des variables en C++ peut s'écrire de trois façons :

— À la C

```
int i = 0, j = 1, k = -2; // Déclaration avec initialisation
double x; // Déclaration uniquement
x = 3.1415; // Puis initialisation
```

— Initialisation par construction : à l'aide d'une parenthèse (C++ 98)

```
int i(0), j(1), k(-2);
double x(3.1415);
int l = int(x+0.5); // Conversion d'un réel en entier
std::vector<double> tableau(10,3.);// Créer un tableau de 10 réels rempli
 de la valeurs trois
```

Remarquer que la dernière initialisation, celle créant un tableau, est impossible à faire en utilisant l'écriture du C, puisque cette initialisation demande deux paramètres (un entier et un réel).

— Initialisation par liste d'initialisation : à l'aide d'une accolade

```
int i{0}, j{1}, k{-2};
```

À partir de C++ 20, les accolades permettent l'initialisation d'une structure en nommant les champs qu'on veut initialiser (dans l'ordre de déclaration sinon on aura une erreur), à l'instar de la norme 99 du langage C :

```
/* Définit le type Client : */
struct client
{
  int age;
  int poids
  int taille;
};
client john{
    .age = 35,
    .taille = 190,
};// Le poids est mis par défaut à zéro car non spécifié.
```

Si elle s'avère pratique, l'usage des accolades parfois peut être aussi source de confusion :

```
std::vector<double> tableau(10,3.);// Créer un tableau de 10 réels rempli
 de trois
std::vector<double> tableau{10,3.};// Créer un tableau de 2 réels :
[10,3] !
```

Il faut donc utiliser cette dernière possibilité avec précaution !

À partir de C++ 11 :

Si l'écriture octale, décimale ou hexadécimale est autorisée depuis le C, il n'était pas possible d'écrire un nombre en binaire. Depuis C++ 11, cet oubli est réparé et il est possible d'écrire un nombre en binaire à l'aide du préfixe `0b` :

```
int a = 037;          // Ecriture octale (préfixe 0), a vaut 31 en décimal (8*3 +
 7)
int b = 103;          // Ecriture décimale
int c = 0xF0;         // Ecriture hexadécimale (préfixe 0x) : c vaut 240 (15*16 +
 0)
int d = 0b01100110; // Ecriture binaire (préfixe 0b) : d vaut 102 (2 + 4 + 32 +
64)
```

Listing 2.21 – Ecriture binaire octale décimale et hexadécimale

À partir de C++ 14 :

La lecture de grand nombres, particulièrement en binaire, peut être pénible lors de la lecture ou l'écriture d'un code et être sources d'erreurs difficiles à détecter !

Considérons le code suivant :

```
double e      = 2.718281828459045;
long long  l  = 1345652343LL;
long long bl  = 0b11011011110110111011110111011101LL;
```

Si il est difficile de lire la valeur de `e` ou `l`, il est encore plus difficile de lire la valeur en binaire de `bl` !

C++ 14 introduit le caractère simple quote (') comme sépérateur de chiffres. Ainsi, en C++ 14 ou supérieur, il est possible de reécrire le code précédent comme suit :

```
double      e  = 2.718'281'828'459'045;
long long   l  = 1'345'652'343LL;
long long  bl  = 0b1101'1011'1101'1011'1011'1101'1101'1101LL;
```

Listing 2.22 – Ecriture avec séparateur de chiffre (C++ 14)

ce qui permet une lecture plus aisée des valeurs de `e`, `l` et `bl` !

Remarquons que le séparateur peut être mis n'importe où entre deux chiffres dans le nombre. Ainsi, il est possible (mais sans intérêt) d'écrire un nombre sous la forme `3'42'3564'3'2LL` ce qui ne facilite pas la lecture (qui est pourtant le but de ce rajout au C++)

2.4.8 Le qualificateur const

Considérons une variable telle que π. De toute évidence, cette variable sera fixée à une certaine valeur (qui sera une bonne approximation de la valeur exacte !) tout le long de l'exécution du programme.

En effet, imaginons le programme suivant :

```
double π = 3.141592653589793;
...
// Bien plus loin dans le code
π = -1.;
...
// Et encore bien plus loin
int rayon = 3;
// Calcul de la circonférence du cercle
double circonférence = 2*π*rayon;
```

Manifestement, le résultat à la fin sera faux ! Pourquoi avoir changé π, me direz-vous ? Hé bien, cela peut arriver pour moultes raisons ! On a voulu changer la valeur d'une autre variable et on a malencontreusement pris π à la place (à cause d'une complétion malheureuse, par exemple), ou encore, dans un contexte de programmation embarquée, on voulait reutiliser une variable déjà existante pour limiter la mémoire prise par le programme, etc…Bref, cela arrive bien plus fréquemment qu'on pourrait se l'imaginer.

Puisque de toute évidence, la valeur de π n'est pas destinée à être modifiée, il est nécessaire de la protéger contre toute modification éventuelle après son initialisation. C'est le rôle du mot clef `const`.

C'est un qualificateur, car il ne modifie pas le type fondamental de la variable. Simplement il lui rajoute une propriété supplémentaire : celle d'être immuable. Il se rajoute avant ou après le type de la variable.

Par exemple :

```
const double π = 3.141592653589793;
double const e = 2.718281828459045;
```

Tenter de modifier la valeur d'une variable constante déclenche une erreur de compilation. C'est bien ce que l'on veut !

Les expressions constantes

Il y a un point que nous n'avons pas abordé sur les variables constantes, c'est le moment où elles sont initialisées. En fait, une variable constante peut être initialisée soit au moment de la compilation soit au moment de l'exécution.

Il est parfois important que les variables constantes soient initialisées au moment de la compilation, particulièrement lorsqu'on utilise les librairies dynamiques qui ne permettent pas de savoir quand une variable constante sera initialisée. Pour garantir que l'initalisation se fait durant la phase de compilation, il existe depuis C++ 11 un mot clef `constexpr`.

Une variable `constexpr` est obligatoirement `const`. Et la seule différence est qu'elle garantit d'être initalisée durant la compilation. On ne peut initialiser une variable `constexpr` qu'à l'aide d'autres variables `constexpr`.

```
constexpr int i = 4; // i est initialisée au moment de la compilation
const int j = 3; // j peut être initalisée au moment de la compilation ou de l'
    exécution
constexpr int k = i/2; // ok, on utilise une variable constexpr pour calculer k
constexpr int l = j*2; // Erreur, j n'est |pas une variable constexpr.
```

Nous verrons plus loin qu'on peut également déclarer des fonctions `constexpr` et que cela permet d'utiliser des fonctions pour initialiser des variables `constexpr`.

2.4.9 Pointeurs et adresse mémoire

Les pointeurs natifs

Nous avons vu, sans le dire, un pointeur lorsque nous avons parlé des chaînes de caractères : `char *`. Un pointeur n'est rien d'autre qu'une adresse mémoire, c'est à dire un nombre donnant un emplacement dans la mémoire. Il peut bien entendu être initialisé à n'importe quel nombre, mais son intérêt est de prendre une valeur correspondant à l'emplacement mémoire d'une variable ! Remarquons qu'un pointeur ne permet pas directement de stocker une valeur, il ne fait que stocker l'adresse où se trouve cette valeur (qui aura donc déjà été stockée dans un espace réservé en mémoire).

Une pointeur se déclare avec un type, et une * accolée au nom de la variable. Attention, dans le cas d'une multi-déclaration, chaque variable de type pointeur doit avoir une * accolée à son nom. Ainsi :

```
int *pt_1, *pt2; // Pointeur sur des entiers
double * pt_3, *pt_4; // Pointeur sur des doubles
char *pt_5, ch; // Attention ch est un char, pas un pointeur !
```

Listing 2.23 – Déclaration de pointeurs

Pour trouver l'adresse d'une variable (son emplacement mémoire), il suffit d'utiliser le symbole `&\verb` :

```
int une_valeur = 3;
int *pt_valeur = &une_valeur; // Adresse de une_valeur
```

Il est également possible de faire de l'arithmétique avec les pointeurs. Dans ce cas, le pointeur tient compte du type de donnée sur lequel il pointe. Par exemple, si un pointeur `pt_tab` pointe sur le premier élément d'un tableau `tab` de `double`, alors `pt_tab+1` pointera sur le deuxième, et plus généralement `pt_tab+i` pointera sur le $(i+1)^e$ élément du tableau `tab` qui correspond à `tab[i]` (n'oublions pas que les indices commencent à zéro en C/C++, voir plus loin pour les tableaux).

Pour accéder à une valeur se trouvant à l'adresse stockée dans le pointeur, il suffit d'utiliser l'opérateur de déférencement `*` :

```
int une_valeur = 3;
int *pt_valeur = &une_valeur; // On pointe sur une_valeur
*pt_valeur += 1; // On incrémente une_valeur pointée par pt_valeur
```

Listing 2.24 – Exemple d'utilisation du déférencement

Les pointeurs sont très utiles pour divers contextes, en particuliers pour parcourir ou stocker des tableaux dynamiquement (voir plus loin).

Il est possible que deux pointeurs pointent sur la même variable. Si on utilise ces deux pointeurs pour lire ou modifier la valeur pointée, cela ne pose pas de soucis en séquentiel. Par contre, si on veut utiliser les deux pointeurs pour libérer l'espace mémoire occupée par la valeur (on verra comment faire plus loin), cela peut être problématique, car on ne peut pas libérer deux fois la même zone mémoire sans obtenir une erreur fatale durant l'exécution du programme.

Enfin, il existe un pointeur spécial, le pointeur nul, qui ne pointe sur aucune valeur. En C++, ce pointeur est défini dans la bibliothèque <memory> et se nomme `std::ptr_null`. Pour ceux qui ont fait du C, attention, ce n'est pas du tout l'équivalent de NULL en C (qui existe également pour la compatibilité avec le C) !

Pour les pointeurs, attention à l'usage du qualificateur `const` avec les pointeurs :

- Placé **avant le symbole** `*` désignant le pointeur, il signifie que le pointeur pointe sur une valeur qu'il considère comme constant (même si elle ne l'est pas)
- Placé **après le symbole** `*` désignant le pointeur, il signifie que le pointeur lui-même est constant, il ne peut pas pointer par la suite sur une autre valeur. Par contre on peut modifier la valeur pointée.
- Il est possible de placer le qualificateur `const` avant et après le symbole `*`. Dans ce cas, on déclare un pointeur constant qui pointe sur une valeur qu'il considère comme constant !

Ainsi :

```
int a = 3, b = 4;
const int c = 5;

const int * pt_a = &a;
*pt_a = -a; // Erreur de compilation ! a considéré constant par le pointeur
pt_a = &b;  // Correct, le pointeur n'est pas constant

int * const pt_b = &b;
*pt_b = -b; // Correct, pt_b pointe sur une valeur considérée comme non constant
pt_b = &a; // Erreur de compilation ! pt_b est un pointeur constant.

int const* const pt_c = &a;
*pt_c = -a; // Erreur de compilation ! a considéré constant par le pointeur
pt_c = &b;  // Erreur de compilation ! le pointeur est constant
```

```
int* pt_d = &c; // Erreur de compilation, on ne peut pas crée un pointeur sur une
    valeur non constante
                // si la valeur pointée est constante.
```

On peut voir qu'il est interdit de crée un pointeur sur une valeur non constant si on initialise ce pointeur sur une valeur constante.

Nous allons voir que C++ possède dans sa bibliothèque standard des pointeurs intelligents qui permettent d'allouer facilement un espace mémoire pour y stocker dynamiquement (c'est à dire durant l'exécution du programme) une valeur, mais aussi garantir qu'on ne peut pas libérer deux fois le même espace mémoire.

Pointeurs partagés

Les pointeurs partagés sont des pointeurs qui "comptent" le nombre de pointeurs qui pointent sur la même variable à l'aide d'une compteur de référence. Lorsqu'on quitte un bloc d'instructions contenant un tel pointeur, ce pointeur décrémente le compteur de référence, et si le compteur atteint la valeur zéro (plus aucun pointeur ne pointe sur cette valeur), il libère l'espace mémoire réservée pour cette variable.

On crée un tel pointeur en utilisant la fonction `std::make_shared` et en spécifiant entre < et > le type de valeur pointée. La fonction réserve l'espace mémoire et initialise dans la foulée la valeur voulue en fonction des paramètres mis entre ses parenthèses.

```
struct ficheEtudiant
{
    std::string prénom, nom;
    std::int32_t âge, numéro_carte_étudiant, promotion;
};
...
int main()
{
    auto pt_i1 = std::make_shared<int>(-4);
    auto pt_fiche1 = std::make_shared<ficheEtudiant>("Gérard","Chambier", 32,
    132493, 2022);
    auto pt_fiche2 = std::make_shared<ficheEtudiant>();
```

Listing 2.25 – Exemple de création de pointeurs partagés

Notons dans le cas de l'initialisation d'une structure pointée par un pointeur partagé, qu'on peut initialiser complètement ou pas du tout les champs de la structure mais qu'il est impossible d'utiliser une initialisation partielle en désignant les champs initialisés. On verra dans le chapître consacré à la programmation objet comment faire pour permettre de ne faire qu'une initialisation partielle.

