

Langage C modernisé

Cours – HAE303E – Licence 2 EEA / 2024

Kada KRIBICH

kada.kribich@umontpellier.fr

Mikhaël MYARA

mikhael.myara@umontpellier.fr



“La plupart des problèmes d'informatique proviennent de l'interface chaise-clavier.”

Klaus Klages

Langage C moderne - Cours - HAE303E - Licence 2 EEA/2024

6 septembre 2024

Programmation en Langage C "modernisé" : Langage C de base avec des éléments empruntés au C++.
par Kada Kribich et Mikhaël Myara, pour le Department EEA, Université de Montpellier, France



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0
Unported License.*

Préface

On propose ici un ensemble de solutions pour vous permettre de pratiquer le C/C++ chez vous. En effet, l'informatique, c'est un peu de réflexion et beaucoup de pratique. Pour s'entraîner il faut pratiquer sur ordinateur, c'est indispensable. Nous proposons 3 possibilités pour travailler chez vous :

1. Une première possibilité 100% en ligne, avec un simple navigateur web, via <http://repl.it>. C'est une bonne solution si on ne veut pas configurer et installer des logiciels. Mais elle s'éloigne un peu des conditions de travail à l'université (même si le langage C/C++ est exactement identique, bien entendu).
2. Une deuxième possibilité consiste à vous connecter sur les ordinateurs de l'université en utilisant **x2go**. C'est une bonne solution pour être chez soi dans les conditions exactes de travail de l'université, qui correspondent donc aussi aux conditions des contrôles et examens. Mais cela suppose une connexion internet de qualité, et malgré cela vous pouvez remarquer des ralentissements certains jours si les serveurs de l'université sont très sollicités. Et certains TP (ceux qui utilisent la bibliothèque "jeu vidéo" notamment) risquent de ne pas pouvoir se faire par ce moyen.
3. Une troisième solution est dédiée aux ordinateurs sous Windows, elle consiste à installer un ensemble d'outils sur votre ordinateur. C'est une bonne solution qui vous plonge dans un environnement de travail vraiment très proche de l'université, et ne dépend plus d'internet une fois l'installation réalisée. Elle est déclinable sous Linux et Mac, mais n'est pas décrite pour ces machines. Prenez contact avec vos enseignants si vous avez besoin d'aide avec ces plateformes.

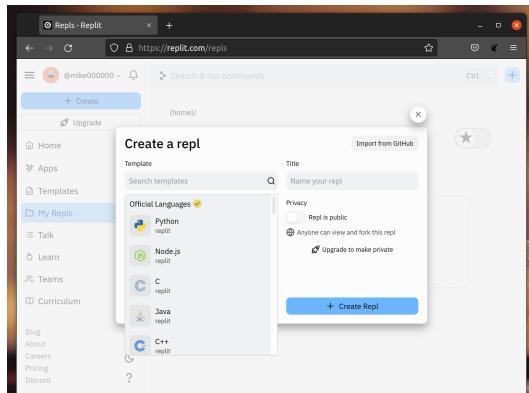
Solution 1 - <http://repl.it>:

C'est un site web qui donne accès en ligne à beaucoup de langages informatiques, notamment le langage du terminal Linux (**bash**), le langage C/C++, mais aussi beaucoup d'autres.

Note importante : Même si le site peut s'utiliser pour programmer en C/C++ sans créer de compte, cette utilisation est limitée et ne donne pas accès à tout. Ainsi, il est **indispensable** de créer un compte sur ce site. Pour créer ce compte, allez simplement sur <http://repl.it> :



Puis allez sur **sign in**, indiquez vos coordonnées et validez le compte en surveillant vos emails. Une fois le compte créé, connectez-vous en retournant sur <http://repl.it>. Vous aboutissez à la fenêtre ci-dessous :



Pour ouvrir le dialogue "Create a repl", on a cliqué sur "+ Create" que vous voyez en arrière-plan, en haut à gauche de la fenêtre. Maintenant, il vous reste à sélectionner "C/C++" et à donner un nom au programme que l'on va créer, puis valider. Et c'est tout. Vous aurez à répéter cette procédure pour chaque nouveau programme.

Solution 2 - X2GO

X2GO est un logiciel qui vous permet de vous connecter aux ordinateurs de l'université de façon sécurisée. Ce faisant, il vous plonge exactement dans l'environnement que vous utilisez en travaux pratiques, qui correspond donc aussi à celui qui sera utilisé pour les contrôles et examens. C'est un très bon outil de travail global pour vous à l'université, le seul défaut de cette solution est qu'elle suppose une bonne connexion internet. Aussi, les serveurs de l'université ne doivent pas être trop sollicités au moment où vous vous connectez : il arrive qu'ils soient beaucoup, pas forcément souvent, mais cela tombe forcément mal.

La procédure d'installation est disponible pour Windows, Linux et Apple/MacOsX sur le Moodle de l'université :

<https://moodle.umontpellier.fr/mod/page/view.php?id=126345>

Notez que l'article porte bizarrement le nom "Ubuntu 20.04" mais si vous descendez dans la page vous verrez la section **Accès à l'environnement graphique via le client X2go** qui contient toute la partie liée à Windows et Apple/MacOsX également. Une fois l'installation terminée, vous devriez aboutir à votre bureau sur les machines de l'université :



Solution 3 - Installation de chocolatey, MSYS2 et Eclipse

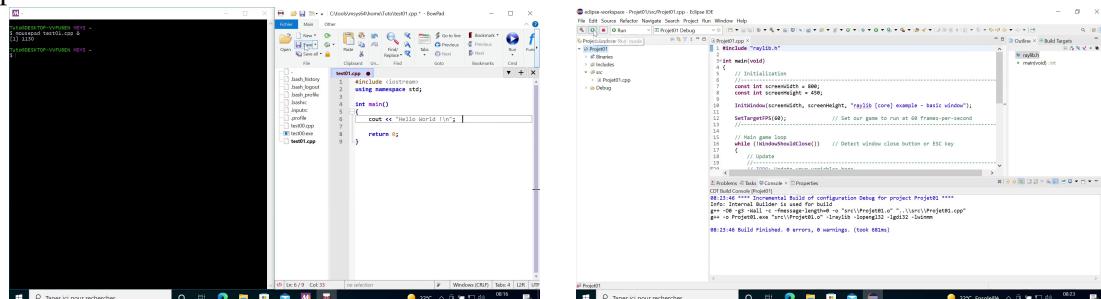
La procédure ci-dessous est dédiée à Windows. Nous avons identifié un ensemble d'outils qui vous permettent malgré tout de travailler chez vous avec des outils adaptés, de façon très similaire à ce que l'on fait sous Linux à l'université. Il y a plusieurs étapes, elles sont toutes décrites dans le tutoriel vidéo :

<http://dl.eea-fds.umontpellier.fr/CppInstall/tuto-install-Cpp.mp4>

Pour suivre ce tutoriel vous aurez besoin d'un package à télécharger :

<http://dl.eea-fds.umontpellier.fr/CppInstall/package.zip>

Vous aboutirez à plusieurs environnements de travail, en pluttot en mode terminal, l'autre orienté Eclipse :



Une fois l'installation réussie, suivez ce tutoriel pour utiliser le terminal, C/C++, un éditeur (Bowpad), un IDE (Eclipse) et la bibliothèque Raylib, c'est-à-dire tous les outils nécessaires pour vous entraîner chez vous avec les outils que nous utiliserons lors de ces TP :

<http://dl.eea-fds.umontpellier.fr/CppInstall/tuto-utilisation-Cpp.mp4>

Enfin, un tutoriel spécifique pour utiliser Raylib sous Eclipse :

<http://dl.eea-fds.umontpellier.fr/CppInstall/tuto-projet-raylib.mp4>

Pour finir : si jamais l'installation se passait mal, il est préférable de tout désinstaller et recommencer plutot qu'essayer de "bricoler". Nous donnons ci-dessous un tutoriel pour tout désinstaller proprement :

<http://dl.eea-fds.umontpellier.fr/CppInstall/tuto-supprimer-Cpp.mp4>

Table des matières

Chapitre I – Introduction Générale	9
I.1 Contexte	9
I.2 Popularité et usage des différents langages de programmation	10
I.3 C'est un cours de C ... "modernisé"?	11
I.4 Et par rapport à Python et à l'assembleur?	12
I.5 Organisation du cours	13
 Les bases du Langage C/C++	
Chapitre II – Comparaison entre code Python et code C/C++ : un survol	15
II.1 Un premier code	15
II.2 Un deuxième exemple	18
II.3 Et ça va vraiment plus vite? Meilleures performances en calcul?	19
II.4 Quelques autres similarités et différences	20
II.5 Python est "interprété", C/C++ est "compilé"?	21
Chapitre III – Les variables en C/C++	23
III.1 Plusieurs types "nombre entier" en C/C++	23
III.2 Les autres types de base du C	26
III.3 Déclaration de variables : typage explicite	26
III.4 Affichage et saisie	28
III.5 Conversion de type	28
Chapitre IV – Tableaux et chaînes de caractères	31
IV.1 Les chaînes de caractères "string"	31
IV.2 Tableaux : "vector"	33
IV.3 Tableaux à 2 dimensions	35
Chapitre V – Les Fonctions	39
V.1 Premier survol : similarités et différences avec Python	39
V.2 Et return?	40
V.3 Un exemple	43
V.4 Un petit Quiz	46

Chapitre VI – Structures de données	49
VI.1 Contexte	49
VI.2 Un premier exemple : les Nombres Complexes	49
VI.3 Quelques autres exemples de structures	51
VI.4 Tableaux de structures	52
VI.5 Fonctions et structures	52
VI.6 Organisation d'un code contenant des structures et des fonctions	53
 Les techniques du Langage C/C++	
Chapitre VII – Introduction à la partie 2	56
Chapitre VIII – Codage des nombres	57
VIII.1 Codage en base 2 : binaire	57
VIII.2 Codage en base 16 : hexadécimal	58
VIII.3 Codage des variables entières en C/C++	60
VIII.4 Codage des nombres à virgule flottante en mémoire	61
VIII.5 Comparaison avec le codage en chaînes de caractères	61
Chapitre IX – Exploration de la mémoire et pointeurs	64
IX.1 Les variables dans la mémoire : contenu et adresse	64
IX.2 Des variables particulières : les pointeurs	67
IX.3 Le type d'un pointeur	68
IX.4 Déréférencement	70
IX.5 Une technique importante : modifier une variable en utilisant un pointeur	71
IX.6 Une chose à ne pas faire : déréférencer un pointeur sans savoir où il pointe	71
IX.7 Tableaux en mémoire	72
IX.8 Au sujet des chaînes de caractères	75
Chapitre X – Fonctions avec plusieurs résultats	76
X.1 Entrées et sorties multiples des fonctions en C/C++	76
X.2 Le passage par adresse en profondeur	78
X.3 Et sans pointeur, ça marche vraiment pas ? (non)	79
X.4 Comparaison return et passage par adresse	80
X.5 Utiliser à la fois return et passage par adresse	84
X.6 Et avec les structures ?	85
Chapitre XI – Compilation multi-fichiers et Bibliothèques	88
XI.1 Compilation multi-fichiers	88
XI.2 Exemple de Bibliothèque non-standard : rayLib	91
Chapitre XII – Gestion des fichiers en C/C++ et Formats de Fichiers	99
XII.1 Lire et écrire des données dans un fichier : de quoi parle-t-on exactement ?	99
XII.2 Rôle du système d'exploitation	100

XII.3 Gestion des fichiers en C/C++ : mode Texte	101
XII.4 Gestion des fichiers en C/C++ : mode Binaire	105
XII.5 Se déplacer dans un fichier : "curseur" sur fichier	109
XII.6 "Formats" de fichiers : une introduction	110
 Pont entre C "modernisé" et C Standard	
Chapitre XIII – Introduction	114
Chapitre XIV – Gestion de la RAM en C standard	115
XIV.1 Organisation de la RAM	115
XIV.2 Allocation dynamique	116
XIV.3 Allocation statique	117
XIV.4 Quelques Exemples	119
Chapitre XV – Gestion des chaines de caractères	125
XV.1 Concept de chaîne de caractères en C standard	125
XV.2 Y'a pas "cout" !	126
XV.3 Y'a pas non plus "cin" !	127
 Mélanger du Python et du C/C++	
Chapitre XVI – Exécuter un code C/C++ à partir de Python	130
XVI.1 Introduction	130
XVI.2 Le code que l'on a envie d'écrire en Python	130
XVI.3 Créer la fonction C/C++	130
XVI.4 Créer la bibliothèque C/C++	131
XVI.5 Importer la bibliothèque dans Python	132
XVI.6 Utiliser la fonction C dans Python	133
XVI.7 Une écriture plus élégante	134

Chapitre I – Introduction Générale

❶ Motivations et objectifs

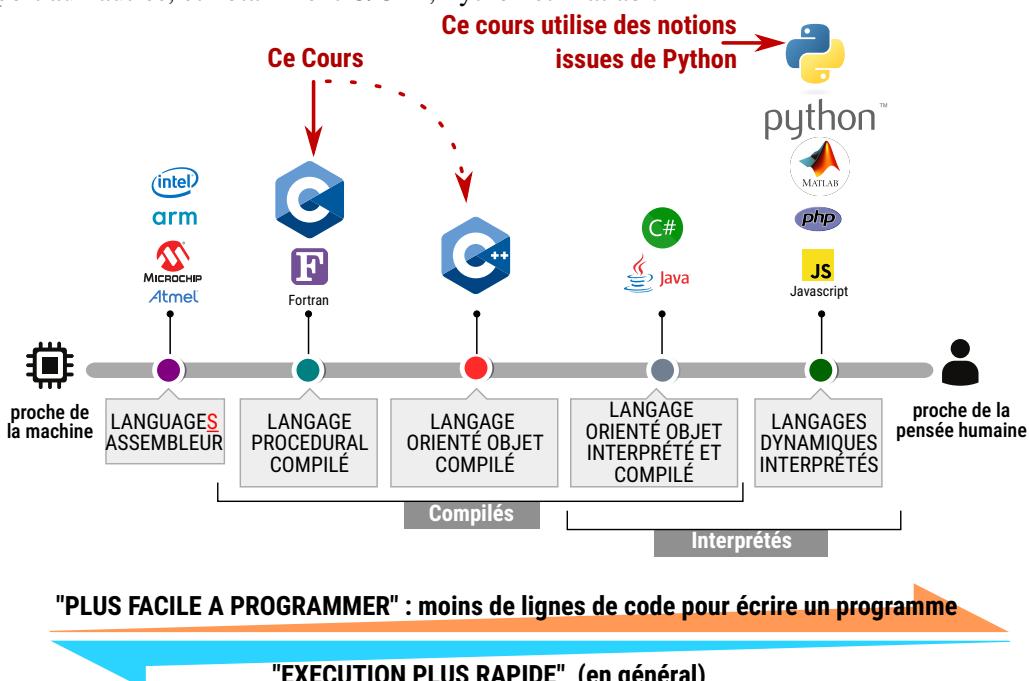
Dans ce chapitre, on veut situer les langages C et C++ dans le contexte des langages de programmation : Assembleur, C, Matlab, Python et d'autres. Nous présentons aussi les contours de ce cours, et pourquoi nous parlons de "C modernisé" plutôt que de C ou de C++.

I.1 Contexte

En tant qu'étudiant en "EEA", vos besoins en matière de programmation sont de deux natures. En effet, vous avez besoin, comme "outils informatiques", de deux types de langages :

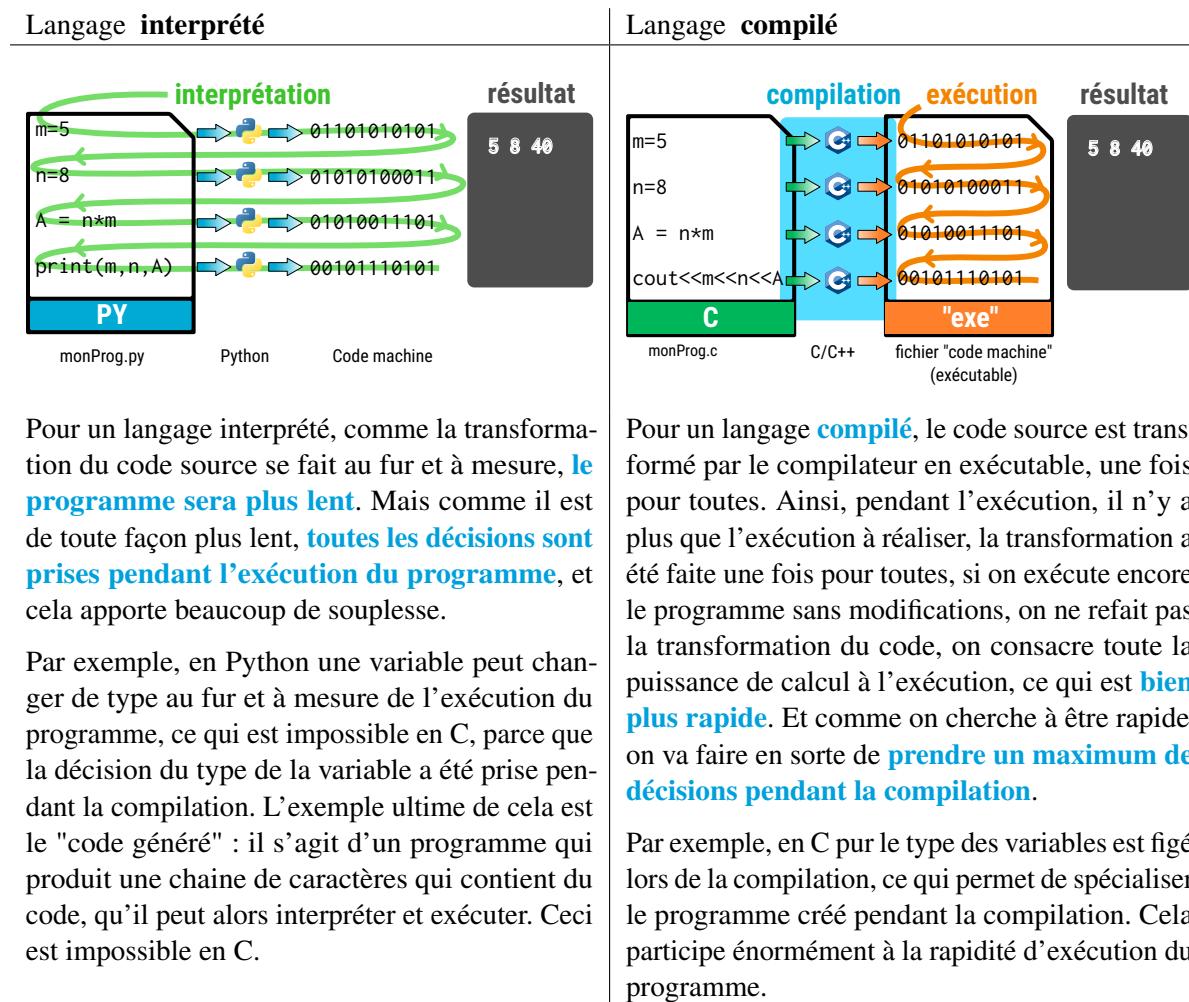
- Un langage dit "de haut niveau", typiquement un langage permettant d'écrire des "scripts", que vous utilisez pour des tâches de nature scientifique : calcul, tracé de courbes, simulation, etc. Dans le contexte dans lequel nous travaillons, on parle essentiellement de Python, pour lequel vous devez avoir des notions, et éventuellement d'un autre langage, assez similaire : Matlab.
- Un langage dit "de bas niveau", typiquement un langage dit "compilé", qui sert dans un contexte plus "matériel", typiquement la programmation des microcontrôleurs¹. Il s'agit de C/C++ et parfois d'un peu d'assembleur.

Or il existe des dizaines de langages informatiques, et on ne peut pas dire que certains soient meilleurs que d'autres : tous ont une utilité. Mais on peut les classifier sur un plan technique pour les situer les uns par rapport aux autres, et notamment C/C++, Python et Matlab :



1. Mais l'usage du C/C++ ne s'arrête pas là, il concerne tout ce qui doit être rapide ou fonctionner sur des machines disposant de peu de puissance de calcul

Sur l'illustration, notez bien les deux flèches, qui classent les langages par "facilité pour la programmation", qui va en général à l'inverse de "rapidité d'exécution" (vitesse du calcul) des programmes créés. Et c'est souvent un des enjeux importants quand on choisit un langage pour une tâche donnée : ce compromis entre "facilité de programmation" et "rapidité du programme". Et pour illustrer cela, on a mis en évidence d'un côté les langages compilés, et de l'autre les langages interprétés. Voyons de quoi il s'agit :

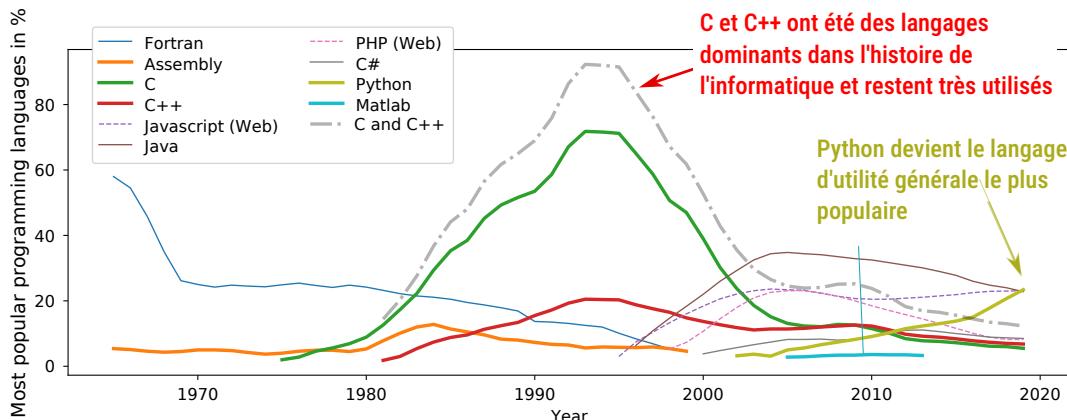


Il est probable que cette comparaison soit un peu obscure pour un apprenti programmeur, mais nous reviendrons sur ces idées dans le déroulé du module. Il faut à ce stade retenir le compromis facilité/rapidité qui existe dans tout langage.

I.2 Popularité et usage des différents langages de programmation

On retrace l'histoire (résumée) de la popularité des langages de programmation ci-dessous² :

2. graphe réalisé à partir de la vidéo "Most Popular programming Languages 1965-2019" de la chaîne youtube "Data is Beautiful", disponible à l'URL : <https://www.youtube.com/watch?v=0g847HVwRSI>



Ce que l'on peut comprendre à partir de ce graphe :

- L'immense popularité passée de C, C++ et Fortran fait qu'enormément de code est disponible pour tout un tas d'usages différents. De nos jours, ces codes prennent la souvent la forme de bibliothèques, qui sont par nature extrêmement rapides. Elles sont souvent utilisées à partir de langages de haut niveau comme Python ou Matlab. Pour anecdote, le cœur de Matlab, qui concerne le calcul de matrices, est basé sur une bibliothèque Fortran open-source, ainsi que sur divers codes C/C++.
- Le fait que C et C++ soient moins utilisés aujourd'hui (en proportion) est lié à deux choses : d'une part à l'évolution de la puissance de calcul des PC, et d'autre part au développement du Web, qui n'utilise pas C et C++. Malgré tout, pour les applications "de bureau" (= non web), C et C++ restent utilisés pour développer des parties de logiciels qui nécessitent de la vitesse.
- L'assembleur n'est plus utilisé que comme moyen d'optimisation ultime. Aujourd'hui, ne sont plus programmés en assembleur que des choses comme des drivers (par exemple driver/pilote de carte video pour gamers) ou des parties de certains logiciels, comme Photoshop par exemple.
- Python est le langage d'utilité générale qui est le plus utilisé au monde aujourd'hui. Il concerne le développement de logiciels "de bureau", d'applications web ou smartphone, usage scientifique, hobbyistes, ...

I.3 C'est un cours de C ... "modernisé"?

L'objectif ici est d'apprendre le Langage C, parce que l'objectif visé à court terme est la programmation de microcontrôleurs, abordée au second semestre. Mais faisons une remarque : On parle depuis le début de C et de C++, mais qu'est ce que cela signifie ? C++ est un langage qui est une évolution importante du C : il est totalement compatible avec C (il "contient" tout ce que le C contient), et apporte de nombreuses améliorations. Parmi elles la plus importante est la "programmation par objet". Mais la programmation par objet n'est pas notre objectif.

En effet, dans ce cours, nous voulons nous centrer principalement sur le C, pour deux raisons :

- D'abord parce que les concepts de "programmation objet" sont difficiles à prendre en main et qu'ils ne sont pas utiles pour ce que l'on veut faire,
- Ensuite, parce que l'un des objectifs visés est la programmation sur microcontrôleurs, et que C++ "pur et dur" peut s'avérer trop gourmand en performances matérielles pour eux.

Malgré tout, il est pratique d'utiliser des éléments du C++ dans un programme en C, sans pour autant aller vers la programmation objet. C'est ce compromis que nous appelons le "**Langage C modernisé**". Cela simplifie un ensemble de choses et rend plus naturelle l'écriture de programmes plus robustes, sans impacter la vitesse d'exécution (ou marginalement), et sans rendre le programme trop lourd pour un microcontrôleur. Il serait donc dommage de s'en priver.

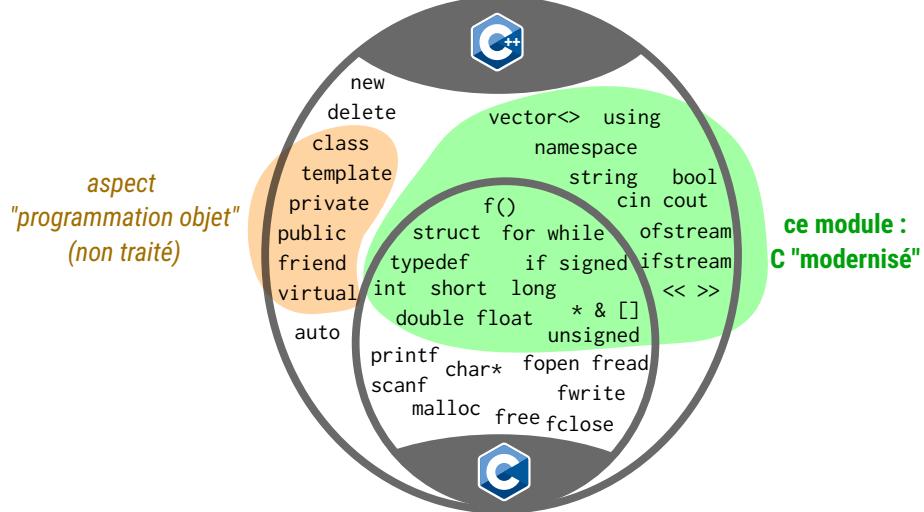
Toutefois, nous n'avons pas poussé la démarche de modernisation jusqu'au bout, pour rester compatibles avec les techniques de "C classique" utilisées sur les bibliothèques pour microcontrôleur. Ce qui est proposé ici, c'est donc une démarche qui permet d'apprendre suffisamment de C pur et dur pour pouvoir

utiliser les bibliothèques des microcontrôleurs, tout en utilisant le C++ lorsqu'il nous permet d'éviter des pièges du C, qui sont nombreux et plus vraiment d'actualité. Seuls de très vieux codes sont encore du "C pur", or programmer en C pur nous expose à des pièges qui mènent à des plantages "incompréhensibles" si on n'est pas un programmeur vraiment expérimenté du C.

Enfin, bien que nous utilisions des morceaux de C++, **nous ne parlerons pas de "programmation objet"**. La programmation objet et le C++ seront abordés en Master.

En résumé, nous ferons simplement de la programmation impérative, qui est le cœur du C "pur et dur" et que vous avez normalement appris en Python, tout en utilisant des ingrédients de C++ pour rendre plus simples et plus robustes certaines choses.

Pour ceux qui auraient des notions sur ces langages, voici une illustration du contour approximatif de ce cours, qui correspond à l'idée de ce que nous appelons du "C modernisé"³ :



Remarque

Même si l'essentiel de ce cours concerne le "C modernisé" que nous appelons C/C++, les derniers chapitres de ce cours sont dédiés aux éléments de syntaxe et à l'état d'esprit qui permettent de s'en sortir avec du C standard. Ainsi, ce cours va suivre la progression suivante :

Python → C modernisé (C/C++) → C standard (C)

Cette progression permet de prendre en main la syntaxe de base du langage, grâce à l'approche C modernisé, sans tomber dans les pièges du C standard dans un premier temps. Et dans un deuxième temps, on arrive au C standard, qui fait partie des objectifs.

I.4 Et par rapport à Python et à l'assembleur?

Le C/C++ sont des langages complémentaires par rapport à Python. Comme mentionné plus haut, le C et C++ sont des langages qui sont faits pour écrire des programmes rapides. Mais l'écriture des programmes reste plus complexe.

D'une certaine façon, en programmation moderne, cela revient à dire que l'objectif que l'on se fixe avec C et C++, c'est de faire des choses plus "simples" mais les faire de façon presque idéale : un programme écrit en C peut s'exécuter 10 000 à 100 000 fois plus vite que le même programme écrit en Python. Il faut faire très attention avec ce genre de chiffre, ce sont des ordres de grandeur que l'on peut constater, mais ils peuvent être bien inférieurs ou bien supérieurs selon ce que l'on fait, et même selon la

3. Notez qu'il existe un peu partout des ouvrages qui parlent de "C moderne" aussi appelé parfois "C+" (et non C ou C++), mais que le contour n'est pas forcément celui choisi pour ce cours

façon que l'on a d'écrire le code. Mais c'est bien la rapidité qui amène à vouloir aller vers du C ou du C++.

Le langage le plus rapide que peuvent exécuter les machines est l'assembleur. En comparaison, le C/C++ produit des programmes qui sont 5 fois plus lents dans le pire des cas. Là encore, le compromis facilité/rapidité reste comparable sur le principe : lors l'écriture d'un programme C est bien bien plus "simple" et aboutit à du code beaucoup plus lisible qu'on ne peut le faire en assembleur, ce qui rend acceptable la perte de performance.

Enfin, notez aussi que les bibliothèques Python se contentent pour beaucoup de faire simplement appel à du code écrit en C, ce qui confère à Python, sur certaines tâches spécifiques, des performances extrêmement bonnes, surtout au regard du peu de code qui a été nécessaire.

I.5 Organisation du cours

Le cours est fait de 2 parties à peu près de même taille :

1. une première partie qui concerne **les bases du langage C/C++**. L'objectif est d'introduire et de vous habituer à la syntaxe du C/C++, en utilisant ce que vous savez de Python comme base générique en programmation. C'est donc un survol qui va nous amener à connaître les ingrédients principaux, mais a des limites
2. une seconde partie, puis difficile mais indispensable, qui concerne les **techniques de programmation du C/C++**. Cette partie nous ramènera plus près du fait que le C/C++ est pensé comme un langage "près de la machine". Dans cette partie, nous reviendrons sur comment un ordinateur fonctionne et introduirons alors un outil puissant du C/C++ : les pointeurs. L'introduction des pointeurs nous permettra de dépasser les limites rencontrées dans la première partie, grâce à un ensemble de techniques de programmation que tout programmeur en C/C++ doit connaître.

❖ En résumé ...

Nous allons apprendre le "langage C", mais sous une forme modernisée, quelque part à mi-chemin entre C et C++. Ce langage est plus "ardu" que Python même si il partage beaucoup de choses avec lui. L'avantage est une capacité à réaliser des calculs bien plus rapidement que ne le fait Python. Autrement dit : on peut soit accélérer des calculs sur PC, soit réaliser des programmes utilisables sur des puissances de calcul plus limitées (et donc consommant beaucoup beaucoup moins d'énergie), comme les microcontrôleurs, qui sont étudiés au second semestre. ■

PARTIE I

Les bases du Langage C/C++

Chapitre II – Comparaison entre code Python et code C/C++ : un survol

② Motivations et objectifs

On montre ici deux programmes simples, qui permettent de démarrer avec la syntaxe du C/C++. On discute aussi de ce que signifie un peu plus concrètement langage "interprété" et langage "compilé".

II.1 Un premier code

Prenons un code Python très simple, et regardons comment on le transforme en C/C++. Ici un programme simple qui calcule $\sin(x) \times \exp(-x)$ pour $x = 1.5$:



```
from math import sin, exp

x = 1.5
y = sin(x)*exp(-x)

print("f(x) pour x=", x, "vaut", y)
```



```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    float x;
    float y;

    x = 1.5;
    y = sin(x)*exp(-x);

    cout << "f(x) pour x= " << x << " vaut " << y << "\n";

    return 0;
}
```

Rien de fou ici : si on lit le code Python :

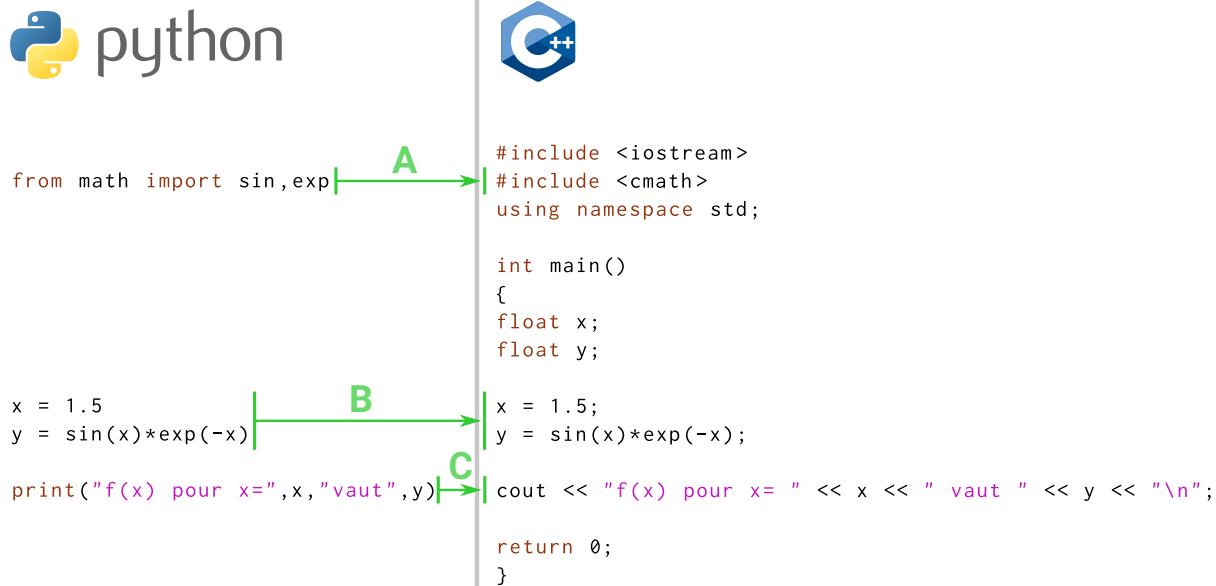
- on commence par importer les fonctions `sin` et `exp` à partir de la bibliothèque `math`,
- on affecte la valeur 1.5 à la variable `x`,
- on calcule le résultat de $\sin(x) \times \exp(-x)$, et on stocke le résultat dans `y`,
- enfin on affiche le résultat, sous la forme :

```
f(x) pour x= 1.5 vaut 0.22257121610821853
```

Et c'est ce qu'affichent les deux programmes, sauf que le C/C++ affiche moins de chiffres après la virgule (mais ça n'est pas important en soi).