Notons également que dans ce cadre, le mot clef `auto` est utilisé à bon escient puisque `std::make_shared` renvoie bien un pointeur partagé et que respécifier, par exemple, que `pt_i1` est un pointeur partagé sur un entier ne ferait qu'alourdir et rendre le code moins lisible :

```
int main()
{
    std::shared_ptr<int> pt_i1 = std::make_shared<int>(-4);
    std::shared_ptr<ficheEtudiant> pt_fiche1 =
        std::make_shared<ficheEtudiant>("Gérard","Chambier", 32, 132493, 2022);
    std::shared_ptr<ficheEtudiant> pt_fiche2 = std::make_shared<ficheEtudiant>();
```

Listing 2.26 – Même exemple sans auto

Pour connaître le nombre de pointeurs faisant référence à une valeur pointée par un vecteur partagé, on utilise la méthode `use_count`. Voici un exemple d'utilisation :

```
auto pt_i = std::make_shared<int>(-4);
std::cout << "Nombre de pointeur se référant à -4 : " << pt_i.use_count() << std::
    endl;
{
  auto pt_j = pt_i;
  std::cout << "Nombre de pointeur se référant à -4 : " << pt_i.use_count() << std
    ::endl;
  {
    auto pt_k = pt_j;
    std::cout << "Nombre de pointeur se référant à -4 : " << pt_i.use_count() <<
    std::endl;
  }
  std::cout << "Nombre de pointeur se référant à -4 : " << pt_i.use_count() << std
    ::endl;
}
std::cout << "Nombre de pointeur se référant à -4 : " << pt_i.use_count() << std::
    endl;
```

Listing 2.27 – Lire le nombre de pointeur faisant référence à une valeur

qui affichera

```
Nombre de pointeur se référant à -4 : 1
Nombre de pointeur se référant à -4 : 2
Nombre de pointeur se référant à -4 : 3
Nombre de pointeur se référant à -4 : 2
Nombre de pointeur se référant à -4 : 1
```

Vous pouvez trouver l'exemple complet dans `Exemples/Pointeurs/shared_pointers.cpp\verb`.

Signalons enfin qu'il est impossible de faire de l'arithmétique de pointeurs avec les pointeurs partagés.

Pointeurs uniques

Les pointeurs uniques sont des pointeurs qui garantissent qu'un et un seul pointeur à la fois peut pointer sur une valeur spécifique en mémoire. Il est ainsi impossible d'avoir une copie d'un vecteur unique. Un pointeur unique en C++ est de type `std::unique_ptr` et comme pour les pointeurs partagés, on spécifie le type de la variable pointée entre < et >.

Comme pour le pointeur partagé, il est possible à la fois de réserver et d'initialiser la variable qui sera pointée par le pointeur unique grâce à la fonction `std::make_unique`.

```
struct ficheEtudiant
{
    std::string prénom, nom;
    std::int32_t âge, numéro_carte_étudiant, promotion;
};
...
int main()
{
    auto pt_i = std::make_unique<int>(-4);
    auto pt_j = std::make_unique<int>();
    auto pt_fiche1 = std::make_unique<ficheEtudiant>("Robert", "Chambier", 24,
    135395, 2022);
    auto pt_fiche2 = std::make_unique<ficheEtudiant>();
}
```

Listing 2.28 – Exemple d'utilisation des pointeurs uniques

Là encore, il est possible d'initialiser une structure à condition d'initialiser tout ses champs.

Comme dit plus haut, il est impossible de recopier un pointeur unique dans un autre pointeur unique. Il est par contre possible de *déplacer* les données d'un pointeur dans un autre pointeur unique. Cela veut dire que le pointeur *déplacé* perdra sa référence sur la valeur pointée. Ainsi :

```
std::unique_ptr<int> pt_k = pt_i; // Erreur de compilation, copie interdite !
std::unique_ptr<int> pt_l = std::move(pt_i);// OK, on déplace les données de pt_i
    dans pt_l
```

Nous avons introduit ici une nouvelle fonction qui permet, non pas de copier la valeur d'une variable dans une autre variable, mais de *déplacer* la valeur d'une variable dans une autre. Ici, après appel de la fonction `std::move`, `pt_i` a perdu sa référence sur la variable `i`. C'est maintenant `pt_l` qui contient la référence.

Nous reviendrons plus en détail sur ces opérations de déplacement qui permettent depuis C++ 11 de simplifier l'interface et de mieux optimiser les codes C++.

2.4.10 Les références

Nous avons vu deux manières de lire ou modifier une valeur : soit un stockant directement cette valeur en mémoire à l'aide d'une variable, soit en stockant dans un pointeur l'adresse du début de la mémoire occupée par cette valeur. Remarquons que dans le deuxième cas, il faut d'une manière ou d'une autre (à l'aide d'une variable ou bien avec l'allocation dynamique qu'on verra plus loin) qu'on ait réservé de la mémoire pour stocker cette valeur avant de pointer dessus.

Il existe une troisième façon d'accéder à une valeur : par référence. Cette manière d'accèder à une valeur peut-être vu comme un "mixte" des deux autres accès : une référence ne permet pas de réserver et de stocker une valeur directement, elle doit pour cela faire référence à une variable ou un emplacement mémoire où la valeur est stockée (comme pour un pointeur), mais elle permet également d'accéder directement à la valeur à la manière d'une variable sans devoir passer par un opérateur de déférencement. Comme son nom l'indique, une référence fait référence à une valeur, en aucun cas elle ne la stocke. De plus, une référence ne peut pas changer de valeur à référer, contrairement à un pointeur !

Le symbole pour définir une référence est le symbole `&`, accolé (ou non) au type. Elle doit être obligatoirement initialisée à sa déclaration sur la valeur dont elle doit faire référence.

```
int i = 3;
int& j = i; // j fait référence à i
j = 4;      // i et j valent maintenant 4 !
```

Il est possible de définir des références avec une déduction automatique du type. Dans le cadre des déductions implicites, il suffit de rajouter le symbole `&` juste après `auto`. Dans le cadre des déductions explicites, il faut rajouter des parenthèses autour de l'expression qui permet de déduire le type :

```
int i = 3;
auto& j = i; // j fait référence à i et est de même type
decltype(i) k = i; // k est une variable recopiant la valeur de i
decltype((i)) l = i; // l est une référence à i.
```

Listing 2.29 – Exemple de référence avec déduction de type automatique

Comme on peut le constater, il est important de ne pas mettre de parenthèse en moins ou en trop lors de la déclaration explicite d'un type !

Attention, il ne faut pas croire qu'une référence est attachée à une variable ! Elle est attachée à une valeur à une certaine position dans la mémoire.

Ainsi, considérons le code suivant :

```
auto pt_x = std::make_unique<double>(0.303);
auto pt_y = std::make_unique<double>(0.0);
double& x = *pt_x;

std::cout << "*pt_x = " << *pt_x << ", *pt_y = " << *pt_y << " et x = " << x << std
    ::endl;
x = -0.303;
std::cout << "*pt_x = " << *pt_x << ", pt_y = " << *pt_y << " et x = " << x << std
    ::endl;
*pt_x = 0.404;
std::cout << "*pt_x = " << *pt_x << ", pt_y = " << *pt_y << " et x = " << x << std
    ::endl;
*pt_y = -1.;
std::cout << "*pt_x = " << *pt_x << ", pt_y = " << *pt_y << " et x = " << x << std
    ::endl;
std::swap(pt_x,pt_y);
std::cout << "*pt_x = " << *pt_x << ", pt_y = " << *pt_y << " et x = " << x << std
    ::endl;
x = 1.414;
std::cout << "*pt_x = " << *pt_x << ", pt_y = " << *pt_y << " et x = " << x << std
    ::endl;
*pt_x = 2.15;
std::cout << "*pt_x = " << *pt_x << ", pt_y = " << *pt_y << " et x = " << x << std
    ::endl;
*pt_y = 2.28;
std::cout << "*pt_x = " << *pt_x << ", pt_y = " << *pt_y << " et x = " << x << std
    ::endl;
```

On aura pour résultat à l'affichage :

```
*pt_x = 0.303, *pt_y = 0 et x = 0.303
*pt_x = -0.303, *pt_y = 0 et x = -0.303
*pt_x = 0.404, *pt_y = 0 et x = 0.404
*pt_x = 0.404, *pt_y = -1 et x = 0.404
*pt_x = -1, *pt_y = 0.404 et x = 0.404
*pt_x = -1, *pt_y = 1.414 et x = 1.414
*pt_x = 2.15, *pt_y = 1.414 et x = 1.414
*pt_x = 2.15, *pt_y = 2.28 et x = 2.28
```

ce qui montre bien que la référence `x` ici est bien attachée à une valeur en mémoire (qui peut être modifiée) et non à une variable !

Nous verrons une utilisation des références lors de l'appel de fonctions.

2.5 Les tableaux

2.5.1 Gestion statique contre gestion dynamique

En C++, il existe deux façon de réserver de la mémoire sur un ordinateur :

— **L'allocation statique** : la place mémoire nécessaire aux données qu'on veut stocker (variable ou autre) est connu à la compilation. C'est donc le compilateur qui lors de la création de l'exécutable, réservera l'espace nécessaire pour stocker les données dans l'exécutable lui-même ;

— **L'allocation dynamique** : la place mémoire nécessaire aux données qu'on veut stocker n'est pas connu au moment de la compilation. La réservation de l'espace mémoire se fera

donc durant l'exécution du programme, à l'aide d'instructions spécifiques permettant cette allocation.

L'allocation statique ne coûte rien en terme de temps d'exécution, au contraire de l'allocation dynamique qui doit faire appel aux services du système d'exploitation pour obtenir un espace mémoire réservé et protégé (des autres applications).

Par contre, sur certains systèmes d'exploitation (en particuliers Windows...), l'allocation statique est limitée à une taille mémoire qui dépend du système d'exploitation (sous Windows, selon la version du processeur et de Windows, cette limitation est entre 256ko et 512ko). Il est donc préférable d'utiliser l'allocation statique que pour de petites quantités !

2.5.2 Gestion dynamique de la mémoire

Il est possible de gérer dynamiquement la réservation et l'initialisation de variables. Pour cela on utilise l'instruction new qui permet d'allouer et d'éventuellement initialiser une variable.

La syntaxe de new est très simple : `pointeur = new type_name;` où `type_name` est le type de variable qu'on veut réserver dynamiquement.

Il est également possible d'initialiser la variable. Dans ce cas, la syntaxe de new est : `pointeur = new type_name(arguments)` où `arguments` sont les arguments permettant d'initialiser la valeur.

Par exemple :

```
auto pt_int = new int; // On crée dynamiquement un entier pointé par pt_int
auto pt_float = new float; // Idem pour un float pointé par pt_float
auto pt_fiche = new ficheEtudiant; // On crée dynamiquement une valeur de type
    ficheEtudiant

auto pt_int2  = new int(3);// Crée dynamiquement entier valant 3 pointé par pt_int2
auto pt_float2= new float(3.14);// idem float valant 3.14 pointé par pt_float2
// Initialisation dynamique d'une nouvelle fiche d'étudiant
auto pt_fiche2= new ficheEtudiant("Robert"s, "Chambier"s, 32, 345341, 2022);
```

Qui dit création dynamique dit également destruction dynamique ! Une variable créée dynamiquement ne se détruira pas et ne libérera pas la mémoire automatiquement. C'est à la charge du programmeur de détruire cette variable. C'est le rôle de l'instruction `delete`.

Ainsi, en prenant l'exemple de création dynamique ci-dessus, on peut détruire et libérer la mémoire prise par ses variables par :

```
delete pt_fiche2;
delete pt_float2;
delete pt_int2;
delete pt_fiche;
delete pt_float;
delete pt_int;
```

Remarquons que la destruction des variables a été sciemment faite dans le sens contraire de leur création. Ce n'est bien sûr par obligatoire, mais c'est conseillé pour ne pas laisser de "trous" inutiles en mémoire qui pénaliserait la place mémoire occupée par l'exécutable (mais c'est un sujet un peu complexe à aborder).

Attention à bien libérer la place mémoire avant de perdre le dernier pointeur sur cette valeur. Sinon, il est impossible d'accèder à la valeur en mémoire qui restera en mémoire jusqu'à la fin de l'exécution du programme. C'est ce qu'on appelle une fuite mémoire (memory leak en anglais). Si cette "fuite mémoire" se fait dans une partie du code appelée très souvent et régulièrement,

la place mémoire prise par l'exécutable deviendra de plus en plus grande jusqu'à ce que le programme "plante" faute de place mémoire. Ce type d'erreur est très difficile à trouver, et des utilitaires comme `valgrind` sous Linux sont une aide très précieuse pour détecter ce type d'erreur !

2.5.3 Autre façon de gérer la mémoire dynamiquement

Si les instructions `new` et `delete` étaient indispensables avant le C++ 11, elles le sont bien moins depuis ! En effet, nous avons vu qu'il existait des pointeurs partagés et uniques qui s'occupent de la réservation mémoire et de l'initialisation des variables.

L'avantage de ces pointeurs est qu'ils vous garantissent qu'il ne peut y avoir de fuite mémoire dans votre programme si vous les utilisez. En effet, ils détruisent et libèrent automatiquement une valeur dès qu'il n'y a plus de pointeurs sur cette valeur !

Ainsi, on aurait pu avantageusement remplacer l'allocation vu précédemment avec `new` par :

```
auto pt_int = std::make_shared<int>();
auto pt_float = std::make_shared<float>();
auto pt_fiche = std::make_shared<ficheEtudiant>;

auto pt_int2  = std::make_shared<int>(3);
auto pt_float2= std::make_shared<float>(3.14);
auto pt_fiche2= std::make_shared<ficheEtudiant>("Robert"s, "Chambier"s, 32, 345341,
    2022);
```

La désallocation de ces variables se fera automatiquement et permet ainsi une simplification du code tout en le rendant moins sensibles aux fuites mémoires !

Remarque : On aurait pu remplacer `std::make_shared` par `std::make_unique` si on sait qu'un seul pointeur

Gestion statique de la mémoire

En C (valable aussi en C++), un tableau statique se déclare de la manière suivante :

```
const int N = 10;
double array1[N]; // Déclation du tableau sans initialisation
double array2[N] = { 1., 2., 3., 4., 5., 6., 7., 8., 9., 10. }; // Avec
    initialisation
double array3[]  = { 1., 3., 7., 11. };// La taille est déterminée par la liste d'
    initialisation { ... }
```

Listing 2.30 – Allocation statique d'un tableau en C/C++

Cette déclaration d'un tableau statique pose quelques problèmes. Le compilateur ne vérifie pas obligatoirement que la liste d'initialisation possède un nombre de valeurs inférieur ou égal à la taille du tableau :

```
double array[4] = {1., 2., 3., 4., 5.};
```

Dans le cas où la taille d'un tableau est déterminée par sa liste d'initialisation, il n'est pas immédiat de trouver la taille du tableau :

```
double array3[]  = { 1., 3., 7., 11. };// La taille est déterminée par la liste d'
    initialisation { ... }
```

```
int size_of_array3 = sizeof(array3)/sizeof(double); // Calcul de la taille du
    tableau statique
```

Listing 2.31 – lire la taille d'un tableau statique en C/C++

De plus, l'accès aux éléments du tableau n'est pas protégé et il est facile de se tromper dans les indices et lire des valeurs en dehors du tableau :

```
double array3[]  = { 1., 3., 7., 11. };// La taille est déterminée par la liste d'
    initialisation { ... }
double x = 0.;
for ( int i = 0; i < sizeof(array3); ++i )// Erreur sur la taille du tableau !!!!
    x += array3[i];                       // Et donc dépassement du tableau et
    mauvais calcul...
```

Listing 2.32 – Erreur d'indice

Remarque 1 : L'exemple ci-dessus à permis d'introduire la liste d'initialisation, c'est à dire un ensemble de valeurs définies entre deux accolades { et } qui permettent d'initialiser une collection de valeurs (comme un tableau). La liste d'initialisation est un objet important du C++ à partir de sa version C++ permettant une grande souplesse dans l'initialisation des objets (voir le cours sur l'objet qu'on fera plus tard.)

Remarque 2 : Attention, dans l'exemple ci–dessus, la variable N est bien déclarée comme **constante**, ce qui permet de définir des tableaux statiques de dimension N. Cependant, si N n'est pas constant, rien ne garantit que votre code compilera sur tous les compilateurs. C'est en fait une extension du compilateur gnu C qui permet d'allouer un tableau statique de façon dynamique (sic !) mais qui ne marchera pas sur des compilateurs comme celui de microsoft, intel, ...

En C++, depuis C++ 11, il existe une solution plus sûre et plus conviviale, permettant de vérifier (en particuliers en mode développement) ou de protéger l'accès aux données du tableau. Il faut pour cela utiliser `array` de la bibliothèque standard du C++.

La construction d'un tableau statique en C++ ressemble beaucoup à la construction faite en C, mais avec beaucoup plus de vérifications quant à la construction du tableau :

```
#include <iostream>
#include <array>

int main()
{
    // Déclaration d'un tableau statique de dimension quatre contenant des entiers
    non initialisés
    std::array<int,4> iarray4;

    // Déclaration d'un tableau statique de dimension cinq contenant des doubles
    initialisés par une liste
    std::array<double,5> darray5({ 1.2, 2.3, 3.4, 4.3, 3.2});

    // Autre écriture pour initialiser un tableau statique de dimension quatre
    contenant des doubles initialisés par une liste
    std::array<double,4> darray4 = { 1.2, 2.3, 3.4, 4.3 };

    // A partir de C++ 17, il est possible d'omettre le type et la taille si on
    donne une liste de valeurs :
    std::array darray6 = { 1., 2., 3., 4., 5., 6.}; // Equivalent à std::array<
    double,6> pour le type

    // Si la taille est précisée, le C++ vérifie qu'on a pas une liste contenant
    plus de valeurs que la dimension du tableau
```

```
    std::array<double,4> darray4_2 = { 1.2, 2.3, 3.4, 4.3, 3.2 };// Erreur à la
    compilation
    ...

}
```

On accède aux éléments du tableau de la même manière qu'un tableau statique en C, mais avec une possibilité de vérification de l'indice passé en plus :

```
    // On lit le premier élément du tableau darray5 : x devient un alias constant
    sur ce premier élément
    const double& x = darray5[0];
    // On lit le cinquième élément d'un tableau de dimension quatre ! Si -
    D_GLIBCXX_DEBUG a été mis en option de compilation
    // l'exécutable s'arrête en signalant qu'un mauvais indice a été passé au
    tableau !
    double y = darray4[4];
    // Bien sûr, on peut également modifié un élément d'un tableau :
    darray4[2] = 3.14;
```

D'autres services non présent pour le tableau statique de type C sont disponibles. Notez la façon d'appeler ces fonctions, appelées méthodes, où on écrit le nom de l'objet sur lequel le traitement doit être effectué, suivi d'un point et du nom de la méthode à appeler sur cet objet :

```
    // Lire la dimension du tableau (son nombre d'éléments)
    auto sz = iarray4.size();
    // Remplir le tableau avec une valeur donnée en paramètre :
    iarray4.fill(-1); // Remplit le tableau iarray4 avec la valeur -1.
    // On peut aussi comparer des tableaux lexicographiquement :
    std::array<int,4> iarray4_2 = {-1, -2, 1, -1};
    if (iarray4_2 < iarray4)
        std::cout << "iarray4_2 inférieur à iarray4" << std::endl;
```

Il est également possible de parcourir de façon "automatique" les éléments d'un tableau. Pour cela, on doit introduire un concept du C++ qui s'applique non seulement aux tableaux statiques, mais à un grand nombre de conteneurs : **les itérateurs**

Un itérateur est un objet qui, pointant sur des éléments d'un tableau ou d'un conteneur possède la propriété d'itérer au travers des éléments du conteneur en utilisant un ensemble d'opérations (l'incrément $++$ en particuliers et l'opérateur de déférencement $*$).

La forme la plus basique d'itérateur est le pointeur en C, qui pointe sur un élément d'un tableau et peut itérer en utilisant l'opérateur $++$ (pré ou post) pour parcourir les éléments du tableau.. Mais d'autres types d'itérateurs sont possibles. Par exemple on peut définir un itérateur permettant de parcourir une liste dont les éléments ne sont pas contigüs en mémoire.

Il faut remarquer qu'un pointeur est un type d'itérateur, mais que ce n'est pas tous les itérateurs qui ont les mêmes fonctionnalités que les pointeurs. On peut classifier les itérateurs en cinq catégories : Les itérateurs d'entrée, de sortie, les itérateurs uni-directionnels, les itérateurs bi-directionnels et les itérateurs à accès randomisés.

1. **Les itérateurs d'entrée et de sortie** : ce sont les itérateurs les plus limités. Ils ne peuvent que parcourir en lisant qu'une seule fois les éléments d'un tableau en lecture ou écriture (pensez à un itérateur sur un fichier par exemple ou des données lues sur un port série, etc.) ;

2. **Les itérateurs uni-directionnels** : Ils ont toutes les fonctionnalités d'un itérateur de sortie et d'entrée si il n'est pas constant. Il est possible de lire ou écrire plusieurs fois le même élément mais on ne peut qu'itérer que du premier ou dernier élément ou du dernier élément au premier. Tous les itérateurs des conteneurs sont au minimum des itérateurs uni-directionnels. Un exemple d'itérateur uni-direcionnel est un itérateur sur une liste simple chaînée.
3. **Les itérateurs bi-directionnels** : similaire aux itérateurs uni-directionnels, sauf qu'il est possible d'itérer en avant ou en arrière (avec l'opérateur —)
4. **Les itérateurs à accès randomisés** : Ce sont les itérateurs qui ressemblent aux pointeurs. Il est possible de sauter plusieurs éléments en avant ou en arrière et dont lire les éléments d'un conteneur de façon "aléatoire" en utilisant une arithmétique semblable à celle des pointeurs.

Pour accéder aux itérateurs d'un conteneur, le conteneur propose de façon standard deux méthodes : `begin()` qui crée un itérateur sur le début du conteneur et `end()` qui crée un itérateur pointant sur l'adresse mémoire suivant le dernier élément du conteneur. Ainsi, dans le cas d'un objet de type `array`, on peut parcourir tous ces éléments comme dans l'exemple suivant :

```
    std::cout << "darray4 : ";
    for ( auto iter = darray4.begin(); iter != darray4.end(); ++iter )
        std::cout << *iter << " ";// <-- notez le déférencement
    std::cout << std::endl;
```

Listing 2.33 – utilisation explicite des itérateurs pour array

Depuis le C++ 11, on peut "cacher" l'utilisation de ces itérateurs à l'aide d'une écriture plus simple de la boucle `for` (et cela est valable pour tout objet possédant des itérateurs). La boucle de l'exemple suivant parcourt automatiquement tous les éléments de `darray5` (en prenant chaque élément en référence constante pour éviter une copie) :

```
    std::cout << "darray5 : ";
    for ( const double& val : darray5 ) std::cout << val << " ";
    std::cout << std::endl;
}
```

Listing 2.34 – utilisation implicite des itérateurs pour array

Gestion dynamique de la mémoire

En C, il n'y a guère le choix : on alloue dynamiquement un tableau à l'aide de l'instruction `malloc` dont on reçoit l'adresse de début dans un pointeur. Il faut ensuite libérer cet espace mémoire lorsqu'on n'en a plus besoin à l'aide de l'instruction `free` :

```
int n;
...
double* pt_array = (double*)malloc(n*sizeof(double));
...
free(pt_array);
```

En C++, pour des raisons d'initialisation de certains objets, il est déconseillé d'utiliser `malloc` et `free`. Il est préférable d'utiliser leur équivalent (à l'initialisation des objets près) : `new` et `delete` dont l'écriture est moins lourde :

```
int * pt_n = new int; // On réserve une variable entière
int& n = *pt_n;

double* pt_pi = new double(3.1415); // Réserve un double et l'initialise à 3.1415
...
double* pt_array = new double[n];// On réserve un tableau de n doubles
...
delete [] pt_array; // On libère un tableau dynamique
delete pt_n;        // On libère une variable dynamique
```

Listing 2.35 – Exemple d'utilisation de new et delete

Attention : Il est interdit de désallouer un tableau allouer avec `malloc` avec un `delete` sous peine d'erreurs mémoires lors de l'éxécution. De même il ne faut pas libérer à l'aide de `free` un tableau alloué avec `new`.

Néanmoins, principalement à partir du C++ 11, l'utilisation de `new` et `delete` n'est utile que dans certains contextes particuliers. On préférera pour créer un tableau dynamique utiliser l'objet `vector` de la bibliothèque du même nom qui permet une gestion automatique de la mémoire sans avoir à s'occuper de la libération de la mémoire (qui se fera automatiquement lors de la sortie du bloc d'instruction où a été déclaré le tableau) :

```
// Réserve un tableau dynamique de trente entiers qui se détruira quand il ne sera
    plus visible
std::vector<int> indices(30);
// Déclare un tableau de doubles :
std::vector<double> tableau;
// Et réserve ensuite 10 éléments :
tableau.resize(10);
// Autre possibilité pour réserver une nouvelle taille
// en perdant les éléments déjà contenus mais c'est plus rapide que resize :
std::vector<double>(20).swap(tableau);
// Créer dynamiquement un tableau de 10 réels donnés dans une liste d'
    initialisation
// Notez que le type n'a pas besoin d'être précisé puisque la liste contient des
    doubles (déduction automatique : C++ 17)
std::vector tab2 = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10. };
// On realloue le tableau tab2 à ving éléments (en conservant les dix premiers
    éléments)
tab2.resize(20);
// Alloue un tableau d'entiers de dimension 100 rempli de -1:
// Là encore, on ne précise pas le type car on passe un entier pour valeur de
    remplissage (déduction automatique) :
std::vector ind2(100,-1);
// Et là on alloue un tableau de réel rempli de 3.14
std::vector pi_array(100,3.14);
// Création d'un tableau recopiant les données d'un autre conteneur via les
    itérateurs :
std::vector<double> copie(tab2.begin(), tab2.end());
// Et plus simple encore, copie du tableau ind2 dans un nouveau tableau d'entier
std::vector<int> ind3(ind2);
```

Pour interagir avec des librairies C ou d'autres langage, il est possible de lire l'adresse du premier élément (valable aussi pour `array` !) :

```
double* pt_tab2 = tab2.data();
```

Il est bien sûr possible de lire ou écrire dans un tableau de type `vector` :

```
// Lit le quatrième élément
double x = tab2[3];
// Ecrit le cinquième élément
tab2[5] = 3.14;
// Lit le premier élément (on peut aussi écrire dessus)
double p = tab2.front();
// Ecrit le dernier élément (on peut aussi seulement le lire)
tab2.back() = -3.14;
```

On peut également changer les données de deux pointeurs à condition qu'ils aient des éléments de même type :

```
// Echange des données entre tab2 et tableau :
tab2.swap(tableau);
```

La gestion mémoire étant optimiser au mieux, il se peut que le tableau que vous manipulez ait une taille inférieure à la réservation mémoire effectivement faite. Pour connaître la taille du tableau ou reallouer le tableau de sorte que la réservation mémoire corresponde à la taille. On peut également préréserver une taille mémoire pour remplir ensuite le tableau élément par élément sans avoir à faire de reallocation.

```
// Lire le nombre d'éléments contenus dans le tableau :
auto sz = tab2.size();
// Realloue le tableau de sorte que la mémoire soit optimisée (C++ 11) :
tab2.shrink_to_fit();
// Création tableau vide :
std::vector<double> an_array;
// Réservation de 10 doubles dans le tableau (mais toujours avec 0 éléments)
an_array.reserve(10);
// Puis on rajoute les éléments un à un :
for ( int i = 0; i < 10; ++i )
    // En rajoutant chaque élément à la fin du tableau :
    an_array.push_back(3.1415*i);
// Le tableau aura bien 10 éléments à la fin avec une seule allocation avec l'appel
    à reserve
```

2.6 Les fonctions

Une fonction est une entité qui peut prendre en entrée des paramètres, exécute un algorithme et retourne éventuellement une valeur. La syntaxe pour déclarer une fonction est `[type retour] [nom fonction]([liste de paramètres]);`. On déclare typiquement une fonction dans un header (un fichier avec l'extension `.hpp`). Pour définir une fonction, c'est à dire mettre en œuvre l'algorithme permettant à la fonction de remplir son contrat, la syntaxe ressemblera à

```
[type retour] [nom fonction]([liste paramètres])
{
  // Bloc d'instructions
}
```

La définition sera écrite dans le fichier `.cpp` correspondant au fichier contenant la déclaration.

Ainsi, si on a écrit une fonction permettant de calculer la racine d'un entier en renvoyant l'entier le plus proche de la racine de ce nombre, on déclarera la fonction dans un header, par exemple `imath.hpp` :

```
#ifndef _IMATH_HPP_
#define _IMATH_HPP_

int sqrt(int i);

#endif
```

Listing 2.36 – Fichier imath.hpp

et le corps de la fonction dans le fichier `imath.cpp` :

```
#include "imath.hpp"

int sqrt(int i)
{
  ... // L'algorithme pour calculer la racine entière
}
```

Listing 2.37 – Fichier imath.cpp

Le include dans le fichier `imath.cpp` n'est pas nécessaire mais permet néanmoins de s'assurer que la signature de la fonction (ses arguments) est la même dans le header que dans le fichier de mise en œuvre.