Ce qui est similaire

On a représenté en vert ce "que l'on peut comprendre" assez intuitivement dans cette conversion :

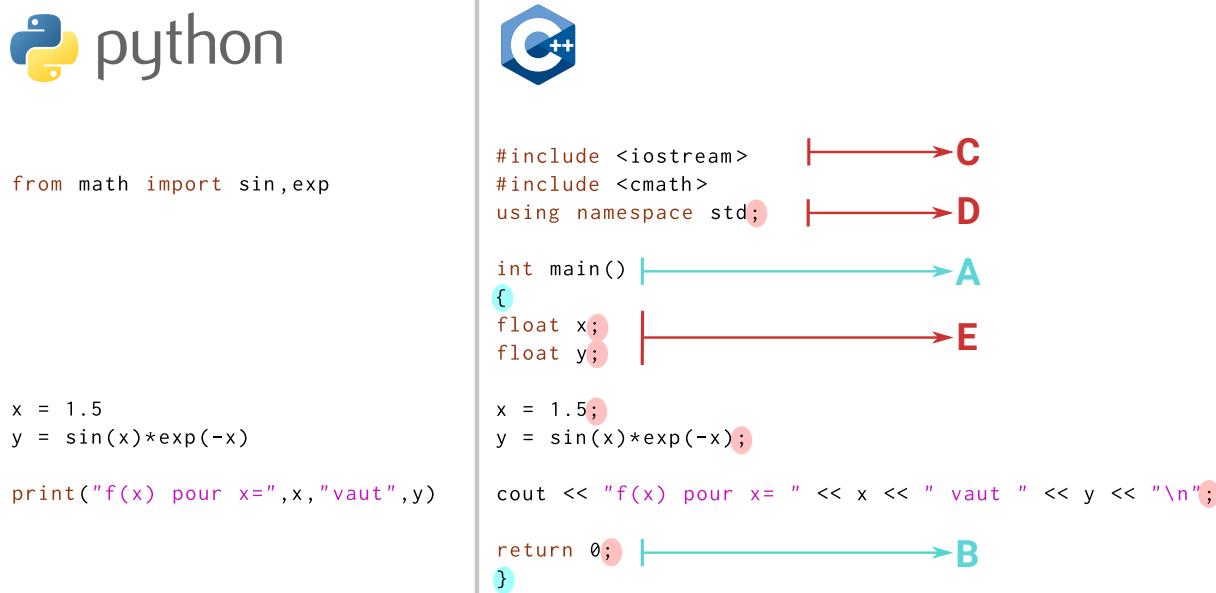


Quelques commentaires :

- **flèche A** : inclusion de la bibliothèque. On voit bien qu'en C/C++ elle s'appelle `cmath` alors qu'elle s'appelle `math` en python, que l'inclusion se fait avec la séquence `#include<...>` en C/C++ alors qu'elle se fait avec `from ... import`. Et une autre remarque : lors de l'importation, en Python on n'importe que certaines parties d'une bibliothèque, alors qu'en C on importe forcément la totalité.
- **flèche B** : L'expression du calcul est la même car issue des mathématiques (même si ça n'est pas forcément exactement pareil dans tous les langages). La seule différence ce sont les " ; " que l'on trouve à la fin de chaque ligne. Nous y reviendrons, ils sont obligatoires en C/C++.
- **flèche C** : On identifie très bien la ligne qui fait l'affichage. La syntaxe est assez différente, en C/C++ on utilise `cout` accompagné par des `<<`, mais l'ordre des éléments est le même. Le `\n` en fin de ligne qui force le retour à la ligne (qui est automatique en Python).

Ce qui est différent

Maintenant regardons ce qui n'est pas intuitif, mais qui est indispensable au fonctionnement du programme C/C++ :



- **flèches A et B + cercles bleus** : Cela définit le "programme principal" pour le C/C++, c'est à dire ce qui est exécuté dès le lancement du programme : le programme commence après l'accolade qui s'ouvre juste en dessous de `int main()`. C'est "comme ça". Dans un premier temps, considérez que cette séquence, formée par :

```

int main()
{
    ...
}
    
```

définit un bloc principal, qui constitue le point de départ d'un programme C/C++, et contient le code. C'est peu de choses, mais c'est indispensable, absolument indispensable.

- **flèche C** : il s'agit de l'inclusion d'une bibliothèque supplémentaire. Il faut comprendre que le Langage C/C++ ne sait même pas afficher un texte si on ne lui explique pas comment faire (alors que Python si). Et donc il faut inclure la bibliothèque qui permet d'ajouter l'équivalent de `print` en Python, c'est à dire `cout`, et cette bibliothèque c'est `iostream`.
- **flèche D** : cette ligne n'est pas évidente à comprendre et on ne rentrera pas dans les détails. Mais de façon simple : elle permet d'utiliser directement ce qui est ajouté par les bibliothèques standard (fournies avec le C/C++), et donc il faut la mettre systématiquement à la fin de tous les `#include<...>` dans les programmes que l'on fait. Elle sera bien justifiée lorsque vous ferez de la programmation objet.
- **flèche E** : c'est un sujet très important : en C/C++, on doit **déclarer les variables**. C'est à dire que l'on ne peut pas écrire simplement, directement, `x=1.5` : avant ça, il faut dire que l'on crée la variable `x`, et dire quel sera son type (ici `float`) : c'est ce qui indique au C/C++ qu'il doit réservé un emplacement mémoire pour elle, et il faut comprendre que c'est l'un des ingrédients du C/C++ qui fait qu'il est rapide, bien plus rapide que Python. ***Il faut aussi avoir en tête que au moment où déclare une variable, on n'a aucune idée de sa valeur !***. Au début, elle donc est simplement **aléatoire** (valeur "au hasard"), et il faut attendre de lui attribuer une valeur pour être certain de ce qu'elle vaut. On reviendra sur cet aspect, pour l'instant retenez qu'il faut absolument déclarer les variables, et que tant qu'on ne leur a pas donné de valeur, elles valent n'importe quoi : même pas "0", mais bien n'importe quoi.

- Et puis il y a des ";" à la fin de chaque ligne de code. En Python on peut en mettre sans obligation, en C/C++ il est obligatoire.

II.2 Un deuxième exemple

On va introduire les boucles et les conditions. On va prendre un exemple simple, qui calcule la "conjecture de Syracuse" ¹, qui utilise une suite définie par :

- si U_n est pair, alors $U_{n+1} = U_n/2$,
- si U_n est impair, alors $U_{n+1} = 1 + 3 \times U_n$,
- on choisit un U_0 entier et positif, par exemple $U_0 = 27$

Notez que c'est juste un exemple, on aurait pu prendre n'importe quel calcul qui nécessite des boucles et conditions. On va la calculer pour 120 termes, avec Python et avec C/C++ :



```

1
2
3
4
5
6
7
8
9 U = 27
10 k=0
11
12 while k < 120:
13     if U%2 == 0:
14         U = U // 2
15     else:
16         U = 3*U+1
17     k=k+1
18     print("U",k,"=",U)
19
20
21
22
23
24
25
26
27 #

```

[code/code-II-2-1.py]



```

1 #include <iostream>
2 using namespace std;
3
4 int main ()
5 {
6     int U;
7     int k;
8
9     U = 27;
10    k = 0;
11
12    while (k < 120)
13    {
14        if (U % 2 == 0)
15        {
16            U = U / 2;
17        }
18        else
19        {
20            U = 3*U+1;
21        }
22        k = k+1;
23        cout << "U" << k << " = " << U << "\n";
24    }
25
26    return 0;
27 }

```

[code/code-II-2-1.cpp]

On voit que les mots **while** et **if** s'utilisent en C/C++ et en Python et pour les mêmes raisons. Ce qui change c'est :

- En C/C++, la condition est maintenant entre parenthèses, et il n'y a pas les ":"

1. Cette conjecture stipule que la suite mathématique définie ici aboutit forcément à des cycles 4-2-1 après un nombre de termes assez grand, quel que soit le U_0 entier positif choisi. Cela a été vérifié jusqu'à des U_0 énormes sans trouver de contre exemple, mais les mathématiciens ne savent pas, à ce jour, démontrer pourquoi le comportement est celui là, ni prouver que cette conjecture est fausse pour certains U_0 . C'est donc en cela que c'est toujours une conjecture : on ne sait ni prouver que c'est vrai, ni prouver que c'est faux.

- Les "blocs" de code associés aux `while` et `if` sont maintenant entourés par "`{`" et "`}`". Autrement dit, à chaque fois que vous ajoutez un niveau de tabulation (décalage) en Python, en C il faut mettre une "`{`". Et à chaque fois que vous retirez un niveau de tabulation de Python (ou *indentation*), il faut mettre une "`}`" en C/C++.
- L'opérateur `//` de Python n'existe pas en C, et ça n'est pas "un détail", il y a quelque chose derrière ca :
 - En Python, la division entière se fait avec `//`, quel que soit le type des nombres à gauche et à droite de l'opération
 - En C, c'est le type des nombres qui définit le type de l'opération et de son résultat. Si on a des nombres flottants de chaque côté de l'opérateur, ici "`/`", c'est une opération en flottant et le résultat sera flottant. Si ce sont des nombres entiers, c'est une opération en nombre entiers et le résultat sera un entier. Si on mélange les deux, c'est le type le plus général qui compte.

II.3 Et ça va vraiment plus vite ? Meilleures performances en calcul ?

On va comparer la vitesse de calcul des deux langages. Pour que l'on puisse y voir quelque chose, on doit transformer un peu le code, pour que le résultat soit significatif :

- Ca parrait idiot, mais il faut d'abord supprimer les affichages. En effet, l'affichage est sous-traité au système d'exploitation, et donc il n'est pas représentatif du temps utilisé par Python ou par le C.
- Ensuite, il faut augmenter beaucoup le nombre de termes que l'on va calculer. On va partir sur cent millions de termes (100 000 000).
- Par ailleurs, pour montrer la vraie performance du C/C++, il faut demander au compilateur "d'optimiser le code", c'est à dire le rendre le plus rapide possible. Pour cela on va le compiler en utilisant l'option `-O3` :

```
g++ -O3 code.c -o code
```

Cette idée "d'optimisation" pendant la compilation ne peut exister qu'en C/C++ et pas en Python, puisque Python est interprété.

- Pour terminer, on va utiliser la capacité du terminal à chronométrier le temps passé dans un programme, et donc on va lancer, successivement, le programme C puis le programme Python :

```
time code
time python3 code.py
```

Et on obtient (sur une station de travail portable de 2021) **pour Python un temps > 2 s, et pour C/C++ un temps non mesurable, < 4ms**. On a donc un rapport de l'ordre de 10000, ce qui signifie que la différence des temps d'exécution se creuse si on augmente le nombre de termes calculés.

Important

On a donc bien mis en évidence la vitesse du C par rapport à Python. Cette "vitesse" peut être utilisée de 3 façons :

- Accélérer des temps de calcul sur PC,
- Rendre possibles des calculs sur PC,
- Utiliser des puissances de calcul faible (et donc peu d'énergie) pour faire des choses déjà évoluées, comme ce sera le cas avec les microcontrôleurs.

Remarque

Malgré tout, on a voulu ici "marquer le coup" et obtenir des rapports de temps "monstrueux". En effet, ce code, tel qu'il est écrit côté Python, est particulièrement propice à faire que Python se mette à ramer. Mais c'est justement cela qu'il faut comprendre : si on veut faire du code optimisé avec Python, il faut un vrai talent, on peut vite tomber dans des pièges qui font que le temps de calcul va littéralement exploser, ce qui peut être déconcertant si on ne maîtrise pas tout ce qui se passe. On pourrait écrire le même code en Python pour qu'il soit bien plus rapide. Mais il ne faut pas se leurrer, sur ce genre de calcul on aura au minimum un rapport de temps 10 à 100 entre Python et C/C++ (en faveur du C/C++ bien entendu) sur le temps d'exécution, même si on connaît parfaitement Python et comment l'optimiser pour réduire les temps de calcul. Alors qu'en C/C++, il n'y a pas vraiment de manière "horrible" d'écrire ce code pour le ralentir : le C/C++ est assez naturellement un langage qui consomme peu de ressources et c'est sa grande force (même si il existe des leviers aussi en C/C++ pour optimiser du code et le faire aller encore plus vite). Et dans tous les cas, il y a donc un certain niveau où C/C++ devient incontournable pour améliorer les temps de calcul de toute façon.

II.4 Quelques autres similarités et différences

Opérateurs et chaînes de caractères

On pourra s'en rendre compte en programmant au fur et à mesure, mais voici quelques autres différences assez courantes :

- Il n'y a pas d'opérateur en C/C++ pour calculer a^b . En Python, on écrit simplement `a**b`. En C/C++, il faut passer par la bibliothèque mathématique et la fonction `pow`
- Les guillemets ("") en C/C++ sont la seule façon de définir une chaîne de caractères. Les autres possibilités offertes par Python n'existent pas en C/C++

La boucle `for`

C'est un sujet qui mérite que l'on s'arrête un peu dessus. Il existe des boucles `for` en C/C++, mais elles ne signifient pas la même chose qu'en Python. En Python, les boucles `for` sont un "raccourci très pratique" qui permet de traiter les "types itérables", comme les `list`, `tuple`, `array`, etc.

En C/C++, le concept de "type itérable" n'existe pas². D'ailleurs en C/C++ les `list`, `tuple` n'existent pas directement³. En C/C++, la boucle `for` est en réalité une écriture compacte du `while` et rien de plus. Si on compare les deux sur un code qui affiche 10 fois "bonjour" :

Avec <code>while</code> :	Avec <code>for</code> :
<pre>int k; k=0; while(k<10) {cout << "Bonjour"; k=k+1; }</pre>	<pre>int k; for(k=0;k<10;k=k+1) {cout << "Bonjour"; }</pre>

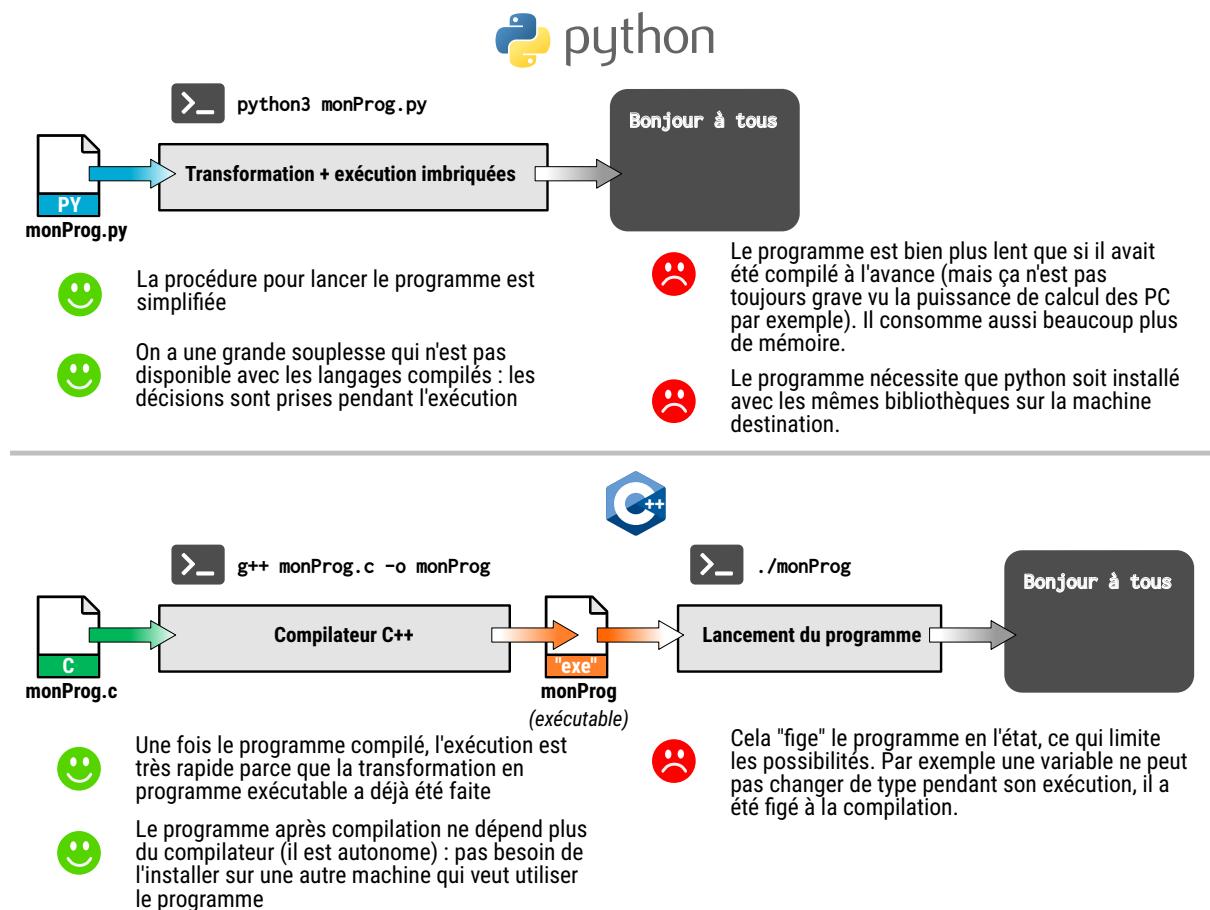
Voilà c'est juste une écriture plus compacte.

2. Il existe bien des itérateurs mais ça commence à être de la programmation assez avancée et la syntaxe est loin d'être très naturelle au début.

3. Cela existe dans des bibliothèques mais là encore cela concerne des usages avancés.

II.5 Python est "interprété", C/C++ est "compilé"?

Regardons ce qui se passe quand on veut lancer un programme Python et quand on lance un programme C/C++.



On est donc sur un compromis entre d'un côté, la simplicité et les fonctionnalités du langage interprété, et de l'autre côté l'efficacité à l'exécution du langage compilé.

Remarque

Il est possible de compiler du Python, avec différents outils (par exemple "Cython"). Bien que Python ne soit pas fait pour être compilé, on obtient alors, sans rien modifier au code, des gains de performance (vitesse) de l'ordre de 20%. C'est déjà "bon à prendre", mais ça reste peu de choses par rapport au gain de performance que l'on peut obtenir avec du C/C++, largement supérieur à 1000 dans la plupart des situations. En substance, ce "faible gain" de 20% est lié à la fois au fait que Python n'a pas été conçu pour être compilé, et donc son code interne n'est pas optimisé dans cette direction (alors que pour le C/C++, si), et en outre, C/C++ a, de par sa syntaxe, des éléments qui font que les programmes seront plus rapides, parce qu'ils obligent le programmeur à plus prendre en compte ce qu'il est en profondeur l'ordinateur, quand Python tend à s'en abstraire.

En résumé ...

On a vu un premier code par comparaison entre C/C++ et Python. À part quelques aspects légers de syntaxe indispensables à connaître et dont on a discuté, il y a 2 points fondamentaux qu'il faut avoir en tête :

- Le programme principal, en C/C++, s'écrit toujours dans l'environnement du `main`, alors que ce n'est pas le cas en Python.
- Le C/C++ nécessite la déclaration des variables, ce dont on va rediscuter au prochain chapitre

On a vu aussi `while` et `if` en C/C++, qui ont exactement le même rôle qu'en Python. Les différences sont dans la syntaxe. Schématiquement sur le `while` C/C++ :

```
while (.....)
{
    ...
}
```

Ainsi, on délimite la condition par des parenthèses en C/C++ au lieu de la délimiter par ":" en fin de condition en Python. De plus, on utilise des accolades "{" et "}" en C/C++ pour définir ce qu'il faut exécuter, en lieu et place du décalage (indentation) que l'on écrit en Python (mais il est d'usage, pour la mise en page, de faire ce décalage quand même en C/C++). De plus, `for` n'a pas la même utilité qu'en Python : c'est ici, en C/C++, juste une écriture compacte de `while`

Enfin, on a discuté des différences entre langages interprétés et langages compilés, l'un étant plus souple et plus intuitif pour l'utilisateur du langage, l'autre nécessitant un processus plus complexe mais aboutissant à un programme **autonome** et par nature **bien plus rapide**. On pu voir que l'on retrouve ces étapes lors de l'exécution du programme. ■

Chapitre III – Les variables en C/C++

⌚ Motivations et objectifs

Le sujet des variables mérite d'être discuté avec un peu de profondeur, pour plusieurs raisons. D'abord, parce que le comportement de C/C++ et de Python au regard des variables est assez différent, même si cela semble inaperçu dans la syntaxe. Ensuite, parce qu'il est utile d'examiner quel type du C/C++ est bien adapté à telle ou telle situation, en particulier au sujet des nombres entiers.

III.1 Plusieurs types "nombre entier" en C/C++

En Python, il y a essentiellement un type entier¹, et il se nomme `int`. En C "de base", il y a 5 types entiers, avec des variantes :

Type	Bits	<code>sizeof</code>	Valeurs si <code>signed</code>	Valeurs si <code>unsigned</code>	Exemple d'application
<code>char</code>	8 bit	1	-128 à +127	0 ... 255	texte, amplitude d'un son en basse qualité, pixels d'une image, ...
<code>short</code>	16 bit	2	-32768 à +32767	0 à 65353	amplitudes d'un son "HiFi", pixels d'une image pour magazine d'art ...
<code>int</code>	32 bit	4	-2 milliard à +2 milliard	0 à 4 milliard	calculs arithmétiques, compteur dans un tableau
<code>long</code>	64 bit	8	-8×10^{18} à $+8 \times 10^{18}$	0 à 16×10^{18}	calculs arithmétiques, compteur dans un tableau

Quelques remarques :

- Il y a dans ce tableau une colonne nommée "sizeof" : `sizeof` est un opérateur du C qui permet de connaître la taille d'un type en *octets*. Ainsi, le `char` par exemple, défini comme entier sur 8 bits, fait 1 octet comme indiqué dans le tableau, et donc le résultat de `sizeof(char)` est 1. Le `long`, défini comme entier sur 32 bits, fait 4 octets, et donc le résultat de `sizeof(long)` est 4. Et ainsi de suite.
- Tous les types entiers peuvent être "signés" (`signed`) ou "non-signés" (`unsigned`), ce qui influence les valeurs possibles, comme indiqué dans le tableau. Si on est "non signé", les valeurs pour un nombre sur N bits sont simplement entre 0 et $2^N - 1$, toutes positives. Si on est "signé", on dispose d'autant de valeurs mais on les répartit autrement, entre -2^{N-1} à $-2^{N-1} - 1$, donc positives ou négatives. Si on décide de travailler avec un `long` non signé, on notera simplement `unsigned long`.
- Si on ne précise ni `signed` ni `unsigned` lors de la déclaration d'une variable, alors par défaut la variable est *signée*, sauf pour `char` qui est non signé.
- le type `int` est le plus souvent utilisé. Malgré tout, son codage dépend de l'ordinateur² ou processeur sur lequel on travaille, mais on considérera qu'il est toujours sur 32 bit en pratique.

Regardons maintenant quelques exemples d'utilisation.

1. On parle ici de Python "de base", mais certaines bibliothèques comme `numpy` donnent accès à ces autres types entiers
2. le `long` peut aussi changer de taille selon les mêmes critères.

Le type char

Le type "char" s'appelle comme ça parce qu'il sert souvent à représenter des caractères. Mais c'est une idée bizarre, on vient de dire que ce sont ... des entiers ?

Important

C'est important, **il faut bien comprendre ceci : pour un ordinateur, tout est une affaire de nombres** : les caractères, les textes, les sons, les images, tout ça n'existe pas vraiment. L'ordinateur ne peut rien traiter d'autre que des valeurs.

Cette remarque semble contradictoire avec notre vie de tous les jours : on utilise tous les jours des ordinateurs pour faire tout autre chose que traiter des nombres, et donc on a l'impression que les textes, images et sons (par exemple) existent vraiment dans un ordinateur. Mais il y a une astuce : *on peut décider d'une convention dans laquelle on donne un sens autre que sa valeur à un nombre*. C'est ce que l'on fait pour le texte : on décide que par exemple la valeur 65 correspond au dessin d'un symbole qui est un "A" en majuscule. Et on voit bien que l'on peut faire une association comme ça pour toutes les lettres de l'alphabet : il suffit que cette convention soit la même pour tout le monde pour que tout le monde se comprenne. Pour les caractères, une telle convention existe, historiquement la "table ASCII" (ici seulement pour les valeurs de 0 à 127, mais elle s'étend jusqu'à 255 en pratique) :

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	-	127	7F	[DEL]

Ainsi, on comprend très bien que la séquence 66-79-78 représente la séquence de caractères B-O-N : c'est le concept de **codage de l'information**. Notez que l'on utilise pour cela le plus petit type entier disponible, celui qui ne permet que 256 combinaisons. Et c'est suffisant parce qu'il n'y a pas des milliers, des milliards ou des milliards de milliards de caractères à représenter dans l'alphabet latin. Et donc autant utiliser le moins de mémoire possible pour représenter ce que l'on a à représenter, ici des caractères. A titre d'exemple : si on avait quelque chose comme 45000 caractères, on utiliserait **short**, parce que c'est l'entier qui permet 65535 combinaisons, juste supérieur à 45000. Notez qu'en réalité, on présente ici les choses à l'envers : l'octet a été défini justement parce qu'une des premières applications grand public de

l'informatique a été de gérer du texte, et il s'est donc imposé comme "unité de mesure" de la quantité de mémoire sur un ordinateur.

En C/C++, on peut créer une variable contenant la lettre 'B' de 2 façons (à taper dans le contexte d'un `main` bien entendu) :

```
char lettre1 = 66;
char lettre2 = 'B';
cout << "Version 1 : "<<lettre1 << "    Version2 : " <<lettre2;
```

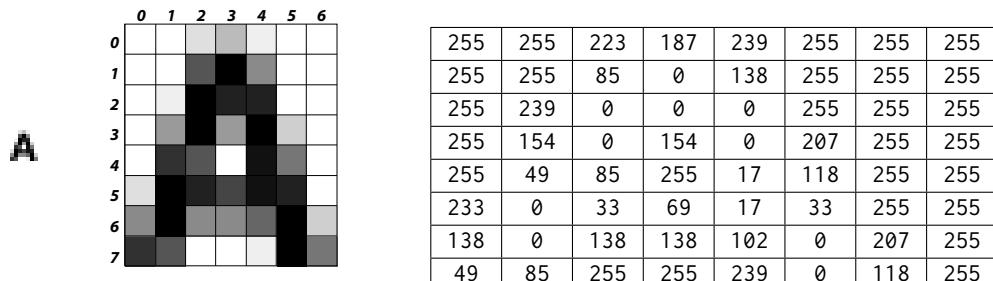
Qui affiche :

```
Version 1 : B    Version2 : B
```

On remarque deux choses :

- en C/C++, le "quote" (guillement simple ou apostrophe) sert à parler d'*un caractère unique* et pas d'une chaîne de caractères, comme c'était le cas en Python. Ainsi, il faut retenir que la séquence 'B' vaut simplement 66, et que la séquence 'ABC' est illégale en C/C++.
- l'affichage fait par `cout` du type `char` ne consiste pas à afficher sa valeur en tant qu'entier, mais à afficher la lettre qui correspond via le code ASCII. On verra comment contourner cela.

Mais le `char` ne se limite pas à cela : on l'utilise par exemple en imagerie. En effet, une image "niveaux de gris" utilise typiquement 8 bit pour chaque pixel. Une valeur faible (proche de 0) définit un point sombre, une valeur forte (proche de 255) représente un point clair. Entre les deux valeurs extrêmes, les points sont interprétés en différents niveaux de gris :



On utilisera donc des `unsigned char`. Avec ce codage, on a une précision un peu meilleure que 0.5 % (1/256) sur l'amplitude, ce qui suffit toutefois en général à satisfaire l'œil. Autrement dit : 256 niveaux de gris suffisent aux besoins de l'œil en matière de nuances.

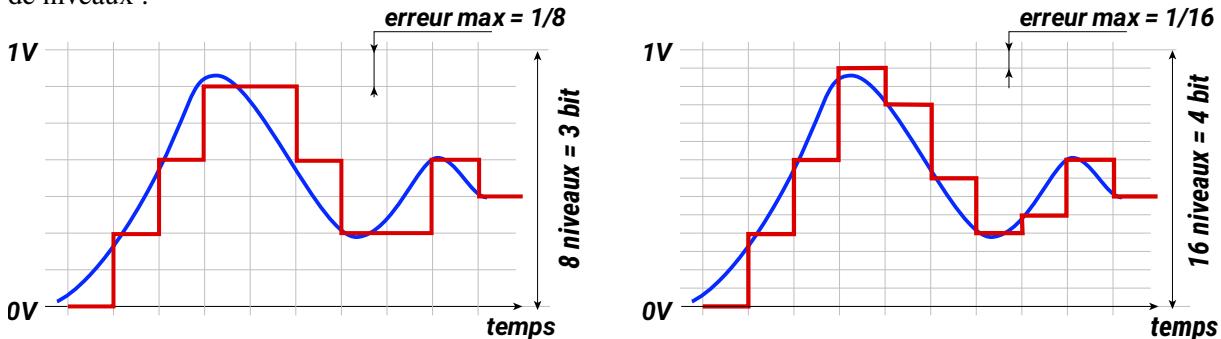
On vient donc de faire le tour de comment un ordinateur affiche du texte : par code, on crée un tableau qui contient par exemple la séquence d'octets 66-79-78, pour "BON" dans la table ASCII. L'ordinateur de son côté dispose de polices de caractères qui pourraient être stockées comme une succession d'images similaires à ce que l'on a décrit ici (une par caractère)³, et donc "afficher un texte" avec `print` en Python ou `cout` en C/C++ revient simplement à recopier le dessin qui correspond aux lettres 66-79-78.

Le type short

Il existe de nombreuses utilisations du type `short`. Par exemple on pourrait vouloir avoir plus de "niveaux de gris" différents dans l'image précédente. C'est assez peu habituel mais dans certains domaines cela se fait, par exemple pour les photographies destinées aux livres d'art ou en imagerie médicale parfois. Mais ça reste peu fréquent, les images sont essentiellement codées sur 8 bit par pixel. L'utilisation la plus courante du type `short` c'est le son : un son de qualité "HiFi" est un son dont chaque amplitude est codée sur 16 bit, ce qui fait $\approx 0.0015\%$ d'erreur (1/65536), et c'est ce qui est nécessaire pour que l'oreille n'entende pas de déformation ! Imaginez dans votre vie quotidienne quoi que ce soit que vous feriez avec

3. En réalité c'est bien plus compliqué, aujourd'hui les polices de caractères sont vectorielles et non "bitmap" comme ici, mais l'idée reste juste et correspond à un certain passé

une précision pareille ! Probablement à peu près rien. Votre oreille, elle, en a besoin. Voici une illustration de comment le nombre de bits influence la précision, ici sur respectivement 3 bits (soit $2^3 = 8$ niveaux) et 4 bits. Il est clair qu'avec 4 bits on va pouvoir se rapprocher plus du signal réel puisqu'on dispose de plus de niveaux :



Les autres types entiers

Il s'agit de **int**, **long** et **long long**. Ils servent la plupart du temps à faire des calculs (sur des entiers) ou à parcourir des tableaux. Puisqu'un tableau peut contenir un nombre de cases assez grand, on serait limité à des tableaux de 256 cases si on utilisait un **char** (par exemple) pour parler du numéro d'une case. Notez que cela n'a rien à voir avec le type des cases du tableau elles-mêmes : on peut très bien parcourir un très grand tableau de **char** par exemple. Dans ce cas, le contenu des cases sera **char**, mais on utilisera plutôt un **int** (c'est classique) pour le numéro des cases.

III.2 Les autres types de base du C

Le C/C++ peut évidemment gérer les nombres à virgule ("nombres réels"), comme Python. Mais il gère 3 types de nombres à virgule. Les nombres à virgule sont toujours signés :

- les **float**, flottants dits "simple précision", qui ont été très utilisés à une époque car plus rapides à gérer que les **double** (décris ci-dessous). Aujourd'hui ils sont plus ou moins obsolètes. Attention toutefois : ce que Python appelle **float** sont ... les **double** du C/C++. C'est logique dans la mesure où Python est un langage plus moderne, qui ne s'encombre pas d'anciennes considérations. Un **float** en C/C++ utilise 4 octets pour coder un nombre à virgule.
- Les **double**, flottants dits "double précision", qui sont le standard pour le calcul scientifique aujourd'hui, et équivalents au **float** de Python. Ils utilisent 8 octets, et peuvent gérer des nombres positifs ou négatifs dont la valeur absolue entre $\approx 10^{-308}$ et $\approx 10^{+308}$, avec une précision relative de 15 chiffres significatifs (!!). Ça paraît "monstreux" et ça l'est, mais c'est parfois insuffisant. Alors ?
- Alors il y a les **long double**, qui sont utilisés dans des cas spécifiques car ils nécessitent plus de temps de calcul aujourd'hui encore. Ils ne sont pas, à ce jour, normalisés, et leur précision peut varier d'un processeur à un autre, elle est juste garantie au moins égale à celle d'un **double**

En complément, le C/C++ dispose d'un type qui n'a rien à voir, le type **bool** pour "booléen", qui peut valoir uniquement 2 valeurs : **true** ou **false**.

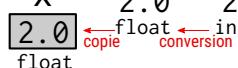
En outre, le C/C++ ne propose pas, par défaut, d'équivalent aux **listes**, **tuples**, etc, de Python.

III.3 Déclaration de variables : typage explicite

Contrairement à Python, en C/C++ une variable ne change **jamais** de type au cours de sa vie. C'est l'un des éléments techniques qui font que le C/C++ est bien plus rapide que Python. C'est ce que nous montrons sur l'illustration ci-dessous, en utilisant un petit code d'exemple :

 python	 C++
<pre> 1 2 3 4 5 6 x=1.5 7 x=2 8 9 print(x) </pre>	<pre> 1 #include<iostream> 2 using namespace std; 3 int main() 4 { 5 float x; 6 x = 1.5; 7 x = 2; 8 cout << x; 9 return 0; 10 } </pre>

Regardons pour les lignes 5,6,7 et 8 ce qui se passe⁴ :

Ligne	python	C++
5		 float x est une "case" mémoire assez grande pour stocker un float (donc 4 octets)
6	 1.5 est dans une nouvelle case mémoire pour "float" que x regarde	 1.5 float x contient maintenant 1.5
7	 1.5 est oublié et une nouvelle case de taille "int" est utilisée x regarde donc un int maintenant !	 2.0 float x contient maintenant 2.0
8	Si on lit la valeur de x, la question se pose de savoir de quel type il est, ici pour adapter l'affichage : ça prend du temps. Python va donc reconnaître que x est un int avant de l'afficher, parce que toutes les variables sont typées en Python.	Ici, le type de x était connu lors de la compilation puisque son type a été déclaré et ne peut pas changer, et donc pendant l'exécution la question ne se pose pas : le code a été compilé spécifiquement pour ce type.

4. Ce qui est décrit est assez simplifié et pas rigoureux mais donne une bonne image mentale des mécanismes en jeu

III.4 Affichage et saisie

On a vu jusqu'à présent comment faire l'affichage d'une variable, avec **cout**. C'est assez simple :

```

1 #include<iostream>
2 using namespace std;
3
4 int main()
{
5     int i;
6     i=9;
7     cout << i;
8
9
10    return 0;
11}
```



[code/code-III-4-4.cpp]

De façon parfaitement symétrique, on peut demander la saisie au clavier d'une variable, en utilisant **cin** et l'opérateur **>>**. Sur un exemple :

```

1 #include<iostream>
2 using namespace std;
3
4 int main()
{
5     int i;
6
7     cin >> i;
8
9     cout << "vous avez tape :" << i << "\n";
10
11    return 0;
12}
```



[code/code-III-4-5.cpp]

On peut faire ça avec tous les types que l'on a vu jusqu'ici sans restriction.

III.5 Conversion de type

On peut réaliser explicitement des opérations de conversion sur des variables en C/C++. Donnons un exemple :

```

1 #include<iostream>
2 using namespace std;
3
4 int main()
{
5     float a = 2.5789;
6
7     cout << "On affiche a normalement : "<< a << "\n";
8     cout << "On affiche a converti en entier : "<< (int) a << "\n";
9     cout << "On affiche encore a normalement : "<< a << "\n";
10
11    return 0;
12}
```



[code/code-III-5-6.cpp]

qui affiche :

```

On affiche a normalement : 2.5789
On affiche a converti en entier : 2
On affiche encore a normalement : 2.5789
```

On s'intéresse aux lignes 8 à 10 :

- Ligne 8, on affiche la variable **a** sans transformation, directement, et elle est affichée avec toutes ses décimales
- Ligne 9, on a ajouté (**int**) devant **a**. C'est cela qui réalise la conversion en **int** (c'est l'équivalent de **int(a)** en Python) et qui fait que la valeur se trouve privée de ses décimales.
- Mais attention : *cela ne change pas le type ou la valeur de a!!!* Il faut comprendre que le type de **a** a été décidé une fois pour toutes quand il a été déclaré, et que l'expression (**int**) **a** est "la valeur de **a** convertie en **int**". C'est ce que prouve la ligne 10.

Il y a un ensemble de situations où les conversions sont indispensables. Typiquement dans certaines situations où on calcule des règles de 3 avec des entiers, ce genre de chose. Donnons le code ci-dessous qui convertit en heures une valeur donnée en minutes :

```

1 #include<iostream>
2 using namespace std;
3
4 int main()
5 {
6     int minutes=50;
7     cout << "convertissons " << minutes << " mn en heures : \n";
8     cout <<    minutes << " mn fait " << minutes/60 << " h.\n";
9
10    return 0;
11 }
```

 [code/code-III-5-7.cpp]

Ce code affiche :

```
convertissons 50 mn en heures :
50 mn fait 0 h.
```

On obtient 0 parce que le nombre de minutes est inférieur à 60, et que les deux valeurs sont entières, donc le C/C++ fait la division entière (contrairement à Python qui a un opérateur dédié nommé `//` pour forcer les divisions en division entière). C'est pas forcément ce que l'on voulait. Alors on a plusieurs solutions : passer les minutes en **double** par exemple, mais peut-être a-t-on d'autres bonnes raisons de vouloir qu'il soient en **int** dans d'autres parties du code non représentées ici. Une autre solution est d'utiliser l'opérateur de conversion (**double**) comme ci-dessous :

```

1 #include<iostream>
2 using namespace std;
3
4 int main()
5 {
6     int minutes=50;
7     cout << "convertissons " << minutes << " mn en heures : \n";
8     cout <<    minutes << " mn fait " << ((double)minutes)/60 << " h.\n";
9
10    return 0;
11 }
```

 [code/code-III-5-8.cpp]

qui affichera :

```
convertissons 50 mn en heures :
50 mn fait 0.833333 h.
```

Notez que ce qui a été converti c'est **minutes**. Ainsi, à gauche du `/` on a un **double**, à droite un **int**, le C/C++ utilise le type le plus général pour faire l'opération : **double**.