Le corps de la fonction, aux arguments près, ressemble beaucoup au `main` vu depuis le début. `main` est en fait une fonction, la fonction principale qui sera toujours appelée en premier à l'exécution.

En général, un header ne va pas contenir une seule fonction, mais plusieurs fonctions regroupée autour d'un même thème. Par exemple, on pourra déclarer et mettre en œuvre dans `imath` toutes les fonctions effectuant des calculs sur les entiers : puissance nième d'un entier, test de primalité, développement p-adique d'un entier, résolution d'équations sur $\mathbb{N}$, etc.

On pourra également regrouper une structure et toutes les fonctions effectuant des traitements sur cette structure au sein d'un même fichier d'entête.

On voit dans la syntaxe d'une fonction qu'on doit préciser le type de valeur qu'elle renvoit. Mais comment fait-on si on veut une fonction qui ne renvoit aucune valeur ? C'est là que le type `void` va intervenir. En effet, pour une fonction ne renvoyant aucune valeur, on précise simplement que le type de valeur renvoyer par la fonction est de type vide (`void`).

Ainsi, on a deux possibilités pour une fonction :

— Soit elle retourne une valeur et dans ce cas, il faut préciser le type de la valeur retournée, puis retourner une valeur de ce type dans le code de la fonction. On récupère ensuite cette valeur dans une nouvelle variable... ou non ! (mais dans ce cas, la valeur retournée est perdue pour la suite). Par exemple :

```
int iracine(int val)
{
   int irac;
   irac = ... // Algorithme calculant l'entier le plus proche de la racine
  de val
   return irac;// On renvoit le résultat (une valeur entière)
}

...
// Ok, on récupère et on stocke le résultat dans une nouvelle variable
```

```
  int isq177 = iracine(177);
  // On affiche le résultat, mais celui-ci est perdu par la suite. Impossible
     à le récupérer !
  std::cout << "racine entière de 193 est : " << iracine(193) << std::endl;
```

— Soit elle ne retourne pas de valeur. Pas de retour d'une valeur à faire. On peut néanmoins appeler return sans donner de valeur si on veut quitter la fonction sans exécuter ce qui peut suivre dans la fonction :

```
void affiche_valeur(double x)
{
  if (x<0)
  {
    std::cout << std::set_precision(5) << std::setw(9) << x;
    return;
  }
  std::cout << std::set_precision(5) << std::setw(8) << x;
}

...
// Appel de la fonction affiche_valeur
affiche_valeur(3.1415);
affiche_valeur(-3.1415);
...
```

Les paramètres d'une fonction sont une copie des valeurs passées lors de l'appel. Il est donc impossible de conserver un paramètre avec une valeur prise dans la fonction après être sorti de cette fonction.Prenons par exemple le code suivant :

```
# include <iostream>
  void produit_par_deux(double x)
  {
    x = 2*x;
  }

  int main()
  {
      double val = 1.5;
      std::cout << "val => " << val << std::endl;
      produit_par_deux(val);
      std::cout << "val => " << val << std::endl;
      return EXIT_SUCCESS;
  }
```

Si on compile et exécute le code, on obtient la sortie suivante :

```
val => 1.5
val => 1.5
```

Visiblement, la fonction `produit_par_deux` n'a pas modifié `val` ! L'explication est simple : au moment de l'appel de la fonction, le C++ va copier la valeur de val dans un autre emplacement mémoire et utiliser cette copie pour exécuter le code de la fonction. Ainsi, c'est une copie de `val` pour qui on multipliera la valeur par deux, et non la valeur de `val` elle-même.

Ici, pour pouvoir obtenir le résultat de la multiplication par deux, le plus simple est de retourner la valeur résultante de l'opération et de récupérer à nouveau le résultat dans `val` :

```cpp
# include <iostream>
double produit_par_deux(double x)
{
  return 2*x;
}

int main()
{
  double val = 1.5;
  std::cout << "val => " << val << std::endl;
  val = produit_par_deux(val);
  std::cout << "val => " << val << std::endl;
  return EXIT_SUCCESS;
}
```

Cette fois-ci on obtient bien la sortie suivante : `val => 1.5` `val => 3`

2.6.1 Retour de plusieurs valeurs

Mais que fait-on si on doit retourner plusieurs valeurs ? Par exemple, supposons que nous voulons écrire une fonction qui calcule la division euclidienne de deux entiers et qui doit renvoyer le résultat et le reste de la division (donc deux entiers).

Une première solution serait, comme en C, de passer un pointeur (natif, partagé ou unique). Par exemple :

```cpp
int division_euclidienne(int p, int q, int* reste)
{
  int résultat = p/q;
  *reste = p - q*résultat;
  return résultat;
}
...
int a = 7, b = 3;
int reste;
int quotient = division_euclidienne(a,b,&reste);
std::cout << "Division de a par b : " << quotient << " avec pour reste : " << reste
    << std::endl;
...
```

Si cette solution répond à notre besoin, elle est tout de même peu commode et fragile :

- Passer un pointeur pour modifier une valeur passée en paramètre n'a rien "de naturel" ;
- Cela alourdit le code, car on est obligé de déférencer la variable dans la fonction et passer l'adresse d'une variable en paramètre ;
- Que se passe-t'il si on passe `nullptr` à la fonction ? Ou bien l'adresse de `quotient` ?

Il existe en fait dans la bibliothèque standard du C++ des types de données permettant de renvoyer plusieurs valeurs.

À commencer par le tableau statique `std::array` ! En effet, pour notre fonction plus haut, rien ne nous interdit de renvoyer un tableau statique de deux entiers :

```cpp
std::array<int,2> division_euclidienne(int p, int q)
{
  int résultat = p/q;
  return {résultat, p - q*résultat};
```

```
}
```

Pour récuper les deux valeurs (le résultat et le reste de la division), on doit normalement recevoir les deux valeur dans un tableau, ce qui est lourd et peu expressif/compréhensible :

```
auto résultat = division_euclidienne(7,3);
std::cout << "Résultat de la division : " << résultat[0]
          << " et le reste : " << résultat[1] << std::endl;
```

Il n'est en effet pas facile de comprendre à la première lecture que le premier élément du tableau est le résultat de la division et le deuxième élément le reste lors de l'appel de la fonction. Néanmoins, depuis le C++ 17, tant qu'on renvoie certains types de valeurs, il est possible de récupérer les valeurs directement dans des variables. Ainsi, le code ci–dessus devient :

```
auto [résultat, reste] = division_euclidienne(7,3);
std::cout << "Résultat de la division : " << résultat
          << " et le reste : " << reste << std::endl;
```

Le code a nettement gagné en expressivité ! Nous avons renvoyé ici deux valeurs entières, mais il est parfaitement possible avec cette technique d'en renvoyer plus. Par exemple, imaginons que nous voulons écrire une fonction qui renvoie toutes les racines cubiques d'un nombre complexe z : en écrivant sous forme polaire $z = ze^{i\theta}$, les racines cubiques sont :

$$z^{\frac{1}{3}}e^{i\frac{theta}{3}}, z^{\frac{1}{3}}e^{i\frac{theta}{3}+\frac{2\pi}{3}}, z^{\frac{1}{3}}e^{i\frac{theta}{3}+\frac{4\pi}{3}}$$

On peut donc écrire la fonction sous la forme suivante :

```
std::array<std::complex<double>,3> racines_cubique(std::complex<double> z)
{
    const double _2s3 = 2*std::numbers::pi/3.;
    const double _4s3 = 4*std::numbers::pi/3.;
    double argument = std::arg(z)/3.;
    double module   = std::cbrt(std::abs(z));// std::cbrt => Racine cubique

    return {
        module * std::exp(1.i*argument), module * std::exp(1.i*(argument + _2s3)),
        module * std::exp(1.i*(argument + _4s3))
    };
}
```

et l'appeler de la manière suivante pour récupérer les trois racines :

```
auto [z1,z2,z3] = racines_cubique(1.+1.i);
std::cout << "\tRacines cubiques de 1+i : " << z1.real() << "+" << z1.imag() << "i,
    "
          << z2.real() << "+" << z2.imag() << "i, " << z3.real() << "+" << z3.imag
    () << "i"
          << std::endl;
```

Cependant, cette solution ne peut marcher que si les données *sont de type homogène*. En effet, un `std::array` n'accepte que des valeurs de même type. Comment faire alors si on veut retourner deux valeurs de type différent ?

La bibliothèque du C++ propose dans ce cas le type `std::pair` de la bibliothèque <utility> qui permet de gérer une paire de valeur :

```
#include <utility>

...

std::pair<int,double> values; // Une paire int/double
values.first = 1;
values.second= 2.45;
```

Le type de chacune des deux valeurs est précisé entre les chevrons < et >. Ce type est très utile pour retourner une paire de valeurs de type hétérogène. Prenons par exemple une fonction retournant dans un tableau de réels passé par pointeur l'élément maximal et sa position. La fonction doit bien retourner un `double` (la valeur maximale) et un `std::int64_t` (la position de cet élément). Voici une possibilité pour écrire cette fonction :

```
std::pair<double,int>
trouve_et_localise_valeur_maximale( int n, const double * valeurs)
{
    int index = 0;
    int valeur_max = valeurs[0];
    for ( int i = 1; i < n; ++i )
    {
        if (valeur_max < valeurs[i])
        {
            valeur_max = valeurs[i];
            index    = i;
        }
    }
    return {valeur_max,index};
}
```

L'appel de cette fonction va ressembler à celle renvoyant un tableau statique :

```
std::array tableau = {1., -3., 5., 7., 2., -8., 6.};
auto [valeur_max, position] = trouve_et_localise_valeur_maximale(tableau.size(),
    tableau.data());
std::cout << "Valeur maximale trouvée : " << valeur_max
          << " pour l'index " << position << std::endl;
```

Remarque 1 : On peut voir que pour le retour de la fonction, nous n'avons retourner qu'une liste d'initialisation. En fait, le C++ connaissant par la partie déclarative de la fonction le type de la valeur de retour, va automatiquement convertir la liste d'initialisation dans le type `std::pair<double,int>`. Une erreur de compilation aurait eu lieu si on avait mis trois valeurs au lieu de deux dans la liste d'initialisation.

Remarque 2 : Pour l'instant, on se contente pour la fonction de prendre un pointeur natif et le nombre d'éléments. On verra au fur et à mesure du cours, comment coder cette fonction d'une façon plus élégante et bien plus sûre sans perdre la généralité de la fonction (et en la généralisant plutôt !).

Le seul cas encore non traité est le cas où on doit renvoyer plus de deux valeurs de type hétérogène. Là encore, on peut utiliser un type générique `tuple` proposé dans la bibliothèque `tuple`.

Un tuple en C++ ressemble beaucoup au tuple de python. C'est une collection fixée de valeurs hétérogènes. Par contre, deux tuples ne contenant pas la même liste de type différents ne sont pas considérés de même type en C++. Le tuple est principalement utilisé pour retourner justement plusieurs données hétérogènes. Pour exemple, considérons une fonction approchant un nombre réel $x \geq 0$ par une fraction (continue mais réduite) et retournant l'erreur faite sur le réel. Cette fonction renvoie donc deux entiers p et q (le diviseur et le dividende de la fraction obtenue) et un réel donnant l'erreur d'approximation relative ε faite sur le réel :

$$\varepsilon = \left| \frac{x - \frac{p}{q}}{x} \right|$$

Le tuple étant une type un peu long à écrire, nous allons utiliser une fonction générique `make_tuple` ainsi qu'une déduction automatique du type de retour de la fonction :

```
auto convertion_en_fraction(double valeur, int nombre_iter_max)
{
    std::int64_t dividende, diviseur;
    assert(valeur>=0.);
    std::int64_t partie_entière = std::int64_t(valeur);
    double reste = valeur - partie_entière;
    if (std::abs(reste)<1.E-14)
    {
        return std::make_tuple(partie_entière, 1LL, 0.);
    }
    if (nombre_iter_max == 0)
    {
        dividende = partie_entière;
        diviseur  = 1LL;
        return std::make_tuple(dividende, diviseur, reste);
    }
    auto [p,q,r] = convertion_en_fraction(1./reste, nombre_iter_max-1);
    dividende = partie_entière*p+q;
    diviseur  = p;
    reste = std::abs(valeur-double(dividende)/double(diviseur));
    return std::make_tuple(dividende, diviseur, reste);
}
```

Remarque 1 : La déduction automatique du type de retour existe depuis le C++ 14. Dans le cadre d'un retour de tuple (parmi d'autres cas), cela s'avère très pratique et permet un allègement du code non négligeable sans pour autant nuire à sa lisibilité. Il ne faut cependant pas abuser de cette déduction si le type de la fonction retournée n'est pas "verbeux" à écrire.

Remarque 2 : Si on utilise la déduction automatique du type de retour, il n'est plus possible de laisser le C++ deviner le type de retour en ne renvoyant qu'une simple liste d'initialisation !

L'usage de cette fonction peut ressembler là encore à celui d'une fonction revoyant un tableau statique :

```
auto [dividende,diviseur,erreur] = convertion_en_fraction(std::sqrt(2.), 20);
```

Dans le cadre d'un tuple, si on veut recopier le résultat dans des variables déjà déclarées ou si on veut ignorer certaines valeurs renvoyer, on peut utiliser la fonction `std::tie` :

```
std::int64_t dividende, diviseur;
double erreur;
std::tie(dividende, diviseur, erreur) = convertion_en_fraction(std::sqrt(2.), 20);
std::tie(dividende, diviseur, std::ignore) = convertion_en_fraction(4./3., 20);
```

Le mot clef `std::ignore` permet d'ignorer une des valeurs de retour. Il n'est malheureusement pas utilisable avec la syntaxe avec les crochets [et] où on doit recevoir les trois valeurs.