Note : On pouvait se passer ici de l'opérateur de conversion en donnant l'expression `minutes/60.0` par exemple.

Un autre exemple pour répondre à un problème que l'on a vu précédemment : la conversion de type

peut nous permettre d'afficher la valeur d'un char plutôt que la lettre à laquelle il correspond dans la table ASCII. De cette façon, on peut écrire un petit code qui affiche un morceau de table ASCII, ici de 32 à 126 (soit les caractères "normaux", non spéciaux) :

```
1 #include<iostream>
2 using namespace std;
3
4 int main()
5 {
6     char lettre=32;
7     while(lettre < 127)
8     {
9         cout << lettre << " " << (int)lettre <<"\n";
10        lettre = lettre + 1;
11    }
12
13 return 0;
14 }
```



[code/code-III-5-9.cpp]

Qui affiche (affichage partiel) :

```
.....
.....
= 61
> 62
? 63
@ 64
A 65
B 66
C 67
D 68
E 69
F 70
G 71
H 72
.....
.....
```

❖ En résumé...

On a vu les types "de base" du Langage C. Mais on a surtout vu que le C/C++ **impose** que l'on déclare les variables par leur type, et c'est une différence forte avec Python, dont la conséquence pratique principale est qu'**une variable ne peut pas changer de type au cours de sa vie**. Cela peut sembler peu de choses, on peut le vivre plutôt comme une sorte de contrainte alors que "Python c'était tellement pratique" (ce qui est vrai). Mais en réalité, il faut comprendre que ce mode de fonctionnement est l'un des ingrédients spécifiques au C/C++ qui fait que les temps de calcul sont grandement réduits. ■

Chapitre IV – Tableaux et chaînes de caractères

⌚ Motivations et objectifs

On va voir ici comment gérer des variables pouvant contenir "plusieurs valeurs" : les chaînes de caractères, qui peuvent contenir plusieurs lettres, et les tableaux **vector**, qui permettent de contenir plusieurs valeurs du type que l'on veut.

IV.1 Les chaînes de caractères "string"

Il ya plusieurs façons de gérer les chaînes de caractères en C et C++. Nous allons voir la façon "C++" (différente de celle du "C", mais bien plus pratique) qui est très proche de Python. Commençons par écrire un tout petit code et décrire ce qu'il signifie :

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main ()
6 {
7     string s1 = "Bonjour !";
8     cout << s1 << "\n";
9
10    return 0;
11 }
```

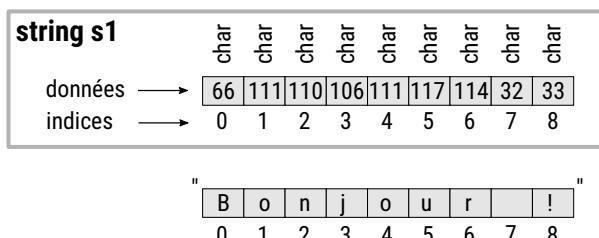
Qui affiche :

Bonjour !

Décrivons rapidement :

- La ligne 2 du code inclut la bibliothèque qui permet de gérer les chaînes de caractères
- La ligne 7 crée une variable **s1** de type **string** (*qui signifie "chaîne de caractères"*) qui contiendra le texte **Bonjour !**.
- La ligne 8 affiche cette chaîne

Plus en profondeur, voici ce que fait exactement la ligne 7 :



Il s'agit donc d'un tableau d'entiers de type **char**, dans lequel chaque case peut prendre une valeur entre 0 et 255. Mais comme c'est une *chaîne de caractères*, les valeurs contenues dans ces cases sont à comprendre par leur correspondance dans la table ASCII. Autrement dit, même si ça ne semble pas évident dans la

syntaxe, la ligne 7 crée en réalité un tableau de **char** assez grand pour contenir le texte donné à droite du =. Et cette opération revient en réalité, en profondeur, à stocker dans chaque case un entier qui correspond au code ASCII de chaque lettre indiquée (voir la table ASCII page 24). *C'est la compréhension profonde qu'il faut avoir de ce qu'est une chaîne de caractères.*

Evidemment ca ne s'arrête pas là. Pour l'illustrer, on montre ci-dessous l'essentiel des opérations possibles sur une chaîne de caractères en C/C++ :

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main ()
6 {
7     string s1 = "Bonjour !";
8     cout << s1 << "\n";
9     s1.resize(8);
10    cout << s1 << "\n";
11
12    s1 = s1 + "a tous !";
13    cout << s1 << "\n";
14    cout << "s1 a maintenant une taille de " << s1.length () << " caracteres
15        .\n";
16
17    s1[10] = 118;
18    cout << s1 << "\n";
19
20    s1[10] = 't';
21    cout << s1 << "\n";
22
23    int k = 10;
24    while (k < 14)
25    {
26        cout << k << " - " << s1[k] << "\n";
27        k = k + 1;
28    }
29
30    return 0;
31 }
```

 [code/code-IV-1-10.cpp]

Qui affiche :

```
Bonjour !
Bonjour
Bonjour a tous !
s1 a maintenant une taille de 16 caracteres.
Bonjour a vous !
Bonjour a tous !
10 - t
11 - o
12 - u
13 - s
```

Expliquons :

- La ligne 2 du code inclut la bibliothèque qui permet de gérer les chaînes de caractères
- La ligne 7 crée une variable **s1** de type **string** (*qui signifie "chaîne de caractères"*) qui contiendra le texte **Bonjour !**.
- La ligne 8 affiche cette chaîne

- La ligne 9 réduit la chaîne à 8 caractères, ce qui revient à supprimer le "!".
- La ligne 12 ajoute du texte à la fin de `s1`. On parle de *concaténation*.
- La ligne 14 utilise `length()` pour obtenir le nombre de caractères.
- La ligne 16 est un peu plus complexe. Il faut comprendre qu'un caractère dans une chaîne de caractères est un `char`. Autrement dit, comme on en a déjà parlé, c'est un entier sur 8 bit. Et donc une chaîne de caractères est juste une série d'entiers, et ces entiers sont à interpréter dans la table ASCII pour en faire du texte. Ce que fait la ligne 16, c'est remplacer le caractère de la ligne 10, c'est à dire le `t` du mot `tous`, par la valeur 118, c'est à dire par le code ASCII du `v` (voir la table ASCII page 24). On note au passage que les chaînes de caractères du C/C++ sont modifiables, contrairement à celles de Python.
- La ligne 19 est similaire, sauf qu'elle montre que l'on peut obtenir la valeur du code ASCII d'une lettre par l'utilisation des guillemets simples. Ainsi, en C/C++ l'expression '`t`' vaut simplement 116, comme on peut le voir dans la table ASCII.
- Enfin, les lignes 22 à 27 parcouruent une sous partie de la chaîne pour extraire les lettres `t-o-u-s`.

Remarque

Remarque 1 : Contrairement à Python, quand on accède à une case avec les `[]`, il n'y a aucune vérification sur le numéro de la case. Si vous cherchez à atteindre une case qui n'existe pas, le C/C++ va simplement aller chercher la case qui correspondrait, en mémoire, si elle appartenait à la chaîne. En effet la chaîne est enregistrée dans une zone de la mémoire, et il y a bien d'autres cases mémoire à la suite, et on peut y accéder, et le C/C++ ne se pose pas plus de question. Notez qu'accéder à une telle case peut aussi aboutir à un plantage du programme ! Ce fonctionnement n'est pas du tout souhaitable pour vous en tant que programmeur, il est périlleux, mais il est le seul qui permette d'obtenir une grande vitesse d'exécution, et c'est le parti pris du C/C++. Le fait que Python vérifie les accès par exemple le ralentit. Et donc, en C/C++ c'est bien à vous de bien faire attention aux indices, le langage ne vous aidera pas.

Remarque 2 : En C "pur et dur", les chaînes de caractères ne se gèrent pas avec des `string` mais avec des `char*`, qui sont bien plus difficiles à gérer. Malgré tout, on peut avoir besoin de repasser en "mode" C dans certaines (rares) circonstances. Et on peut le faire facilement, avec la fonction `c_str()`, comme montré sur l'exemple ci-dessous :

```
string chaineCpp = "Hello";
chaineCpp = chaineCpp + " World !";
char* chaineC = chaineCpp.c_str();
// chaineC est maintenant la chaîne "C pur et dur"
// obtenue à partir de la string C++
```

Remarque 3 : On est loin d'avoir tout vu mais ça suffira. Une note malgré tout : on peut comparer 2 chaînes de caractères dans un `if` avec l'opérateur de comparaison traditionnel `==`.

IV.2 Tableaux : "vector"

Les tableaux en C pur et dur sont un sujet qui peut être difficile et que l'on n'abordera pas. On ne va parler que des tableaux en C++. Tout se passe avec la bibliothèque `vector`. Voici sur un exemple très simple :

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

```

7   vector<int> v1= {1,8,3,5,6};
8   int k;
9   k=0;
10  while(k < v1.size())
11  {
12      cout << v1[k] << " ";
13      k=k+1;
14  }
15
16  return 0;
17 }
```

qui affiche :

1 8 3 5 6

Analysons un peu :

- **ligne 2** : on inclue la bibliothèque pour les tableaux.
- **ligne 7** : on déclare, on crée et on initialise le tableau.
- la fin du code consiste à afficher le contenu du tableau. Il n'existe en effet, en C/C++, pas de façon d'afficher un tableau complet en une seule instruction, on ne peut afficher que chacun de ses éléments. On note l'utilisation de `size()`, qui permet de connaître le nombre de cases du tableau.
- On note aussi l'utilisation de l'opérateur "[...]" du C/C++ à la **ligne 12** (dans l'expression `v1[k]`), qui permet, comme en Python, d'accéder au contenu des cases.

Mais reprenons ce qui se passe ligne 7 :

- La déclaration :

```
vector<int> v1;
```

seule, c'est à dire sans ce qui est à droite du égale de la ligne 7, elle signifie que l'on crée un tableau dont le nom est `v1` dont chaque case sera un `int`. En remplaçant `int` par n'importe quel autre type, on crée un tableau dont chaque case est cet autre type. Le nombre de cases, en absence d'initialisation, est 0 cases, mais on pourra le modifier ultérieurement.

- la partie à droite du égale est une initialisation, qui va faire que `v1` va faire 5 cases, dont le contenu sera, dans l'ordre, 1,8,3,5,6, comme montré ci-dessous :

<code>vector<int> v1</code>					
	1	8	3	5	6
données →					
indices →					

Remarque

Les `vector<...>` sont le type de base pour parler de tableaux en C/C++. Il est analogue au `array` de `numpy` en Python, mais ne dispose pas des actions du type "opérations mathématiques" (somme, division, etc) parce qu'il correspond à un concept de tableau au sens large, pas dédié au calcul mathématique, alors que c'est le cas avec `numpy` de Python. Cela signifie que pour ajouter deux tableaux ensemble, il faut les parcourir avec une boucle.

Aussi, en C/C++, les types de base de Python comme `tuple`, `list`, `dict`, etc, n'existent pas directement. Ils existent via des bibliothèques mais ne sont pas si faciles que ça à manipuler, et pas nécessairement utiles pour ce que l'on veut faire en C/C++. En C/C++, l'essentiel des programmes utilisent de `vector<...>` parce qu'on est en recherche de performance et que c'est ce qu'il y a de plus rapide.

Donnons un exemple plus complet : le calcul de la suite de Syracuse, comme précédemment, mais au lieu d'afficher les valeurs, on les stocke au fur et à mesure dans un tableau, et tout n'est affiché qu'une fois tous les termes calculés :

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main ()
6 {
7     int U;
8     int k;
9
10    U = 27;
11    k = 1;
12    vector<int> valeurs;
13    valeurs.resize(120);
14    valeurs[0] = 27;
15    while (k < 120)
16    {
17        if (U % 2 == 0)
18        {
19            U = U / 2;
20        }
21        else
22        {
23            U = 3*U+1;
24        }
25        valeurs[k] = U;
26        k = k+1;
27    }
28
29
30    k = 0;
31    while (k < 120)
32    {
33        cout << valeurs[k] << " ";
34        k = k+1;
35    }
36
37    return 0;
38 }
```

 [code/code-IV-2-11.cpp]

Notez l'utilisation, ligne 13, de `resize`, qui redimensionne le tableau à la taille que l'on veut. Malgré tout il faut faire attention : exactement comme pour n'importe quelle variable, redimensionner le tableau ne signifie pas qu'il y a des 0 partout. C'est en effet différent de `np.zeros` en Python. Toujours le même point de vue : le C/C++ ne prend pas l'initiative de remplir un tableau avec quelque chose si ça n'a pas d'utilité ensuite (rien ne dit qu'on voudra des 0), dans le but de simplement gagner du temps.

IV.3 Tableaux à 2 dimensions

On peut facilement faire des tableaux à 2 dimensions : il suffit de créer un tableau dont chaque case est elle-même un tableau. Par exemple si on veut créer des matrices, on peut écrire :

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
```

```

5 int main()
6 {
7 vector< vector<double> > M1;
8
9 return 0;
10 }
```

Et voilà. Quelques remarques :

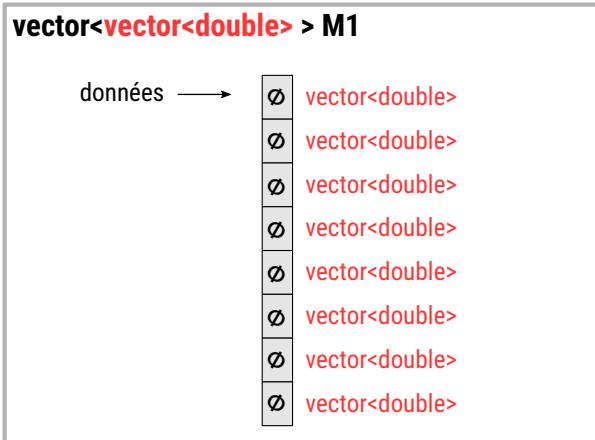
- On a utilisé `vector<double>` comme contenu pour chaque case de `M1`.
- Notez bien l'espace entre les `>` ligne 7, parce que sans l'espace C/C++ confondrait avec l'opérateur `>>` et produirait une erreur de compilation.
- Ici on a fait un tableau à 2 dimensions de 0 cases ... on va améliorer ça.

On va faire un tableau de 8 lignes par 5 colonnes. Commençons par créer les 8 lignes :

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main()
6 {
7 vector< vector<double> > M1;
8 M1.resize(8);
9 return 0;
10 }
```

On obtient :



Notez que dans ce tableau on ne peut rien stocker : tout ce qu'on a fait c'est dire qu'il faut 8 lignes. Continuons en ajoutant les colonnes. Pour cela, il nous faut redimensionner chaque `vector<double>` de `M1` pour qu'il fasse une taille de 5 :

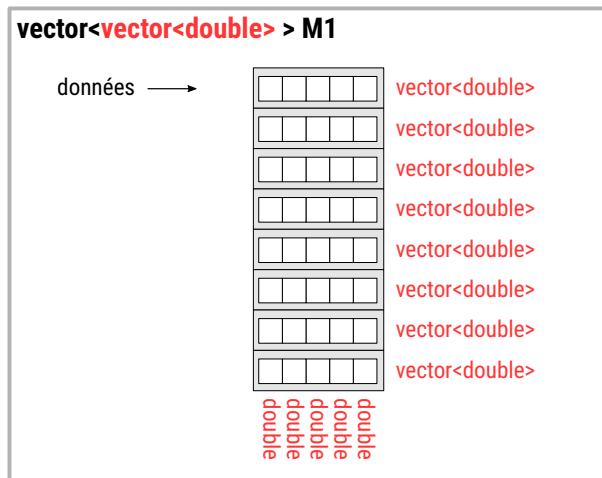
```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main()
6 {
7     vector< vector<double> > M1;
8     M1.resize(8);
9     int ligne = 0;
10    while(ligne < M1.size())
11    {
12        M1[ligne].resize(5);
13        ligne = ligne+1;
14    }
15 }
```

```
16 }  
17 return 0;  
}
```



Ce qui aboutit à :



Et maintenant imaginons que l'on veuille remplir cette matrice avec l'identité, on ajouterait au code :

```
int i, j;  
i=0;  
  
while(i<M1.size())  
{  
    j=0;  
    while(j<M1[i].size())  
    {  
        if (i==j)  
        {  
            M1[i][j]=1;  
        }  
        else  
        {  
            M1[i][j]=0;  
        }  
        j=j+1;  
    }  
    i=i+1;  
}
```



 [code/code-1v-3-13.cpp]

Remarque

On aurait aussi bien inverser le rôle des lignes et colonnes. On aurait créé le tableau de la façon suivante :

```
#include<iostream>
#include<vector>
using namespace std;

int main()
{
    vector< vector<double> > M1;
    M1.resize(5);
    int colonne = 0;
```

```
    while(colonne < M1.size())
    {
        M1[colonne].resize(8);
        colonne = colonne+1;
    }

    return 0;
}
```

 [code/code-IV-3-14.cpp]

Ce qui impose que dans la suite du code, on s'intéresse aux cases en écrivant :

M1[j][i]

au lieu de :

M1[i][j]

Le choix de quelle est la ligne et quelle est la colonne est une question de convention. Pour du calcul mathématique, il est habituel de vouloir le premier indice comme étant la ligne, le second serait la colonne. En imagerie on préfèrera l'inverse. Et donc il n'y a pas de règle. Autrement dit, compte tenu du fait qu'on voulait 8 lignes par 5 colonnes, le premier code est dans le "style" mathématique, le second est dans le "style" imagerie. L'essentiel étant de choisir une convention, peu importe ce qu'elle est, et de rester cohérent avec cette convention dans le reste du code.

❖ En résumé ...

On a vu l'utilisation des **string** et des **vector**, qui sont les façons les plus simples de gérer les chaînes de caractères et les tableaux en C/C++, tout étant d'une immense efficacité. On a vu comment créer des tableaux simples, mais aussi des tableaux plus évolués comme des tableaux à 2 dimensions. ■

Chapitre V – Les Fonctions

⌚ Motivations et objectifs

On montre comment écrire une fonction en C/C++, à partir de ce que vous savez déjà des fonctions en Python. Pour une grande part, les choses sont très similaires entre C/C++ et Python : le concept de fonction est universel en programmation, et elles ont la même utilité. Malgré tout, il y a un ensemble de différences que nous devons regarder.

V.1 Premier survol : similarités et différences avec Python

Le concept de **fonction** existe dans les deux langages et représente les mêmes idées :

- au premier abord : "ranger du code" sous un nom pour pouvoir ensuite l'utiliser en ne mentionnant que ce nom
- au deuxième abord : créer des "boîtes noires" qui permettent de produire des sorties à partir d'entrées.

La syntaxe n'est pas extrêmement éloignée mais le C/C++ a quelques "pièges" et complique un peu les choses. C'est au moins pour une part lié au parti pris du C/C++ d'être un langage rapide. Comparons sur un petit exemple :

 python	 C++
<pre>1 2 3 4 def afficheBonjour(nb): 5 6 k = 0 7 while k<nb: 8 9 print("Bonjour ! ") 10 k=k+1 11 12 13 14 15 afficheBonjour(5) 16 17 # 18 #</pre>	<pre>1 #include <iostream> 2 using namespace std; 3 4 void afficheBonjour(int nb) 5 { 6 int k = 0; 7 while(k<nb) 8 { 9 cout << "Bonjour ! \n"; 10 k=k+1; 11 } 12 } 13 14 int main () 15 { 16 afficheBonjour(5); 17 return 0; 18 }</pre>

Si on se réfère au code python, de façon claire ce code crée une fonction, appelée `afficheBonjour`. Cette fonction prend en paramètre `nb`, qui est le nombre de fois que l'on va afficher "Bonjour !". L'affichage lui même est fait dans une simple boucle. Plus loin, on fait appel à la fonction (ligne 16) en imposant la valeur 5 pour `nb`.

Le code C/C++ fait exactement la même chose et aux mêmes lignes. Mais comme on le voit il y a des nuances :

- D'abord : **le mot def n'existe pas en C/C++**. Sur cet exemple, on pourrait avoir l'impression que `def` est remplacé par `void` en C/C++ *c'est complètement faux!*. `void` signifie autre chose, et nous en discuterons plus loin.
- Ensuite, toujours ligne 4 : comme en C/C++ il faut déclarer le type de toute variable en indiquant son type, et donc forcément **il faut indiquer le type des paramètres d'entrée de la fonction** ! sur cet exemple, il donc faut déclarer le type de `nb`.
- Le contenu de la fonction n'est pas indiqué par une tabulation comme en Python mais par des accolades. Ca on a l'habitude, puisque c'est ce que l'on fait déjà pour `while` et `if`.
- Et pour la suite il y a une grande différence : Python, si on peut placer des lignes de code en dehors de blocs : cela signifie que la ligne appartient au programme principal. En C/C++, ceci est **totalelement impossible!**. En effet, **toute opération ou appel de fonction en C/C++ doit appartenir à une fonction** ! Et ici, ça commence à se voir, le `int main()` que l'on "traine" depuis le début n'est rien de plus que la définition d'une fonction, appelée `main`, qui signifie "principal" en anglais. Autrement dit, l'exécution du programme commence forcément à l'intérieur de la fonction `main`, quoi qu'on écrive avant. Autrement dit, en C/C++, le programme principal est complètement identifié et rassemblé à un seul endroit. En Python, il faut parcourir l'ensemble du script pour comprendre ce qu'est le programme principal.
- L'appel de fonction, ligne 16, est très similaire à ce que l'on fait avec Python, il consiste à fixer les valeurs des paramètres.
- Enfin, on note dans le `main` un `return`, qui évoque le `return` de Python. Et en effet ça signifie plus ou moins la même chose, mais son utilisation n'est pas aussi souple qu'en Python. On va y revenir

Donc rien de trop bizarre dans ce premier survol, à part ce `void` que l'on n'a pas justifié, et que la fonction `main()` ne contient pas (alors que c'est une fonction), et ce `return 0` que l'on n'a jamais justifié depuis le début mais qui fait quand même penser aux fonctions en Python.

V.2 Et `return`?

Rappel :

Faisons une analogie entre math et python :

Math	python
$f(x,y) = \begin{cases} \sin(xy) & \text{si } x > 3 \\ \exp(x)\exp(y) & \text{sinon} \end{cases}$	 python <pre> import math as m def f(x,y): if x>3: val = m.sin(x*y) else: val = exp(x)*exp(y) return val </pre>

Dans ce code python, comme dans cette expression mathématique, on **définit** une fonction. On dit :

- que son nom est f ,
- que les paramètres d'entrée sont x et y ,
- ce qu'elle fait (ici ce qu'elle calcule),
- et quel est son résultat

En Python, le résultat est identifié par **return** : c'est la dernière ligne de la fonction ici. Mais qu'est ce que cela signifie "son résultat" ? Pour bien comprendre cela, il faut utiliser la fonction. En effet, jusqu'ici, on a juste **défini** la fonction f , que ce sont côté Math ou côté Python : on a dit comment elle se nomme, ce qu'elle fait, etc. Mais on ne l'a pas utilisé. Alors regardons côté Math et côté Python comment on utilise la fonction :

Math	 python
$z = f(5, 8)$	<code>z = f(5, 8)</code>

C'est exactement pareil. Ce qui se passe ici, c'est que l'on fixe les valeurs de x et y , pour que la fonction puisse vraiment faire un calcul, et que l'on récupère le résultat de ce calcul dans une nouvelle variable z . Et cela se nomme "**faire appel à la fonction**". Et ce que l'on veut montrer ici, c'est que c'est grâce au **return** de la fonction `f` que Python a su que le résultat à communiquer à la variable `z`, à l'extérieur de `f`, était la valeur de la variable `var` de Python. Ainsi, `z` va simplement récupérer la valeur de `var` de Python.

Et en C/C++?



Important

En C/C++, **return** peut s'utiliser comme en Python MAIS avec une limite très forte : on ne peut avoir qu'une seule sortie, et pas plusieurs. Pour l'instant, on va se limiter à des cas avec une seule sortie en C/C++, mais nous verrons dans un chapitre plus loin comment obtenir malgré tout plusieurs sorties. ■

Procédons par analogie avec Python sur un petit exemple qui calcule une approximation¹ de la valeur de π :

1. Cette façon de calculer π est l'une des plus anciennes et converge lentement. Voir l'article Wikipedia https://fr.wikipedia.org/wiki/Approximation_de_%CF%80 pour en savoir plus sur ce sujet. Aussi, vous pouvez essayer de programmer cette autre série (*formule de John Machin*) : $S_n = 4 \sum_{k=0}^n \frac{(-1)^k}{2k+1} \left(\frac{1}{5^{2k+1}} - \frac{1}{239^{2k+1}} \right)$ et constaterez qu'elle converge beaucoup plus vite. Autrement dit : à temps de calcul constant, elle permet d'obtenir plus de décimales de π .

 python	
<pre> 1 2 3 4 5 def calcPi(nb): 6 7 valPi = 0 8 9 for k in range(0, nb): 10 11 valPi = valPi + ((-1)**(12 k))/(2*k+1) 13 14 return valPi*4 15 16 17 18 19 pi1 = calcPi(1) 20 pi2 = calcPi(100) 21 pi3 = calcPi(100000) 22 print(pi1, "\n", pi2, "\n", pi3) 23 pi4 = calcPi(30000000) 24 print(pi4) 25 26 27 # </pre> <p> [code/code-V-2-2.py]</p>	<pre> 1 #include <iostream> 2 #include <cmath> 3 using namespace std; 4 5 double calcPi(int nb) 6 { 7 double valPi = 0; 8 int k = 0; 9 for(k=0; k<nb; k=k+1) 10 { 11 valPi = valPi + pow 12 (-1,k)/(2*k+1) 13 } 14 return 4*valPi; 15 16 int main() 17 { 18 double pi1,pi2,pi3,pi4; 19 pi1 = calcPi(1); 20 pi2 = calcPi(100); 21 pi3 = calcPi(100000); 22 cout <<pi1 <<"\n" <<pi2 << 23 "\n" << pi3 <<"\n"; 24 pi4 = calcPi(30000000); 25 cout <<pi4; 26 return 0; </pre> <p> [code/code-V-2-15.cpp]</p>

Quelques commentaires :

- D'abord, ca n'est pas vraiment "le sujet" de ce que l'on montre ici, mais autant en parler. En C/C++, on la déjà dit, il n'y a pas d'équivalent à l'opérateur `**` de Python. Pour calculer x^y en C/C++, il faut utiliser la fonction `pow` de la bibliothèque `cmath`. C'est ce que l'on voit au niveau des inclusions (ligne 2) et au niveau de la ligne 11, dans la formule mathématique.
- Ensuite, bien entendu le code C/C++ est plus rapide, environ 15 fois.
- Maintenant venons-en au sujet qui nous interesse. On voit que cette fois ci, dans la traduction en C/C++ de `calcPi`, il y a un `return` ligne 13. Et donc oui, `return` existe et sert à faire le même genre de choses qu'en C/C++. Mais avec une limite forte : **return en C/C++ ne peut transmettre qu'une seule sortie, et pas plusieurs**. Et c'est une limite du langage que nous apprendrons à résoudre dans la partie 2 de ce document.
- Comme en C/C++ tout est typé, le C/C++ a besoin de connaître le type du résultat transmis par `return`. Ca n'a rien de surprenant. Et cela on le fait à gauche du nom de la fonction. Si on regarde ligne 5, en effet, à gauche de `calcPi`, on a écrit `double` et pas `void`. C'est dû au fait que l'on renvoie `4*valPi`, et que le résultat de cette expression est un `double` puisque `valPi` est un `double`. Et il faut reporter cela à gauche du nom de la fonction.
- Enfin, au niveau du `main`, on a créé des variables `pi1`, `pi2`, `pi3` et `pi4` pour recevoir les différents résultats de `calcPi`. Et tous sont de type `double`, parce que justement la fonction est indiquée comme renvoyant un `double`.



Important

Il y a 4 points fondamentaux dont il faut se souvenir avec `return` :

1. `return` ne permet de faire sortir qu'un seul résultat d'une fonction.
2. Jusqu'ici, nous avions systématiquement écrit `void` à gauche du nom d'une fonction que l'on définit, parce que nos fonctions n'utilisaient pas `return` et c'est la seule raison !!! En effet, répétons le, **void n'est pas du tout l'équivalent C/C++ de def en Python !** Il signifie seulement que la fonction n'utilise pas `return` et donc ne renvoie rien !
3. En conséquence, pour le C/C++, si la fonction utilise `return`, il faut aussi indiquer le type de l'expression utilisée par `return` à la place du `void` habituel.
4. Enfin, pour faire appel à la fonction et récupérer son résultat, il faut créer une variable qui correspond au type renvoyé par la fonction.

■

V.3 Un exemple

On va traiter à travers un exemple une méthodologie pour créer une nouvelle fonction en C/C++. On repart d'un exemple donné en §IV.2 :

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main ()
6 {
7     int U;
8     int k;
9
10    U = 27;
11    k = 1;
12    vector<int> valeurs;
13    valeurs.resize(120);
14    valeurs[0] = 27;
15    while (k < 120)
16    {
17        if (U % 2 == 0)
18        {
19            U = U / 2;
20        }
21        else
22        {
23            U = 3*U+1;
24        }
25        valeurs[k] = U;
26        k = k+1;
27    }
28
29
30    k = 0;
31    while (k < 120)
32    {
33        cout << valeurs[k] << " ";
34        k = k+1;
35    }
36
```

```

37     return 0;
38 }

```



[code/code-V-3-16.cpp]

On se propose de séparer le code en fonctions :

1. Une première fonction qui réalise le calcul, avec comme **entrées** la valeur du premier terme U_0 et le nombre de termes n , avec comme **sor tie** le tableau des valeurs,
2. Une seconde capable d'afficher le tableau de valeurs,
3. Un programme principal.

La fonction de calcul

Commençons par la fonction qui réalise le calcul. Appelons la **calcSyraccuse**, parce qu'elle calcule l'évolution de la suite de Syracuse :

```

1 vector<int> calcSyraccuse(int U0, int n)
2 {
3     int U=U0;
4     int k=1;
5     vector<int> valeurs;
6     valeurs.resize(n);
7     valeurs[0] = U0;
8     while (k < valeurs.size())
9     {
10         if (U % 2 == 0)
11         {
12             U = U / 2;
13         }
14         else
15         {
16             U = 3*U+1;
17         }
18         valeurs[k] = U;
19         k = k+1;
20     }
21     return valeurs;
22 }

```



[code/code-V-3-17.cpp]

Analysons un peu :

- d'abord regardons la première ligne :

```
|vector<int> calcSyraccuse(int U0, int n)
```

Cette ligne est appenée "**prototype**" de la fonction. Elle résume ce que seront ses entrées et son éventuelle sortie. Juste en lisant cette ligne et sans même regarder l'ensemble du code, on sait que la fonction est paramétrée par 2 valeurs (**U0** et **n**, c'est normal pour une suite !), que ces deux paramètres sont des **int**, que la fonction s'appelle **calcSyraccuse**, et qu'elle produira un résultat (parce qu'il n'y a pas marqué **void**), et que ce résultat sera un tableau d'entiers. C'est donc très informatif !

- Les lignes 5 à 20 sont une copie exacte des lignes 12 à 27 du code du programme complet.
- La ligne 3 est équivalente aux lignes 7 et 10 du code précédent, mais avec une adaptation : on ne commence pas forcément avec **U0=27** mais avec le **U0** que l'on veut, puisqu'il pourra être donné à la fonction comme paramètre.
- Enfin, la ligne 21 contient le return sur la variable **valeurs**. Et il faut regarder les choses ainsi : Le type de **valeurs** doit **ABSOLUMENT** correspondre au type indiqué à gauche du prototype à la ligne 1. C'est normal puisque cette partie du prototype indique le type du résultat. Vérifions le : le type de **valeurs** est donné ligne 5, et c'est bien **vector<int>**. Tout va bien.

La fonction d'affichage

Le code de la fonction d'affichage est donné ci-dessous :

```

1 void afficheVector(vector<int> T)
2 {
3     int k=0;
4     cout << "Valeurs du tableau : \n";
5     while( k < T.size() )
6     {
7         cout << T[k] << "\n";
8         k=k+1;
9     }
10    cout << "\n";
11 }
```

 [code/code-V-3-19.cpp]

Regardons cela :

- D'abord le prototype :

```
|void afficheVector(vector<int> T)
```

Ce prototype indique que la fonction va prendre en paramètre un `vector<int>`, qui s'appellera `T` à l'intérieur de la fonction. Il indique que la fonction s'appelle `afficheVector`, mais aussi que **la fonction ne renvoie aucun résultat**, et c'est, encore une fois, le sens qu'il faut donner à `void`.

- Pour le reste, le code est un copier/coller exact des lignes 30 à 35, à une légère exception : on a du déclarer la variable `k`. En effet, dans le `main`, on avait une seule variable `k` qui servait aux deux sous-parties (le calcul d'abord lignes 15—27, l'affichage ensuite, lignes 30—36). Mais maintenant que `k` est dans une fonction, il devient une **variable locale** à la fonction, ce qui nécessite de le déclarer. Il ne faut pas se tromper : la variable `k` de `calcSyraccuse` et celle de `afficheVector` ne sont pas la même variable !. Chacune est locale à sa fonction, chacune occupe un espace mémoire qui lui est propre, elles sont indépendantes. Elles portent juste le même nom, localement : l'une est "le `k` de `calcSyraccuse`", l'autre est "le `k` de `afficheVector`".

Le programme complet

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 vector<int> calcSyraccuse(int U0,int n)
6 {
7     int U=U0;
8     int k=1;
9     vector<int> valeurs;
10    valeurs.resize(n);
11    valeurs[0] = U0;
12    while ( k < valeurs.size() )
13    {
14        if (U % 2 == 0)
15        {
16            U = U / 2;
17        }
18        else
19        {
20            U = 3*U+1;
21        }
22        valeurs[k] = U;
23        k = k+1;
24    }
25    return valeurs;
```

```

26 }
27
28 void afficheVector(vector<int> T)
29 {
30     int k=0;
31     cout << "Valeurs du tableau : \n";
32     while( k < T.size() )
33     {
34         cout << T[k] << "\n";
35         k=k+1;
36     }
37     cout << "\n";
38 }
39
40 int main()
41 {
42     vector<int> resultat;
43     resultat = calcSyraccuse(27,120);
44     afficheVector(resultat);
45     return 0;
46 }
```



[code/code-V-3-21.cpp]

Dans ce programme complet on voit la définition de 3 fonctions :

- **calcSyraccuse** (ligne 5),
- **afficheVector** (ligne 28),
- **main** (ligne 40), c'est à dire là où l'exécution débutera.

Il n'y a rien d'évident qui permet d'identifier que ce sont des définitions de fonction, c'est une difficulté du C/C++. Il faut être attentif et regarder si on trouve une séquence du type :

```

| typederetour nom(type1 nom1, type2 nom2, type3 nom3 ...)
| {
| ....
| }
```

Ensuite, le code est structuré avec les fonctions en premier et la fonction **main** à la fin. Ce n'est pas un hasard et on y reviendra : pour l'instant ayez en tête que le **main** doit être à la fin. C'est incomplet de penser les choses ainsi mais pour l'instant ça peut suffire.

Enfin, regardons les lignes 43 et 44 :

- La ligne 43 récupère le résultat produit par **calcSyraccuse** : c'est ce que dit le **resultat=**, ça fonctionne comme en Python. Mais en C/C++, à cause du typage, il faut vérifier que c'est cohérent. La cohérence ici tient dans le fait que **resultat** du **main** est du type **vector<int>** (c'est écrit ligne 41), et que ligne 5 (qui contient le prototype de **calcSyraccuse**) on a bien indiqué que le résultat de **calcSyraccuse** produit un résultat qui est un **vector<int>**. Et au passage, rappelons que la fonction **calcSyraccuse** renvoie **valeurs** qui est aussi du même type. Tout ça n'est pas une coïncidence, c'est indispensable : pour que **resultat** du **main** puisse récupérer **valeurs** de **calcSyraccuse**, il faut que les deux soient du même type.
- La ligne 44 est plus simple à analyser mais malgré tout : le fait qu'il y ait écrit **void** à gauche dans le prototype de **afficheVector** (ligne 28) fait que écrire quelque chose du genre **a=afficheVector(resultat);** est impossible : ce **void** signifie que la fonction ne produit aucun résultat et donc, dans le **main**, il est impossible de récupérer un résultat.

V.4 Un petit Quiz

On propose ici un ensemble de prototypes de fonctions. Les fonctions elles-mêmes ne sont pas écrites. On indique ensuite des appels à cette fonction. Dire lesquels sont corrects et lesquels ne le sont pas :

Question 1 : double cos(double x);

- a) double y;
y=cos(5.0);
- b) cos(5.0);
- c) double y;
y = cos(5.0,8.2);
- d) int y;
y=cos(5.0);
- e) double y;
y = cos();

Question 2 : void disp(string s);

- a) string M="Hello";
disp(M);
- b) string M="Hello";
int k;
k = disp(M);
- c) string M="Hello";
void k;
k = disp(M);
- d) int k = 2;
disp(k);
- e) string M="Hello";
void disp(string M);

Question 3 : vector<float> calc();

- a) vector<float> T;
calc(T);
- b) vector<float> T;
T = calc(T);
- c) vector<float> T;
T = calc();
- d) vector<int> calc();
- e) vector<float> calc(void);

Question 4 :

float calc2(vector<float> R, int n);

- a) vector<float> T;
float calc2(R,2);
- b) vector<float> T;
float m;
m = calc2(2,T);
- c) vector<float> T;
float m;
m = calc2(T,3);
- d) vector<float> T;
calc2(T,3);
- e) vector<float> T;
float m;
m, T=calc2(T,3);

Corrigé Question 1 :

- a) **VRAI** c'est la meilleure réponse : **cos** prend bien un double (ici 5.0) en paramètre. Au sujet du résultat, **cos** renvoie un **double**, qui va être reçu par **y** qui est aussi un **double**. C'est donc parfait
- b) **POSSIBLE** mais pas intéressant : on fait calculer le cosinus avec le bon type, mais on ne récupère jamais le résultat. Ca ne sert à rien avec une fonction comme cosinus.
- c) **IMPOSSIBLE** parce que la fonction **cos** ne prend qu'un seul paramètre, comme on le voit sur le prototype
- d) **POSSIBLE** mais moyennement intéressant : le cosinus renvoie une valeur entre 0 et 1. Comme ici **y** est un **int**, on va recevoir la plupart du temps 0 au lieu de la valeur du cosinus (on perd les chiffres après la virgule). Ca pourrait être souhaité et volontaire, mais a priori ça semble surtout tordu.
- e) **IMPOSSIBLE** parce que la fonction **cos** ne prend pas zero mais un paramètre.

Corrigé Question 2 :

- a) **VRAI** c'est la seule bonne réponse : `disp` prend bien un `string` en paramètre, et on n'a pas de résultat à récupérer.
- b) **IMPOSSIBLE** parce que `disp` n'a pas de résultat (le `void` du prototype), et que l'on cherche à récupérer son résultat dans `k`.
- c) **IMPOSSIBLE** pour deux raisons : parce qu'on ne peut pas déclarer une variable de type "vide" ou "sans type" (le `"void"` devant `k`). Et si on pouvait déclarer une telle variable, `disp` n'a de toute façon pas de résultat, et la syntaxe `k=` devant `disp` n'est pas permise
- d) **IMPOSSIBLE** parce le seul paramètre que `disp` prend est un `string` alors qu'on lui donne un `int` (qui est `k`).
- e) **IMPOSSIBLE** parce qu'il ne faut pas redonner les types des variables pendant l'appel. Ce qui est écrit là ressemblerait plutôt à un prototype de fonction, pas à un appel de fonction.

Corrigé Question 3 :

- a) **IMPOSSIBLE** : la fonction `calc` ne prend pas de paramètre.
- b) **IMPOSSIBLE** : la fonction `calcne` prend pas de paramètre, même si elle a bien un `vector<float>` comme résultat.
- c) **VRAI** c'est bien la bonne réponse, la fonction renvoie bien un `vector<float>`, et le résultat sera récupéré par la variable `T`
- d) **FAUX** Ce n'est pas un appel de fonction mais une déclaration de prototype.
- e) **FAUX** C'est exactement la même chose que d) : c'est une déclaration de prototype. La nuance c'est le `void` entre parenthèses. Il signifie "rien", autrement dit la fonction ne prend pas de paramètre.

Corrigé Question 4 :

- a) **IMPOSSIBLE** : On ne doit pas répéter le type de retour lors de l'appel de fonction. De plus, la variable `R` n'existe nulle part dans le contexte de cet appel de fonction
- b) **IMPOSSIBLE** : Le premier paramètre de la fonction `calc2` n'est pas un `n` est pas un `int` mais un `vector<float>`, et le deuxième n'est pas un `vector<float>` mais un `int`. Cet appel donne l'impression que celui qui a écrit cette ligne a inversé les deux paramètres.
- c) **VRAI** c'est bien la bonne réponse, la fonction renvoie bien un `float`, qui sera récupéré par `m` qui est bien un `float` elle aussi, et les paramètres sont du bon type et dans le bon ordre.
- d) **POSSIBLE** C'est le même appel que la c), sauf qu'on ne récupère pas le résultat. Pourquoi pas, tout dépend de ce que fait précisément la fonction `calc2`. Par exemple peut-être qu'elle affiche des choses et qu'on a juste besoin de cet affichage, mais pas du résultat.
- e) **FAUX** contrairement à Python, on ne peut pas récupérer deux résultats avec une fonction en C/C++.

En résumé ...

On a vu que pour une grande part, les fonctions C/C++ fonctionnent comme les fonctions Python. Il y a malgré tout une nuance importante quand on veut avoir des **sorties**. Si avec un code Python on peut avoir plusieurs résultats sans difficulté, pour l'instant, en C/C++, on ne sait transmettre qu'un seul résultat avec une fonction. On verra plus loin qu'il faut user de techniques spécifiques pour résoudre ce problème.

Enfin, on a vu (superficiellement) comment organiser un code C/C++ comportant plusieurs fonctions, en combinant lisibilité et besoins du compilateur. ■

Chapitre VI – Structures de données

⌚ Motivations et objectifs

On a vu que le langage C/C++ peut être enrichi au niveau des fonctions : il vient avec un ensemble de fonctions de base importées à partir des bibliothèques, et l'on peut en créer autant que l'on veut, selon les besoins. Nous allons voir ici comment enrichir le Langage C/C++ au niveau des **types**. En effet, si le langage C/C++ est fourni avec quelques types de base (**char**, **short**, **double**, **long**, **int**, ...), il offre également la possibilité de créer vos propres types de données, grâce au concept de **structure**.

VI.1 Contexte

Les structures vont permettre de regrouper en tant qu'une seule variable l'ensemble des caractéristiques d'un concept à décrire. Dit comme ça c'est sûrement un peu abstrait, alors nous allons illustrer l'idée qui est derrière sur quelques exemple.

VI.2 Un premier exemple : les Nombres Complexes

En Langage C "standard", il n'existe pas de type "nombre complexe". Si nous avons des opérations à faire sur des nombres complexes, il faut impérativement créer une variable pour la Partie Réelle, une pour la Partie Imaginaire, et tenir compte, lorsque l'on écrit un calcul, le traitement du "nombre imaginaire" (noté i en mathématiques, j en physique).

Prenons un exemple : on veut effectuer le calcul $Z_1 \times Z_2$. Avec ce que nous savons, la seule solution est¹ :

```
1 #include <iostream>
2 using namespace std;
3 int main(void)
4 {
5     double R1,I1,R2,I2,R3,I3; /*parties réelles et imaginaires des
6                                2 nombres ainsi que du résultat*/
7
8     cout << "Partie Réelle du premier nombre ?";
9     cin >> R1;
10    cout << "Partie Imaginaire du premier nombre ?";
11    cin >> I1;
12    cout << "Partie Réelle du deuxième nombre ?";
13    cin >> R2;
14    cout << "Partie Imaginaire du deuxième nombre ?";
15    cin >> I2;
```

1. En effet, au sujet du produit, si on développe :

$$(R_1 + i \times I_1) \times (R_2 + i \times I_2) = \underbrace{R_1 R_2 - I_1 I_2}_{\text{Partie réelle du résultat}} + i \underbrace{(R_2 I_1 + R_1 I_2)}_{\text{Partie imaginaire du résultat}}$$

```

17 R3=R1*R2-I1*I2;
18 I3=R2*I1+R1*I2;
19
20 cout << "Le Resultat vaut : "<< R3<<"+"<< I3 << "\n";
21
22 return 0;
23 }
```



[code/code-VI-2-23.cpp]

Le défaut de cette façon d'écrire le code est que l'on ne voit pas que l'on manipule des nombres complexes : à part le nom des variables rien n'associe R1 et I1 comme formant un seul et unique nombre complexe. Il serait préférable d'avoir trois variables Z1, Z2 et Z3 comprenant chacune une partie réelle et une partie imaginaire par exemple.

En C/C++, on peut imposer ce type d'association avec une **structure**, comme suit :

```

typedef struct
{
    double Re;
    double Im;
} complexe;
```

Que signifie cela ? Cela signifie que l'on peut, dès maintenant, parler de **complexe** comme un nouveau type, c'est à dire que l'on peut créer de nouvelles variables dont le type sera **complexe**. De plus, un **complexe** sera défini comme étant une variable "composite", contenant deux **double**, un pour la partie réelle, un pour la partie imaginaire.

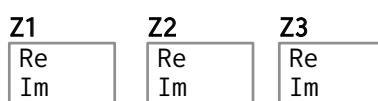
Ainsi, une fois la structure écrite, on peut écrire :

```

int main()
{
    complexe Z1, Z2, Z3;

    return 0;
}
```

Et quand on écrit ça, automatiquement, on a créé 3 variables Z1, Z2 et Z3, chacune correspondant au modèle défini par la structure **complexe**. C'est à dire que chacune de ces 3 variables contient son propre **Re** et son propre **Im**. On peut s'en faire la représentation suivante :



Pour chacune de ces variables, on peut récupérer la partie réelle et la partie imaginaire avec l'opérateur `".".` Et avec ça, on peut réécrire le code précédent :

```

1 #include <iostream>
2 using namespace std;
3
4 typedef struct
5 {
6     double Re;
7     double Im;
8 } complexe;
9
10 int main(void)
11 {
12     complexe Z1, Z2, Z3;
13
14     cout << "Partie Reelle du premier nombre ?";
15     cin >> Z1.Re;
16     cout << "Partie Imaginaire du premier nombre ?";
```

```

17 cin >> Z1.Im;
18 cout << "Partie Reelle du deuxieme nombre ?";
19 cin >> Z2.Re;
20 cout << "Partie Imaginaire du deuxieme nombre ?";
21 cin >> Z2.Im;
22
23 Z3.Re = Z1.Re*Z2.Re - Z1.Im*Z2.Im;
24 Z3.Im = Z2.Re*Z1.Im + Z1.Re*Z2.Im;
25
26 cout << "Le Resultat vaut : "<< Z3.Re <<"+" << Z3.Im << "\n";
27
28 return 0;
29 }
```

 [code/code-VI-2-24.cpp]

On note des séquences du type **Z1.Re**, utilisant l'opérateur **".**", un peu partout dans le code, qui signifient, sur cet exemple, que l'on va chercher la sous partie **Re** du complexe **Z1**.

Remarque

On aurait été tenté d'écrire des choses comme :

```

    cin >> Z1; // a la place des lignes 8 et 10
    cout << Z3; // a la place de la ligne 19
    Z3 = Z1*Z2; // A la place des lignes 16 et 17
```

Mais c'est un mauvais réflexe. En effet, ca n'est pas parce que l'on définit **complexe** comme étant l'association de deux **double** que le C/C++ "prend conscience" d'un seul coup de ce qu'est, mathématiquement, un complexe, ni de quelle façon il faut l'afficher. Et il faut donc traiter chaque élément de la structure séparément.

VI.3 Quelques autres exemples de structures

On peut répéter ce principe avec beaucoup de concepts. On donne quelques exemples :

Date et heure

Supposons que l'un veuille représenter une date avec l'heure. On pourrait écrire les choses ainsi :

```

typedef struct
{
    int jour,mois,annee,heure;
    float secondes;
}dateHeure;
```

On a choisi ici des entiers pour tout sauf pour les secondes, par exemple parce qu'on voudrait aussi les millisecondes, etc.

Point dans un espace à 3 dimensions

Un point dans espace à 3 dimensions c'est 3 coordonnées : *x*, *y* et *z* :

```

typedef struct
{
    double x,y,z;
}point3D;
```

Fiche "contact"

Si on veut représenter un "contact", on peut avoir besoin de choses comme son nom, son prénom, sa date de naissance, son adresse, son téléphone, son email, etc. On peut écrire ça comme ça :

```
typedef struct
{
    string nom;
    string prenom;
    dateHeure naissance;
    string adresse;
    string tel;
    string mail;
}contact;
```

On a utilisé des chaînes de caractères pour un peu tout, y compris pour le téléphone parce que le téléphone peut prendre des formes très variées (même si on pense que c'est juste un ensemble de chiffres) : le fait que sa forme change beaucoup d'un pays à l'autre, ou les conventions internationales avec par exemple "+33" pour appeler en France à partir de l'étranger, etc, tout ça fait qu'on ne peut pas représenter ça par quelques entiers. La seule exception c'est la date de naissance, on a choisi de réutiliser la structure **dateHeure** définie plus haut.

VI.4 Tableaux de structures

Une fois qu'une structure, par exemple **complexe**, est définie, on peut l'utiliser à la manière de n'importe quel autre type du C/C++. Par exemple on peut créer des tableaux de structures, comme suit :

```
vector<complexe> T1;
T1.resize(10);

T1[2].Re = 5;
T1[2].Im = -3;
```

Ce code :

- Crée un tableau de 10 **complexe**,
- modifie le 3ième complexe du tableau pour lui attribuer la valeur $5 - 3i$.

Ce qui revient à dire que la situation à la fin du programme est celle-ci :

T1[0]	T1[1]	T1[2]	T1[3]	T1[4]	T1[5]	T1[6]	T1[7]	T1[8]	T1[9]
Re : ??	Re : ??	Re : 5	Re : ??						

Im : ??	Im : ??	Im : -3	Im : ??						
---------	---------	---------	---------	---------	---------	---------	---------	---------	---------

VI.5 Fonctions et structures

De la même façon que l'on peut utiliser une structure à la place de n'importe quel type pour créer un tableau, on peut passer ou renvoyer une structure à une fonction comme n'importe quel type, que ce soit en entrée ou en sortie. Ainsi, imaginons que l'on veuille maintenant mettre le code de la multiplication des nombres complexes dans une fonction par exemple. Remarquons d'abord qu'une fonction qui réalise la multiplication de deux nombres complexes a 2 entrées (les complexes à multiplier) et une sortie (le complexe qui est le résultat de cette multiplication). Voici ce que ça donne sur un petit exemple :

```
1 #include <iostream>
2 using namespace std;
3
4 typedef struct
5 {
6     double Re;
7     double Im;
8 } complexe;
9
10 complexe produitCplx(complexe A, complexe B)
11 {
12     complexe C;
```

```

13     C.Re = A.Re*B.Re - A.Im*B.Im;
14     C.Im = B.Re*A.Im + A.Re*B.Im;
15
16     return C;
17 }
18
19 void afficheCplx(complexe A)
20 {
21     char signeIm = '+';
22     if(A.Im <0) {
23         signeIm = '-';
24         A.Im = -A.Im;
25     }
26     cout << A.Re << signeIm << A.Im << "i\n";
27 }
28
29 int main()
30 {
31     complexe Z1,Z2,Z3;
32     Z1.Re = 7;
33     Z1.Im = 5;
34     Z2.Re = -2;
35     Z1.Im = 3;
36     Z3 = produitCplx(Z1,Z2);
37     afficheCplx(Z3);
38 }
```

 [code/code-VI-5-25.cpp]

VI.6 Organisation d'un code contenant des structures et des fonctions

En matière de "style d'écriture du code" on préfèrerait avoir le **main** le plus haut possible dans le code, simplement parce que, comme c'est là que démarre l'exécution du programme, on a envie de le lire en premier quand on regarde un programme existant. Pour atteindre ce résultat en C/C++, il faut satisfaire les besoins du compilateur. Pour cela, il est préférable d'organiser le code comme suit :

1. Au début les inclusion de toutes bibliothèques utiles,
2. Ensuite les définitions de toutes les structures nécessaires,
3. Puis viennent les prototypes des fonctions
4. Puis le **main**
5. Enfin la définition des fonctions correspondant aux prototypes écrit au point 3.

Voici ce que cela donne par exemple sur le code précédent :

```

1 // 1. Bibliotheques
2 #include <iostream>
3 using namespace std;
4
5 //2. Structures
6 typedef struct
7 {
8     double Re;
9     double Im;
10 } complexe;
11
12 // 3. Prototypes
13 complexe produitCplx(complexe A, complexe B);
14 void afficheCplx(complexe A);
```

```

15
16 //4. main
17 int main()
18 {
19     complexe Z1,Z2,Z3;
20     Z1.Re = 7;
21     Z1.Im = 5;
22     Z2.Re = -2;
23     Z2.Im = 3;
24     Z3 = produitCplx(Z1,Z2);
25     afficheCplx(Z3);
26 }
27 //5. Definition des fonctions
28 complexe produitCplx(complexe A, complexe B)
29 {
30     complexe C;
31     C.Re = A.Re*B.Re - A.Im*B.Im;
32     C.Im = B.Re*A.Im + A.Re*B.Im;
33
34     return C;
35 }
36
37 void afficheCplx(complexe A)
38 {
39     char signeIm = '+';
40     if(A.Im <0) {
41         signeIm = '-';
42         A.Im = -A.Im;
43     }
44     cout << A.Re << signeIm << A.Im << "i\n";
45 }
```

 [code/code-VI-6-26.cpp]

En organisant le code ainsi, le compilateur a toutes les informations nécessaires au fur et à mesure de la compilation, et le `main` est "aussi haut" qu'il le peut dans le code.

❖ En résumé ...

Nous avons vu un concept du C/C++ qui n'existe pas vraiment, pas sous cette forme du moins, en Python : **les structures**. Les structures permettent d'associer un ensemble de valeurs, peu importe leur type, sous la forme d'une seule variable. Cela permet de définir des concepts cohérents au niveau des données.

On a vu qu'une fois la structure/type définie, on peut créer des variables correspondant à sa description, et on dispose de l'opérateur `."` pour accéder aux sous-parties. De plus, la structure peut s'utiliser comme n'importe quel autre type du C/C++, et on peut notamment :

- Créer des variables, bien entendu, décrites par la structure,
- Créer des tableaux de structures
- Passer des structures à des fonctions,
- Renvoyer des structures avec `return`

De cette façon, on peut définir non seulement un concept par ses données, mais en plus le rendre facilement utilisable par les fonctions que l'on crée autour. ■

PARTIE II

Les techniques du Langage C/C++

Chapitre VII – Introduction à la partie 2

Nous avons fait un bon survol du C/C++, mais nous avons surtout vu ce qui était similaire à Python pour l'essentiel, à part les structures. Maintenant nous allons regarder des choses qui sont plus orientées vers "l'état d'esprit" du Langage C/C++.

Il faut ici rappeler que le C/C++ est un langage considéré aujourd'hui comme proche de l'électronique, on dit aussi "bas niveau", à l'inverse de Python qui est loin de l'électronique et dont on dit qu'il est "haut niveau". Alors, pour mieux comprendre ce qui fait la teneur du C/C++, on a besoin de revenir sur des considérations liées au codage des nombres et à la structure de la mémoire. Cela nous amènera à définir ce qu'est une **adresse** pour la mémoire, à partir de laquelle on définira le concept de **pointeur**. Les pointeurs correspondent à une définition simple, mais cette définition a énormément de conséquences et d'impact sur la façon de programmer en C/C++, avec une syntaxe qui peut sembler difficile à interpréter parfois (mais qui n'est évidemment pas ambiguë). Tout cela fait que le concept de pointeur est souvent considéré comme un sujet difficile en C/C++. Malgré tout il est indispensable à tout programmeur en C/C++ parce que certaines techniques les utilisent massivement. En outre, ces concepts sont cruciaux pour comprendre la programmation sur microcontrôleurs, qui vous concerne en tant qu'étudiant "EEA", et vous sera enseignée au second semestre.

Chapitre VIII – Codage des nombres

💡 Motivations et objectifs

On a vu que l'on pouvait créer des variables, des chaînes, des tableaux, etc. Mais on n'a pas encore une grande idée de comment cela se représente en mémoire vive. Pour cela, on a besoin de notions de base sur comment les nombres sont représentés en mémoire, c'est ce que nous regardons ici.

VIII.1 Codage en base 2 : binaire

Au sein de la machine, les nombres sont codés en "binaire", c'est à dire en base 2. Pour comprendre, il faut déjà se souvenir de ce que signifie l'expérience que l'on a tous les jours des nombres en base 10.

Nombres en base 10 : nombres décimaux

Par exemple prenons 2763. Que signifie le fait que 2763 soit un nombre "en **base 10**" ?

- Chacun de ses chiffres peut être choisi parmi une liste de **10 symboles** : 0,1,2,3,4,5,6,7,8,9. Et c'est tout.
- Plus le chiffre est situé vers la gauche, plus il est le coefficient d'une **puissance de 10** grande.

Si on fait une représentation :

chiffre	2	7	6	3
position	3	2	1	0
poids	10^3	10^2	10^1	10^0

Et donc, en utilisant ce tableau, on peut dire que ce qui fait la "valeur" du nombre 2463 est le calcul ci-dessous :

$$2763_d = 2 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 3 \times 10^0$$

Notez le *d* en indice qui indique que ce nombre est représenté en base 10, on dit aussi qu'il est "décimal".

💡 Remarque

Dans ce nombre, on peut remarquer que plus on considère un chiffre situé à droite, moins il va compter (si je modifie le 3 en 9 par exemple, je modifie peu la valeur puisque j'obtiens 2469), et si on modifie un chiffre situé à gauche, on modifie beaucoup le nombre (si je change le 2 en 9 je change beaucoup la valeur puisque j'obtiens 9463). A cause de cela, on dit que la partie droite est "**le poids faible**", et que la partie gauche est "**le poids fort**".

Nombres en base 2 : nombres binaires

Un nombre binaire, c'est exactement la même chose, sauf que c'est en **base 2**. Autrement dit :

- Chacun de ses chiffres peut être choisi parmi une liste de **2 symboles** (au lieu de 10) : 0,1. Et c'est tout.
- Plus le chiffre est situé vers la gauche, plus il est le coefficient d'une **puissance de 2** grande (au lieu d'une puissance de 10).

Si je prends le nombre binaire 10011101, alors :

chiffre	1	0	0	1	1	1	0	1
position	7	6	5	4	3	2	1	0
poids	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Et donc d'après le tableau, sa valeur est :

$$\begin{aligned} 10011101_b &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 128 + 0 + 0 + 16 + 8 + 4 + 0 + 1 \\ &= 157_d \end{aligned}$$

Chaque chiffre est appelé un "bit", et on dit que celui qui est à gauche est le "bit de poids fort" et celui à droite est le "bit de poids faible".

Le type **char**

Pour des raisons liées à l'histoire de l'informatique¹, les mémoires informatiques sont "alignées" par paquets de 8 bits, et on appelle cela un **octet** ou **byte** (en anglais), et en C il s'agit du type **char**. Tous les autres types utilisent forcément une quantité de mémoire qui est multiple de ça. Et donc la séquence 10011101 que nous venons de regarder serait celle qui correspondrait à la ligne C :

```
char a;
a = 157;
cout << (int)a;
```

Note : l'opérateur (**int**) au niveau du **cout** qui force **cout** à afficher la valeur de **a** plutôt que la lettre correspondante dans la table ASCII.

De plus, en C/C++, on peut aussi écrire directement des nombres binaires. Ainsi, l'expression précédente peut aussi s'écrire :

```
char a;
a = 0b10011101;
cout << (int)a;
```

Notez le **0b** qui précède le nombre binaire, c'est de cette façon que le C/C++ comprend que vous allez écrire un nombre binaire. Et il faut bien comprendre que dans les deux cas, le nombre *a* est exactement le même : qu'il soit donné au C/C++ en binaire ou en décimal, de toute façon, dans la mémoire il sera en binaire. Quant à **cout**, lui affiche toujours en décimal, et affiche donc simplement 157, peu importe que l'on ait indiqué le nombre en binaire ou en décimal dans le code.

VIII.2 Codage en base 16 : hexadécimal

On peut élargir ce raisonnement à des séquences de 16, 32 ou 64 bits pour les **short**, **long** et **long long**. Et là arrive un problème : un nombre de 32 bits par exemple est très difficile à lire ou à mémoriser, c'est "pas pratique" : imaginez vous rappeler par cœur d'une séquence de 32 valeurs (0 ou 1). C'est pas pratique du tout. Alors il est commun d'utiliser une notation condensée du binaire : **l'hexadécimal**. L'hexadécimal, c'est la **base 16**. Décrivons la :

1. déjà discuté : utilisation des ordinateurs pour le texte

- Chacun de ses chiffres peut être choisi parmi une liste de **16 symboles** : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. On note que comme on n'avait plus de chiffre disponible après 9, on a complété arbitrairement avec de nouveaux symboles que l'on a pris dans l'alphabet latin (lettres de A à F), mais on aurait pu choisir n'importe quoi d'autre. Juste l'hexadécimal a été défini comme ça, avec A qui vaudrait 10 en décimal, B vaudrait 11, et ainsi de suite.
- Plus le chiffre est situé vers la gauche, plus il est le coefficient d'une **puissance de 16** grande.

Par exemple, prenons le nombre hexadécimal $2FA5_h$:

chiffre	2	F	A	5
position	3	2	1	0
poids	16^3	16^2	16^1	16^0

Et donc $2FA5_h$ vaut :

$$\begin{aligned} 2FA5_h &= 2 \times 16^3 + F \times 16^2 + A \times 16^1 + 5 \times 16^0 \\ &= 2 \times 4096 + 15 \times 256 + 10 \times 16 + 5 \times 1 \\ &= 12197_d \end{aligned}$$

Cela explique ce qu'est l'hexadécimal mais pas en quoi l'hexadécimal est une écriture condensée du binaire. Pour faire la relation, il faut remarquer qu'en hexadécimal on a 16 symboles, et qu'en binaire on en a 2. A chaque fois qu'on ajoute un bit dans un nombre binaire, on multiplie par 2 le nombre le plus grand que l'on peut atteindre. Et on voit très bien que $16 = 2 \times 2 \times 2 \times 2 = 2^4$. Autrement dit, avec 4 bits on peut représenter les mêmes valeurs qu'avec un seul nombre hexadécimal. Autrement dit, on peut facilement convertir le nombre $2FA5_h$ par exemple, en binaire, en décomposant chaque nombre hexadécimal, individuellement, en binaire :

chiffre hexadécimal	2	F	A	5
correspondance en binaire	0010	1111	1010	0101

D'où on peut écrire :

$$2FA5_h = 0010\ 1111\ 1010\ 0101_b$$

Autrement dit $2FA5_h$ est une écriture condensée de $0010\ 1111\ 1010\ 0101_b$.

On pourrait prendre le problème à l'envers pour se convaincre. Prenons la séquence de 14 bit :

$$11\ 1101\ 0010\ 1011_b$$

Pour trouver le nombre hexadécimal qui correspond, il suffit de le découper en séquences de 4 bits en partant de la droite, et de convertir chaque séquence de 4 bits en un seul chiffre hexadécimal. Faisons le :

correspondance en binaire	(00)11	1101	0010	1011
valeur en décimal	3	13	2	11
valeur en hexadécimal	3	D	2	B

D'où $3D2B_h = 11\ 1101\ 0010\ 1011_b$. On a donc en gros 4 fois moins de symboles en hexadécimal qu'en binaire, et c'est bien normal d'après la définition de l'hexadécimal.



Remarque

Remarque 1 : On peut représenter un octet avec 2 chiffres hexadécimaux, un short avec 4 chiffres hexadécimaux, un long avec 8 chiffres hexadécimaux, etc.

Remarque 2 : Nous n'avons vu que des nombres positifs, autrement dit les nombres **unsigned** du C/C++. Les nombres **signed** utilisent le bit le plus à gauche du nombre pour indiquer signe : lorsqu'il vaut 0, le nombre est positif, s'il vaut 1, le nombre est négatif. Pour les nombres positifs, le codage est exactement le même que pour les nombres **unsigned**. Pour les nombres négatifs, on ne le décrit pas ici,

si vous cherchez à vous renseigner on parle de codage en "complément à 2", ce qui revient à inverser l'état de tous les bits du codage de sa valeur absolue.

Reprenez comment le C/C++ voit tout ça, et repartons avec $2FA5_h$. On pourrait calculer que $2FA5_h = 12197_d$, ce qui revient à dire que pour mettre cette valeur dans une variable, on pourrait écrire en C/C++ :

```
short a;  
a = 12197;  
cout << a;
```

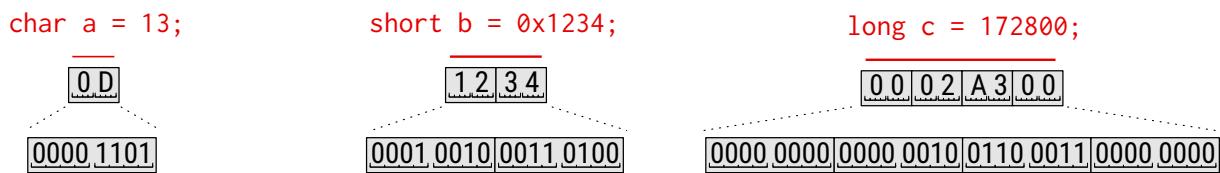
Mais en C/C++, on peut aussi écrire directement des nombres sous la forme hexadécimale. Ainsi, l'expression précédente peut aussi s'écrire :

```
short a;  
a = 0x2FA5;  
cout << a;
```

Notez le **0x** qui précède le nombre hexadécimal, c'est de cette façon que le C/C++ comprend que vous allez écrire un nombre en hexadécimal. Et il est important d'être convaincu que dans les deux cas, **a** vaut exactement la même chose, et est représenté en mémoire exactement de la même façon. Simplement, à l'affichage, **cout** affiche les nombres en décimal, parce que c'est pratique dans la plupart des utilisations que l'on en a.

VIII.3 Codage des variables entières en C/C++

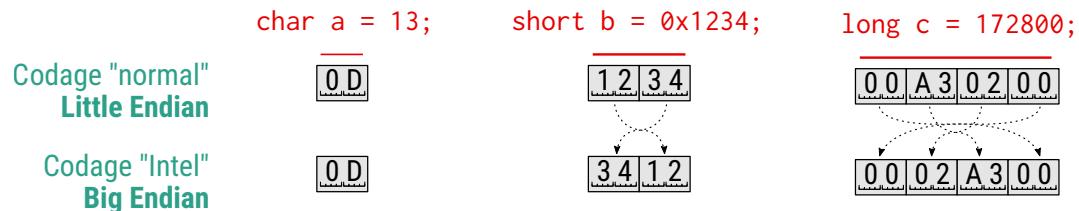
On a vu que l'unité de mémoire la plus petite est le "bit", un chiffre qui vaut 1 ou 0. Pour représenter les grandes nombres binaires de façon plus concise, on regroupe souvent les bits par paquets de 4, chaque paquet étant alors représenté par un chiffre hexadécimal. Au dessus de tout ça, la mémoire, elle, est découpée en octets, soit 8 bits ou encore 2 chiffres hexadécimaux. On représente donc ci-dessous, pour les types entiers les plus courants du C, ce que cela donne en mémoire :



Cette représentation des nombres en mémoire se nomme "**Little Endian**" et correspond exactement à ce que l'on trouve dans un microcontrôleur ou sur la plupart des processeurs. Mais spécifiquement sur PC, il existe un autre codage des nombres entiers. C'est ce dont on discute ci-dessous.

Big Endian et Little Endian

On décrit ici une bizarrerie de l'histoire des technologies, qui ne s'appuie sur rien de scientifique, mais qui existe en pratique à cause certains méandres de l'histoire de l'informatique. Il se trouve que les PC utilisent une puce de calcul fabriquée par Intel. Et à une époque très ancienne (dans les années 70), Intel a décidé de "changer le codage des nombres" à un format qui est le leur, en inversant le poids des octets d'un nombre. On appelle ce codage "**Big Endian**". Sur l'exemple précédent, cela donne :



Ce changement n'aboutit à absolument aucun bénéfice technologique (cela revient à "croiser des fils" dans le câblage interne d'un processeur, rien de fou), juste intel a décidé de ne pas faire "comme tout le monde" pour des raisons (sûrement fantasmées) de propriété intellectuelle. C'est un peu comme si un

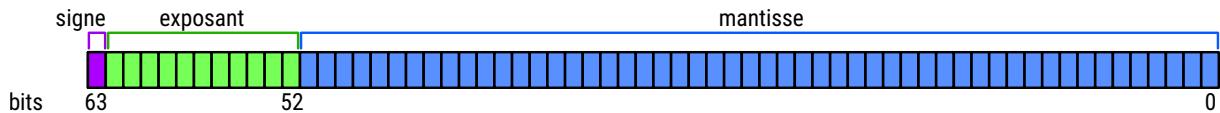
mathématicien décidait que 1234_d doit s'écrire désormais 3412_d sans justification. C'est juste une source de confusion², mais on doit vivre avec. Ainsi, à l'époque à laquelle nous vivons, il y a donc deux codages pour les nombres :

- le codage "little Endian", ("la fin est petite", autrement dit "le poids faible est à la fin" donc "à droite"), qui est le codage normal,
- le codage "big Endian", ("la fin est grande" autrement dit "le poids fort est à la fin", donc "à droite"), qui est le codage introduit par Intel et qui est habituel sur PC

Ça n'a souvent pas de conséquence, sauf quand on explore la mémoire, comme au prochain chapitre, ou si un processeur intel et un processeur "autre" doivent communiquer ensemble (par exemple quand on branche une imprimante sur un PC, pour que la communication se fasse bien il faut réencoder les nombres lors de la transmission ...)

VIII.4 Codage des nombres à virgule flottante en mémoire

On va regarder le cas des **double**. Ils sont codés sur 8 octets, donc sur 64 bit. Leur structure est donnée ci-dessous :



La valeur absolue du nombre réel est donnée par :

$$\text{valeur} = \text{mantisse} \times 2^{\text{exposant}}$$

On voit sur l'illustration que la mantisse est codée par les bits 0 à 51, elle fait donc 52 bits de long, et c'est sa valeur absolue puisque le signe est codé sur le bit en violet. Un nombre de 52 bits non signé peut aller jusqu'à $2^{52} - 1 \approx 2 \times 10^{15}$. C'est pourquoi les nombres ont une précision de 15 chiffres significatifs. L'exposant utilise les bits de 52 à 62, soit 11 bits, dans lesquels il faut retirer le bit de signe³, ce qui fait 10 bits en valeur absolue. Autrement dit l'exposant peut valoir, en valeur absolue, au maximum $2^{10} - 1 = 1023$. Ce qui fait que 2^{exposant} vaut au maximum $2^{1023} \approx 10^{308}$. Ainsi, comme on l'avait annoncé, on peut avoir des nombres positifs ou négatifs variant, en ordre de grandeur, de 10^{-308} à 10^{308} , avec une précision de 15 chiffres significatifs.

VIII.5 Comparaison avec le codage en chaines de caractères

Si "tout est clair", vous avez compris que les valeurs des variables ne sont pas du tout codées sous forme de chaines de caractères. En effet, il faut bien distinguer ces deux choses :

Codage Binaire	Codage par Chaine de caractères														
<code>short k=25696;</code>	<code>string s1="25696";</code>														
<u><code>short k</code></u> <table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>6</td><td>4</td><td>6</td><td>0</td></tr></table>	6	4	6	0	<u><code>string s1</code></u> <table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>3</td><td>2</td><td>3</td><td>5</td><td>3</td><td>6</td><td>3</td><td>9</td><td>3</td><td>6</td></tr></table> <p>'2' '5' '6' '9' '6' ← caractère ASCII pour chaque case</p>	3	2	3	5	3	6	3	9	3	6
6	4	6	0												
3	2	3	5	3	6	3	9	3	6						

2. Et pour se moquer de cette situation, lorsque les expressions "little Endian" et "big Endian" ont traversé l'Atlantique pour venir en France, elles ont été transformées en "grand indien" et "petit indien".

3. ce n'est pas codé réellement avec un bit de signe pour des raisons techniques mais ça revient au même

On voit bien que le codage associé n'est pas du tout le même. Or : il est souvent utile de convertir l'un en l'autre, et réciproquement. Par exemple : afficher une variable à l'écran fait déjà intervenir une telle conversion. Prenons en effet ce petit code d'exemple :

```
short k=25696;
cout << k;
```

Tout ce que l'on peut afficher dans le terminal, c'est une chaîne de caractères. Mais comme **k** est un short, il est codé en binaire, comme montré ci-dessus, et donc il y a bien quelque chose dans **cout** qui fait la conversion de quelque chose qui prend la forme de **k** en quelque chose qui prend la forme de **s1** dans l'illustration ci-dessus⁴.

Imaginons que l'on veuille récupérer une chaîne qui contient le codage de **k** sous forme de chaîne de caractères. Les bibliothèques du C/C++ permettent de réaliser cette conversion. Ça marche sur le même principe que **cout**. Voici d'abord ce que l'on aurait écrit avec **cout** :

```
1 #include <iostream>
2 #include <sstream>
3 using namespace std;
4
5 int main ()
6 {
7     short k=25696;
8     cout << k << "\n";
9     return 0;
10 }
```

On va séparer **cout** en deux parties :

- Une partie qui crée la chaîne de caractères,
- Une partie qui affiche la chaîne de caractères générée. Voici ce que ça donne :