Il est néanmoins possible d'ignorer par la suite une des valeurs avec cette dernière syntaxe, mais pour éviter d'avoir un message de prévention (warning), il faut utiliser la directive de compilation `[[maybe_unused]]` :

```
[[maybe_unused]] auto [dividende,diviseur,erreur] = convertion_en_fraction(4./3.,
    20);
```

Passage par référence

Nous avons vu dans une section précédente la notion de référence. Il s'avère que cette notion est très utile pour le passage de certains paramètres, à savoir :

— des paramètres dont la valeur doit être modifiée en sortie de la fonction ;

— des paramètres prenant beaucoup de place mémoire ou prenant du temps à être copiés.

Puisqu'une référence crée un alias sur une variable, il est nécessaire de s'assurer que cette variable existe au moins tant que la référence sur cette variable existe. Ainsi, si il est parfaitement possible de retourner une référence à la sortie d'une fonction, il faut **que la variable retournée ne soit pas locale à la fonction sous peine de graves problèmes à l'exécution**.

Ainsi, le code suivant est valide et ne pose aucun problème :

```
#include <iostream>

int& max(int N, int* array)
{   // Retourne une référence sur la valeur la plus grande du tableau
    int imax = 0;
    for ( int i = 1; i < N; ++i) {
        if (array[i]>array[imax])
            imax = i;
    }
    return array[imax];
}

void display_array(int N, const int* array )
{
    std::cout << "[ ";
    for ( int i = 0; i < N; ++i ) std::cout << array[i] << " ";
    std::cout << "]" << std::endl;
}

int main()
{
    const int N = 8;
    int array[N] = {1,2,5,6,3,9,10,8};
    display_array(N, array);

    for ( int n = 0; n < 10; ++n ) {
        int& a = max(N,array);
```

```
        a = a - 1;
        display_array(N, array);
    }
    return EXIT_SUCCESS;
}
```

Listing 2.38 – Exemple complet et valide de retour d'une référence par une fonction

Par contre, le code suivant va avoir un comportement aléatoire pouvant mener presque certainement à un plantage du code :

```
#include <iostream>
#include <cmath>

// La fonction n'est pas valide car on retourne
double& max_root_quadratic_function(double a, double b, double c)
{
  assert(a!= 0);
  double delta = b*b-4*a*c;
  double sol;
  if (delta  < 0) sol = std::nan;
  else if (delta == 0) sol =  -b/(2*a);
  else sol = (-b+std::sqrt(delta))/2;

  return sol; // Erreur, on retourne une variable locale en référence
}

int main()
{
    double& root = max_root_quadratif_function(1,2,-3);
    std::cout << "Racine max de x²+2x-3 : " << root << std::endl;
    return EXIT_SUCCESS;
}
```

Listing 2.39 – Exemple invalide de retour d'une référence par une fonction

En effet, la variable `sol` est locale à la fonction `max_root_quadratic_function` et sera par conséquent détruite à la sortie de la fonction. La référence retournée sera donc une référence sur une variable qui n'existe plus !

Passage par référence

En C, pour éviter une copie d'une variable contenant beaucoup de données (typiquement une grosse structure de donnée), ou pour modifier à la sortie d'une fonction un argument passé à la fonction, il fallait obligatoirement passer par un pointeur :

```
#include <iostream>

void iter_syracuse(int* un)
{
    if ((*un)%2 == 0) (*un) /= 2;
    else (*un) = (3*(*un)+1)/2;
}

int main()
{
    std::cout << "Suite de syracuse : ";
    int n = 1433;
    std::cout << n << " ";
    while (n != 1)
```

```
    {
        iter_syracuse(&n);
        std::cout << n << " ";
    }
    std::cout << std::endl;
    return EXIT_SUCCESS;
}
```

Listing 2.40 – fonction C avec modification des arguments d'entrée

Plusieurs remarques à propos du code C :

1. La manipulation d'un pointeur dans la fonction la rend peu lisibile (confusion possible entre le signe multiplié et le symbole de déférencement) ;
2. Le code n'est pas sûr : que se passe-t'il si le pointeur est nul ? où bien pointe sur une variable qui n'existe pas ?

On peut bien sûr vérifier que le pointeur est non nul à l'aide d'une assertion dans la fonction (valable aussi en C) :

```
#include <iostream>
#include <cassert>

void iter_syracuse(int* un)
{
    // Précondition
    // En mode debug, (-DDEBUG ou -g à la compilation), arrête le programme si la
    condition n'est pas vérifiée
    assert(un != nullptr);
    assert((*un) > 0);

    if ((*un)%2 == 0) (*un) /= 2;
    else (*un) = (3*(*un)+1)/2;
    // Postcondition
    assert((*un) > 0);
}
```

Listing 2.41 – fonction C avec modification des arguments d'entrée

Remarques :

1. À partir du C++ 11, la macro NULL du C a été remplacé avantageusement par un objet `nullptr` possédant son propre type. C'est particulièrement utile lors de l'écriture de fonctions template.
2. Rappelons qu'il est vivement conseillé d'utiliser des assertions dans son code, principalement pour les pré et post-conditions ! Cela permet de gagner un temps fou à la phase de débogage du code.

En C++, on peut remplacer avantageusement le pointeur par une référence pour le paramètre de la fonction :

```
#include <iostream>
#include <cassert

void iter_syracuse(int& un)
{
    // Précondition
    assert(un > 0);

    if (un%2 == 0) un /= 2;
```

```
    else un = (3*un+1)/2;

    // Postcondition
    assert(un > 0);
}

int main()
{
    std::cout << "Suite de syracuse : ";
    int n = 1433;
    std::cout << n << " ";
    while (n != 1)
    {
        iter_syracuse(n);
        std::cout << n << " ";
    }
    std::cout << std::endl;
    return EXIT_SUCCESS;
}
```

Listing 2.42 – fonction C avec modification des arguments d'entrée

Lors de l'appel à la fonction, le paramètre `un` devient un alias du paramètre `n` passé à la fonction. Par contre, il est obligatoire dans le cas d'une simple référence de passer une variable et non directement une valeur à la fonction, puisque `un` doit devenir un alias d'une variable. Ainsi `iter_syracuse(1433)` générerait une erreur du compilateur.

L'usage de la référence dans la fonction a donc permis :

1. D'écrire un code beaucoup plus clair puisqu'on n'utilise plus de symbole de déférencement ;
2. D'avoir un code plus sûr puisque le mécanisme de la référence nous oblige à appeler la fonction avec une variable pré-existante.

Le deuxième usage de la référence, à l'instar des pointeurs, et de pouvoir passer un paramètre dont le type contient beaucoup de données sans effectuer une copie de ce paramètre à l'appel (rappelons que le C passe ses paramètres par valeur). Passer en référence (ou en référence constante comme nous allons le voir ci–dessous), c'est l'équivalent de passer par adresse la variable. Néanmoins, il existe plusieurs situations où on ne désire pas que le paramètre passé par référence puisse être modifié par inadvertance par le programmeur. On utilise dans ce cas là une référence constante :

```
#include <iostream>
#include <cmath>

struct rational
{
    int nom;
    unsigned denom;
};

inline double eval(const rational& r)
{
    return r.nom/double(r.denom);
}

rational approx_rationnal(double x, const rational& pn, const rational& qn, double
    tol = 1.E-6 )
{
    rational median{.nom=pn.nom+qn.nom,.denom=pn.denom+qn.denom};
```

```
    double y = eval(median);
    if (std::abs(x-y) < tol) return median;
    if (y < x) return approx_rationnal(x, median, qn);
    else return approx_rationnal(x, pn, median);
}

int main()
{
    double x = std::acos(-1);

    auto rx = approx_rationnal(x, {.nom=int(x),.denom=1}, {.nom=int(x+1),.denom=1})
    ;
    std::cout << "Approximation de " << x << " par un rationnel -> " << rx.nom << "
    /" << rx.denom << std::endl;

    return EXIT_SUCCESS;
}
```

Listing 2.43 – Exemple d'utilisation de la référence constante

Plusieurs commentaires à faire sur ce code :

1. Les paramètres de type `rational` ont bien été passés par référence afin d'éviter à chaque coup une copie de deux entiers ;
2. La fonction `eval` a été déclarée `inline`. Cela permet d'éviter la surcharge d'un appel de fonction pour une fonction ne faisant presque rien. `inline` permet de remplacer l'appel de la fonction directement par le code machine assemblé pour la fonction.
3. Dans la fonction `main`, nous avons passé deux rationnels en valeur directement dans l'appel de la fonction ce qui est interdit dans le cas d'un passage par référence simple mais permis dans le cas d'un passage par référence constante ! En fait, passer par référence constante assure qu'on ne peut pas modifier ce paramètre au sein de la fonction, et donc la fonction peut donc effectuer un alias en lecture seule avec les deux valeurs mises sur la pile d'appel (qu'on ne peut pas modifier).

Enfin, le retour par une fonction d'un tableau de type `vector` n'effectue une copie du tableau retourné que si la variable de type `vector` retournée n'est pas locale à la fonction, sinon elle effectue un *déplacement*, c'est à dire que la variable recevant le retour de la fonction "vole" le pointeur et la mémoire réservée par la variable retournée (ce qui n'a pas d'incidence puisque cette variable aurait dû être détruite de toute façon… :

```
#include <vector>

// Retourne le plus grand des vecteurs :
vector bad_max(const vector& u, const vector& v)
{
  if (u>v) return u; else return v;
}

// Idem. Notez qu'on retourne une référence, valide ici puisque u et v ne sont pas
    local à cette fonction
const vector& good_max(const vector& u, const vector& v)
{
  if (u>v) return u; else return v;
}


// Retourne l'addition de deux vecteurs
vector add(const vector& u, const vector& v)
```

```
{
  assert(u.size() == v.size());
  vector w(u.size());
  for ( std::size_t i = 0; i < u.size(); ++i )
  {
    w[i] = u[i] + v[i];
  }
  return w;
}

int main()
{
  vector u  = {1., 2., 3., 4., 5.};
  vector v  = {2., 3., 4., 5., 1.};
  // L'appel de bad_max provoque une copie au retour de la fonction car u et v ne
    sont
  // pas locaux à la fonction bad_max.
  auto   w1 = bad_max(u,v);
  // Ici, pas de copie puisqu'on a retourné une référence constante par la foncton
    good_max;
  auto   w2 = good_max(u,v);
  // Et ici pas de copie puisqu'on le vecteur retourné est local à la fonction add
  auto   w3 = add(u,v);
}
```

Listing 2.44 – retour d'une variable de type vector

Comme on voit sur l'exemple ci–dessus, si on ne veut pas avoir une copie effectuée lors du retour d'une variable global d'une fonction, il suffit de retourner une référence (constante ou non, ou les deux en surchargeant la fonction pour les tableaux constants et les tableaux non constants) :

```
// Surcharge de la fonction good_max pour avoir une référence si les deux tableaux
    ne sont pas constants
vector& good_max( vector& u, vector& v )
{
  if (u>v) return u; else return v;
}
```

Il existe bien d'autres méthodes pour les objets de type `vector`, dont vous pouvez trouver la documentation dans les deux sites **cppreference**et **cplusplus.com**. La chose importante à retenir est surtout que la libération de la mémoire se fait de façon automatique dès que l'objet de type vecteur cesse d'être visible.

2.6.2 Surcharge de fonctions

Il arrive souvent, en calcul scientifique en particuliers, de devoir écrire plusieurs versions d'une même fonction pour des types différents. Par exemple, en algébre linéaire, il est utile d'avoir une fonction permettant de faire l'opération $y \leftarrow y + a.x$ où x et y sont des vecteurs d'un certain espace vectoriel $\mathbb{K}^n$ (où $\mathbb{K}$ est un corps tel que les rationnels, les réels, les complexes, $\mathbb{Z}/p.\mathbb{Z}$ (p premier), etc. et a un scalaire appartenant au corps $\mathbb{K}$.

On aimerait pouvoir écrire des fonctions pour cette opérations pour au moins trois types de scalaires différents en C++ : les réels simple précision, les réels double précision, les complexes simple précision et les complexes double précision. Dans un langage tel que le Fortran ou le C, nous serions obligé d'appeler ces quatre fontions par des noms différents (et de fait, dans

la bibliothèque d'algébre linéaire BLAS, les quatre fonctions effectuant cette opération sur les quatre types de scalaires suscités sont nommées saxpy, daxpy, caxpy et zaxpy). En C++, il est possible d'appeler ces quatre fonctions par le même nom, du moment que le type ou le nombre de paramètre est différent. Lors de l'appel de la fonction, le type ou le nombre de paramètres passés permettra sans ambigüité au compilateur de savoir quel fonction appelée.

Ainsi en C++, il est possible d'écrire les deux fonctions ci–dessus :

```
// Fonction avec flottants simple précision
void axpy(int N, float a, const float* x, float* y)
{   // Opération y <- y + a.x sur des vecteurs x,y avec a scalaire
    int i;
    for (i = 0; i < N; ++i ) y[i] += a*x[i];
}

// Fonction avec flottants double précision
void axpy(int N, double a, const double* x, double* y)
{   // Opération y <- y + a.x sur des vecteurs x,y avec a scalaire
    int i;
    for (i = 0; i < N; ++i ) y[i] += a*x[i];
}
```

Listing 2.45 – Exemple de surcharge d'une fonction C++

Ici, le compilateur décidera quelle fonction appeler en fonction du type des paramètres passés, ainsi

```
void main()
{
    float fx[] = {1.f, 2.f, 3., 4.f};
    float fy[] = {0.f, -1.f, -2.f, -3.f};
    axpy(4, 2.f, fx, fy);// Appel la fonction avec les flottants simple précision

    double dx[] = {1., 2., 3., 4.};
    double dy[] = {0., -1., -2., -3.};
    axpy(4, 2., dx, dy); // Appel la fontion avec les flottants double précision

    axpy(4,2.f, dx, dy);// Erreur de compilation, mélange simple et double
    précision !
                        // Le compilateur ne peut décider quel fonction appeler...
}
```

Listing 2.46 – Appel à des fonctions surchargées

Il est important de bien respecter le type des paramètres passés à l'appel de la fonction sous peine d'avoir des erreurs de compilation parfois fort peu compréhensibles !