```
1 #include <iostream>
2 #include <sstream>
3 using namespace std;
4
5 int main ()
6 {
7     short k=25696;
8
9     stringstream ss;
10    ss << k << "\n";
11
12    string s1 = ss.str();
13
14    cout << s1 ;
15    return 0;
16 }
```

Ainsi :

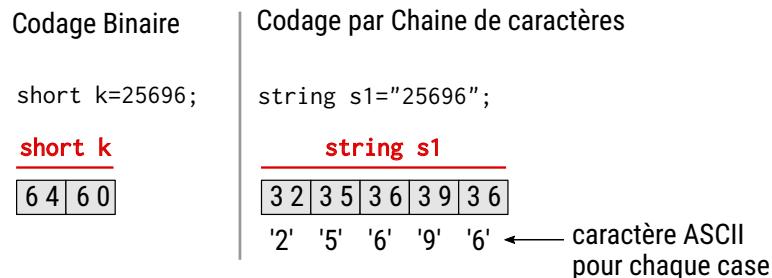
- Lignes 9 et 10, on définit la variable **ss** comme étant un **stringstream**. Ce **stringstream** va recevoir, ligne 10, tout ce que l'on aurait donné à **cout** pour un affichage simple. Ainsi, **ss** va emmagasiner tout ce que **cout** aurait affiché.
- Ligne 12 : on crée la chaîne de caractères **s1** pour recevoir tout ce que **ss** a emmagasiné, grâce à la fonction **str()**

4.

Et si vous réfléchissez un peu, cette transformation n'est pas facile du tout ! Un point de départ possible serait de considérer qu'un **short** ne peut pas dépasser 5 caractères (sa plus grande valeur est 32767). Donc on peut commencer par diviser par 10000 pour avoir le premier chiffre, prendre le reste de cette division, diviser le reste par 1000 pour avoir le second chiffre, en prendre le reste, et ainsi de suite.

- Ligne 14 : on vérifie que la chaîne `s1` contient bien le nombre `k`, mais mis sous forme de chaîne de caractères.

On vient donc bien de réaliser cette conversion :



Maintenant imaginons que l'on veuille faire l'opération inverse : transformer une chaîne de caractères en la valeur qui lui correspond dans une variable binaire. Sur l'exemple ci-dessous :

```

1 #include <iostream>
2 #include <sstream>
3 using namespace std;
4
5 int main ()
6 {
7     stringstream ss;
8     ss << "1.564";
9
10    double f;
11    ss >> f;
12
13    cout << f << '\n';
14    return 0;
15 }
```

Cette fois le `stringstream` a pris la place de `cin` à la ligne 8 (revoir la section III.4 si ça n'est pas clair), et la transformation en `double` s'est fait ligne 11.

❖ En résumé ...

On a revu ce qu'est une "base de numération", on l'a appliquée à la base 10 puis aux deux bases qui sont plus naturelles pour un processeur : la base 2, appelée aussi "binaire", qui est la base de l'informatique simplement parce que c'est la base la plus simple de toutes puisqu'elle ne repose que sur 2 symboles (même si elle est moins naturelle pour nous), et la base 16, appelée aussi "hexadécimal", qui est très commune parce qu'on peut la voir comme une écriture condensée du binaire. En effet, un chiffre hexadécimal est simplement équivalent à 4 chiffres binaires, ce qui réduit d'un facteur 4 l'écriture des nombres. Mais c'est simplement parce que c'est "plus pratique" pour nous, il n'y a rien de très conceptuel derrière cette idée.

On a aussi comparé avec un "piège" classique, qui est de croire que les nombres sur lesquels on fait des calculs peuvent être représentés par des chaînes de caractères. En effet, un calculateur ne travaille pas avec des chaînes de caractère pour faire du calcul, et on se doit d'en réaliser la conversion vers des nombres binaires avant. ■

Chapitre IX – Exploration de la mémoire et pointeurs

⌚ Motivations et objectifs

On va s'intéresser à la représentation des données en mémoire à une échelle plus large que simplement "les nombres". On va définir deux concepts importants en langage C/C++ et encore plus importants pour la programmation microcontrôleurs :

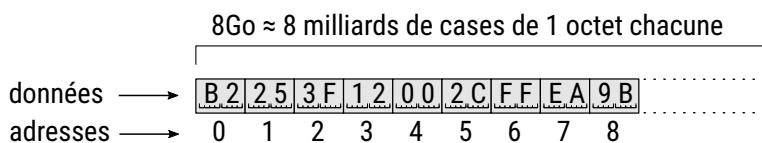
- le concept d'**adresse**,
- et le concept de **pointeur**,

Le concept de pointeur est parmi les concepts difficiles à comprendre en C/C++, pas parce qu'il est complexe en soi, mais parce que la syntaxe peut sembler confuse (alors qu'elle ne l'est pas), et que les pointeurs sont un outil puissant qui est beaucoup utilisé en C/C++, dans nombre de situations, et on en verra quelques unes dans les chapitres suivants.

IX.1 Les variables dans la mémoire : contenu et adresse

Données et adresses

En C/C++, on est plus proche de l'ordinateur qu'on ne l'est en Python, et l'illustration la plus habituelle de ce fait est que l'on a besoin de parler d'*adresse*, un concept assez absent du langage Python alors qu'il est fondamental pour le C/C++. Mais pour comprendre, il faut se faire une idée du monde tel qu'il est vu par un programme. En simplifiant, pour l'essentiel, la puce qui exécute les calculs, le microprocesseur, voit tout son environnement comme "un énorme tableau", dont chaque case fait la dimension d'un octet. La plus grande partie de ce tableau est la RAM, autrement dit le lieu où sont stockés les programmes et les variables. Chacune de ces cases a un contenu et un numéro qui indique la position de la case. Quand on crée une variable par exemple, le microprocesseur utilise quelques cases de cet immense tableau, pour représenter / mémoriser la variable en question. Ce qu'il faut voir, c'est que, le "monde" dans lequel on travaille, quand on le regarde vu d'un processeur et donc vu du programmeur, ressemble à ça :



La RAM ("mémoire vive") est donc un immense tableau dont chaque case contient un octet¹, c'est-à-dire une séquence de 8 bits, qui peut être représentée par 2 chiffres hexadécimaux, comme sur le dessin. Le contenu de la case est appelé "**donnée**" (data en anglais).

Chacune de ces cases est située à une certaine position, et cette position est appelée "**adresse**" (puisque c'est le "lieu" où elle "habite").

Un PC moyen aujourd'hui contient 8 Go de RAM, ce qui signifie environ 8 milliards de ces cases, chacune pouvant contenir 1 octet ou 8 bit. Comme on a 8 milliards de cases, l'adresse est une valeur située entre 0 et 8 milliards.

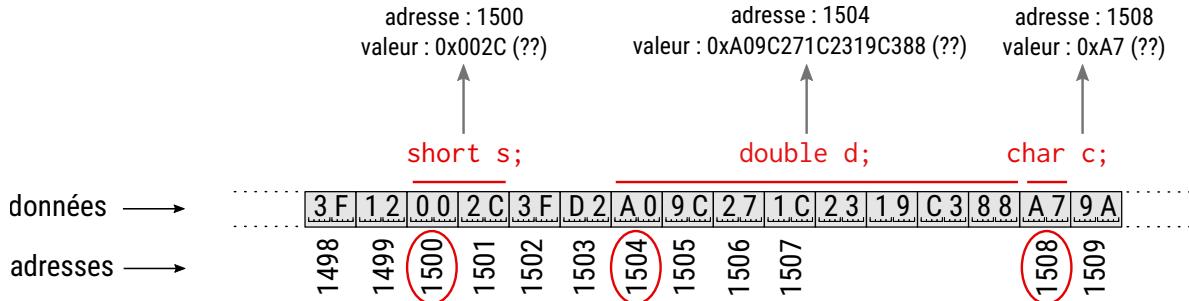
1. Ca n'est pas obligatoire, il existe ou du moins il a existé des ordinateurs fabriqués autrement, par exemple avec 16 bits par case, mais aujourd'hui l'essentiel de l'informatique fonctionne comme ça

Variables en mémoire

Tout ce qui est présent dans la mémoire est sous cette forme, et les variables que l'on crée ne font pas exception. Examinons ce qui se passe avec le petit code suivant :

```
1 short s;
2 double d;
3 char c;
```

Le C/C++ va "réserver" de la mémoire pour chacune de ces variables, c'est à dire qu'il va décider de la position en mémoire de chaque variable. Regardons ce que ça donne sur le petit code ci-dessus :

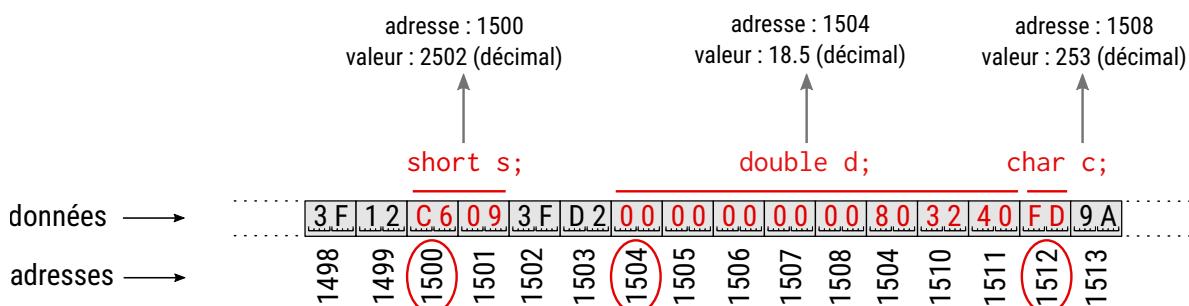


Quand on crée une variable, le C/C++ ne fait que réserver la mémoire, c'est à dire qu'il se contente d'attribuer une plage de mémoire à une variable, et donc le contenu "vaut ce qu'il vaut" : on ne peut pas faire confiance à cette valeur pour mener un calcul par exemple. Les valeurs données sur l'illustration ne servent donc "à rien", juste à illustrer qu'une case mémoire vaut forcément quelque chose, mais que ce quelque chose correspond à rien d'exploitable. Et de la même façon, les adresses ont été choisies par le C/C++ (encerclées en rouge sur l'illustration), et on ne peut pas décider ce qu'elles sont.

Voici maintenant ce qui se passe si on complète le programme en attribuant des valeurs :

```
1 short s;
2 double d;
3 char c;
4
5 c = 253;
6 s = 2502;
7 d = 18.5;
```

On aboutit à :



Commençons par regarder le plus simple : **ligne 5**. La valeur $253_d=FD_h$. C'est assez facile à voir, le plus grand nombre que l'on peut représenter avec 8 bit est $FF_h=255_d$ et il faut retrancher 2, aussi bien en hexadécimal qu'en décimal. Et donc on se retrouve avec ce qui est représenté sur le dessin.

Un peu plus embêtant : **ligne 6**. Le codage hexadécimal de $2050_d=09C6_h$. Et on voit qu'en mémoire on a obtenu cela mais dans un autre ordre : C609. C'est l'effet du codage Big Endian, décrit en §[VIII.3](#)

On ne va pas regarder ce qui se passe **ligne 7** : ce serait compliqué à décoder parce que le codage des "doubles" est un peu complexe comme nous l'avons montré plus haut. Mais au final on a un certain codage hexadécimal.

Important

A ce stade, il faut bien comprendre que, vu de la mémoire, les `int` ou `float` ou `char` n'a aucune signification. La mémoire est juste une succession d'octets, et c'est parce que nous utilisons un langage tel que le C/C++ que ces données brutes prennent un sens, par la gestion que fait le langage des variables. Il faut donc bien comprendre que la mémoire n'est rien de plus qu'un "flux binaire" d'informations, auxquels il faut "quelque chose en plus", typiquement un programme, ou quelque chose qui décrit son organisation, pour lui donner du sens. ■

L'opérateur adresse du C

Pour finir, le C/C++ nous donne aussi accès à l'adresse des variables, avec l'opérateur `&`. Voici sur un exemple :

```
1 short s;
2 double d;
3 char c;
4
5 c = 253;
6 s = 2502;
7 d = 18.5;
8
9 cout << &d << "\n";
10 cout << &s << "\n";
11 cout << &c << "\n";
```

qui affiche :

```
0x7ffe2c72c398
0x7ffe2c72c395
0x7ffe2c72c396
```

Ce sont les adresses de ces variables au moment où nous avons lancé le programme. Elles "remplacent" les 1500, 1504 et 1512 sur l'exemple dessiné. Elles sont affichées en hexadécimal par `cout`, ce qui est classique en programmation.

Important

En C/C++, toute expression doit avoir un type. Prenons la variable `s` par exemple. cette variable a pour type `short`, c'est marqué ligne 1. Mais maintenant, quel est le type de `&s` ? Evidemment pas simplement `short`, sinon ça signifierait que l'adresse de `s` ne peut pas dépasser 65000, ce qui est stupide en soi, et ne correspond pas à ce que nous avons affiché dans l'exemple plus haut. Donc ce n'est pas `short`. En C/C++, le type de l'adresse d'une variable est le type de la variable suivi d'une étoile `*`. Ainsi, les types de `&s`, `&d`, `&c` sont, respectivement, `short*`, `double*`, et `char*`. **C'est important à mémoriser**, on va en reparler sous peu. ■

Remarque

Sur un PC dit "64 bits", les adresses sont codées en 64 bit. C'est une des raisons pour lesquelles on dit que ce sont des PC 64 bit. Autrement dit elle permet de gérer des quantités de RAM allant jusqu'à $2^{64} \approx 10^{19}$ octets, largement plus que ce que contient un PC actuel, de l'ordre de $8Go \approx 10^9$ octets : on

a donc beaucoup de marge avant d'arriver à une limite technologique de ce côté.

Dire qu'une adresse est codée sur 64 bits, c'est dire qu'elle est codée sur 8 octets et nécessite donc 16 chiffres hexadécimaux pour sa représentation. On voit ci-dessus que les adresses utilisent ici 12 chiffres hexadécimaux : c'est parce que les 4 premiers valent 0 et ne sont donc pas représentés. Ils valent 0 justement parce qu'un PC moderne n'a pas assez de RAM pour atteindre les adresses les plus grandes.

IX.2 Des variables particulières : les pointeurs

Important

C'est un concept qui n'existe pas du tout en Python mais qui est fondamental en C/C++, et tout autant pour les microcontrôleurs !

Un pointeur est par définition :

- **une variable**, autrement il occupe une quantité de mémoire, comme n'importe quel **short** ou **float**, il a une certaine valeur, et on peut réaliser des opérations dessus,
- cette variable représente **une adresse**. Autrement dit la valeur qu'elle vaut, est à comprendre comme une adresse.

Première prise en main : avec un pointeur **void****

Pour comprendre ce que cela signifie, reprenons la situation précédente, que l'on complète pour utiliser maintenant un pointeur **void*** :

```
1 short s;
2 double d;
3 char c;
4
5 c=253;
6 s=2502;
7 d=18.5;
8
9
10 void* p;
11
12 p=&s;
```

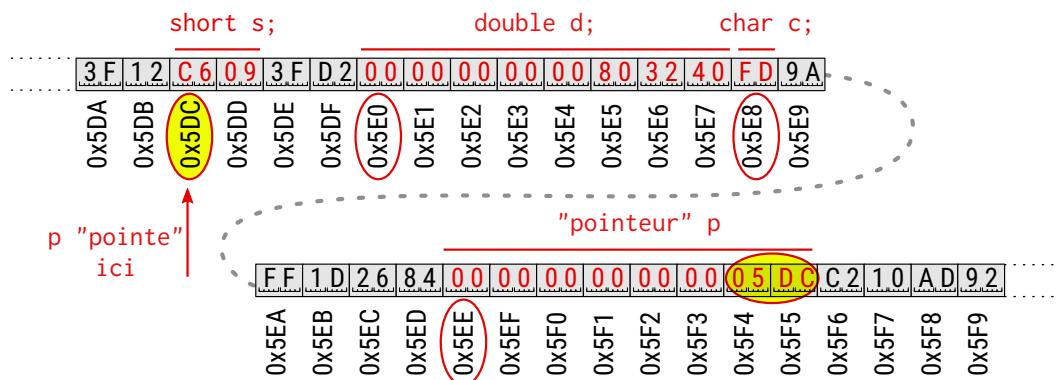
 [code/code-IX-2-27.cpp]

Dans ce code, la ligne 10 crée un pointeur : on identifie que c'est un pointeur par le *. Et à la ligne 12, on affecte à ce pointeur l'adresse de s.

On donne plus bas une représentation graphique de ce qui se passe avec p :

- Comme c'est une variable, le C va la stocker "quelque part", supposons (arbitrairement) en **0x5EE** sur cet exemple.
- Maintenant, la question à se poser est "combien d'octets sont nécessaires ?". La réponse est : comme on travaille sur un PC 64 bit, il faut pouvoir stocker une adresse qui elle-même fait 64 bit, et donc cela va nécessiter 8 octets. Et n'oublions pas que le code affecte à p l'adresse de s soit sur cet exemple **0x5DC**.

Regardons ce que ça donne :



On retrouve bien la variable "pointeur" **p** qui contient **0x5DC**. Mais comme la valeur d'un pointeur est à considérer comme une adresse, cela revient à dire que le **0x5DC** dont on parle correspond à l'adresse de **s**. Autrement dit **p** "pointe" à l'adresse de **s**, *on dit que "p pointe sur s"* : c'est ce qui est illustré par la flèche rouge sur le dessin.

Mais à quoi ça sert ? Qu'est ce qu'on peut faire avec ça ? Avec un pointeur **void***, tel que ce pointeur **p**, on ne peut rien faire de très utile. C'est juste une première approche qui a servi à se rassurer sur le fait qu'on peut bien créer des variables capables de contenir l'adresse d'une autre variable, et que cela revient à dire que le pointeur **p** "pointe" sur la variable **s**. Mais on ne peut rien faire de plus, juste stocker une adresse. Alors voyons comment faire plus.

IX.3 Le type d'un pointeur

Le pointeur **void*** que nous avons créé manque d'utilité parce que aucun type ne lui est associé. En effet, on sait que **void** tout seul signifie "aucun type" en C/C++. Et donc **void*** est le pointeur sans type. Et c'est pour cela qu'on ne peut pas faire grand chose avec, à part stocker une adresse. Alors on va introduire le typage des pointeurs, qui est une notion fondamentale.



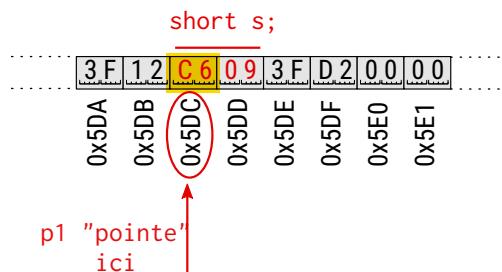
Important

Et donc en C/C++, les pointeurs ont un type. Pourquoi ? Ca semble bizarre : jusqu'ici, avec des variables "classiques", quand on donne un type à un entier, cela représente le nombre d'octets qu'il va utiliser. Or ici il fait de toute façon 8 octets puisque qu'il contient une adresse. Alors ?

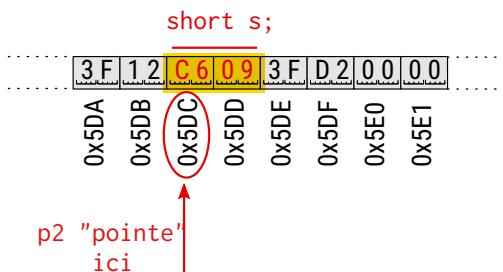
Alors : le type d'un pointeur est le type de la façon dont il regarde la mémoire. C'est une phrase très importante : cela signifie que c'est le type du pointeur qui définit l'interprétation qu'il fait des données qu'il voit en mémoire. En effet : la mémoire n'est rien d'autre qu'une succession d'octets, et le fait qu'une zone soit interprétée comme étant un **short** ou **double** est l'effet de la gestion des variables par le langage. Et à travers un pointeur qui a un type, on peut décider de l'interprétation que l'on donne à une zone de mémoire. ■

On illustre cette idée en prenant un pointeur **p1** "typé" **char** et un pointeur **p2** "typé" **short**, chacun regardant la même zone de mémoire :

ce que **p1** voit si **p1** est de type **char** :



ce que **p2** voit si **p2** est de type **short** :



Ce que l'on voit, c'est que la "zone" qui intéresse le pointeur dépend de son type : si il est de type **char**, alors il regarde la mémoire "par paquets" de 1 octet, si il est de type **short**, alors il regarde la mémoire "par paquets" de 2 octets. Si c'était un **long**, ce serait "par paquets" de 4 octets, et ainsi de suite.

Alors regardons comment écrire ça en C/C++ :

```

1 short s;
2 double d;
3 char c;
4
5 c = 253;
6 s = 2502;
7 d = 18.5;
8
9 char* p1;
10 short* p2;
11
12 p2 = &s;
13 p1 = (char*)&s;
```

Décrivons ce petit programme :

- Le début du programme est le même que jusqu'à présent : il déclare 3 variables de types différents, et leur donne des valeurs.
- les lignes 9 et 10 montrent comment on **crée un pointeur avec un type**. Ce qui identifie le fait que l'on crée un pointeur, au moment de la déclaration, c'est le ***** qui apparaît. Il faut lire les choses ainsi :
 - Ligne 9, on crée un pointeur dont le nom est **p1**, et qui va **regarder la mémoire comme une succession de char**, autrement dit par "paquets" de 1 octet : c'est ce que signifie **char*** au moment où on déclare un nouveau pointeur.
 - Ligne 10, on crée un pointeur dont le nom est **p2**, et qui va **regarder la mémoire comme une succession de short**, autrement dit par "paquets" de 2 octets : c'est ce que signifie **short*** au moment où on déclare un nouveau pointeur.
- les lignes 12 et 13 **placent les pointeurs** **p1** et **p2** sur **s**. Sur notre exemple, cela revient à dire que **p1** et **p2** vont pointer sur l'adresse 0x5DC, et c'est exactement le résultat que l'on obtient. Mais la syntaxe impose quelques nuances :
 - La ligne 12 est simple : **p2** va se positionner sur l'adresse **&s**.
 - De façon similaire, on pourrait "résumer" ce que fait la ligne 13 en :

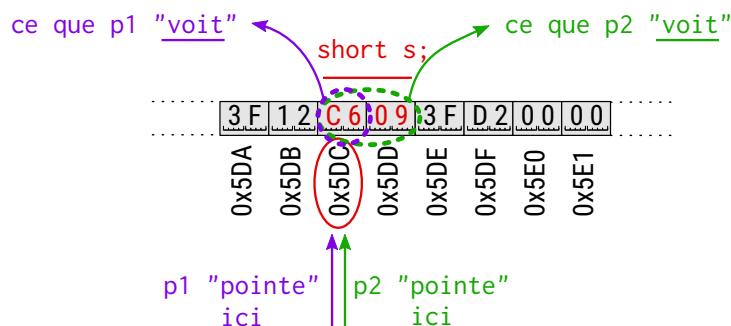
```
p1 = &s;
```

Parce que c'est exactement ce que fait la ligne 13. Mais l'écrire ainsi n'est pas possible, le compilateur C/C++ va refuser. En effet, quand on affecte un pointeur à une position, il faut *absolument* que le type à gauche du **=** soit identique au type à droite. Autrement dit, il faut que **p1** et **&s** soient du même type. Or le type de **p1** est **char***, mais le type de **&s** est **short***, comme

on l'a vu en page 66. Et donc pour forcer cette égalité à se faire, on invoque l'opérateur de conversion `char*`, qui en réalité "ne fait rien" au sens où il ne transforme pas l'adresse (c'est exactement la même adresse), mais permet à l'égalité de se faire. Ca semble tordu mais au sens du C, c'est ce que nous faisons maintenant qui est inhabituel et contre lequel le C/C++ veut nous protéger. Mais là, sur cet exemple, c'est bien ce que l'on veut faire.

- revenons sur la ligne 12 : il est donc normal qu'il n'y ait rien de plus à faire que ce qui est écrit, puisque `&s` et `p2` sont du même type, soit `short*`. Notez que c'est le cas que l'on rencontre le plus souvent dans les programmes, il est assez rare d'avoir besoin de faire des conversions de types sur des pointeurs. Mais pour ce que l'on veut montrer c'est indispensable.

On aboutit donc à la situation ci-dessous :



IX.4 Déréférencement

Pour l'instant on n'a pas l'impression d'avoir fait quelque chose de plus qu'avec les pointeurs `void*`. En effet, maintenant :

- On sait comment créer un pointeur avec un type,
- On sait comment le placer à un endroit précis,
- Sauf que maintenant on sait "ce qu'il voit" et "comment il le voit", et ça va tout changer.

Ce qui va changer, c'est que maintenant on peut demander au pointeur de récupérer ou modifier "ce qu'il voit". Ce qu'il voit, on l'obtient avec l'opérateur `*` : on appelle cette opération le **déréférencement**. Par exemple, si on veut afficher ce que chacun des deux pointeurs voit :

```

1 short s = 2502;
2
3 char* p1;
4 short* p2;
5
6 p2 = &s;
7 p1 = (char*)&s;
8
9 int a, b;
10 a = *p1;
11 b = *p2;
12
13 cout << a << "\n";
14 cout << b << "\n";

```

l'opérateur est utilisé sur les lignes numérotées 1 et 2 dans le code ci-dessus. En résultat, il affichera :

```
-58
2502
```

où $-58_d=C6_h$ si on considère des variables "signées" (ce qui est le cas de `a` qui est `int`), et $2502_d=C609_h$. Ceci conforte que l'on comprend bien ce que voient `p1` et `p2`.



Important

Une erreur qui est un grand classique du C : penser que "*" signifie "pointeur.
NON!!! "*" ne signifie pointeur QUE lors de la déclaration du pointeur.

Une fois le pointeur créé, "*" signifie "déréférencement", c'est-à-dire récupérer ce que "voit" le pointeur.

Conséquence : Dans le petit exemple ci-dessous :

```
1 long a=5;
2 long* p;
3 p = &a;
4 cout << *p;
```

p est un **long*** (c'est écrit ligne 2), ce qui signifie que p est un pointeur qui regarde la mémoire comme des **long**, et ***p** (ligne 4) est un **long**, parce que c'est la "chose" vue par p : c'est bien le sens de l'expression ***p**.

On peut se mémoriser la signification de * en C/C++ par l'illustration ci-dessous :

long *p;	long *p;
long* est le type de p	long est le type de *p



IX.5 Une technique importante : modifier une variable en utilisant un pointeur

Enfin, on peut, à travers le pointeur, modifier la variable vue par le pointeur. Par exemple :

```
short c;
short* p1 = &c;
*p1 = 8; // met 8 dans c
```



Important

Ce petit code met 8 dans la variable c parce que p1 pointe sur c. Autrement dit, comme p1 "voit" c dans son intégralité, modifier "ce qui est pointé" par p1 revient à modifier c. Si ça semble un peu stupide de vouloir modifier c indirectement, **c'est une technique que nous emploierons dans certaines circonstances parce que nous n'aurons pas le choix.** Aussi c'est important et à mémoriser. ■

IX.6 Une chose à ne pas faire : déréférencer un pointeur sans savoir où il pointe



Important

Tout est écrit dans ce titre mais c'était utile de mettre en évidence. Voici un exemple de code qui fait planter "de temps en temps" un programme, et qui, quand il ne plante pas, fait toujours n'importe quoi :

```
short* p1 ;
*p1 = 8; // ou a-t-on mis ce 8 ???
```

Pourquoi ce petit code pose-t-il problème ? Voici ce qu'il fait :

- Au niveau de la première ligne, on crée le pointeur p1. Et quand on le crée, comme toute variable du C/C++, on ne sait pas "ce qu'il vaut". Autrement dit : on ne sait pas où il pointe. Ça pourrait aussi bien être dans les cases mémoire d'un autre programme par exemple, ou en un endroit de

notre programme où il n'a rien à faire,

- à la deuxième ligne, on affecte la valeur **8** aux cases mémoires que "voit" le pointeur. Mais ... a-t-on le droit de le faire ? La réponse est non ! On n'a pas le droit d'écrire arbitrairement des choses dans des cases mémoire qui ne sont pas liées à des variables ou à des tableaux que l'on a créé avant ! Or là, on ne maîtrise rien.

Résultat : il est possible que le système d'exploitation le détecte ^a, et si c'est bien le cas, il va simplement arrêter votre programme en le faisant planter.

Bilan : A chaque fois que l'on écrit un ***** devant un pointeur, on vérifie que l'on maîtrise le lieu où il pointe. *C'est un morceau de méthodologie indispensable avec les pointeurs!* ■

a. c'est une façon imagée de le dire, pour comprendre il faut savoir ce qu'est une "MMU" pour "Memory Management Unit" dans un processeur, notion dont vous entendrez sûrement parler plus tard

IX.7 Tableaux en mémoire

On va maintenant explorer la mémoire des tableaux **vector**. D'abord regardons si on comprend ce que sont les adresses des cases :

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main ()
6 {
7
8 vector<short> v1;
9 vector<double> v2;
10
11 v1.resize(20);
12 v2.resize(20);
13
14 cout<<"Tableau de short\n";
15 int k=0;
16 while(k<20) {
17     cout << &v1[k] <<"\n";
18     k=k+1;
19 }
20
21 cout<<"\nTableau de double\n";
22 k=0;
23 while(k<20) {
24     cout << &v2[k] <<"\n";
25     k=k+1;
26 }
27 return 0;
28 }
```



Ce code crée 2 tableaux de 20 cases, l'un contient des **short** (qui fait 2 octets), l'autre contient des **double** (qui fait 8 octets). Puis, le code affiche les adresses des cases. Voici un extrait de ce que l'on obtient :

1 Tableau de short 2 0x55dfb017ceb0 3 0x55dfb017ceb2 4 0x55dfb017ceb4 5 0x55dfb017ceb6	6 0x55dfb017ceb8 7 9 Tableau de double 10 0x55dfb017cee0 11 0x55dfb017cee8
--	--

```

12 | 0x55dfb017cef0
13 | 0x55dfb017cef8
14 | 0x55dfb017cf00
15 | .....

```

On remarque bien que les adresses des cases du tableau de **short** sont espacées de 2 octets, et que les adresses du tableau de **double** sont espacées de 8 octets. C'est conforme à la définition du **short** et **double**, et "rassure" sur le fait que les cases d'un tableau sont consécutives en mémoire.

Maintenant regardons ça à travers des pointeurs, avec ce code :

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main ()
6 {
7     vector<unsigned short> v1;
8     v1.resize(20);
9
10 // remplissage du tableau
11     int k=0;
12     while(k<20) {
13         v1[k] = k;
14         k = k+1;
15     }
16
17 // exploration avec des pointeurs
18     unsigned char* p1;
19     unsigned short* p2;
20     p1 = (unsigned char*)&v1[0];
21     p2 = &v1[0];
22     k=0;
23     while(k<20) {
24         cout << (void*) p1 << " " << (int)(*p1) << " " << p2 << " " << *p2 << "\n"
25             ";
26         k=k+1;
27         p1 = p1+1;
28         p2 = p2+1;
29     }
30
31     return 0;
}

```

 [code/code-IX-7-29.cpp]

D'abord une remarque : nous avons choisi des types "non signés" pour ne pas s'embêter avec le codage des nombres négatifs. C'est pour faire un exemple.

Ensuite, à partir de la ligne 16, il ne devrait pas être compliqué de comprendre que l'état du début de **v1** est le suivant (en prenant en compte l'inversion du codage Intel) :

v1[0]	v1[1]	v1[2]	v1[3]	v1[4]	v1[5]	v1[6]	v1[7]	v1[8]	v1[9]	...
00	00	01	00	02	00	03	00	04	00	05
00	00	00	00	00	00	00	00	00	00	00

0xEB0	0xEB1	0xEB2	0xEB3	0xEB4	0xEB5	0xEB6	0xEB7	0xEB8	0xEB9	0xEBA
0xECB	0xECB1	0xECB2	0xECB3	0xECB4	0xECB5	0xECB6	0xECB7	0xECB8	0xECB9	0xECBA
0xECB	0xECB1	0xECB2	0xECB3	0xECB4	0xECB5	0xECB6	0xECB7	0xECB8	0xECB9	0xECBA

Maintenant, on place **p1** et **p2** à l'adresse de la case 0 (on a noté comme précédemment le besoin de conversion, ligne 20, qui n'existe pas ligne 21). C'est à dire que l'on aboutit à cette situation :

The diagram illustrates a memory dump with 10 entries labeled **v1[0]** through **v1[9]**. Each entry contains a pair of hex values. Below the dump, two pointers are shown: **p1** pointing to the start of the dump, and **p2** pointing to the value at **v1[1]**.

v1[0]	v1[1]	v1[2]	v1[3]	v1[4]	v1[5]	v1[6]	v1[7]	v1[8]	v1[9]										
0.0	0.0	0.1	0.0	0.2	0.0	0.3	0.0	0.4	0.0	0.5	0.0	0.6	0.0	0.7	0.0	0.8	0.0	0.9	0.0

Pointers:

- p1** "pointe" (purple arrow) points to the start of the dump.
- p2** "pointe" (green arrow) points to the value at **v1[1]**.

Puis on rentre dans la boucle. Expliquons la ligne 24 où on affiche un ensemble de choses :

- D'abord **p1**. Il y a devant une conversion en (**void***) qui semble bizarre. On n'a en gros jamais besoin de faire ce genre de "bricolage" sauf avec les pointeurs **char*** affichés par **cout**. C'est lié à "l'ancien" fonctionnement des chaînes de caractères en C/C++. Sans rentrer dans les détails, (**void***)**p1** vaut la même adresse que **p1** tout seul, mais le (**void***) permet de l'afficher correctement avec **cout**. C'est un cas particulier.
 - Ensuite on affiche la donnée vue par **p1** avec ***p1**. Mais on a ajouté (**int**) devant pour l'afficher en tant que valeur. En effet, ***p1** est un **char** (puisque **p1** est un **char***, non il n'y a pas d'erreur avec les étoiles ici), et **cout** veut l'afficher comme un caractère, à travers la table ASCII. Or ici on veut afficher la valeur, et donc on convertit en **int** pour qu'il soit affiché comme une valeur.
 - puis on affiche **p2** et ***p2** et ça se fait normalement parce que les **short** et **short*** sont gérés "normalement" par **cout**.
 - Enfin, on fait "avancer" les pointeurs **p1** et **p2** de "1"

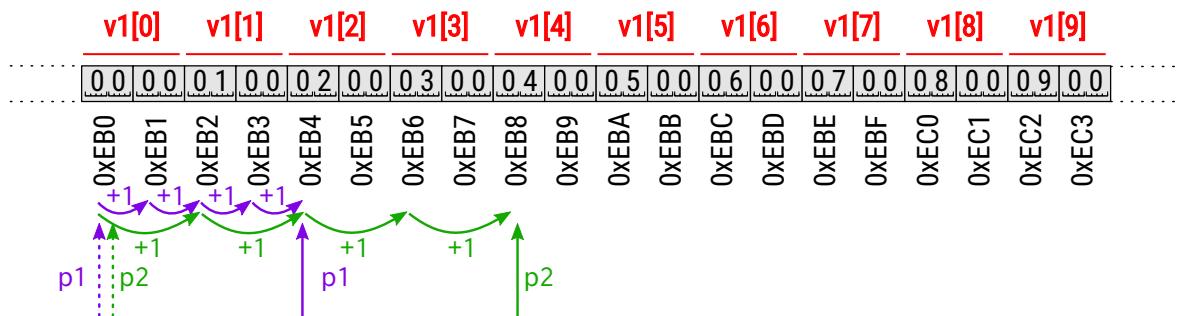
Et voici ce qui est affiché :

0x55da4c693eb0	0	0x55da4c693eb0	0
0x55da4c693eb1	0	0x55da4c693eb2	1
0x55da4c693eb2	1	0x55da4c693eb4	2
0x55da4c693eb3	0	0x55da4c693eb6	3
0x55da4c693eb4	2	0x55da4c693eb8	4
0x55da4c693eb5	0	0x55da4c693eba	5
0x55da4c693eb6	3	0x55da4c693ebc	6
0x55da4c693eb7	0	0x55da4c693ebe	7
0x55da4c693eb8	4	0x55da4c693ec0	8
0x55da4c693eb9	0	0x55da4c693ec2	9
0x55da4c693eba	5	0x55da4c693ec4	10
0x55da4c693ebb	0	0x55da4c693ec6	11
0x55da4c693ebc	6	0x55da4c693ec8	12
0x55da4c693ebd	0	0x55da4c693eca	13
0x55da4c693ebe	7	0x55da4c693ecc	14
0x55da4c693ebf	0	0x55da4c693ece	15
0x55da4c693ec0	8	0x55da4c693ed0	16
0x55da4c693ec1	0	0x55da4c693ed2	17
0x55da4c693ec2	9	0x55da4c693ed4	18
0x55da4c693ec3	0	0x55da4c693ed6	19

Comme on s'y attend, une colonne sur deux est une adresse, et une sur deux une valeur. Ensuite, regardons un peu mieux ces valeurs et ces adresses. Et rappelons nous que le pointeur **p1** est de type **unsigned char***, et donc regarde la mémoire "octet par octet", et **p2** est de type **unsigned short***, et regarde la mémoire "deux octets par deux octets" :

- Au sujet des adresses : la première colonne correspond à **p1** (revoir la ligne 24 du code). Et dans le code, on voit que **p1** avance "par pas de 1". Et on voit bien que l'adresse augmente de 1 en 1. Bon ça semble logique.
 - Toujours au sujet des adresses : la troisième colonne correspond à **p2**. Et dans le code, on voit que **p2** avance aussi "par pas de 1". Or ... Dans la troisième colonne, les adresses avancent ... 2 par 2 ...

Bizarre ! Et en fait non. Il faut se souvenir que les pointeurs regardent la mémoire en fonction de leur type. Autrement dit : quand `p1` avance, il avance par "pas" de 1 `char`, donc de 1 octet, et quand `p2` avance, c'est par "pas" de 1 `short`, donc par pas de 2 octets ! et donc ce comportement est normal. On peut l'illustrer comme suit sur 4 répétitions de la boucle :



- Ensuite : les valeurs. Regardons la colonne 4, c'est le plus simple. Comme le tableau est un tableau de `unsigned short` et que le pointeur est de type `unsigned short*`, alors une case du tableau correspond exactement à un `unsigned short`, et ce que "voit" le pointeur fait aussi la même taille. Donc les deux se correspondent, et le pointeur voit, tour à tour, exactement les valeurs qui ont été mises dans le tableau, aussi parce qu'il se déplace, implicitement, d'un `unsigned short` à chaque passage dans la boucle (ligne 27).
- Pour la colonne 2, il faut avoir conscience que `p1` regarde la mémoire octet par octet et se déplace aussi d'un seul octet à chaque fois (ligne 26). Et donc il voit le tableau octet par octet, et donc décompose chaque `unsigned short` de `v1` octet par octet. Et donc la colonne 2 affiche dans l'ordre chaque case de 1 octet du tableau, comme montré sur le dessin.

IX.8 Au sujet des chaines de caractères

Pour les chaines de caractères `string` c'est la même chose : une chaîne de caractère revient à un tableau de caractères `char`, et donc les caractères seront dans des cases consécutives de 1 octet de long.

❖ En résumé ...

On a introduit un concept fondamental du C/C++ : le concept de **pointeur**. Ce concept met hélas en échec beaucoup de débutants en programmation, et il faut être vigilant : ce n'est pas forcément très difficile en soi, il y a peu de définitions ou de choses à savoir, mais il faut être rigoureux à chaque instant sur ce que l'on fait.

Notamment, il y a la difficulté du symbole `*` : il peut signifier soit que l'on crée un pointeur (par exemple : `int* p;`), ou que nous faisons un déréférencement (= on accède à la valeur pointée) pour un pointeur existant (par exemple : `*p`). C'est la plus grande source de confusion si on n'est pas attentif.

Une fois que l'on a dit cela, on ne s'est attardé qu'à la syntaxe. Les pointeurs sont très utiles en C/C++ et donnent lieu à des techniques de programmation qu'il faut absolument connaître, notamment avec les fonctions (prochain chapitre) ou encore dans la programmation par microcontrôleur au semestre prochain. ■

Chapitre X – Fonctions avec plusieurs résultats

① Motivations et objectifs

On a décrit le concept de pointeur. Même si vous avez (nous l'espérons) compris de quoi il s'agissait, vous avez pu retenir qu'un pointeur c'est juste un "jouet" pour "jouer avec la mémoire" et c'est tout. C'est se tromper beaucoup. Même si en C/C++ "modernisé", qui est l'objet de ce cours, les pointeurs interviennent moins qu'en C pur et dur, il y a une technique du C/C++ et sur les pointeurs sans laquelle on ne peut pas travailler : le "**passage par adresse**" dans les fonctions, qui permet de gérer des résultats multiples avec une fonction. C'est très important et à connaître absolument. Voyons cela.

X.1 Entrées et sorties multiples des fonctions en C/C++

Ce que l'on voudrait faire en C/C++, c'est pouvoir écrire (par exemple) l'équivalent du petit code Python ci-dessous, qui comporte une fonction avec plusieurs résultats, ce que l'on ne sait pas encore faire :

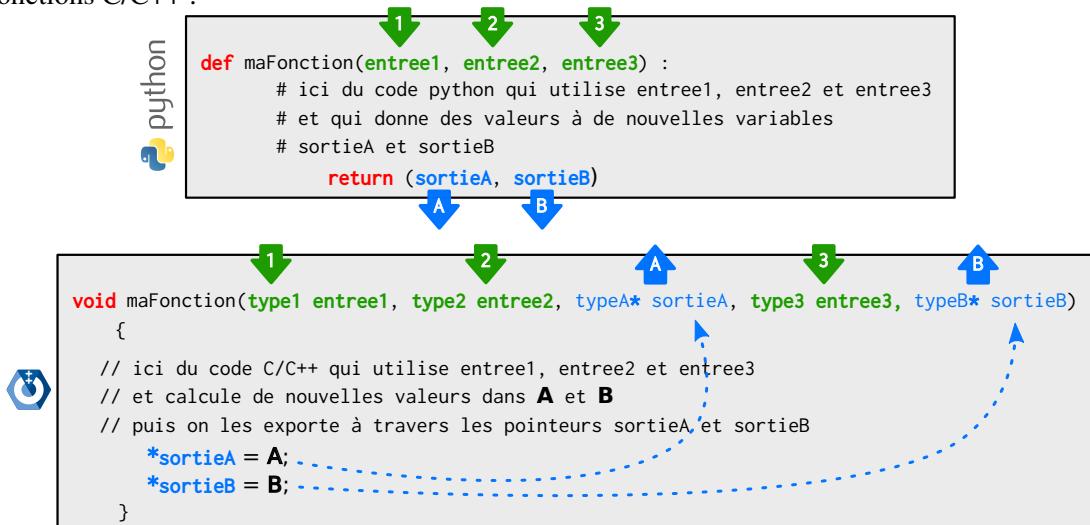