Nous verrons au chapitre des templates qu'il peut être important d'appeler des fonctions faisant le même traitement mais avec des types différents par le même nom.

2.6.3 Fonction générique (C++ 2020)

Si on observe attentivement les codes écrits dans leurs versions simple ou double précision (mais également en complexe), on se fera vite la remarque que ces fonctions possèdent exactement le même code, au type de variable près !

Une possibilité serait de définir un type en amont de la fonction, qu'on pourrait changer selon le besoin :

```
using scalar_t = double;

...
```

```
void axpy(int N, scalar_t a, const scalar_t* x, scalar_t* y)
{   // Opération y <- y + a.x sur des vecteurs x,y avec a scalaire
    int i;
    for (i = 0; i < N; ++i ) y[i] += a*x[i];
}
```

Tant que le code utilisant cette fonction ne s'en sert que pour un type de scalaire, c'est une solution viable, bien que peu pratique (il faut penser à modifier le type de `scalar_t` pour chaque nouveau programme qu'on veut compiler), et comment faire dès lors qu'une application a besoin de la fonction pour différents types de scalaires ?

C'est là que la notion de fonction générique intervient. Une fonction générique est une sorte de patron (template) de fonction, sur lequel le compilateur va s'appuyer pour générer des fonctions selon les types de paramètres passés à la fonction. Si on passe des réels simple précision, une fonction sera générée pour considérer des réels simples précisions, si des doubles sont employées, une fonction sera générée pour considérer des réels double précisions, etc.

Nous verrons plus loin dans le cours un moyen de faire cela en C++, avec contrôle des types permis, à l'aide des patrons (template) qui sont un concept donnant une grande puissance au langage C++ mais aussi les plus beaux mal de tête du monde ! (On peut faire des choses très (trop ?) complexes avec les templates). Mais depuis C++ 20, il est possible à l'aide d'une écriture bien plus simple d'écrire des fonctions génériques, et cela grâce au mot clef `auto` !

```
// Fonction générique pour tout type de vecteur
void axpy(int N, auto a, const auto x, auto y)
{   // Opération y <- y + a.x sur des vecteurs x,y avec a scalaire
    for (int i = 0; i < N; ++i ) y[i] += a*x[i];
}

void main()

    float fx[] = {1.f, 2.f, 3., 4.f};
    float fy[] = {0.f, -1.f, -2.f, -3.f};
    axpy(4, 2.f, fx, fy);// Appel la fonction avec les flottants simple précision

    double dx[] = {1., 2., 3., 4.};
    double dy[] = {0., -1., -2., -3.};
    axpy(4, 2., dx, dy); // Appel la fonction avec les flottants double précision

    axpy(4,2.f, dx, dy);// Appel la fonction avec a en simple précision, dx et dy
    en double
}
```

Listing 2.47 – utilisation du mot clef auto pour déclarer les paramètres de la fonction

Ici la fonction `axpy` a été généralisée pour prendre n'importe quel type pour `a, x` et `y`. Au fur et à mesure des différents appels à cette fonction "générique", le compilateur produire plusieurs versions de la fonction qui prendra en paramètre les divers types passés pour `a, x` et `y`. Ainsi, dans l'exemple ci-dessus, trois versions de la fonction seront générées : une avec trois réels simple précision, une avec trois réels double précision et la dernière avec `a` réel simple précision et `x` et `y` déclarées comme réels double précision.

Les types (qui peuvent être tous différents) de `a, x` et `y` seront valides tant que l'opération `y[i] += a*x[i]` reste valide pour les types passés pour les trois arguments. Ainsi le programme suivant ne compilera pas :

```
void main()
```

```
{
    double dx[] = {1., 2., 3., 4.};
    double dy[] = {0., -1., -2., -3.};
    axpy(4, "toto", dx, dy); // Ne compile pas ! Pas de sens de faire y[i] += "toto"*x[i]
}
```

Remarque : Si le mot clef `auto` en C++ 20 permet de créer facilement des fonctions génériques, il ne permet pas un contrôle fin des types permis contrairement aux templates que l'on verra plus tard. Néanmoins, dans beaucoup de cas, cette généricité est suffisante et bien plus légère à l'écriture que les templates…

Par contre, si vous avez défini ce que veut dire par exemple la multiplication d'un entier avec une chaîne de caractère (ce qui est possible en C++, on verra cela plus loin lorsqu'on parlera des opérateurs), il est tout à fait légitime d'écrire la ligne suivante :

```
axpy(4, 3, "toto"s, "titi"s);
```

Bien sûr, il est déconseillé de le faire, car cela rajoute plus de confusion au code qu'autre chose...

Valeur par défaut

Si nous pouvons déjà être satisfait de notre fonction "générique", on pourra vous faire néanmoins remarquer que si votre fonction s'applique bien à des vecteurs dont les cœfficients sont contigüs en mémoire, votre fonction ne pourra effectuer son opération dès lors que les cœfficients de votre vecteur sont espacés régulièrement en mémoire. Par exemple, si vous avez une matrice rangée par ligne (l'indice des colonnes varie le plus vite), vous pouvez souhaiter effectuer votre opération `axpy` sur deux colonnes de votre matrice. Seulement, les coefficients sont alors espacés du nombre de colonnes que possède votre matrice ! Pour généraliser votre fonction aux vecteurs lignes ou colonnes des matrices, il suffit donc de rajouter deux paramètres, `incx` et `incy` qui vous donnent le nombre d'éléments à "sauter" pour trouver le prochain cœfficient d'un des deux vecteurs.

Votre fonction devient donc :

```
void axpy(int N, auto a, const auto* x, auto* y, int incx, int incy)
{
    for ( int i = 0; i < N; ++i ) y[i*incy] += a*x[i*incx];
}

int main()
{
    const int N = 4;
    double A[N][N] = { {1,2, 4,  8},
                       {1,3, 9, 27},
                       {1,4,16, 64},
                       {1,5,25,125} };
    // On soustrait quatre fois la colonne 1 à la colonne 3 de la matrice :
    axpy(4, -4., A, A+2, N, N);
    // Rajout de la deuxième colonne à la quatrième ligne :
    axpy(4, 1., A+1, &A[3][0], N, 1);
}
```

Listing 2.48 – Généralisation de la fonction axpy

Pour l'exemple donné ci–dessus, il est clair que les paramètres `incx` et `incy` sont indispensables. Néanmoins, cela alourdit votre code, principalement pour les vecteurs dont les éléments sont contigüs en mémoire, ce qui représente la majorité des cas rencontrés. En effet, il faut à chaque fois pour ce dernier cas, rajouter deux paramètres valant un à la fonction, sans parler du risque de bogue en rajoutant ces deux paramètres (en mettant par erreur deux au lieu de un pour l'un des paramètres par exemple).

Heureusement, C++ prévoit ces cas, et il est possible de fournir pour ces paramètres des valeurs par défaut ce qui permettra de les omettre lors d'un appel "usuel".

Dans le cas de la fonction `axpy`, on vient de voir que dans la majorité des appels à cette fonction, les paramètres `incx` et `incy` valent un. On peut donc définir notre fonction en rajoutant pour ces deux paramètres un pour valeur par défaut :

```
void axpy(int N, auto a, const auto* x, auto* y, int incx = 1, int incy = 1)
{
    for ( int i = 0; i < N; ++i ) y[i*incy] += a*x[i*incx];
}

int main()
{
    const int N = 4;
    double A[N][N] = { {1,2, 4,  8},
                       {1,3, 9, 27},
                       {1,4,16, 64},
                       {1,5,25,125} };
    double x[N] = {1,2,3,4};
    double y[N] = {4,3,2,1};
    // On soustrait quatre fois la colonne 1 à la colonne 3 de la matrice :
    axpy(N, -4., &A[0][0], &A[0][2], N, N); // incx = N, incy = N
    // Rajout de la deuxième colonne à la quatrième ligne :
    axpy(N, 1., &A[0][1], &A[3][0], N);// incx = N, incy = 1
    // Rajout de la deuxième ligne à la quatrième colonne :
    axpy(N, 1., &A[1][0], &A[0][3], 1, N); // incx = 1, incy = N
    // Opération sur les vecteurs x et y :
    axpy(4, 1., x, y); // incx = 1, incy = 1;

    return EXIT_SUCCESS;
}
```

Listing 2.49 – Valeurs par défaut pour la fonction axpy

Remarques :

1. Les paramètres ayant des valeurs par défaut doivent **impérativement** être déclarée en dernier dans les paramètres de la fonction ;
2. L'ordre des paramètres par défaut doit être respecté à l'appel : si un paramètre possédant une valeur par défaut doit être défini avec une valeur spécifique, **tous les paramètres précédents**, même ceux ayant une valeur par défaut, doivent également avoir une valeur spécifique définie par l'utilisateur. Ainsi, dans l'exemple ci–dessus, on ne peut pas définir une valeur différente de un pour `incy` sans définir explicitement la valeur un pour `incx` à l'appel !

2.6.4 Les fonctions inline

Parfois une fonction possède peu d'instructions et est assez rapide pour que l'appel à cette fonction est un temps d'exécution non négligeable par rapport à l'exécution de la fonction elle-même. Il est possible dans ce cas de déclarer la fonction `inline`. Les fonctions `inline` sont des

fonctions dont le code d'exécution sera directement insérer à l'endroit de l'appel de la fonction, évitant ainsi le coût d'un appel. Les fonctions `inline` doivent être déclarées et définies dans le fichier d'entête.

Il faut faire attention cependant à ne pas trop abuser des fonctions `inline` sous peine d'arriver à pénaliser l'exécution du code (alors qu'elles ont pour but de l'optimiser !) en grossisant la taille de code compilé qui peut, dans une boucle par exemple, ne plus pouvoir être contenu dans le cache d'instructions du processeur.

```
inline double sqr(double x)
{
  return x*x;
}
```

Listing 2.50 – Exemple de fonction inline

2.6.5 Les fonctions constexpr

Nous avons vu les variables déclarées constexpr. En fait, il est parfaitement possible de déclarer des fonctions constexpr. Cela permet en particuliers de pouvoir utiliser des fonctions pour évaluer des variables constexpr durant la compilation.

Par exemple, considérons la fonction factorielle. On aimerait pouvoir évaluer des factorielles soit durant l'exécution, soit durant la compilation (quand cela est possible). Nous allons alors écrire une fonction constexpr :

```
constexpr std::int64_t factorielle( std::int64_t n )
{
    if (n == 0) return 1LL;
    return n * factorielle(n-1);
}

int main()
{
    std::cout << "10! = " << factorielle(10) << std::endl << std::flush;
    return EXIT_SUCCESS;
}
```

Listing 2.51 – Exemple de fonction constexpr

Si on compile et on exécute le code, nous aurons sans surprise l'affiche suivant :

```
10! = 3628800
```

Par contre, si on regarde la table des symboles de l'exécutable (à l'aide sous Unix d'une commande tel que `nm -C`), on constatera que la fonction `factorielle` n'existe pas !

Les plus courageux pourront avoir la curiosité de voir l'assembleur généré par le compilateur, et pourront constater que l'appel à `factorielle(10)` a été remplacé directement par la valeur de 10 ! (c'est à dire 3628800).

Si on modifie maintenant légèrement la fonction principale :

```
int main()
{
  std::int64_t n1 = 5LL;
  constexpr std::int64_t n2 = 6LL;
  std::cout << "10! = " << factorielle(10) << std::endl << std::flush;
  std::cout << "5!  = " << factorielle(n1) << std::endl << std::flush;
  std::cout << "6!  = " << factorielle(n2) << std::endl << std::flush;
}
```

Listing 2.52 – Exemple de fonction constexpr utilisé par une variable non const

On constate maintenant dans la table des symboles qu'une fonction `factorielle` existe bien et si on regarde l'assembleur généré, on peut voir que pour 10 ! et 6 !, on n'affiche que la valeur obtenue par la fonction tandis que `x!`, on appelle bien la fonction récursive.

Plusieurs règles doivent être respectées lors de la définition d'une fonction `constexpr` :

— Elle ne peut appeler que des fonctions elles-mêmes `constexpr`. Lorsqu'on veut utiliser une fonction de la librairie standard, il faut donc regarder si la norme précise qu'elle est déclarée `constexpr` ou non ;

— La fonction ne peut retourner que des valeurs qu'on peut construire à la compilation (les types scalaires mais on verra aussi des objets plus complexes sous certaines conditions) ;

— On ne peut pas utiliser des exceptions dans une fonction `constexpr`, ni des `goto`, des variables non `constexpr`, etc.

Les `constexpr` seront essentiels lors du chapître sur les templates !

Voici un exemple un peu plus complexe de fonction constexpr :

```
constexpr std::int64_t isqrt( std::int64_t n )
{
  std::int64_t a = n;
  std::int64_t b = 0;
  std::int64_t c = (a+b)/2;
  while ( (c*c-n != 0) && (a-b>1) )// Tant que c n'est pas la racine carrée de a ou
     proche de la racine
  {
    if (c*c-n<0)// Si c est plus petit que la racine carrée de a
    {
      b = c;
      c = (b+a)/2;
    }
    else
    {
      a = c;
      c = (b+a)/2;
    }
  }
  return c;
}
```

Listing 2.53 – Racine carrée entière en constexpr par dichotomie

Si on utilise cette fonction directement avec des valeurs constantes de n (soit directement des valeurs, soit des valeurs dans des variables `constexpr`), le compilateur évaluera la racine entière de n durant la compilation !

2.6.6 Surcharge des opérateurs

De même qu'il est possible de surcharger des fonctions avec de nouveaux types définis par l'utilisateur, il est également possible de redéfinir les symboles

```
+ += - -= * *= / /= ++ -- () [] ! ~ & | ^ = == < > <= >= << >>
```

et d'autres encore, moins courants. Néanmoins, l'ordre des opérateurs restent celui imposé par le C et le C++, impossible de redéfinir les priorités sur les opérateurs !

Notons de plus un opérateur supplémentaire en C++ 20, l'opérateur "vaisseau spatial" <=> qui permet de regrouper en un seul opérateur tous les opérateurs de comparaison. Pour des soucis d'efficacité, il est néanmoins possible de surcharger cet opérateur par certains des opérateurs standards du C++ qui prendront la priorité sur l'opérateur "vaisseau spatial".

Pour apprendre à manipuler la redéfinition des opérateurs, concevons une petite bibliothèque manipulant des vecteurs dans $\mathbb{R}^3$, avec laquelle on pourra :

- Additionner et soustraire
- Effectuer un produit scalaire ou vectoriel
- Effectuer une homothétie
- Afficher le vecteur
- Pouvoir lire une composante du vecteur de deux manières :
  - Par composante : `double x = u.x;` par exemple
  - Sous la manière d'un tableau : `double x = u[0];` par exemple

On verra ainsi au fur et à mesure comment redéfinir certains opérateurs. Commençons par définir notre vecteur et la façon d'accéder aux composantes du vecteur :

```
struct vecteur
{
  double x, y, z;
};

int main()
{
    vecteur u{.x = 3, .y = 5, .z = -1};
    vecteur v{.x = 5, .y = 1, .z =  7};
    std::cout << "vecteur u : " << u.x << ", " << u.y << ", " << u.z << std::endl;

    return EXIT_SUCCESS;
}
```

Listing 2.54 – définition d'un vecteur en trois dimensions

On peut donc déjà avec cette simple définition définir des vecteurs et accéder à leurs composantes. Pour accéder à la manière d'un tableau aux composantes des vecteurs, nous allons utiliser l'opérateur `[]`, un des rares opérateurs qui doit être impérativement être défini à l'intérieur de la structure avec l'opérateur `()` (opérateur d'évaluation) :

```
struct vecteur
{
    double x, y, z;

    // Le const à la fin de la ligne ci-dessous signifie que cet opérateur
    marchera pour
    // un vecteur constant (ce qui n'est pas le cas si on ne met pas const à la fin
    )
    double operator [] ( int i ) const
    {
        assert(i>=0);
        assert(i <3);
        if (i==0) return x;
        if (i==1) return y;
        assert(i==2);// Normalement toujours vrai !
        return z;
    }

    // Remarque : le C++ sait distingué les deux car celui au dessus est défini pour
    // un vecteur constant et celui ci-dessous pour un vecteur non constant.
    // Notez que dans le cas non constant, on renvoie une référence sur l'élément
    du
    // vecteur afin qu'il puisse être modifiable (par exemple en écrivant u[0] =
    3.; )
    double& operator [] ( int i )
    {
```

```
        assert(i>=0);
        assert(i <3);
        if (i==0) return x;
        if (i==1) return y;
        assert(i==2);// Normalement toujours vrai !
        return z;
    }
};

int main()
{
    vecteur u{.x = 3, .y = 5, .z = -1};
    vecteur v{.x = 5, .y = 1, .z =  7};
    std::cout << "vecteur u : " << u.x << ", " << u.y << ", " << u.z << std::endl;
    std::cout << "vecteur v : ";
    for (int i = 0; i < 3; ++i ) std::cout << v[i] << " ";
    std::cout << std::endl;

    return EXIT_SUCCESS;
}
```

Listing 2.55 – définition d'un vecteur en trois dimensions avec opérateur d'accés

Rajoutons maintenant l'addition de deux vecteurs à l'aide du symbole + et la soustraction de deux vecteurs à l'aide du symbole – :

```
vecteur operator + ( const vecteur& u, const vecteur& v )
{
  return vecteur{.x = u.x + v.x,
                 .y = u.y + v.y,
                 .z = u.z + v.z };
}

vecteur operator - ( const vecteur& u, const vecteur& v )
{
  return vecteur{.x = u.x - v.x,
                 .y = u.y - v.y,
                 .z = u.z - v.z };
}

int main()
{
  ...
  vecteur w;
  w = u + v;
  vecteur t;
  t = u - v;
}
```

Remarquons que le premier paramètre représente le vecteur à gauche du symbole + ou du symbole –, et le second celui à droite. Lorsqu'on écrit dans le main une ligne comme `w = u + v;`, cette ligne est directement traduite par le C++ en `w = operator + (u,v);`.

De même, il est aussi facile de définir le produit scalaire à l'aide du symbole |, le produit vectoriel à l'aide du symbole ^ et l'homothétie à l'aide du symbole * :

```
double operator | ( const vecteur& u, const vecteur& v )
{
  return u.x*v.x + u.y*v.y + u.z*v.z;
}
```

```
vecteur operator ^ ( const vecteur& u, const vecteur& v )
{
  vecteur w;
  w.x = u.y*v.z - u.z*v.y;
  w.y = u.z*v.x - u.x*v.z;
  w.z = y.x*v.y - u.y*v.x;
  return w;
}

vecteur operator * ( double alpha, const vecteur& u )
{
  return vecteur{ .x = alpha * u.x,
                  .y = alpha * u.y,
                  .z = alpha * u.z };
}

int main()
{
  ...
  double scal = (u|v);
  vecteur orth = u^v;
  vecteur double_u = 2. * u;
  ...
}
```

Il ne nous reste plus qu'à afficher le vecteur de la même manière qu'on affichera un entier, un double, etc.

```
std::ostream& operator << ( std::ostream& out, const vecteur& u )
{
  out << "< " << u.x << ", " << u.y << ", " << u.z << " >";
  return out;
}

int main()
{
  ...
  std::cout << "Le vecteur u vaut : " << u << std::endl;
  ...
}
```

Le symbole << est le symbole utilisé en C++ pour décrire un flux vers un périphérique de sortie (on aura le même type d'écriture si on dirige le flux vers un fichier, une imprimante, etc.). Ce symbol utilisé comme opérateur de flux (il peut également servir à d'autre choses), prend à gauche de l'opérateur une sortie (fichier, console, etc.) qui sera en C++ de type `std::ostream` et qui sera modifié (en avançant dans le fichier par exemple), et à droite par le type d'objet qu'on veut utiliser. Le fait de renvoyer en retour de la fonction la sortie `out` permet de chaîner les opérateurs de flux.

En effet, si on décompose la ligne `std::cout << "u vaut : " << u << std::endl;`, cette ligne est traduite comme suit par le C++ :

```
  operator <<(operator <<(operator <<(std::cout, "u vaut"),u      ), std::endl);
//                |          |         std::ostream&,char*  |           |
//                |       std::ostream&                    ,vecteur     |
//             std::ostream&                                        ,   char
```

ce qui montre que retourner une variable de type `std::ostream&` est nécessaire puisque l'opérateur attend à gauche un objet de ce type.

Vous trouverez un exemple complet de ce qu'on vient d'exposer (avec un ou deux opérateur en plus) dans `exemple_vecteur_3d_struct.cpp`.

À partir du C++ 11 :

Il est autorisé de définir des structures, des classes ou des unions dans un corps de fonction. La visibilité de la structure sera celle du bloc dans laquelle elle a été définie.

```
void f()
{
    struct triplet { double x,y,z; };

    triplet t{.x=2,.y=4,.z=5};// initialisation structure C++ 20 ou supérieur

}
```

Listing 2.56 – déclaration d'une structure au sein d'une fonction (C++ 11 ou supérieur)

Le cas particuliers de l'opérateur "spaceship"

Cet opérateur a été introduit à partir du C++ 20. Il permet d'éviter de devoir définir tous les opérateurs de comparaisons (ormi l'opérateur $==$) et les regrouper en un seul opérateur en y rajoutant quelques subtilités...

Ainsi, soit les éléments à comparer sont fortement ordonnés, c'est à dire que deux valeurs identiques sont exactement les mêmes, soit les éléments à comparer sont faiblement ordonnés, c'est à dire que deux valeurs "identiques" appartiennent à la même classe d'équivalence (au sens mathématiques du terme), soit les éléments sont partiellements ordonnées (certains éléments ne sont pas comparables entre eux).

Par exemple, supposons qu'on ait défini une structure `rationnel` contenant deux entiers représentant respectivement le numérateur et le dénominateur (qui doivent être premiers entre eux) :

```
struct rationnel
{
  std::int64_t numérateur, dénominateur;
}
```

Listing 2.57 – Structure définissant un rationnel

On veut, entre autre, pouvoir définir les opérateurs de comparaison pour les rationnels. On définit alors les opérateurs $==$ et "spaceship"(avec une ordonnancement fort) :

```
bool operator == ( const rationnel& p, const rationnel& q )
{
  if ( (p.numérateur == q.numérateur) && (p.dénominateur == q.dénominateur) )
      return true;
  return false;
}

decltype(1LL<=>2LL) operator <=> ( const rationnel& p, const rationnel& q )
{
  if ( (p.numérateur == q.numérateur) && (p.dénominateur == q.dénominateur) )
    return std::strong_ordering::equal;
```

```
  if (p.numérateur * q.dénominateur <  q.numérateur * p.dénominateur)
    return std::strong_ordering::less;
  return std::strong_ordering::greater;
}
```

Listing 2.58 – Définition des opérateurs de comparaison

Remarquez que le type de valeur renvoyer par l'opérateur <=> n'étant pas très clair, on a utilisé ici une déclaration de type explicite à l'aide de `decltype` !

Avec ces deux opérateurs, on a défini les opérateurs <, >, <=, >=, != et ==.

On verra lors du chapître sur la programmation objet, qu'il est possible en fait de demander à ce que l'opérateur == soit défini par rapport à l'opérateur "spaceship".

2.7 Entrées/sorties

En C (mais aussi en C++), l'affichage sur un terminal d'un texte se fait à l'aide de la fonction `printf` de la bibliothèque `stdio.h` (`cstdio` en C++). Cependant, l'utilisation de cette fonction est loin d'être facile et peut être génératrice de bogues :

```
#include <cstdio> // En C++, mais on peut aussi inclure stdio.h, ça marche aussi
                  // Remarque : pas d'extension pour l'inclusion des fichiers en C
    ++ !
int main()
{
    int n1 = 3;
    double x = 3.14;
    // Inversion de %d et %lg => bogue attendu et affichage bizarre
    printf("x = %d, n1 = %lg\n",x, n1);
    return 1;
}
```

En C++, il existe une bibliothèque d'entrée sortie simplifiant grandement l'affichage sur terminal (mais aussi sur fichier) sans se préoccuper du type de valeurs à afficher :

```
#include <iostream> // Bibliothèque de gestion d'entrée/sortie du C++

int main()
{
    int n1 = 3;
    double x = 3.14;
    // std::cout permet d'afficher sur un terminal la sortie standard
    // std::endl permet un retour à la ligne précédé par un flush pour forcer l'
    affichage
    std::cout << "Il est facile d'afficher du texte, mais aussi des variables comme
     x = " << x
              << " ou encore n1 = " << n1 << std::endl;
}
```

On peut également formater les entrées sorties pour une précision donnée, un nombre de caractère fixe, etc.

Voici un exemple exhaustif des possibilités de formatage :

```
#include <iostream>
#include <iomanip> // Pour jouer avec le format des nombres
```

```cpp
int main()
{
    double x = 3.141516;
    int    n = 63123;

    std::cout << n << " en hexadecimal donne " << std::hex << n << std::dec << std
    ::endl;
    for ( int i = 5; i < 10; ++i )
        std::cout << std::setw(i) << n << std::endl;// Modif. nbre de caractère
    pour afficher n
    for ( int i = 5; i < 10; ++i )
        std::cout << std::setfill('0') << std::setw(i) << n << std::endl; // Idem
    mais remplissage par des zéros

    std::cout << std::setprecision(5) << x << std::endl;// Affichage de x avec une
    précision donnée
    std::cout << std::setprecision(9) << x << std::endl;
    std::cout << std::fixed;// Impose que le nombre de chiffre après la virgule
    soit fixe
    std::cout << std::setprecision(5) << x << std::endl;
    std::cout << std::setprecision(9) << x << std::endl;

    return 1;
}
```

Si on compile le programme, on trouve en sortie :

```
63123 en hexadecimal donne f693
6312
 63123
  63123
   63123
    63123
63123
063123
0063123
00063123
000063123
3.1415
3.141516
3.14152
3.141516000
```

Enfin il est possible d'écrire littéralement un booléen plutôt que d'écrire sa représentation entière (1 ou 0) :

Le programme suivant :

```cpp
int number = 5;
bool is_even = (number % 2 == 0);
bool is_odd  = (number % 2 == 1);
std::cout << "Affichage standard des booleens : " << std::endl;
std::cout << number << " est pair   ? " << is_even << std::endl;
std::cout << number << " est impair ? " << is_odd  << std::endl;
std::cout << "Affichage alphanumerique des booleens : " << std::endl;
std::cout << number << " est pair   ? " << std::boolalpha << is_even << std::endl;
std::cout << number << " est impair ? " << std::boolalpha << is_odd  << std::endl;
```

affichera :

```
Affichage standard des booleens :
5 est pair   ? 0
5 est impair ? 1
Affichage alphanumerique des booleens :
5 est pair   ? false
5 est impair ? true
```

Chapitre 3

Programmation objet avec le C++

Tout d'abord, il ne faut pas confondre programmation objet avec langage orienté objet.

La programmation objet est un "style" de programmation, une façon de penser les choses et des les programmer.

Un langage orienté objet est un langage permettant certaines facilités pour faire de la programmation objet.

Il est cependant tout à fait possible de faire de la programmation objet dans un langage qui n'est pas orienté objet. Ainsi, il est tout à fait possible de faire de la programmation objet en C. Un exemple de bibliothèque objet écrite en C est l'interface graphique X11.

La programmation objet se prête bien à certaines situations, beaucoup moins dans d'autres. Il ne faut donc pas chercher à tout prix à programmer objet, sous peine d'avoir des lourdeurs ou encore un code qui s'avère peu efficace. Il est remarquable d'ailleurs de noter que la grande majorité de la bibliothèque standard du C++ n'est pas programmer en objet (très peu d'héritage, etc.) pour des raisons d'efficacité principalement.

Il ne faut pas non plus croire que le C++ est un langage principalement orienté objet. Si la programmation orienté objet fait parti du C++ depuis le début, cette partie du langage a très peu évolué depuis contrairement à d'autres aspects du langage.