```
def somDiff(a,b):  
    s = a+b  
    d = a-b  
    return s,d
```

```
r1,r2 = somDiff(5,8)
```

Cette fonction calcule la somme et la différence de deux valeurs, et les renvoie. Autrement dit cette fonction a 2 entrées (**a** et **b**) et 2 sorties (**s** et **d**).

On a vu que, même si **return** existe en C/C++, il ne permet d'avoir qu'une seule sortie. Pour augmenter le nombre de sorties dans un programme C/C++, le principe général, en C/C++, est d'utiliser **des paramètres supplémentaires** qui sont des "**pointeurs**" pour obtenir des sorties. On justifiera pourquoi plus tard. Schématiquement cela revient à faire la comparaison suivante entre les fonctions en Python et les fonctions C/C++ :



Ainsi, alors qu'en Python, les entrées étaient entre les parenthèses et les sorties indiquées par `return`¹, en C/C++ on peut décider que l'on fait "tout" via les parenthèses (donc via les "paramètres") :

Pour qu'un paramètre soit utilisable comme un sortie, il faut qu'il soit sous la forme d'un pointeur, sinon c'est forcément une entrée. On parle de paramètre de sortie, alors que les autres sont des paramètres d'entrée.

Notez que dans la mesure où on n'a plus besoin de `return` pour avoir des sorties, la fonction peut très bien ne rien renvoyer, d'où le `void` dans son prototype.

Maintenant voyons sur un code complet. Convertissons le code Python donné plus haut en C/C++ :

 python	
<pre> 1 2 3 4 def somDiff(a,b): 5 6 s = a+b 7 d = a-b 8 9 return s,d 10 11 12 13 14 r1,r2 = somDiff(5,8) 15 16 # 17 </pre>	<pre> 1 #include <iostream> 2 using namespace std; 3 4 void somDiff(int a,int b,int* s, int* d) 5 { 6 int calc_s = a+b; 7 int calc_d = a-b; 8 *s = calc_s; 9 *d = calc_d; 10 } 11 12 int main () 13 { 14 int r1,r2; 15 somDiff(5,8,&r1,&r2); 16 return 0; 17 }</pre>

Cela mérite beaucoup d'explications :

- Comme annoncé, la fonction a 2 entrées et 2 sorties, on le voit ligne 4. En C/C++, il faut que tous les paramètres soient accompagné de leur type, et donc on a 2 entrées `a` et `b` qui sont passées "simplement", on dit qu'elles sont passées "par valeur", et deux sorties `s` et `d`, passées par des pointeurs, on dont on dit qu'elles sont passées "par adresse".
- Ensuite le code de la fonction se situe lignes 5 à 10. Dans la version C/C++, ce calcul est fait dans de nouvelles variables (lignes 6 et 7), `calc_s` et `calc_d`. Puis, la phase qui sert exporter le résultat de ces calculs est représenté lignes 8 et 9, où on déréférence `s` et `d`. Comme on le voit, en C/C++ on n'utilise pas `return` pour cela.
- Au niveau du programme principal : en C/C++ on crée des variables `r1` et `r2` pour recevoir les résultats de la fonction. Et ligne 15, on voit que l'on transmet à `somDiff` la valeur 5 pour `a`, la valeur 8 pour `b`, et l'adresse de `r1` pour le pointeur `s`, ainsi que l'adresse de `r2` pour le pointeur `d`. C'est donc `r1` et `r2` qui recevront les résultats de `somDiff`, de façon analogue à ce qui est fait dans le code Python. Il est important de remarquer que le type de `&r1`, par exemple, correspond bien au type de `s`.

Note : On pouvait très bien écrire la fonction `somDiff` en C/C++ de façon plus compacte :

```

void somDiff(int a,int b,int* s, int* d)
{
    *s = a+b;
    *d = a-b;
}
```

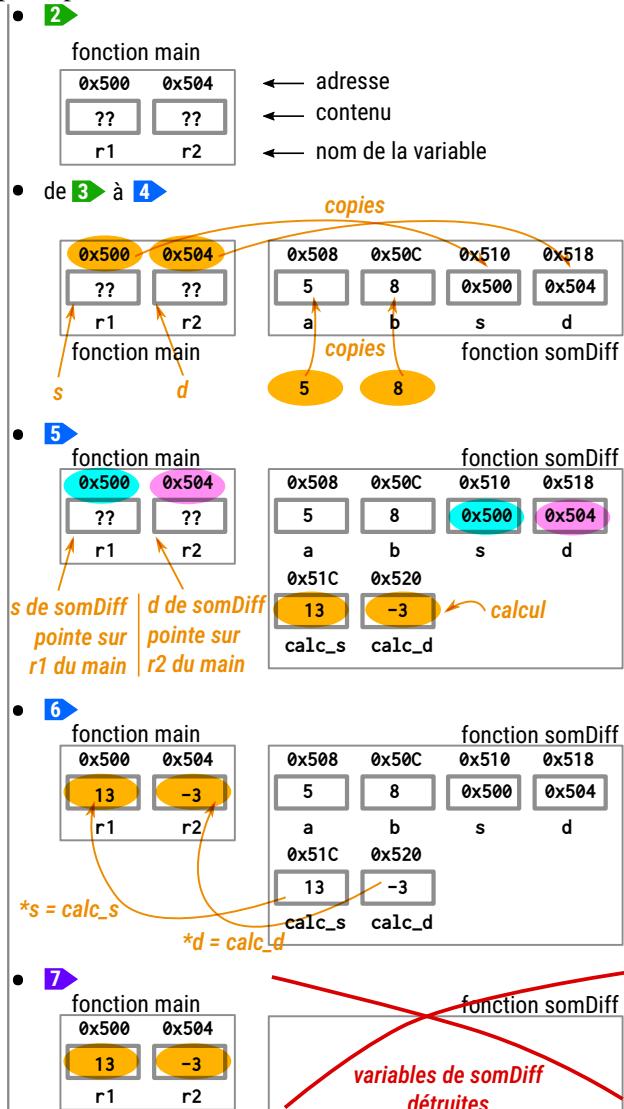
1. En toute rigueur c'est faux, on peut aussi avoir des sorties entre les parenthèses en Python mais cela dépend du type des variables, c'est lié au concept de "mutabilité". Ici on simplifie le discours en première approche.

X.2 Le passage par adresse en profondeur

On va maintenant expliquer pourquoi on donne des sorties aux fonctions C/C++ en utilisant le **passage par adresse**. Pour cela, exécutons le programme pas à pas.

```

1 #include <iostream>
2 using namespace std;
3
4 void somDiff(int a, int b, int* s, int* d)
5 {
6     int calc_s = a+b;
7     int calc_d = a-b;
8     *s = calc_s;
9     *d = calc_d;
10 }
11
12 int main ()
13 {
14     int r1, r2;
15     somDiff(5, 8, &r1, &r2);
16     cout << r1 << " " << r2 << "\n";
17     return 0;
18 }
```



Quelques commentaires :

- 1 (ligne 12) : L'exécution du programme commence au niveau du **main**
- 2 (ligne 14) : la déclaration des variables réserve la mémoire nécessaire pour les variables **r1** et **r2** du **main**. C'est ce qui est représenté dans la colonne de droite : les deux variables sont chacune à une certaine adresse, et leur contenu n'est pas connu, il est aléatoire.
- 3 (ligne 15) : on fait l'appel de fonction. Pour bien voir ce qui se passe, il faut comparer la ligne 15 et la ligne 4. On voit que **a** de **somDiff** vaudra 5, **b** de **somDiff** vaudra 8, que **s** prendra l'adresse de **r1**, et **d** prendra l'adresse de **r2**. Il faut aussi avoir en tête une chose très importante au niveau du C/C++ : le passage des paramètres se fait en faisant des copies à partir des originaux. Il faut aussi avoir en tête l'état dans lequel sont les variables **s** et **d** : **s** de **somDiff** pointe sur **r1** du **main**, de même que **d** de **somDiff** pointe sur **r2** du **main**. Et donc, grâce à ces pointeurs, on peut modifier, indirectement, les deux variables du **main**. C'est la raison précise pour laquelle on a fait du passage par adresse et c'est cela qu'il faut comprendre. Tout cela est représenté dans la colonne de droite à l'étape "de 3 à 4".
- 5 (ligne 8) : Avant l'exécution de la ligne 8, on a créé la mémoire pour les variables **calc_s** et **calc_d**, et on a calculé leurs valeurs selon les formules indiquées lignes 6 et 7.

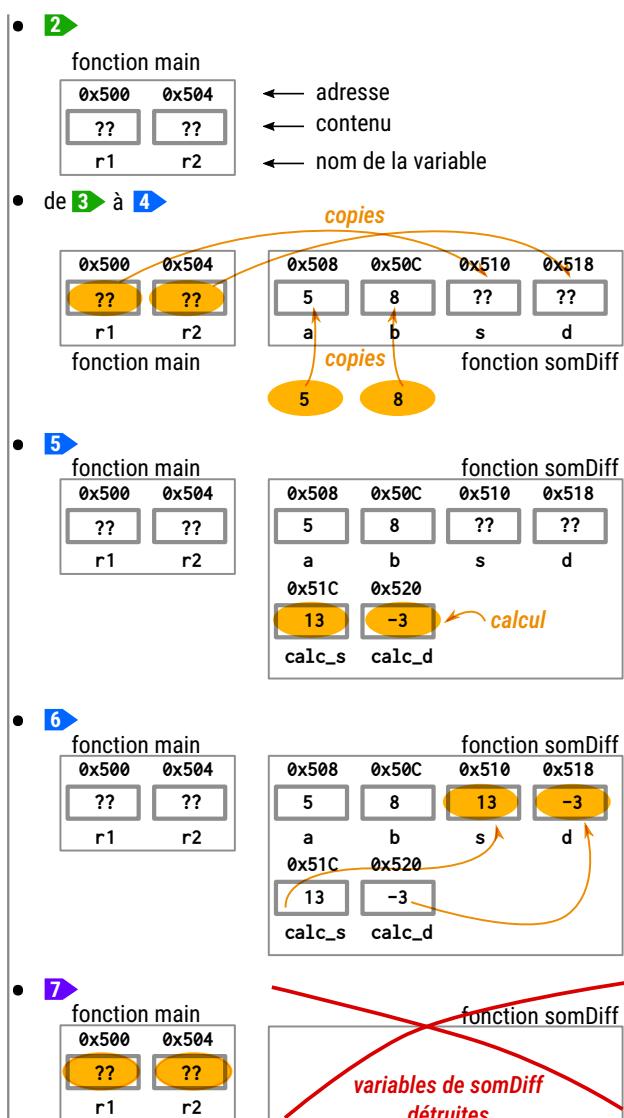
- 6) (ligne 10) : on a exécuté les lignes 8 et 9, et donc on utilise le fait que **s** et **d** de **somDiff** pointent sur les variables **r1** et **r2** du **main** : en déréférençant ces pointeurs, on modifie directement les variables du **main** à partir de la fonction. Et c'est cela, précisément, qui permet de dire que **r1** et **r2** ont récupéré des sorties de **somDiff**, puisque le passage dans **somDiff** en a modifié le contenu. C'est la façon que l'on a, en C/C++, d'avoir des sorties avec une fonction. C'est en effet déjà assez technique, mais "c'est comme ça", et **il faut le comprendre et savoir le faire**. On réutilisera souvent cette technique.
- 7) (ligne 16) : Enfin, après l'exécution de la fonction **somDiff**, le programme revient au niveau du **main**, juste après l'appel de fonction, et affiche les valeurs de **r1** et **r2** calculées dans la fonction, soit respectivement 13 et -3.

X.3 Et sans pointeur, ça marche vraiment pas? (non)

Reprendons maintenant exactement le même code mais sans pointeurs et regardons comment cela fonctionne :

```

1 #include <iostream>
2 using namespace std;
3
4 void somDiff(int a, int b, int s, int d)
5 {
6     int calc_s = a+b;
7     int calc_d = a-b;
8     s = calc_s;
9     d = calc_d;
10 }
11
12 int main ()
13 {
14     int r1,r2;
15     somDiff(5,8,r1,r2);
16     cout << r1 << " " << r2 << "\n";
17     return 0;
18 }
```



Vérifiez que l'on a supprimé les pointeurs : on a simplement passé les variables **r1** et **r2**, comme des variables "normales", et pas "par adresse". Commentons :

- 2) (ligne 14) : Tout se passe exactement comme précédemment, on aboutit à **r1** et **r2** créés, et disposant d'un contenu aléatoire.

- 3> (*ligne 15*) : on fait l'appel de fonction. Pour bien voir ce qui se passe, il faut comparer la ligne 15 et la ligne 4 : On voit que **a** de **somDiff** vaudra 5, **b** de **somDiff** vaudra 8, que **s** sera **r1**, et **d** sera **r2**. C'est encore plus important que précédemment : il faut bien avoir en tête que le passage des paramètres en C/C++ se fait **toujours** en faisant des copies à partir des originaux. Cette fois-ci, **on a juste recopié la valeur** de **r1** et **r2**. Tout cela est représenté dans la colonne de droite à l'étape "de 3> à 4>".
- 5> (*ligne 8*) : Avant l'exécution de la ligne 8, on a créé la mémoire pour les variables **calc_s** et **calc_d**, et on a calculé leurs valeurs selon les formules indiquées lignes 6 et 7 : rien de nouveau ici.
- 6> (*ligne 10*) : on a exécuté les lignes 8 et 9, et donc **s** et **d** prendront les valeurs calculées dans **calc_s** et **calc_d**. Mais il faut bien remarquer que cette fois, **s** et **d** **n'ont plus aucun lien avec r1 et r2 du main** : à l'appel de fonction, ce n'étaient que des copies !! Alors que dans le code précédent, ces variables contenaient les adresses de **r1** et **r2** : elles pouvaient donc les modifier. Ici ce n'est plus le cas, et c'est précisément pour cela que ce code ne fera rien d'intéressant.
- 7> (*ligne 16*) : Enfin, après l'exécution de la fonction **somDiff**, le programme revient au niveau du **main**, juste après l'appel de fonction, et affiche les valeurs de **r1** et **r2**. Mais cette fois ci, **r1** et **r2** n'ont pas récupéré le résultat de **calc_s** et **calc_d**, et ne contiennent donc rien d'intéressant.

C'est donc un échec, et les pointeurs étaient donc absolument indispensables pour obtenir le fonctionnement voulu.

X.4 Comparaison return et passage par adresse

On reprend un exemple donné en §IV.2 :

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main ()
6 {
7     int U;
8     int k;
9
10    U = 27;
11    k = 1;
12    vector<int> valeurs;
13    valeurs.resize(120);
14    valeurs[0] = U;
15    while (k < 120)
16    {
17        if (U % 2 == 0)
18        {
19            U = U / 2;
20        }
21        else
22        {
23            U = 3*U+1;
24        }
25        valeurs[k] = U;
26        k = k+1;
27    }
28
29    k = 0;

```

```

31   while (k < 120)
32   {
33     cout << valeurs[k] << "    ";
34     k = k+1;
35   }
36
37   return 0;
38 }
```

 [code/code-X-4-30.cpp]

Comme on l'a fait précédemment, on se propose de mieux organiser ce code en le découplant en fonctions :

1. Une première fonction qui réalise le calcul, avec comme **entrées** la valeur du premier terme U_0 et le nombre de termes n , avec comme **sortie** le tableau des valeurs,
2. Une seconde capable d'afficher le tableau de valeurs,
3. Un programme principal.

La fonction de calcul

Commençons par la fonction qui réalise le calcul. Appelons la **calcSyraccuse**, parce qu'elle calcule l'évolution de la suite de Syraccuse. On peut l'écrire de deux façons. Pas parce qu'elles sont algorithmiquement différentes (c'est le même calcul), mais parce qu'en C/C++ il y a deux mécanismes de sortie pour les fonctions : le **passage par adresse** et **return**. C'est ce que l'on vient de voir. Dans tous les cas :

- on a deux entrées : **int U0** et **int n**, comme indiqué plus haut,
- et une sortie, le tableau, donc **vector<int>**.

Ecrivons donc maintenant les deux versions :

En utilisant return	En utilisant le passage par adresse
<pre> 1 vector<int> calcSyraccuse(int U0 , int n) 2 { 3 int U=U0; 4 int k=1; 5 vector<int> valeurs; 6 valeurs.resize(n); 7 valeurs[0]=U0; 8 while (k < valeurs.size()) 9 { 10 if (U % 2 == 0) 11 { 12 U = U / 2; 13 } 14 else 15 { 16 U = 3*U+1; 17 } 18 valeurs[k] = U; 19 k = k+1; 20 } 21 return valeurs; }</pre> <p> [code/code-X-4-31.cpp]</p>	<pre> 1 void calcSyraccuse(int U0, int n, vector<int>* valeurs) 2 { 3 int U=U0; 4 int k=1; 5 (*valeurs).resize(n); 6 (*valeurs)[0]=U0; 7 while(k<(*valeurs).size()) 9 { 10 if (U % 2 == 0) 11 { 12 U = U / 2; 13 } 14 else 15 { 16 U = 3*U+1; 17 } 18 (*valeurs)[k] = U; 19 k = k+1; 20 }</pre> <p> [code/code-X-4-32.cpp]</p>

On voit bien que la version qui utilise `return` indique, sur la ligne 1, que ce qui va être renvoyé est du type `vector<int>` (c'est normal, c'est le type de la sortie), et cela correspond à la variable `valeurs` (ligne 21), qui est notée pour être de type `vector<int>` dans sa déclaration, ligne 5. Donc tout est bon.

Pour la fonction qui utilise le passage par adresse pour la sortie : comme on n'utilise pas `return`, le type indiqué comme type de retour est `void` (ligne 1). C'est normal, pas de `return`, pas type de retour. En revanche, on a passé un pointeur sur un vecteur : c'est le paramètre `vector<int>* valeurs` indiqué ligne 1 : c'est justement là que l'on utilise le passage par adresse pour la sortie : c'est ce qui va remplacer ce que l'on a fait avec `return` dans la version précédente. La fonction va donc modifier `valeurs`, au lieu de le créer comme c'était le cas dans la version précédente. Et donc, maintenant, pour utiliser la variable `valeurs`, on a forcément besoin d'utiliser l'opérateur `*`, comme on le voit lignes 6, 7 et 18 : on s'intéresse à "ce qui est pointé" par `valeurs`, et qui est un `vector<int>` qui sera passé en paramètre dans le `main`. Et donc dans cette version, la variable pointée sera modifiée par `calcSyracuse`.

Le programme complet

Donnons maintenant le code complet pour chaque version :

En utilisant return	En utilisant le passage par adresse
<pre> 1 #include <iostream> 2 #include <vector> 3 using namespace std; 4 5 vector<int> calcSyraccuse(int U0 6 , int n) 7 { 8 int U=U0; 9 int k=1; 10 vector<int> valeurs; 11 valeurs.resize(n); 12 valeurs[0]=U0; 13 while (k < valeurs.size()) 14 { 15 if (U % 2 == 0) 16 { 17 U = U / 2; 18 } 19 else 20 { 21 U = 3*U+1; 22 } 23 valeurs[k] = U; 24 k = k+1; 25 } 26 return valeurs; 27 } 28 void afficheVector(vector<int> T 29) 30 { 31 int k=0; 32 cout << "Valeurs du tableau 33 : \n"; 34 while(k < T.size()) 35 { 36 cout << T[k] << "\n"; 37 k=k+1; 38 } 39 cout << "\n"; 40 } 41 int main() 42 { 43 vector<int> resultat; 44 resultat = calcSyraccuse 45 (27,120); 46 afficheVector(resultat); 47 return 0; 48 }</pre> <p> [code/code-X-4-33.cpp]</p>	<pre> 1 #include <iostream> 2 #include <vector> 3 using namespace std; 4 5 void calcSyraccuse(int U0,int n, 6 vector<int>* valeurs) 7 { 8 int U=U0; 9 int k=1; 10 (*valeurs).resize(n); 11 (*valeurs)[0]=U0; 12 while(k<(*valeurs).size()) 13 { 14 if (U % 2 == 0) 15 { 16 U = U / 2; 17 } 18 else 19 { 20 U = 3*U+1; 21 } 22 (*valeurs)[k] = U; 23 k = k+1; 24 } 25 } 26 27 28 void afficheVector(vector<int> T 29) 30 { 31 int k=0; 32 cout << "Valeurs du tableau 33 : \n"; 34 while(k < T.size()) 35 { 36 cout << T[k] << "\n"; 37 k=k+1; 38 } 39 cout << "\n"; 40 } 41 int main() 42 { 43 vector<int> resultat; 44 calcSyraccuse(27,120,& 45 resultat); 46 afficheVector(resultat); 47 return 0; 48 }</pre> <p> [code/code-X-4-34.cpp]</p>

Dans ces deux codes on voit la définition de 3 fonctions :

- **calcSyraccuse** (ligne 5),
- **afficheVector** (ligne 28),
- **main** (ligne 40), c'est à dire là où l'exécution débutera.

Ensuite, au niveau du **main**, les deux versions font la même chose :

- créer une variable **resultat** (ligne 42), qui est un **vector<int>**, qui va servir à récupérer le résultat de **calcSyraccuse** dans les deux cas,
- faire appel à la fonction **calcSyraccuse** (ligne 43), à chaque fois conformément à la définition qui correspond. Ici on peut faire une petite comparaison :
 - Pour la version qui utilise **return**, le résultat est récupéré en écrivant **resultat=** à gauche de l'appel de fonction : **resultat** récupérera donc le résultat de **valeurs**, renvoyé par la fonction **calcSyraccuse**, comme indiqué ligne 25. Pour faire son travail, la fonction **calcSyraccuse** reçoit également 27 pour **U0** et 120 pour **n**.
 - Pour la version qui utilise le passage par adresse, le 3ième paramètre de la fonction est l'adresse d'un **vector<int>** (ligne 5), et donc on fournit l'adresse de **resultat** comme 3ième paramètre de **calcSyraccuse** lors de l'appel de fonction ligne 43. Comme précédemment, la fonction reçoit aussi des valeurs pour **U0** et **n**.
- Puis dans les deux codes, on fait appel à la fonction **afficheVector** (ligne 44), avec pour seul paramètre **resultat**, qui sera considéré comme **T** par **afficheVector**
- Puis le programme principal se termine.

X.5 Utiliser à la fois return et passage par adresse

On a tout à fait le droit d'utiliser en même temps **return** et passage par adresse dans la même fonction. Par exemple, on peut très bien réécrire la fonction **somDiff** sous une forme mixte :

Version passage par adresse	Version return + passage par adresse
<pre> 1 #include <iostream> 2 using namespace std; 3 4 void somDiff(int a,int b,int* s, 5 int* d) 6 { 7 int calc_s = a+b; 8 int calc_d = a-b; 9 *s = calc_s; 10 *d = calc_d; 11 } 12 13 int main () 14 { 15 int r1,r2; 16 somDiff(5,8,&r1,&r2); 17 return 0; 18 }</pre>	<pre> 1 #include <iostream> 2 using namespace std; 3 4 int somDiff(int a,int b,int* d) 5 { 6 int calc_s = a+b; 7 int calc_d = a-b; 8 9 *d = calc_d; 10 return calc_s; 11 } 12 13 int main () 14 { 15 int r1,r2; 16 r1 = somDiff(5,8,&r2); 17 return 0; 18 }</pre>

 [code/code-X-5-35.cpp]

 [code/code-X-5-36.cpp]

Dans ce cas ça n'est pas très élégant. Le "style" d'écriture de code que l'on rencontre souvent, quand on a de nombreuses sorties, consiste à utiliser le passage par adresse pour la plupart des sorties et le **return** pour un "code d'erreur", qui renseigne si l'exécution de la fonction s'est bien déroulée ou pas. Typiquement, on renvoie 0 si tout s'est bien passé, sinon une valeur autre qui est associée à la nature de l'erreur : c'est la convention utilisée par le **main**, avec le fameux **return 0**.

X.6 Et avec les structures?

Ça fonctionne exactement de la même façon. La seule chose qui peut vous surprendre, c'est qu'une fois que l'on a créé une structure, on peut très bien créer un pointeur dessus. Par exemple on peut réécrire la fonction de multiplication de deux nombres complexes en fournissant le résultat par adresse :

Version return	Version passage par adresse
<pre> 1 #include <iostream> 2 using namespace std; 3 4 typedef struct 5 { 6 float Re; 7 float Im; 8 } complexe; 9 10 complexe produitCplx(complexe A, 11 complexe B) 12 { 13 complexe C; 14 C.Re= A.Re*B.Re - A.Im*B.Im; 15 C.Im= B.Re*A.Im + A.Re*B.Im; 16 17 return C; 18 } 19 20 void afficheCplx(complexe A) 21 { 22 char signeIm = '+'; 23 if(A.Im <0) { 24 signeIm = '-'; 25 A.Im = -A.Im; 26 } 27 cout << A.Re << signeIm << A 28 .Im << "i\n"; 29 } 30 31 int main() 32 { 33 complexe Z1,Z2,Z3; 34 Z1.Re = 7; 35 Z1.Im = 5; 36 Z2.Re = -2; 37 Z1.Im = 3; 38 Z3 = produitCplx(Z1,Z2); 39 afficheCplx(Z3); 40 }</pre> <p> [code/code-X-6-37.cpp]</p>	<pre> 1 #include <iostream> 2 using namespace std; 3 4 typedef struct 5 { 6 float Re; 7 float Im; 8 } complexe; 9 10 void produitCplx(complexe A, 11 complexe B, complexe* R) 12 { 13 complexe C; 14 C.Re= A.Re*B.Re - A.Im*B.Im; 15 C.Im= B.Re*A.Im + A.Re*B.Im; 16 17 *R = C; 18 } 19 20 void afficheCplx(complexe A) 21 { 22 char signeIm = '+'; 23 if(A.Im <0) { 24 signeIm = '-'; 25 A.Im = -A.Im; 26 } 27 cout << A.Re << signeIm << A 28 .Im << "i\n"; 29 } 30 31 int main() 32 { 33 complexe Z1,Z2,Z3; 34 Z1.Re = 7; 35 Z1.Im = 5; 36 Z2.Re = -2; 37 Z1.Im = 3; 38 produitCplx(Z1,Z2,&Z3); 39 afficheCplx(Z3); 40 }</pre> <p> [code/code-X-6-38.cpp]</p>

Notez que les seules lignes qui ont changé sont les lignes 10, 16 et 36. Vous voyez bien que l'on a créé un pointeur **complexe*** et qu'on l'a utilisé comme n'importe quel autre pointeur. Le reste est exactement ce que nous avons déjà décrit.

Optimisation

Dans le cas des structures, le passage par adresse peut être aussi utilisé pour accélérer l'appel aux fonctions. En effet, comme on l'a vu au §X.2, une fonction fait des copies des paramètres qu'on lui

transmet. C'est "comme ça" que fonctionne le C/C++, c'est en réalité un mécanisme vertueux dans la plupart des cas (on verra ça plus en TP). Mais si on regarde, la recopie n'a pas le même coût avec une variable habituelle du C/C++ et avec une structure. En effet, les variables habituelles du C/C++ font quelque chose comme quelques octets : `sizeof(char)` vaut 1, `sizeof(short)` vaut 2, `sizeof(long)` vaut 4, `sizeof(double)` vaut 8, et ensuite les pointeurs, par exemple `sizeof(void*)` ou encore `sizeof(short*)` valent 8 octets sur un PC 64 bits. Une structure `complex` fait déjà 2 fois la taille d'un `double`, donc 16 octets, et c'est une toute petite structure. Imaginez qu'il y ait des dizaines de champs, on pourrait arriver à des centaines d'octets sans problème. Et donc on augmente la taille des recopies, donc la mémoire et le temps nécessaires au passage des paramètres. C'est dommage, et ça peut impacter la puissance de calcul si on utilise de telles fonctions à l'intérieur d'une boucle (sur un appel seul l'effet n'est pas mesurable en général).

Alors il arrive (souvent) que l'on utilise le passage par adresse pour accélérer le temps d'appel à une fonction. Ainsi, on pourrait accélérer le code en passant systématiquement tout par adresse :

```
1 #include <iostream>
2 using namespace std;
3
4 typedef struct
5 {
6     float Re;
7     float Im;
8 } complexe;
9
10 void produitCplx(complexe* A, complexe* B, complexe* R)
11 {
12     complexe C;
13     C.Re= (*A).Re*(B).Re - (*A).Im*(B).Im;
14     C.Im= (*B).Re*(A).Im + (*A).Re*(B).Im;
15
16     *R = C;
17 }
18
19 void afficheCplx(complexe* A)
20 {
21     char signeIm = '+';
22     if( (*A).Im <0 ) {
23         signeIm = '-';
24         (*A).Im = - (*A).Im;
25     }
26     cout << (*A).Re << signeIm << (*A).Im << "i\n";
27 }
28
29 int main()
30 {
31     complexe Z1,Z2,Z3;
32     Z1.Re = 7;
33     Z1.Im = 5;
34     Z2.Re = -2;
35     Z1.Im = 3;
36     produitCplx(&Z1,&Z2, &Z3);
37     afficheCplx(&Z3);
38 }
```

 [code/code-X-6-39.cpp]

Remarque

Utiliser des pointeurs sur des structures est donc excessivement courant en C/C++. C'est tellement courant que la syntaxe que l'on a souvent utilisé ici, par exemple ligne 13 (`(*A).Re`), peut être remplacée par une autre syntaxe qui est dédiée à ce genre de situation. On peut l'écrire avec un autre opérateur, l'opérateur `->`, et donc on peut écrire : `A->Re`. Pour utiliser cet opérateur, il faut absolument avoir un pointeur sur une structure à gauche, et un champ de la structure à droite. Si vous ne vous sentez pas à l'aise avec les pointeurs, continuez à utiliser l'autre syntaxe (`(*A).Re`), la connaissance de l'opérateur `->` n'est pas exigée ici.)

❖ En résumé ...

On a vu que pour une grande part, les fonctions C/C++ fonctionnent comme les fonctions Python. Il y a malgré tout une nuance importante quand on veut avoir des **sorties**. On peut avoir autant de sorties que l'on veut avec une fonction C/C++, comme en Python, sauf que cela ne peut se faire qu'en utilisant le **passage par adresse**, qui est une technique fondamentale du C/C++ et qu'**il faut maîtriser absolument**. Malheureusement, cette technique nécessite l'utilisation de **pointeurs**, qui peut s'avérer difficile au premier abord. C'est malgré tout l'une des raisons parmi les plus fréquentes pour laquelle on rencontre des pointeurs en langage C/C++.

Dans des cas "plus simples" où on a une seule variable de sortie, on peut utiliser **return**, comme en Python, mais à ce moment là il faut bien prendre garde de renseigner le type de sortie à côté (à gauche) du nom de la fonction lors de l'écriture du prototype.