3.1 C'est quoi, la programmation objet

La programmation objet (ou orientée objet) est un paradigme de programmation visant à définir des briques logicielles nommées *objet* et leurs interactions : un objet est une entité représentant un concept, une idée, la modélisation d'une entité du monde physique comme une voiture, une personne, un moteur, une pédale, etc. Un objet possède une structure interne et un comportement, et il est possible qu'il puisse interagir avec ses pairs.

L'interaction entre les objets via leurs relations permet de concevoir et de réaliser des fonctionnalités attendues, afin de mieux résoudre les problèmes visés. L'étape de modélisation est une étape très importante de la programmation orientée objet.

Le premier langage a avoir introduit les bases, encores utilisées aujourd'hui, de la programmation orientée objet est le langage *simula* paru en 1967. Mais c'est définitivement *smalltalk* en 1971 puis en 1980 qui fixe les principes de la programmation orienté objet.

A noter qu'un "vrai" langage orienté objet ne peut être qu'interprété, ce qui est le cas de simula ou smalltalk. Les langages compilés tels que C++ ont forcément des limitations quant à la modélisation et la programmation objet (en particuliers l'introspection en cours d'exécution par exemple).

3.1.1 Définition d'un objet

Un objet est une entité qui contient des données et qui propose une interface permettant de manipuler ces données (sous forme de fonctions par exemple).

La programmation orientée objet possède son propre vocabulaire. Ainsi,

— Les données contenues dans un objet sont appelées *les attributs* de l'objet ;

— Les fonctions associées au traitement des données de l'objet sont appelées *l'interface* ou encore les *méthodes de l'objet*.

Prenons par exemple un objet modélisant un nuage de points dans le plan. Les coordonnées des points sont représentées par des réels X et Y.

Il existe de nombreuses façons de représenter un nuage des points en mémoire :

1. Les coordonnées des points sont dans deux tableaux distincts : un tableau X contenant les composantes x_i des points P_i et un tableau y contenant les composantes y_i des points P_i ;
2. Les coordonnées des points sont stockées par couple (x_i, y_i) dans un seul tableau
3. On stocke les points dans un quadtree qui permet de partitionner itérativement les points dans le plan
4. etc.

Imaginons que nous souhaitons mettre en œuvre les services suivants :

— Récupérer les abcisses des points du nuage ;

— Récupérer les ordonnées des points du nuage ;

— Lire le nombre de points contenus par le nuage ;

— Trouver tous les points se trouvant en dessous d'un point P donné.

Avec la première représentation, la mise en œuvre des services pourrait ressembler à ce qui suit :

```
// Représentation du nuage de points à l'aide de deux tableaux X et Y
using nuage_de_points = std::pair<std::vector<double>, std::vector<double>>;
// Fonction pour créer un nuage_de_points :
nuage_de_points créer_nuage(int N)
{
   return std::make_pair(std::vector<double>{N},std::vector<double>{N});
}
// Récupération dans un tableau des abcisses des points :
std::vector<double> lire_abcisses( const nuage_de_points& nuage )
{
    return nuage.first;
}
// Récupération dans un tableau des ordonnées des points :
std::vector<double> lire_ordonnées( const nuage_de_points& nuage )
{
    return nuage.second;
}
// Nombre de points contenu par le nuage :
std::size_t nuage_size( const nuage_de_points& nuage )
{
    return nuage.first.size();
}
// Fonction retournant les points en dessous d'une certaine ordonnéee ainsi que
    leurs indices dans le nuage
auto yLowerThan( const nuage_de_points& crds, double yval )
{
    coordinates c;
    std::vector<int> r;
```

```
    for ( std::size_t i = 0; i < crds.first.size(); ++i)
    {
        if (crds.second[i] < val )
        {
            c.first.push_back(crds.first[i]);
            c.second.push_back(crds.second[i]);
            r.push_back(i);
        }
    }
    return std::make_tuple(c,r);
}
```

Listing 3.1 – Réalisation des services avec la première représentation des données pour le nuage de points

Pour la deuxième réalisation, une mise en œuvre possible est :

```
// Réalisation du nuage de point par un tableau contenant des pairs de double :
using nuage_de_points=std::vector<std::pair<double,double>>;
// Fonction pour créer un nuage_de_points :
nuage_de_points créer_nuage(int N)
{
   return std::vector<std::pair<double,double>>{N};
}
// Récupération dans un tableau des abcisses des points :
std::vector<double> lire_abcisses( const nuage_de_points& nuage )
{
    std::vector<double> x;
    x.reserve(100);
    for ( const auto& c : coords ) x.push_back(c.first);
    return x;
}
// Récupération dans un tableau des ordonnées des points :
std::vector<double> lire_ordonnées( const nuage_de_points& nuage )
{
    std::vector<double> y;
    y.reserve(100);
    for ( const auto& c : coords ) y.push_back(c.second);
    return y;
}
// Nombre de points contenu par le nuage :
std::size_t nuage_size( const nuage_de_points& nuage )
{
    return nuage.size();
}
// Fonction retournant les points en dessous d'une certaine ordonnéee ainsi que
    leurs indices dans le nuage
auto yLowerThan( const nuage_de_points& crds, double yval )
{
    coordinates c;
    std::vector<int> r;
    for ( int i = 0; i < crds.size(); ++i)
    {
        if (crds[i].second < val )
        {
            c.push_back(crds[i]);
            r.push_back(i);
        }
    }
    return std::make_tuple(c,r);
}
```

Listing 3.2 – Réalisation des services avec la seconde représentation des données pour le nuage de points

On peut constater que la mise en œuvre des divers services est très différente entre la première et la deuxième réalisation. Les algorithmes choisis sont guidés par la représentation des données en mémoire. Par contre, l'interface permettant d'accèder aux services est strictement la même !

L'idée principale de la programmation orientée objet est que l'interface permettant de manipuler des données au cours du développement d'un logiciel est immuable (on sait ce que l'on veut faire !) mais que la représentation mémoire de ces données peut varier au cours du développement (on ne sait pas toujours comment faire !).

Lorsqu'on développe un logiciel, on suit un *cahier des charges*. Un cahier des charges est un document décrivant ce que doit faire le logiciel. Il peut se présenter sous diverses formes : un manuel d'utilisation du logiciel (qui n'est pas encore écrit !), un document décrivant les diverses fonctionnalités d'une bibliothèque donnée, etc.

À partir de ce cahier des charges, le programmeur va concevoir une interface répondant aux besoins exprimés par le cahier des charges, c'est la phase d'analyse/conception. En pratique, durant cette phase, il y a de nombreux aller retours entre le cahier des charges et la phase d'analyse, les besoins exprimés dans le cahier des charges étant souvent au mieux incomplets, voire parfois contradictoires.

Pour concevoir l'interface, diverses méthodes ont été proposées. Celle actuellement la plus répandue est la méthode TDD (Test Driven Development) qui consiste à d'abord écrire des maquettes servant principalement aux tests (sous forme de fonction principale ou d'interface graphique) afin de trouver l'interface la plus naturelle et simple possible. Cette méthode apporte beaucoup d'avantages :

- Elle permet d'éclaircir certains points du cahier des charges ;
- Trouver des points contradictoires dans le cahier des charges ;
- Se concentrer uniquement sur l'interface, point crucial de votre logiciel, pour obtenir une interface claire et naturelle ;
- Obliger d'écrire une batterie de tests qui par la suite permettront de s'assurer du bon fonctionnement du logiciel en suivant les divers scénarii possibles.

Un cas concret

Dans l'optique d'une utilisation multiple dans divers logiciels, on vous demande de créer un module permettant de :

- Créer un polynôme de degré n passant par $n+1$ points (x_i, y_i) ;
- Créer un polynôme de degré n passant par $n-1$ points (x_i, y_i) et dont la dérivée est nulle aux points (x_0, y_0) et (x_n, y_n) ;
- Construire une base orthonormale de polynôme dans l'espace des polynômes $\mathscr{R}^n[X]$ de degré n muni du produit scalaire

$$< P|Q >= \int_a^b P(x)Q(x)\omega(x)dx$$

  où $\omega(x)$ est une fonction poids de l'intégrale (éventuellement la fonction constante 1) ;
- Évaluer un polynôme en un point x ;
- Additionner, soustraire, multiplier un polynôme ;
- EFfectuer la division euclidienne d'un polynôme (quotient + reste) ;
- Calculer le pgcd et le ppmc de deux polynômes ;

— Calculer la dérivée ou la primitive d'un polynôme ;
— Afficher un polynôme de façon lisible par un humain ;
— Charger ou sauvegarder un polynôme dans un fichier.

La première chose à faire est donc d'écrire une programme testant les diverses fonctionnalités demandées (même si on ne peut rien tester pour l'instant vu qu'on n'a encore rien écrit !).

Ce n'est pas préciser explicitement dans le cahier des charges, mais visiblement, les coefficients des monômes constituant le polynôme sont réels (sinon on aurait du mal à trouver un polynôme interpolant n points).

```cpp
#include "polynome.hpp"
using namespace algebra;

int main()
{
    std::vector points{ std::pair{-1., 2.}, {0., 1.}, {1., 2.} }; // Points d'interpolation définissant P(x)=x²+1 pour lagrange
                                                                   // et -x^{4}+2.x²+1 pour hermite

    polynome lagrange(points); // Par défaut, interpolation de lagrange
    polynome hermite (points, polynome::interpolation::hermite); // Sinon, on précise

    std::cout << "interpolation de Lagrange : " << std::string(lagrange) << std::endl;
    std::cout << "interpolation d'hermite   : " << std::string(hermite)  << std::endl;

    // Polynômes de Legendre jusqu'au degré cinq :
    auto legendre = polynome::create_orthonormalized_base(5, {-1.,1.}, [](double x) { return 1. });
    std::cout << "Base orthonormée de Legendre : " << std::endl;
    for (auto p : legendre)
        std::cout << "\t" << std::string(p) << std::endl;

    // Polynômes de Tchebychev jusqu'au degré cinq :
    auto tchebychev = polynome::create_orthonomalized_base(5, {-1.,1.}, [](double x) { return 1./std::sqrt(1-x*x); });
    std::cout << "Base orthonormée de Tchebychev : " << std::endl;
    for ( auto p : tchebychev)
        std::cout << "\t" << std::string(p) << std::endl;

    std::cout << "Degré du polynôme de Lagrange : " << lagrange.degre() << std::endl;
    std::cout << "Nombre de monômes : " << lagrange.size() << std::endl;
    std::cout << "polynôme de lagrange. Evaluation en 2. : " << lagrange(2.) << std::endl;

    auto pol1 = lagrange + hermite;
    auto pol2 = lagrange - hermite;
    auto pol3 = lagrange * hermite;
    auto [quotient, reste] = hermite/lagrange;
    auto plus_grand_commun_diviseur = pgcd(hermite, lagrange);
    auto plus_petit_commun_multiple = ppcm(hermite, lagrange);

    auto dP = lagrange.derivate();
    auto pP = lagrange.primitive();// Primitive à une constante près

    // Pour la gestion des fichiers, il serait mieux sûrement de prévoir un service
```

```
    permettant
    // de sauvegarder une collection de polynômes (ici une base orthonormée)
    std::ofstream fichier_sauvegarde("Tchebychev.dat", "w");
    fichier_sauvegarde << tchebychev.size() << std::endl;
    for ( auto p : tchebychev )
        fichier_sauvegarde << p << std::endl;
    fichier_sauvegarde.close();

    std::ifstream fichier_chargement("Tchevychev.dat", "r");
    std::size_t nombre_de_polynomes;
    fichier_chargement >> nombre_de_polynomes;
    std::vector<polynome> base; base.reserve(nombre_de_polynomes);
    for (int i = 0; i < nombre_de_polynomes; ++i )
        base.emplace_back(fichier_chargement); // Lit les polynômes un par un...

    return EXIT_SUCCESS;
}
```

L'exemple ci-dessus n'est bien sûr qu'une façon parmi d'autres de faire l'interface. Cependant, on peut déjà faire quelques remarques :

- Faire une interface demande une bonne maitrise du langage de programmation associé. Il faut en effet déjà avoir une bonne idée de ce que le langage permet de faire et ne permet pas de faire...
- Certains choix dans l'interface ci-dessus sont contestables. Par exemple, utilisé l'opérateur `/` pour faire la division euclidienne de deux polynômes (donnant le quotient et le reste) peut être vu comme trompeur ou mal pensé pour certains. En fait, chaque individu a sa propre vision de ce que peut être une bonne interface, bien qu'on retrouve en général beaucoup de points commun. Ce à quoi il faut arriver, c'est arriver à un *consensus*, c'est à dire à une interface qui ne choque personne dans l'équipe de développement ou le client et à laquelle tout le monde adhère (même si certains penseront toujours que leur vision de l'interface est la bonne...) ;
- Au cours de la mise au point de l'interface, il arrive souvent de trouver de nouvelles fonctionnalités à rajouter pour que l'interface puisse fonctionner et soit simple à utiliser ;
- Quelques règles néanmoins doivent être respectées lors de l'élaboration de l'interface :
  - **La règle de moindre surprise** : Lorsqu'on utilise un langage, il est généralement fourni avec une bibliothèque standard qui propose une interface bien spécifique. Par exemple, lorsqu'on utilise un objet qui représente une collection de valeur (liste, tableau dynamique ou statique, ensemble, etc.), on utilise la méthode `size` pour connaître le nombre d'éléments contenus dans la collection. Dès lors, si notre objet (ici un polynôme) définit une méthode donnant le nombre d'éléments (ici des monômes) contenus dans cet objet, pourquoi appeler la méthode correspondante autrement que `size` ? Avoir une unification des noms pour les méthodes (ou autre) apportent plusieurs avantages :
    1. Le lecteur éventuel de votre code sera immédiatement ce que fait votre méthode ;
    2. Dans le cas d'une fonction générique, cela permet d'avoir des objets dont les interfaces sont compatibles entre elles pour pouvoir être utilisées avec cette fonction.

    Remarquez que pour obtenir le degré du polynôme, il ne fallait surtout pas appeler cette méthode `size` puisque le degré est le nombre de monômes contenus par le polynôme auquel on retranche la valeur un !
  - Essayez d'être expressif quant aux choix des noms des fonctions, types et méthodes.