Enfin, on a vu comment organiser un code C/C++ comportant plusieurs fonctions, en combinant lisibilité et besoins du compilateur. ■

Chapitre XI – Compilation multi-fichiers et Bibliothèques

⌚ Motivations et objectifs

Un "gros" programme en C/C++ se découpe sur plusieurs fichiers, pour améliorer la lisibilité, séparer les différents aspects d'un programme, pour rendre tout ça plus pratique. En profondeur, c'est aussi l'idée qui se cache derrière l'idée de bibliothèque, sauf que le travail est déjà fait. C'est ce que l'on veut comprendre ici

Pour cela, on va commencer par découper un "gros" programme d'exemple en plusieurs fichiers, pour montrer les mécanismes principaux, pour ensuite aller vers la démarche que l'on doit utiliser quand on utilise des bibliothèques. Au sujet des bibliothèques, on va traiter deux sujets :

- Un exemple avec la bibliothèque standard du C/C++, où on va apprendre à créer des programmes capables de lire et d'écrire dans un fichier, et faire l'analyse de ce que cela signifie,
- Un exemple avec une bibliothèque non-standard, ici "rayLib", une bibliothèque simple d'accès dédiée au développement des jeux video.

XI.1 Compilation multi-fichiers

Partons d'un exemple de base très simple :

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 void combienDeZeros(double a) {
6     cout << "\n" << "log(" <<a << ") = " << log10(a) ;
7     cout << " parce qu'il y a " << (int) (log10(a) +1) << " chiffres\n";
8 }
9
10 int main()
11 {
12     combienDeZeros(100);
13
14     return 0;
15 }
```

 [code/code-XI-1-40.cpp]

qui affiche :

```
log(100) = 2 parce qu'il y a 3 chiffres
```

Supposons que ce code soit trop long et qu'on veuille le découper en plusieurs fichiers. On séparerait alors le code dans deux fichiers .cpp, par exemple `main.cpp` et `calcul.cpp`, et on organiseraient les choses comme suit :

fichier **main.cpp**

```

1 int main()
2 {
3 combienDeZeros(100);
4
5 return 0;
6 }

```

 [code/code-XI-1-41.cpp]
fichier **calcul.cpp**

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 void combienDeZeros(double a) {
6 cout << "\n" << "log(" << a << ") = " << log10(a);
7 cout << " parce qu'il y a " << (int)(log10(a)
+1) << " chiffres\n";
8 }

```

 [code/code-XI-1-42.cpp]

En effet : **main.cpp** n'a ni besoin de **iostream** ni de **cmath** pour fonctionner. Pour le reste c'est juste du copier/coller dans 2 fichiers séparés.

Malgré tout, si on compile l'ensemble :

```
g++ main.cpp calcul.cpp
```

On aboutit à une erreur de compilation :

```
main.cpp: In function 'int main()':
main.cpp:3:1: error: 'combienDeZeros' was not declared in this scope
  3 | combienDeZeros(100);
   | ^~~~~~
```

Et en effet : dans la fonction **main** du fichier **main.cpp**, il n'y a aucune information sur l'existence de la fonction **combienDeZeros**. Pour que le compilateur fonctionne, il faut qu'il sache au moins, dans **main.cpp**, quels sont les paramètres et les retours de la fonction **combienDeZeros**. Autrement dit il suffit d'indiquer le prototype de la fonction **combienDeZeros**, comme ci-dessous :

fichier **main.cpp**

```

1 void combienDeZeros(
2     double a);
3
4 int main()
5 {
6 combienDeZeros(100);
7
8 return 0;
}

```

 [code/code-XI-1-43.cpp]
fichier **calcul.cpp**

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 void combienDeZeros(double a) {
6 cout << "\n" << "log(" << a << ") = " << log10(a);
7 cout << " parce qu'il y a " << (int)(log10(a)
+1) << " chiffres\n";
8 }

```

 [code/code-XI-1-44.cpp]

Maintenant, si on compile, on réobtient exactement le même exécutable que lorsque tout le code était dans un seul fichier : Ca fonctionne.

Améliorons le "style"

Dans un gros projet, il n'est pas pratique de devoir indiquer les prototypes des fonctions que l'on utilise et provenant d'autres fichiers **.cpp**. En général, on préfère créer un fichier supplémentaire, dont l'extension est **.h**, qui contient la liste des prototypes des fonctions + les structures utilisées dans le fichier **.cpp** qui correspond. Et donc, on ferait la chose suivante :

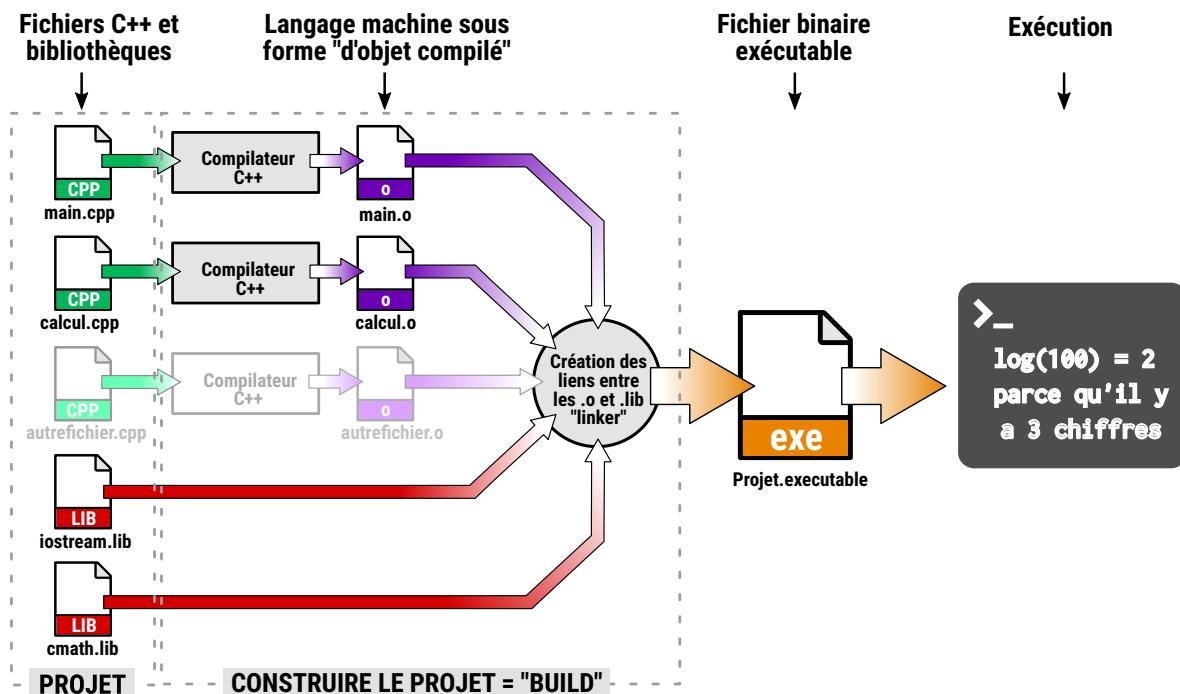
fichier main.cpp <pre> 1 #include "calcul.h" 2 3 int main() 4 { 5 combienDeZeros(100); 6 7 return 0; 8 } </pre>  [code/code-XI-1-45.cpp]	fichier calcul.h <pre> 1 #pragma once 2 3 void combienDeZeros(double a); </pre>  [code/code-XI-1-46.cpp]
fichier calcul.cpp <pre> 1 #include <iostream> 2 #include <cmath> 3 using namespace std; 4 5 void combienDeZeros(double a) { 6 cout << "\n" << "log(" << a << ")" = " << log10(a); 7 cout << " parce qu'il y a " << (int)(log10(a) 8 + 1) << " chiffres\n"; </pre>  [code/code-XI-1-47.cpp]	

Comme on le voit ci-dessus, dans le fichier **calcul.h**, on a juste mis le prototype de **combienDeZeros**, et on a ajouté autour une précaution, qu'on doit indiquer au début et à la fin de chaque fichier .h, qui permet d'éviter "*d'inclure des fichiers .h de façon circulaire*". Même si dit comme ça c'est obscur, on verra ça en TP. Il faut se souvenir que c'est une précaution à prendre systématiquement.

Ensuite, grâce au `#include "calcul.h"`, on ajoute au **main.cpp** le contenu de **calcul.h**, soit le prototype de **combienDeZeros**, ce qui permet la compilation.

Schéma global de compilation d'un projet

On a représenté ci-dessous ce qui se passe lorsque l'on compile un projet C/C++ comportant un ensemble de fichiers .cpp et qui inclue un ensemble de bibliothèques.



Note : Ce schéma de compilation est résumé. En effet, une phase préliminaire, appelée "précompilation" n'a pas été représentée. Lors de cette phase, les informations contenues dans les fichiers référencés dans les commandes `#include` sont intégrées aux différents fichiers `cpp`.



Important

Lorsque vous écrivez dans un code `#include<iostream>` ou `#include<cmath>`, vous avez l'impression "d'inclure" une bibliothèque. En réalité il n'en n'est rien. Tout ce que vous incluez ce sont les déclaration des fonctions qui permettent la compilation. Comme le montre le schéma ci-après, la compilation utilise aussi des fichiers supplémentaires, dont l'extension est souvent `.lib` (mais ça peut changer selon les systèmes d'exploitations et compilateurs), qui contiennent une version déjà compilée de ces fonctions. Ensuite, il reste juste à relier par le **linker** l'ensemble des codes utiles issus de tous ces fichiers pour former le programme exécutable final.

Notez aussi que pour les bibliothèques standard, le C/C++ "se débrouille" pour inclure le fichier de la bibliothèque. Mais si on utilise une bibliothèque externe, non standard, qui ne fait pas partie de l'installation "de base" du langage, alors il faut indiquer au **linker** que l'on a besoin de faire cet ajout au programme. On reviendra sur cet aspect. ■

XI.2 Exemple de Bibliothèque non-standard : rayLib

On va regarder comment on travaille avec une bibliothèque qui n'est pas fournie avec le C/C++ de base. On passe sur son installation mais on va parler de son utilisation.

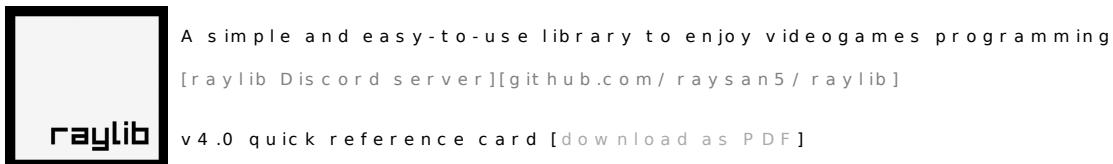
D'abord présentons la bibliothèque **rayLib**, la bibliothèque que l'on prend pour exemple. C'est une bibliothèque assez simple d'emploi qui donne tout ce qu'il faut pour programmer un jeu video. Cela signifie qu'elle contient un ensemble de **fonctions et structures** tout faits qui facilitent la programmation de jeux video.

Pour l'utiliser il faut savoir comment se documenter. La plupart des bibliothèques sont documentées de 3 façons :

- En donnant un **ensemble d'exemples de base** d'utilisation de la bibliothèque : des petits programmes écrits pour illustrer un aspect de la bibliothèque
- Avec une **documentation de référence** qui décrit une par une chaque fonction et structure de la bibliothèque.
- Parfois c'est en **lisant le code source de la bibliothèque** que l'on comprend comment elle fonctionne. Ça n'est pas toujours si difficile que ça, mais c'est une source d'information à ne pas négliger quand elle est disponible, surtout si le code est commenté de façon pertinente.

Extrait de la "documentation de référence"

En ce qui concerne **rayLib**, il y a de très nombreux exemples sur le site officiel sur internet <http://www.raylib.com>, et c'est la source principale de documentation. Il y a aussi un document de référence mais qui comporte peu d'informations. Regardons rapidement un extrait de cette documentation (disponible ici : <https://www.raylib.com/cheatsheet/cheatsheet.html>) :



```
// Window-related functions
void InitWindow(int width, int height, const char *title); // Initialize window and OpenGL context
bool WindowShouldClose(void); // Check if KEY_ESCAPE pressed or Close icon pressed
void CloseWindow(void); // Close window and unload OpenGL context
bool IsWindowReady(void); // Check if window has been initialized successfully
bool IsWindowFullscreen(void); // Check if window is currently fullscreen
bool IsWindowHidden(void); // Check if window is currently hidden (only PLATFORM_DESKTOP)
bool IsWindowMinimized(void); // Check if window is currently minimized (only PLATFORM_DESKTOP)
bool IsWindowMaximized(void); // Check if window is currently maximized (only PLATFORM_DESKTOP)
bool IsWindowFocused(void); // Check if window is currently focused (only PLATFORM_DESKTOP)
bool IsWindowResized(void); // Check if window has been resized last frame
bool IsWindowState(unsigned int flag); // Check if one specific window flag is enabled
void SetWindowState(unsigned int flags); // Set window configuration state using flags
void ClearWindowState(unsigned int flags); // Clear window configuration state flags
void ToggleFullscreen(void); // Toggle window state: fullscreen/windowed (only PLATFORM_DESKTOP)
void MaximizeWindow(void); // Set window state: maximized, if resizable (only PLATFORM_DESKTOP)
void MinimizeWindow(void); // Set window state: minimized, if resizable (only PLATFORM_DESKTOP)
void RestoreWindow(void); // Set window state: not minimized/maximized (only PLATFORM_DESKTOP)
void SetWindowIcon(Image image); // Set icon for window (only PLATFORM_DESKTOP)
void SetWindowTitle(const char *title); // Set title for window (only PLATFORM_DESKTOP)
void SetWindowPosition(int x, int y); // Set window position on screen (only PLATFORM_DESKTOP)
void SetWindowMonitor(int monitor); // Set monitor for the current window (fullscreen mode)
void SetWindowMinSize(int width, int height); // Set window minimum dimensions (for FLAG_WINDOW_RESIZABLE)
void SetWindowSize(int width, int height); // Set window dimensions
void *GetWindowHandle(void); // Get native window handle
int GetScreenWidth(void); // Get current screen width
int GetScreenHeight(void); // Get current screen height
int GetMonitorCount(void); // Get number of connected monitors
int GetCurrentMonitor(void); // Get current connected monitor
Vector2 GetMonitorPosition(int monitor); // Get specified monitor position
int GetMonitorWidth(int monitor); // Get specified monitor width (max available by monitor)
int GetMonitorHeight(int monitor); // Get specified monitor height (max available by monitor)
int GetMonitorPhysicalWidth(int monitor); // Get specified monitor physical width in millimetres
int GetMonitorPhysicalHeight(int monitor); // Get specified monitor physical height in millimetres
int GetMonitorRefreshRate(int monitor); // Get specified monitor refresh rate
Vector2 GetWindowPosition(void); // Get window position XY on monitor
Vector2 GetWindowSizeDPI(void); // Get window scale DPI factor
const char *GetMonitorName(int monitor); // Get the human-readable, UTF-8 encoded name of the primary monitor
void SetClipboardText(const char *text); // Set clipboard text content
const char *GetClipboardText(void); // Get clipboard text content

// Cursor-related functions
void ShowCursor(void); // Shows cursor
void HideCursor(void); // Hides cursor
bool IsCursorHidden(void); // Check if cursor is not visible
void EnableCursor(void); // Enables cursor (unlock cursor)
void DisableCursor(void); // Disables cursor (lock cursor)
bool IsCursorOnScreen(void); // Check if cursor is on the screen

// Drawing-related functions
void ClearBackground(Color color); // Set background color (framebuffer clear color)
void BeginDrawing(void); // Setup canvas (framebuffer) to start drawing
void EndDrawing(void); // End canvas drawing and swap buffers (double buffering)
void BeginMode2D(Camera2D camera); // Begin 2D mode with custom camera (2D)
void EndMode2D(void); // Ends 2D mode with custom camera
void BeginMode3D(Camera3D camera); // Begin 3D mode with custom camera (3D)
void EndMode3D(void); // Ends 3D mode and returns to default 2D orthographic mode
void BeginTextureMode(RenderTexture2D target); // Begin drawing to render texture
void EndTextureMode(void); // Ends drawing to render texture
void BeginShaderMode(Shader shader); // Begin custom shader drawing
void EndShaderMode(void); // End custom shader drawing (use default shader)
void BeginBlendMode(int mode); // Begin blending mode (alpha, additive, multiplied, subtract, custom)
void EndBlendMode(void); // End blending mode (reset to default: alpha blending)
void BeginScissorMode(int x, int y, int width, int height); // Begin scissor mode (define screen area for following drawing)
void EndScissorMode(void); // End scissor mode
void BeginVrStereoMode(VrStereoConfig config); // Begin stereo rendering (requires VR simulator)
void EndVrStereoMode(void); // End stereo rendering (requires VR simulator)
```

On voit juste la liste des fonctions avec leur prototype et, sous la forme d'un commentaire C/C++, sommairement ce qu'elle fait.

C'est très peu et beaucoup d'autres bibliothèques donnent plus d'informations que ça. Mais ça n'est pas le parti-pris de rayLib : rayLib est une bibliothèque qui se veut "facile à utiliser" (c'est pourquoi nous l'avons choisis comme "première expérience"), et se documente essentiellement via des exemples. Malgré tout, ce début de documentation nous indique au moins, pour chaque fonction, combien de paramètre elle prend, ce qu'elle renvoie, etc. Nous verrons en TP que l'on peut exploiter cela de façon efficace avec Eclipse.

Un exemple issu du site officiel de rayLib

Regardons alors un exemple de **rayLib**. Ils sont tous disponibles sur le site officiel de la bibliothèque, à partir d'ici : <https://www.raylib.com/examples.html>. Alors prenons le premier, le plus simple, qui affiche juste un texte dans une fenêtre graphique¹ :

```

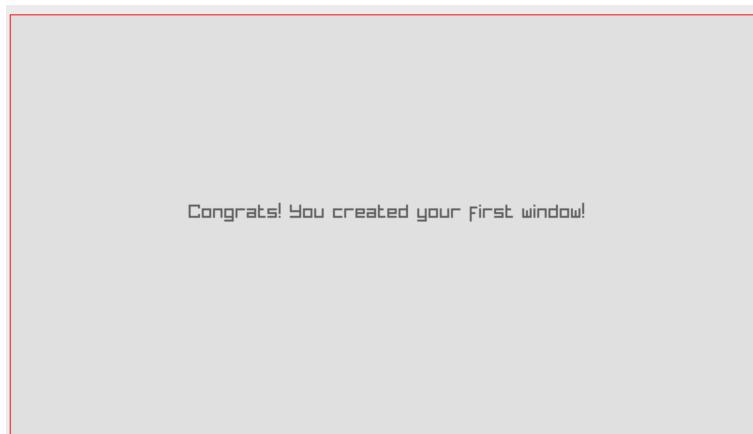
1  ****
2  *
3  *      raylib [core] example - Basic window
4  *      ...
5  *
6  ****
7
8 #include "raylib.h"
9
10 int main(void)
11 {
12     // Initialization
13     //-----
14     const int screenWidth = 800;
15     const int screenHeight = 450;
16
17     InitWindow(screenWidth, screenHeight, "raylib [core] example - basic
18         window");
19
20     SetTargetFPS(60);    // Set our game to run at 60 frames-per-second
21     //-----
22
23     // Main game loop
24     while (!WindowShouldClose()) // Detect window close button or ESC key
25     {
26         // Update
27         //-----
28         // TODO: Update your variables here
29         //-----
30         // Draw
31         //-----
32         BeginDrawing();
33
34             ClearBackground(RAYWHITE);
35
36             DrawText("Congrats! You created your first window!", 190, 200,
37                     20, LIGHTGRAY);
38
39         EndDrawing();
40         //-----
41     }
42
43     // De-Initialization
44     //-----
45     CloseWindow();           // Close window and OpenGL context
46     //-----
47     return 0;
48 }
```



[code/code-XI-2-48.cpp]

Et ce code affiche :

1. on a supprimé un ensemble de commentaires pour réduire le nombre de lignes



Si on sait lire l'anglais, entre les noms des fonctions, les quelques commentaires, et le résultat obtenu, on comprend assez bien ce que fait ce code et comment il le fait. Commentons un peu :

- Ligne 8 on inclue la bibliothèque
- ligne 17, on Initialise la fenêtre de dessin (en effet, les jeux dont on parle sont en mode graphique), en lui donnant une largeur et une hauteur de fenêtre (par les constantes définies lignes 14 et 15), et une chaîne de caractères qui s'avère être le titre de la fenêtre
- La ligne 19 est un peu obscure pour l'instant. Le commentaire dit que ça correspond à la fréquence de rafraîchissement du jeu. Passons pour le moment.
- Ligne 23 débute une boucle qui a l'air de s'arrêter quand on ferme la fenêtre. Autrement dit, tout ce qui est à l'intérieur de la boucle s'exécute de façon répétitive tant que la fenêtre n'est pas fermée.
- Les lignes 31 à 37 sont encadrées par **BeginDrawing** et **EndDrawing**. Ça suggère que tout ce que l'on fait comme dessin se fait forcément entre ces deux appels, autrement dit pour nous entre les lignes 32 et 35
- La ligne 33 donne l'impression qu'on efface le fond, avec une couleur appelée "**RAYWHITE**", qui doit être "le blanc de RayLib".
- Ligne 35 on fait appel à **DrawText**, qui doit être une fonction qui sert à ... dessiner du texte. De façon pas surprenante, on retrouve le texte à dessiner, on se doute que 190 et 200 sont une position x et y probablement, le 20 on peut imaginer que c'est la taille des lettres que l'on veut, et **LIGHTGRAY** serait la couleur, un gris clair. On peut se rassurer sur cette intuition en allant regarder la documentation de référence, qui dit :

```
void DrawText(const char *text, int posX, int posY, int fontSize,  
    Color color);  
// Draw text (using default font)
```

et on retrouve bien la façon "C pur et dur" de parler de chaînes de caractères avec **char*** au lieu de **string**

- Ligne 43, une fois que la fermeture de fenêtre a été demandée, on ferme la fenêtre pour de bon.

Ces observations sont assez compatibles avec ce qui a été affiché par l'application. Bien. On a donc déjà l'ossature d'une application écrite avec rayLib. Mais on n'a pas fait tout ça juste pour afficher un texte, on veut faire plus !

Allons un peu plus loin

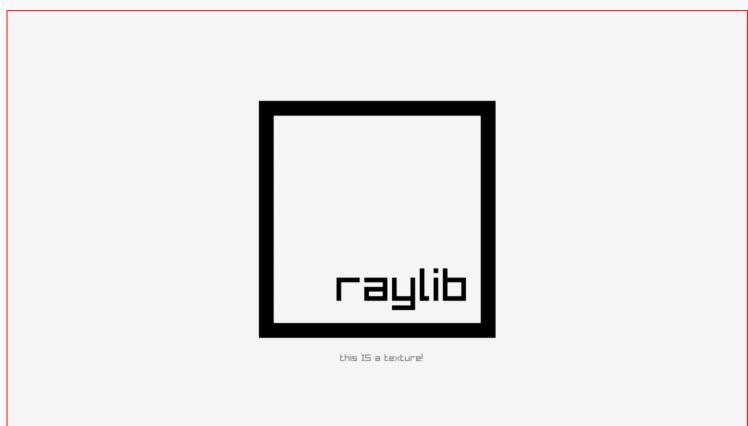
Imaginons maintenant que l'on veuille expérimenter une des fonctionnalités de base des jeux video : afficher des images. Alors prenons un nouvel exemple parmi ceux fournis avec rayLib :

```
1  ****  
2  *  
3  *      raylib [textures] example - Texture loading and drawing  
4  *
```

```
5 ****
6
7 #include "raylib.h"
8
9 int main(void)
10 {
11     // Initialization
12     //-----
13     const int screenWidth = 800;
14     const int screenHeight = 450;
15
16     InitWindow(screenWidth, screenHeight, "raylib [textures] example - "
17                 texture loading and drawing");
18
19     // NOTE: Textures MUST be loaded after Window initialization (OpenGL
20     // context is required)
21     Texture2D texture = LoadTexture("resources/raylib_logo.png");           //
22     // Texture loading
23
24     //-----
25
26     // Main game loop
27     while (!WindowShouldClose()) // Detect window close button or ESC key
28     {
29         // Update
30         //-----
31         // TODO: Update your variables here
32         //-----
33
34         // Draw
35         //-----
36         BeginDrawing();
37
38         ClearBackground(RAYWHITE);
39
40         DrawTexture(texture, screenWidth/2 - texture.width/2,
41                     screenHeight/2 - texture.height/2, WHITE);
42
43         DrawText("this IS a texture!", 360, 370, 10, GRAY);
44
45         EndDrawing();
46         //-----
47
48     // De-Initialization
49     //-----
50     UnloadTexture(texture);          // Texture unloading
51
52     CloseWindow();                  // Close window and OpenGL context
53
54     return 0;
55 }
```

 [code/code-XI-2-49.cpp]

Et qui affiche :



Quelques commentaires :

- On voit que la globalité du code est la même : même initialisation, même boucle principale, etc. C'est rassurant.
- Ligne 19, on a :

```
Texture2D texture = LoadTexture("resources/raylib_logo.png");
// Texture loading
```

De par la syntaxe, on doit comprendre immédiatement que l'on vient de créer une variable nommée **texture**, dont le type est **Texture2D**, et que cette variable récupère le résultat d'une fonction **LoadTexture** qui prend en paramètre le nom d'un fichier **png** sous la forme d'une chaîne de caractères. On peut donc penser que le programme a chargé un fichier **png**, et le référence en mémoire à travers la variable **texture**. *Et c'est absolument considérable ! charger un fichier PNG est quelque chose de complexe, c'est un format d'image compressé qui gère la transparence, et la bibliothèque le fait pour nous !*

- Ligne 36, on a la ligne :

```
DrawTexture(texture, screenWidth/2 - texture.width/2, screenHeight/2 -
            texture.height/2, WHITE);
```

qui prend notamment en paramètre la variable **texture**, puis ce qui est manifestement des coordonnées, puis une couleur. On imagine assez bien que cette ligne sert à dessiner l'image qui a été chargée précédemment. Et on comprend au passage que **texture** est en réalité une structure dans laquelle on peut récupérer, sûrement entre autres, la largeur et la hauteur de l'image. On peut aller regarder dans le code de la bibliothèque quelle est la description de cette structure :

```
// Texture, tex data stored in GPU memory (VRAM)
typedef struct Texture {
    unsigned int id;           // OpenGL texture id
    int width;                 // Texture base width
    int height;                // Texture base height
    int mipmaps;               // Mipmap levels, 1 by default
    int format;                // Data format (PixelFormat type)
} Texture;

// Texture2D, same as Texture
typedef Texture Texture2D;
```

Et donc on voit qu'il y a diverses informations, notamment un **id**, qui doit référencer en mémoire l'image.

- Pour finir, on a à la fin du code, ligne 45, un **UnloadTexture** qui doit supprimer la texture de la mémoire.

Donc voilà. Avec cet exemple, on a appris qu'il y avait 3 fonctions pour gérer les images : **LoadTexture**, **DrawTexture**, et **UnloadTexture**.

C'est de cette façon que l'on progresse dans la connaissance d'une bibliothèque.

Programmons un jeu vidéo

En TP on va programmer un petit jeu vidéo. Voici le genre de chose que ça va donner :



Les traits jaunes permettent de révéler un morceau de la structure du jeu. La carte est donc constituée de "motifs" parmi 3 catégories : on a l'herbe, les arbres et la montagne, et on réutilise ces 3 motifs, qui font chacun 64 x 64 pixels, pour dessiner la carte complète. Ca semble rudimentaire mais beaucoup de jeux vidéo utilisent ce principe, avec bien entendu une plus grande variété de dessins.



Ensuite, on a un personnage, un viking, qu'il faut pouvoir animer. On dispose donc d'une grande quantité d'images pour toutes les attitudes et positions du personnage. On appelle cela des "sprites" ² :



Et donc le programme va consister à d'une part dessiner la carte à partir des 3 motifs et d'une description de la carte, et à dessiner un sprite parmi tous ceux qui sont fournis en fonction de l'action du joueur (par exemple sur quelle touche il appuie) et du temps qui passe. On devra donc se documenter sur la gestion du temps / des animations avec la bibliothèque, ainsi que sur l'utilisation du clavier.

❖ En résumé ...

2. Sous licence opensource . Réalisé avec les logiciels opensource **inkscape** pour le dessin et **OpenToonz** pour l'animation.

On a vu que les `#include` que l'on utilise servent surtout à faire référence à la bibliothèque mais n'incluent pas la bibliothèque elle-même. La bibliothèque correspond à un autre fichier, qui contient en général du code machine / binaire, que l'on va inclure (`#include`) dans le projet pour satisfaire les besoins du linker pour générer l'exécutable.

On a aussi appris à séparer un code long en C/C++ en plusieurs codes séparées et "joints" dans un seul et même projet. Ces fichiers peuvent "connaître" les contenus des autres par la création de fichiers `.h` que l'on peut ensuite `include` dans chacun des fichiers C/C++ qui en a besoin.

Enfin, on a commencé la prise en main de la bibliothèque `rayLib` que nous utiliserons en TP et qui nous servira à programmer un petit jeu video. Les concepts basiques ont été définis : carte, sprites, animation, etc. ■

Chapitre XII – Gestion des fichiers en C/C++ et Formats de Fichiers

⌚ Motivations et objectifs

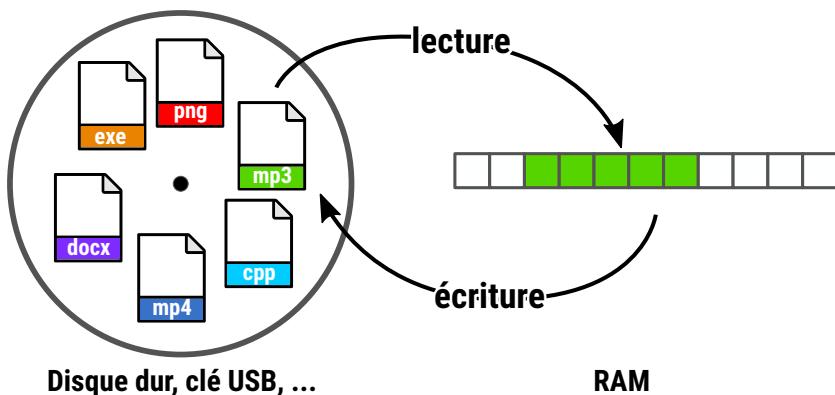
Comme avec d'autres langages, le C/C++ permet de "gérer" des fichiers, c'est à dire qu'il permet de réaliser le chargement et l'écriture d'informations, de "données", sur disque dur. Cela passe par le concept de "fichiers", qui lui-même est un concept complètement lié à la présence du système d'exploitation. On va donc voir un peu plus en profondeur comment cela fonctionne, et aussi nous confronter au codage interne des fichiers, en s'intéressant à l'idée de ce que signifie un "format de fichier".

XII.1 Lire et écrire des données dans un fichier : de quoi parle-t-on exactement ?

Un ordinateur exécute un programme sur son processeur / CPU. Et donc, les seules données qu'il peut traiter sont celles qui sont accessibles au CPU, autrement dit pour l'essentiel il s'agit de ce qui est contenu en RAM.

Ainsi, lorsque l'on récupère des informations dans un fichier ou que l'on écrit des informations dans un fichier, on ne le fait jamais directement : on passe toujours par de la RAM en intermédiaire. Ainsi, on peut essentiellement réaliser deux opérations sur un fichier :

- **lire** des informations **à partir d'un fichier**, et cela consiste à copier des informations contenues dans le fichier vers la RAM.
- **écrire** des informations **vers un fichier**, et cela consiste à copier des informations contenues dans un bloc de RAM vers le fichier. C'est ce que l'on représente sur l'illustration ci-dessous :



Quelques exemples :

- Quand on charge un fichier texte, par exemple un code C/C++ ou un code Python avec un logiciel, il est intégralement copié en RAM

- C'est le cas avec la plupart des choses que l'on fait avec un PC : charger un logiciel, une image, ou encore un fichier Word ou Powerpoint
- Mais c'est assez différent avec les fichiers qui concernent des "flux", typiquement des fichiers video (MP4 par exemple) ou des fichiers son (MP3). Prenons le cas des fichiers video. Une video en assez haute résolution fait une taille qui peut varier mais elle fait facilement quelques Go ou quelques dizaines de Go. Or on peut très bien lire une telle video avec un PC ou un téléphone portable qui ne disposerait "que" de quelques Go, soit une taille qui peut être inférieure. Alors dans ce genre de cas, on prend en compte le fait que l'on n'a pas besoin d'avoir copié en RAM la totalité du fichier pour le jouer : à chaque instant, on a juste besoin d'avoir en mémoire quelques images, celles que l'on est en train de jouer. Il suffit que le chargement puisse se faire au fur et à mesure, et on peut oublier à chaque étape les images précédentes. Cela permet de jouer des films en consommant très peu de RAM, si on suppose que la vitesse de chargement à partir du disque est suffisamment rapide (ce qui est en pratique le cas).

XII.2 Rôle du système d'exploitation

Il faut bien comprendre que le concept de "Fichier" n'est pas naturel pour un ordinateur. Un disque dur, exactement comme la RAM, peut être vu simplement comme une série de cases de mémoire, formant au total une quantité de mémoire bien supérieure à celle d'une RAM et surtout permettant un stockage permanent de ces informations.

Le concept de "fichiers et répertoires" est un certain codage, maintenu par le **système d'exploitation**, qui permet de les gérer, de façon abstraite. Mais c'est déjà quelque chose d'évolué et qui n'est pas géré par l'électronique du disque dur. C'est précisément à cela que sert le "formattage" d'un disque dur : à créer sur le disque dur l'organisation qui permettra de gérer les fichiers et répertoires. Et cette organisation est propre à chaque système d'exploitation : le formattage du disque dur n'est pas fait de la même façon sous Windows, sous Mac ou sous Linux¹.

Malgré tout, toutes ces considérations n'ont pas énormément d'importance pour nous : en ce qui concerne la gestion des fichiers, **nous utilisons le système d'exploitation comme une bibliothèque**, c'est à dire que l'on dispose d'un ensemble de fonctions qui permettent de gérer les fichiers, sans nous soucier de comment le système s'en charge en détail. Mais encore mieux : le C/C++ propose une façon uniformisée de gérer les fichiers, qui n'est même plus dépendante du système d'exploitation : le C/C++ "normalise" donc ces accès, et cela permet d'écrire des programmes "universels", qui peuvent fonctionner sur n'importe quelle plateforme. Il suffira juste de les recompiler.

Les systèmes d'exploitation fonctionnent tous sur la même base et attendent tous du programmeur qu'il respecte le protocole ci-dessous, en 3 étapes, pour accéder à un fichier :

1. Une étape préliminaire appelée "**Ouverture du fichier**". Cette étape consiste à indiquer au système d'exploitation *avec quel fichier on va travailler* (c'est à dire son nom et son chemin), ainsi que le type d'opération que l'on compte faire, c'est à dire si on va faire *des lectures ou des écritures*. Cela permet au système d'exploitation de réservé l'utilisation de ce fichier à notre programme, en interdisant son accès aux autres programmes (notamment pour gérer les "écritures concurrentes").
2. Ensuite, on réalise les opérations que l'on veut vraiment faire : les "**lectures et écritures**" que l'on a besoin de faire, elles-mêmes.
3. Quand le travail est terminé, on doit l'indiquer au système d'exploitation, on parle de "**Fermeture du fichier**". Cela permet un ensemble de choses, en premier lieu de permettre aux autres programmes d'accéder au fichier, mais aussi de garantir que toutes les écritures sont terminées²

1. Non seulement cela mais cette organisation change aussi au fil des versions des différents systèmes d'exploitation ...

2. En effet, lorsque l'on demande une écriture dans un fichier, celle-ci n'est pas forcément réalisée instantanément, elle peut être "bufferisée", c'est à dire stockée en RAM, pour être réalisée ultérieurement par le système d'exploitation. Ce mécanisme permet, pour des raisons qu'on ne décrit pas ici, d'être plus efficace et rapide sur l'écriture des fichiers

XII.3 Gestion des fichiers en C/C++ : mode Texte

On va donc retrouver en C/C++ des éléments pour répondre à chacune de ces phases.

Exemple : lecture d'un fichier texte

On donne ci-dessous un petit code qui sert à charger un fichier "lisezmoi.txt" qui contient du texte.

```

1 #include <fstream>
2 #include <iostream>
3
4 using namespace std;
5
6 int main()
7 {
8     ifstream monFichier;
9     monFichier.open("lisezmoi.txt");
10    monFichier >> std::noskipws; // paramétrage
11
12    char uncaractere;
13    while(monFichier.eof()==false)
14    {
15        monFichier >> uncaractere;
16        cout << uncaractere;
17    }
18    monFichier.close();
19    return 0;
20 }
```



[code/code-XII-3-50.cpp]

Laissons pour l'instant de côté la ligne 10 qui est une subtilité, indispensable malgré tout. Décrivons l'essentiel de ce code :

- Ligne 1 : on inclut la bibliothèque **fstream** qui permet de gérer les fichiers. Elle fait partie des bibliothèques standard du C/C++, il n'y a rien à installer,
- Ligne 8 : on crée une variable de type **ifstream**, apportée par la bibliothèque **fstream**, qui permet de gérer les fichiers **en lecture**. Il existe un équivalent pour gérer les fichiers en écriture, nommé **ofstream**. On verra plus loin.
- Ligne 9 : On ouvre le fichier nommé "**lisezmoi.txt**"
- Ligne 12 : on déclare une variable qui permet de stocker un caractère. En effet, on va charger le fichier caractère par caractère, et afficher au fur et à mesure.
- Ligne 13 : on fait une boucle, qui s'arrêtera quand on aura lu tous les caractères du fichier. C'est ce que teste **monfichier.eof()==false**.
- Ligne 15 : on stocke un caractère du fichier dans la variable **uncaractere**. Au prochain tour de la boucle, ce sera le suivant qui sera lu, et ainsi de suite. Cette gestion est automatique.
- Ligne 16 : on affiche le caractère lu dans le fichier.
- Ligne 18 : on ferme le fichier.
- Remarque sur la ligne 10 : Par défaut, C/C++ ignore les caractères "espace", "tabulation" ainsi que les retours à la ligne. Cette ligne permet d'imposer la lecture de ces caractères. *A noter que ce comportement ne concerne que la lecture de fichiers contenant exclusivement du texte ! La ligne 10 est inutile dans d'autres cas.*

Organisation des données dans le fichier

Exactement comme on l'a montré lorsqu'on a exploré la mémoire, on peut explorer le contenu du fichier octet par octet pour comprendre l'organisation des données à l'intérieur. Supposons que le fichier "lisezmoi.txt" contienne simplement le texte :

Un peu de texte dans un fichier.

On peut regarder le contenu octet par octet avec la commande du terminal (sous Linux) `hexdump`³ :

```
# hexdump -C lisezmoi.txt

offset (hex) 0 1 2 3 4 5 6 7 8 9 A 0 C D E F 0123456789ABCDEF
00000000 55 6e 20 70 65 75 20 64 65 20 74 65 78 74 65 20 |Un peu de texte |
00000010 64 61 6e 73 20 75 6e 20 66 69 63 68 69 65 72 2e |dans un fichier.|
00000020 0a
00000021
```

Chaque ligne contient 16 octets du fichier : la première les 16 premiers, la seconde les 16 suivants, etc. Cela correspond à 0x10 octets en hexadécimal, et c'est ce que représente la colonne indiquée `offset` en rouge.

Ces octets sont représentés sur chaque ligne de 2 façons :

- d'abord par leur codage hexadécimal. Par exemple le tout premier octet du fichier a pour valeur 0x55, le second 0x6e, etc.
- Et dans la colonne de droite, on trouve la représentation à travers la table ASCII de chacun de ces octets. Ainsi, la lettre correspondant à 0x55 est le 'U' majuscule, celui de 0x6e est le 'n' minuscule, et ainsi de suite.

Et donc on voit bien dans cette décomposition octet par octet, que le codage d'un fichier texte consiste juste à stocker, les uns après les autres, les codes ASCII de chacun des caractères qui le composent. Ainsi, on comprend très bien le petit code que l'on a écrit précédemment : il a consisté à juste lire chaque octet du fichier (puisqu'on lit un un seul "char" à chaque étape), dans l'ordre, et à l'afficher les uns après les autres.

Autre exemple : écriture dans un fichier

Imaginons que l'on veuille stocker les valeurs que l'on a calculé pour la conjecture de Syracuse précédemment dans un fichier. On pourrait écrire les choses comme suit :

3. sous windows, vous pouvez utiliser des logiciels comme WinHex, HxD ou encore FreeHexEditorNeo <https://www.hhdsoftware.com/free-hex-editor>

Code qui affiche les valeurs de la conjecture de Syracuse

```

1 #include <iostream>
2 using namespace std;
3
4 int main ()
5 {
6     int U;
7     int k;
8
9     U = 27;
10    k = 1;
11
12    while (k < 120)
13    {
14        if (U % 2 == 0)
15        {
16            U = U / 2;
17        }
18        else
19        {
20            U = 3*U+1;
21        }
22        k = k+1;
23        cout << "U" << k << " = " << U << "\n";
24    }
25
26    return 0;
27 }
```

 [code/code-XII-3-51.cpp]

Code qui stocke dans un fichier les valeurs de la conjecture de Syracuse

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main ()
6 {
7     ofstream monCalcul;
8     monCalcul.open("syracuse.txt")
9     ;
10    int U;
11    int k;
12
13    U = 27;
14    k = 1;
15
16    while (k < 120)
17    {
18        if (U % 2 == 0)
19        {
20            U = U / 2;
21        }
22        else
23        {
24            U = 3*U+1;
25        }
26        k = k+1;
27        monCalcul << U << "\n";
28    }
29
30 }
```

 [code/code-XII-3-52.cpp]

Il n'y a pas grand chose à dire au sujet du code, tout est assez similaire avec la lecture :

- Ligne 7 on a créé un `ofstream` au lieu d'un `ifstream`, parce que maintenant nous voulons écrire dans un fichier,
- Ligne 8 on a ouvert le fichier,
- Ligne 26 on stocke la valeur de U sous forme d'une chaîne de caractère, sous la forme de son codage ASCII, suivi d'un retour à la ligne. *Il faut bien comprendre que les informations vont s'ajouter au fichier, comme on va le constater plus loin, et pas être remplacées!*
- Ligne 28 on ferme le fichier.

Et regardons aussi ce que l'on obtient dans le fichier :

```
# hexdump -C syracuse.txt
00000000  38 32 0a 34 31 0a 31 32  34 0a 36 32 0a 33 31 0a | 82.41.124.62.31. |
00000010  39 34 0a 34 37 0a 31 34  32 0a 37 31 0a 32 31 34 | 94.47.142.71.214 |
00000020  0a 31 30 37 0a 33 32 32  0a 31 36 31 0a 34 38 34 | .107.322.161.484 |
00000030  0a 32 34 32 0a 31 32 31  0a 33 36 34 0a 31 38 32 | .242.121.364.182 |
00000040  0a 39 31 0a 32 37 34 0a  31 33 37 0a 34 31 32 0a | .91.274.137.412. |
00000050  32 30 36 0a 31 30 33 0a  33 31 30 0a 31 35 35 0a | 206.103.310.155. |
00000060  34 36 36 0a 32 33 33 0a  37 30 30 0a 33 35 30 0a | 466.233.700.350. |
00000070  31 37 35 0a 35 32 36 0a  32 36 33 0a 37 39 30 0a | 175.526.263.790. |
00000080  33 39 35 0a 31 31 38 36  0a 35 39 33 0a 31 37 38 | 395.1186.593.178 |
```

```

00000090 30 0a 38 39 30 0a 34 34 35 0a 31 33 33 36 0a 36 | 0.890.445.1336.6 |
000000a0 36 38 0a 33 33 34 0a 31 36 37 0a 35 30 32 0a 32 | 68.334.167.502.2 |
000000b0 35 31 0a 37 35 34 0a 33 37 37 0a 31 31 33 32 0a | 51.754.377.1132. |
000000c0 35 36 36 0a 32 38 33 0a 38 35 30 0a 34 32 35 0a | 566.283.850.425. |
000000d0 31 32 37 36 0a 36 33 38 0a 33 31 39 0a 39 35 38 | 1276.638.319.958 |
000000e0 0a 34 37 39 0a 31 34 33 38 0a 37 31 39 0a 32 31 | .479.1438.719.21 |
000000f0 35 38 0a 31 30 37 39 0a 33 32 33 38 0a 31 36 31 | 58.1079.3238.161 |
00000100 39 0a 34 38 35 38 0a 32 34 32 39 0a 37 32 38 38 | 9.4858.2429.7288 |
00000110 0a 33 36 34 34 0a 31 38 32 32 0a 39 31 31 0a 32 | .3644.1822.911.2 |
00000120 37 33 34 0a 31 33 36 37 0a 34 31 30 32 0a 32 30 | 734.1367.4102.20 |
00000130 35 31 0a 36 31 35 34 0a 33 30 37 37 0a 39 32 33 | 51.6154.3077.923 |
00000140 32 0a 34 36 31 36 0a 32 33 30 38 0a 31 31 35 34 | 2.4616.2308.1154 |
00000150 0a 35 37 37 0a 31 37 33 32 0a 38 36 36 0a 34 33 | .577.1732.866.43 |
00000160 33 0a 31 33 30 30 0a 36 35 30 0a 33 32 35 0a 39 | 3.1300.650.325.9 |
00000170 37 36 0a 34 38 38 0a 32 34 34 0a 31 32 32 0a 36 | 76.488.244.122.6 |
00000180 31 0a 31 38 34 0a 39 32 0a 34 36 0a 32 33 0a 37 | 1.184.92.46.23.7 |
00000190 30 0a 33 35 0a 31 30 36 0a 35 33 0a 31 36 30 0a | 0.35.106.53.160. |
000001a0 38 30 0a 34 30 0a 32 30 0a 31 30 0a 35 0a 31 36 | 80.40.20.10.5.16 |
000001b0 0a 38 0a 34 0a 32 0a 31 0a 34 0a 32 0a 31 0a 34 | .8.4.2.1.4.2.1.4 |
000001c0 0a 32 0a 31 0a 34 0a 32 0a 31 0a | .2.1.4.2.1. |
000001cb

```

Sous linux, le "retour à la ligne" est codé comme `0x0a`, on le voit par exemple à l'octet numéro 2, à l'octet numéro 5, etc. Autrement dit, comme c'est indiqué dans le code, on a stocké chaque valeur sur une ligne dans le fichier. Pour les valeurs elles-mêmes, ce qui a été stocké est le "code ASCII" de chacune des valeurs. Par exemple pour la valeur "82", ce qui a été stocké c'est le code ASCII du 8 et le code ASCII du 2. Si on regarde maintenant le fichier dans un éditeur de texte, on voit (on n'a représenté que le début) :

```

82
41
124
62
31
94
47
142
71
214
107
322
..... etc

```

Ce qui est conforme à ce que l'on attend.

Relecture de syracuse.txt par un programme C/C++

On pourrait vouloir relire un tel fichier avec le C/C++. C'est même l'objectif d'un fichier : si on l'écrit c'est bien pour pouvoir le lire à un moment. Mais il y a une difficulté : il faut "comprendre", par code, que chaque valeur est séparée de la suivante par le caractère `0x0a`, soit un retour à la ligne. On pourrait reprendre le tout premier code que l'on a écrit au début de ce chapitre et chercher à détecter ce caractère, et ça marcherait. Mais ce serait un peu fastidieux, et c'est une opération tellement courante que C/C++ met à notre disposition une fonction qui gère le problème : la fonction `getline`. Voici comment on écrirait le code :

```

1 #include <iostream>
2 #include <fstream>
3 #include <sstream>
4 using namespace std;
5

```

```

6 int main ()
7 {
8     ifstream monCalcul;
9     monCalcul.open("syraccuse.txt");
10    while(monCalcul.eof()==false)
11    {
12        string ligne;
13        getline(monCalcul, ligne);
14        int valeur;
15        stringstream conversion;
16        conversion << ligne;
17        conversion >> valeur;
18        cout << valeur << "\n";
19    }
20    monCalcul.close();
21    return 0;
22 }
```

 [code/code-XII-3-53.cpp]

Quelques commentaires :

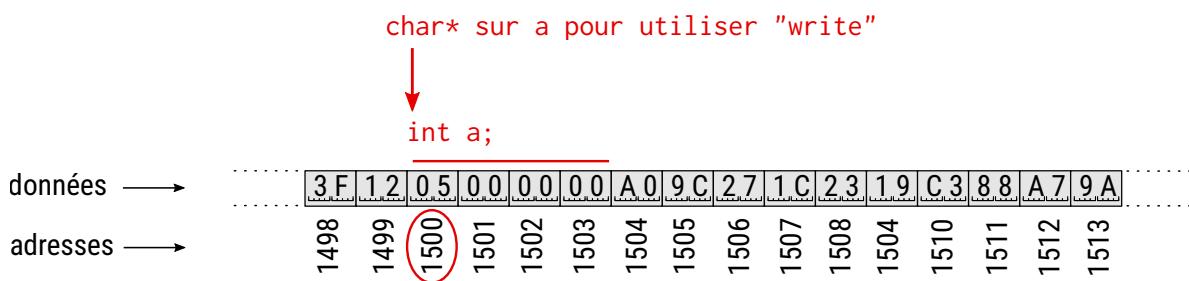
- On a créé un `ifstream` pour ouvrir le fichier en lecture
- On utilise `getline` à la ligne 12. Le résultat est que la variable `ligne` va contenir une ligne du fichier, et à chaque tour de la boucle ce sera la ligne suivante.
- La fin de la boucle, lignes 13 à 16, sert à convertir la chaîne `ligne` en valeur numérique. Pour cet exemple ça n'était pas indispensable, on pouvait se contenter d'afficher `ligne`, ça ne changerait rien. Mais ici on le fait pour montrer que le chargement fait que l'on obtient une chaîne de caractères, et que si on veut l'utiliser en tant que valeur, il faut procéder à une conversion. C'est ce qui permettrait, par exemple, de comparer les valeurs avec d'autres, ou de réaliser des opérations mathématiques dessus, ce qui est impossible tant que la valeur est codée sous forme de chaîne de caractères.

XII.4 Gestion des fichiers en C/C++ : mode Binaire

Jusqu'ici nous avons parlé de fichiers "textes", qui sont les plus simples à aborder, mais qui manquent parfois d'efficacité. En réalité, la plupart des formats de fichiers auxquels vous êtes confronté, ceux qui nécessitent de "la performance" (= taille réduite / rapidité à lire et écrire) ne sont pas en format "texte", mais en format "binaire". C'est le cas de MP4, PNG, JPEG, MP3, et beaucoup d'autres. Alors regardons comment on fait et ce que cela change.

Ecriture de données en mode Binaire

En C/C++, on dispose d'une fonction dédiée à l'écriture en mode binaire, elle se nomme simplement `write`. Elle prend en paramètre un pointeur de type `char*`, pour indiquer le début de la zone mémoire à écrire, et un entier qui indique la taille à écrire. Et c'est assez normal : la quantité de mémoire "de référence" est l'octet, comme on l'a vu pour la RAM, et on va donc donner le début de la zone en question, avec le pointeur, et la quantité à écrire. Par exemple si on veut écrire un entier contenu dans une variable `a` de type `int` (par exemple elle vaut 5, ici en codage Big Endian), située "quelque part" en mémoire RAM, on va se placer dans la situation ci-dessous :



Et donc pour écrire le codage binaire d'un entier dans le fichier, on a besoin, pour `write`, du pointeur `char*` indiqué sur la figure, positionné sur l'adresse de `a`, et on va demander l'écriture de 4 octets, puisque c'est la taille de `a`, comme le montre là encore la figure.

Pour illustrer ce principe, on va encore reprendre l'exemple de la sauvegarde des valeurs de la conjecture de Syracuse, mais pour générer un fichier binaire. Voici ce que cela donne :

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main ()
6 {
7     ofstream monCalcul;
8     monCalcul.open("syracuse.dat",ios::binary);
9     int U;
10    int k;
11
12    U = 27;
13    k = 0;
14
15    while (k < 120)
16    {
17        if (U % 2 == 0)
18        {
19            U = U / 2;
20        }
21        else
22        {
23            U = 3*U+1;
24        }
25        k = k+1;
26        char* p = (char*)(&U);
27        monCalcul.write(p,4);
28    }
29    monCalcul.close();
30    return 0;
31 }
```

[code/code-XII-4-54.cpp]

Quelques commentaires :

- Sur la ligne 8, on doit préciser `ios::binary` pour que l'écriture se fasse sans erreurs. En effet, par défaut l'écriture se fait en mode "texte", et il se peut que certaines conversions soient réalisées à cette occasion, pour des raisons que l'on n'évoquera pas ici. Le seul mode dans lequel on est sûrs d'écrire rigoureusement ce que l'on demande est le mode `ios::binary`, quelles que soient les données écrites, leur type, etc.
- Mais ce qui nous intéresse vraiment sont les lignes 26 et 27 :

- à la ligne 26, on crée un **char*** nommé **p** dont l'objectif est de stocker l'adresse de la variable **u** (à stocker puisque c'est celle qui contient les valeurs de la suite). On note que l'on a du passer par une conversion avec le **(char*)**, parce que pour le C/C++ il n'est pas naturel de vouloir associer un pointeur **char*** (soit **p**) à un pointeur **int*** (soit **&u**). On doit donc le forcer.
- puis on exécute **write** en précisant que l'on va écrire ce qui est situé à l'adresse indiquée par **p** (comme montré sur le dessin) et sur 4 octets.

Regardons maintenant le fichier généré :

```
$ hexdump -C syracuse.dat
00000000  52 00 00 00 29 00 00 00  7c 00 00 00 3e 00 00 00  |R....)....|....>...
00000010  1f 00 00 00 5e 00 00 00  2f 00 00 00 8e 00 00 00  |....^..../....|
00000020  47 00 00 00 d6 00 00 00  6b 00 00 00 42 01 00 00  |G.....k...B...|
00000030  a1 00 00 00 e4 01 00 00  f2 00 00 00 79 00 00 00  |.....y....|
00000040  6c 01 00 00 b6 00 00 00  5b 00 00 00 12 01 00 00  |1.....[....|
00000050  89 00 00 00 9c 01 00 00  ce 00 00 00 67 00 00 00  |.....g....|
00000060  36 01 00 00 9b 00 00 00  d2 01 00 00 e9 00 00 00  |6.....|
00000070  bc 02 00 00 5e 01 00 00  af 00 00 00 0e 02 00 00  |.....^....|
00000080  07 01 00 00 16 03 00 00  8b 01 00 00 a2 04 00 00  |.....|
00000090  51 02 00 00 f4 06 00 00  7a 03 00 00 bd 01 00 00  |Q.....z....|
000000a0  38 05 00 00 9c 02 00 00  4e 01 00 00 a7 00 00 00  |8.....N....|
000000b0  f6 01 00 00 fb 00 00 00  f2 02 00 00 79 01 00 00  |.....y....|
000000c0  6c 04 00 00 36 02 00 00  1b 01 00 00 52 03 00 00  |1...6.....R...|
000000d0  a9 01 00 00 fc 04 00 00  7e 02 00 00 3f 01 00 00  |.....~...?...|
000000e0  be 03 00 00 df 01 00 00  9e 05 00 00 cf 02 00 00  |.....|
000000f0  6e 08 00 00 37 04 00 00  a6 0c 00 00 53 06 00 00  |n...7.....S...|
00000100  fa 12 00 00 7d 09 00 00  78 1c 00 00 3c 0e 00 00  |.....}...x...<...|
00000110  1e 07 00 00 8f 03 00 00  ae 0a 00 00 57 05 00 00  |.....W....|
00000120  06 10 00 00 03 08 00 00  0a 18 00 00 05 0c 00 00  |.....|
00000130  10 24 00 00 08 12 00 00  04 09 00 00 82 04 00 00  |.$......|
00000140  41 02 00 00 c4 06 00 00  62 03 00 00 b1 01 00 00  |A.....b....|
00000150  14 05 00 00 8a 02 00 00  45 01 00 00 d0 03 00 00  |.....E....|
00000160  e8 01 00 00 f4 00 00 00  7a 00 00 00 3d 00 00 00  |.....z...=...|
00000170  b8 00 00 00 5c 00 00 00  2e 00 00 00 17 00 00 00  |.....\....|
00000180  46 00 00 00 23 00 00 00  6a 00 00 00 35 00 00 00  |F...#...j...5...|
00000190  a0 00 00 00 50 00 00 00  28 00 00 00 14 00 00 00  |.....P...(.....|
000001a0  0a 00 00 00 05 00 00 00  10 00 00 00 08 00 00 00  |.....|
000001b0  04 00 00 00 02 00 00 00  01 00 00 00 04 00 00 00  |.....|
000001c0  02 00 00 00 01 00 00 00  04 00 00 00 02 00 00 00  |.....|
000001d0  01 00 00 00 04 00 00 00  02 00 00 00 01 00 00 00  |.....|
000001e0
```

En regardant ça, on doit se souvenir que chaque entier utilise 4 octets. les 4 derniers octets, par exemple, doivent correspondre au dernier terme calculé de la suite, qui se termine par 4-2-1. Et c'est exactement ce que l'on voit : les 4 derniers octets codent la valeur "1" (mais en codage "big endian" puisqu'on est sur processeur intel), les 4 précédents la valeur "2", et les 4 précédents la valeur "4". On peut aussi regarder les valeurs au début, on voit la valeur 0x52 et 0x29 l'une après l'autre, qui correspondent aux valeurs 82 et 41 qui sont, là encore, le début de la suite.

Maintenant : si on regarde dans la partie droite de ce qui est affiché, on voit bien que tout ça n'est pas du tout lisible en ASCII. Et c'est bien normal, on a stocké les valeurs exactement comme elles sont en mémoire, on a déjà discuté de ce codage en §VIII.2 et de la différence avec le codage ASCII en §VIII.5.

Donc tout va bien, tout est clair. Mais alors : pourquoi a-t-on fait ça ? Pourquoi est-ce que ça peut être "mieux" que ce que l'on faisait avec le codage ASCII que nous avons utilisé précédemment ? C'est un peu rapide mais "parce que c'est souvent plus efficace". On peut noter par exemple que stocker dans un fichier

un nombre sous forme ASCII lui donne un codage dont la longueur peut changer selon que le nombre est grand ou pas : "12" prend 2 caractères, alors que "401453035" prend 9 caractères. Disposer d'une taille fixe (et souvent réduite) est déjà intéressant. Mais en plus, en stockant les nombres en binaire, on n'a plus besoin de réaliser des conversions pour pouvoir les réutiliser pour du calcul, et ces conversions, si elles sont faites en grand nombre, peuvent coûter "cher" en temps de calcul.

Lecture de données en mode Binaire

Alors regardons maintenant comment on peut charger un tel fichier. Les choses vont se passer plus ou moins de la même façon, sauf qu'on va ouvrir le fichier en mode "lecture binaire", puis on va, de la même façon que précédemment, faire pointer un pointeur **char*** sur la variable **int** qui va recevoir l'entier que l'on lit, et on va le donner en paramètre à une nouvelle fonction : **read**, qui joue un rôle symétrique à **write** utilisée précédemment. En matière de code, voici ce que cela donne :

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main ()
6 {
7     ifstream monFichier;
8     monFichier.open("syracuse.dat",ios::binary);
9     int U;
10
11    while (monFichier.eof() == false)
12    {
13        char* p = (char*)(&U);
14        monFichier.read(p,4);
15        cout << U << " ";
16    }
17    monFichier.close();
18    return 0;
19 }
```

 [code/code-XII-4-55.cpp]

Quelques commentaires :

- Ligne 7 et 8 on a ouvert le fichier en lecture (**ifstream**) et en mode binaire (**ios::binary**).
- Ligne 12 on boucle tant qu'on n'atteint pas la fin du fichier,
- Lignes 13 et 14 on récupère en entier en mode binaire à partir du fichier. Comme précédemment, on crée un **char***. Ce pointeur pointera sur la variable qui recevra les données binaires lues. La ligne 13 utilise la fonction **read** et lui demande la lecture de 4 octets à stocker à l'adresse pointée par **p**.
- Ligne 16 affiche l'entier lu.

Voici ce qu'il affiche :

```

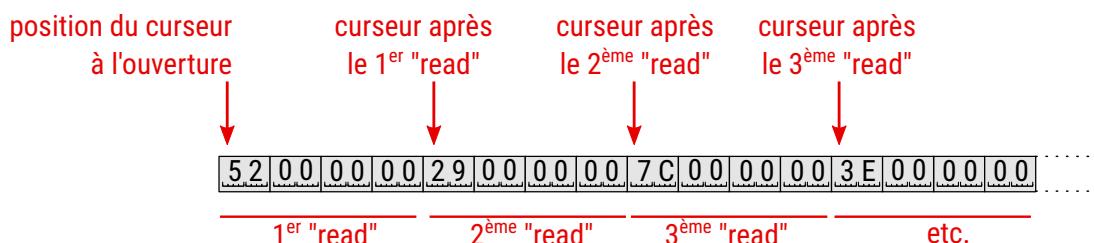
82 41 124 62 31 94 47 142 71 214 107 322 161 484 242 121 364 182 91 274 137
412 206 103 310 155 466 233 700 350 175 526 263 790 395 1186 593 1780 890
445 1336 668 334 167 502 251 754 377 1132 566 283 850 425 1276 638 319 958
479 1438 719 2158 1079 3238 1619 4858 2429 7288 3644 1822 911 2734 1367
4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866 433 1300 650 325 976
488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1 4 2
1 4 2 1 4 2 1 1
```

On retrouve bien les valeurs lues dans le fichier.

XII.5 Se déplacer dans un fichier : "curseur" sur fichier

On a vu que lorsque l'on fait par exemple une lecture dans un fichier, la prochaine lecture débute là où la précédente s'est terminée. C'est liée au fait qu'un curseur sur le fichier est géré et mémorisé automatiquement. Illustrons par exemple ce qui s'est passé lors de la relecture du fichier binaire "syracuse.dat" généré précédemment :

début de "syracuse.dat"



Et on a la possibilité de le déplacer directement, en utilisant la fonction **seekg**. Par exemple, imaginons que l'on veuille lire directement le 3ième **int** stocké dans le fichier, on peut écrire le code suivant :

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main ()
6 {
7     ifstream monFichier;
8     monFichier.open("syracuse.dat", ios::binary);
9     int U;
10    monFichier.seekg(2*sizeof(int), ios_base::beg);
11    char* p = (char*)(&U);
12    monFichier.read(p, 4);
13    cout << U << " ";
14    monFichier.close();
15
16    return 0;
17 }
```

[code/code-XII-5-56.cpp]

qui affiche "124" qui est bien la 3ième valeur. La seule chose nouvelle ici se situe ligne 10 : c'est la ligne qui déplace le curseur. La fonction **seekg** prend 2 paramètres :

- Le premier concerne la quantité d'octets dont il faut se déplacer. Comme on veut lire le 3ième, il faut sauter une quantité de mémoire qui vaut 2 fois la taille d'un entier, ce qui est indiqué par l'expression **2*sizeof(int)**
- Le second paramètre exprime "à partir d'où" ce déplacement se fait. **ios_base::beg** signifie "à partir du début du fichier", mais il existe d'autres modes :

symbole	signification
ios_base::beg	à partir du début du fichier : déplacement absolu. Les valeurs de déplacement sont forcément positives.
ios_base::cur	à partir de la position courante : déplacement relatif. Les valeurs de déplacement sont soit positives soit négatives selon que l'on veut se déplacer en direction de la fin ou en direction du début du fichier.
ios_base::end	à partir de la fin du fichier : déplacement absolu. Dans ce cas les valeurs de déplacement sont forcément négatives.

On peut aussi connaître la position courante du curseur sur le fichier avec la fonction `tellg`. Cette fonction donne la position du curseur en octets à partir du début. Par exemple, écrire :

```
cout << monFichier.tellg() << "\n";
```

entre les lignes 12 et 13 afficherait "12", parce qu'on a sauté 2 entiers et lu un troisième, on est donc à $3 \times 4 = 12$ octets du début. Un autre exemple : Si on veut connaître la taille d'un fichier, on peut simplement faire :

```
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{
    ifstream monFichier;
    monFichier.open("syraccuse.dat", ios::binary);
    monFichier.seekg(0, ios_base::end);
    cout << monFichier.tellg();
    monFichier.close();

    return 0;
}
```

Dans le cas du fichier "`syraccuse.dat`", cela affiche 480, qui est bien la taille en octets de ce fichier : on sait qu'on a calculé 120 termes et donc stocké 120 entiers, chacun faisant 4 octets, on a bien $120 \times 4 = 480$.

XII.6 "Formats" de fichiers : une introduction

Jusqu'à présent on a stocké dans les fichiers des informations simples : il s'agissait juste de valeurs les unes après les autres, ou de "caractères" les uns derrière les autres. Mais on utilise tous les jours des fichiers à différents "formats" : MP3, MP4, JPG, PNG, DOCX, CPP, etc, et ces fichiers contiennent des informations qui ne se limitent pas juste à une série simple de chiffres.

Par exemple un MP3 contient les données de la musique en effet, mais avec une structure complexe ("compressée") qui n'est pas vraiment un simple tableau de chiffres qui représente directement la musique, avec en plus des informations complémentaires comme le nom de la musique, de l'album, du musicien, la durée complète du morceau, etc. Donc beaucoup d'informations complémentaires. Et c'est tout cela qui définit le format du fichier : c'est "le format du fichier" qui décrit quelle organisation des données a été décidée pour stocker ensemble tous ces éléments, et qui donc fait qu'un MP3 est un MP3, un JPG un JPG, etc.

Sans avoir la prétention de faire un cours très abouti sur ce sujet qui pourrait nous amener très très loin, nous allons décrire un format plus simple que les formats actuels mais qui décrit correctement l'état d'esprit : le format BMP, un "vieux" format disponible sous Windows depuis que Windows existe ou presque. Si on se documente sur le "format BMP", voici le genre d'information que l'on peut avoir :

Name	Size	Offset	Description
Header	14 bytes		Windows Structure: BITMAPFILEHEADER
Signature	2 bytes	0000h	'BM'
FileSize	4 bytes	0002h	File size in bytes
reserved	4 bytes	0006h	unused (=0)
DataOffset	4 bytes	000Ah	Offset from beginning of file to the beginning of the bitmap data
InfoHeader	40 bytes		Windows Structure: BITMAPINFOHEADER
Size	4 bytes	000Eh	Size of InfoHeader =40
Width	4 bytes	0012h	Horizontal width of bitmap in pixels
Height	4 bytes	0016h	Vertical height of bitmap in pixels
Planes	2 bytes	001Ah	Number of Planes (=1)
Bits Per Pixel	2 bytes	001Ch	Bits per Pixel used to store palette entry information. This also identifies in an indirect way the number of possible colors. Possible values are: 1 = monochrome palette. NumColors = 1 4 = 4bit palletized. NumColors = 16 8 = 8bit palletized. NumColors = 256 16 = 16bit RGB. NumColors = 65536 24 = 24bit RGB. NumColors = 16M
Compression	4 bytes	001Eh	Type of Compression 0 = BI_RGB no compression 1 = BI_RLE8 8bit RLE encoding 2 = BI_RLE4 4bit RLE encoding
ImageSize	4 bytes	0022h	(compressed) Size of Image It is valid to set this =0 if Compression = 0
XpixelsPerM	4 bytes	0026h	horizontal resolution: Pixels/meter
YpixelsPerM	4 bytes	002Ah	vertical resolution: Pixels/meter
Colors Used	4 bytes	002Eh	Number of actually used colors. For a 8-bit / pixel bitmap this will be 100h or 256.
Important Colors	4 bytes	0032h	Number of important colors 0 = all
ColorTable	4 * NumColors bytes	0036h	present only if Info.BitsPerPixel less than 8 colors should be ordered by importance
Red	1 byte		Red intensity
Green	1 byte		Green intensity
Blue	1 byte		Blue intensity
reserved	1 byte		unused (=0)
repeated NumColors times			
Pixel Data	InfoHeader.ImageSize bytes		The image data

Cette documentation indique la description de "à quelle position" se situe chaque information dans le fichier, en nombre d'octets par rapport au début du fichier. En effet, il est indispensable de savoir exactement à quelle position on stocke telle information dans un fichier, parce qu'il faut se souvenir qu'une fois les données écrites, l'ensemble n'est qu'un ensemble d'octets et on a perdu la "trace" de leur origine : il s'agit de **données brutes**. Notez que c'est la même chose quand on transmet des informations par le réseau par exemple. C'est la raison qui fait qu'un logiciel comme **hexdump**, que nous avons utilisé, ne sait rien faire d'autre que montrer les octets qui constituent les fichiers que nous avons créé : il ne peut pas "deviner" que ce sont en réalité des entiers ou des chaînes de caractères que nous avons stocké, l'information n'est pas présente.

❖ En résumé ...

Nous avons appris à utiliser les bibliothèques standard du C/C++ pour écrire et lire dans des fichiers.
On a vu les 3 phases : ouverture, lecture/écriture et fermeture, indispensables dans cette gestion.

On a vu que l'on pouvait envisager surtout 2 façons de coder l'information dans un fichier : la

façon "texte" et la façon "binaire", chacune résultat d'un choix, choix qui peut avoir des conséquences sur la complexité du code à écrire ou sur la performance au chargement ou à l'écriture du fichiers, le mode binaire étant en général plus optimisé.

On a aussi réalisé l'analyse binaire du contenu d'un fichier pour en comprendre la structure. Tout cela nous a conduit à nous interroger sur le concept de "format de fichier", et sur ce que cela signifie. ■

PARTIE III

Pont entre C "modernisé" et C Standard

Chapitre XIII – Introduction

Nous avons vu jusqu’ici un langage C simplifié par une approche qui fait intervenir du C++. Cela a rendu tout un ensemble de choses bien plus simples. Malgré tout, bien que le C standard date des années 80, il n’est pas obsolète et de nombreux codes sont écrits en C standard, et il faut pouvoir au moins les relire pour les comprendre. Mais au delà : les habitudes sont têtues, et de nombreuses entreprises maintiennent des codes en C standard, pour des raisons historiques, et il est possible ou probable que vous soyez confronté au besoin de continuer un projet écrit en C standard, sans “avoir le droit” d’utiliser les éléments C++ que nous avons donné ici. Ce qui va changer beaucoup, c’est que les pointeurs sont alors partout.

On va donc revenir vers du C standard. Ca n’est pas à comprendre une régression : ça va être l’occasion de mieux comprendre encore ce qui se cache derrière la façon dont un processeur voit un programme et la mémoire, et vous rapproche donc des cours d’architecture et de l’assembleur. Allons y.

Chapitre XIV - Gestion de la RAM en C standard

② Motivations et objectifs

En C++, nous n'avons pas vraiment eu à "gérer la mémoire" : quand nous avons eu besoin de mémoire, nous avons soit créé des variables, soit créé des **vector**. On n'a jamais trop discuté de où était prise cette mémoire et comment elle était gérée : les ingrédients que l'on vous a donné sont assez universels et fonctionnent partout. En revanche, en C standard, on ne peut pas disposer d'une telle approche, et on a besoin d'en savoir plus sur le fonctionnement intime de la façon dont la mémoire est gérée.

XIV.1 Organisation de la RAM

En C/C++, on a réservé des zones de mémoires de 2 façons :

- Pour de petites quantités de mémoire, comme des variables, on a fait de simples déclarations :

```
int k;  
double f;  
matrice22 M; // si on a défini la structure matrice22
```

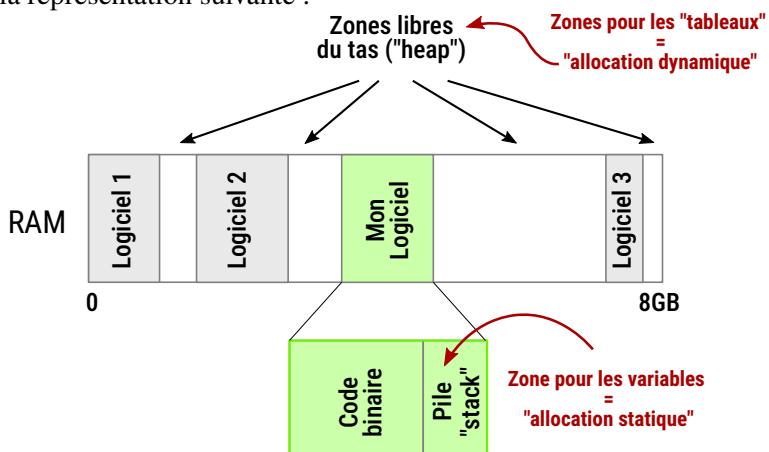
On appelle cela de "**L'allocation statique**" : elle concerne une zone mémoire assez petite, appelée "la pile" (ou "stack").

- Pour de grandes quantités de mémoire, comme des tableaux, on a utilisé les **vector** :

```
vector<double> tab;  
tab.resize(100);
```

On appelle cela de "**L'allocation dynamique**" : elle concerne une zone de mémoire appelée "le tas" (ou "heap").

On peut s'en faire la représentation suivante :



Ce que l'on peut retenir, c'est que :

- La "pile" est un **petit** espace de mémoire (quelques Mo) dédié au stockage des variables, qui fait partie de la mémoire du programme lui-même. Il y a une "pile" différente, locale, pour chaque programme en cours d'exécution dans l'ordinateur,

- Le "tas" est un **grand** espace mémoire (plusieurs Go) qui contient la totalité des programmes et zones mémoires utilisées et libres de l'ordinateur. Dans cet espace, on peut réserver de grands blocs de mémoire. Le plus grand espace que l'on peut réserver en un bloc fait la taille du plus gros "trou" dans les espaces inoccupés de la RAM (parties blanches sur l'image ci-dessus).

XIV.2 Allocation dynamique

C'est le cas le plus simple à traiter pour nous, parce qu'il ressemble aux **vectors** (sans être identique). L'allocation dynamique se base sur **malloc**, qui appartient à la bibliothèque **stdlib.h**, qui ne nécessite pas le "using namespace std" du C/C++. Voici sur un exemple la comparaison entre les deux versions (C/C++ à gauche, C standard à droite) :

fichier **exemple.cpp**

```

1 #include <vector>
2 using namespace std;
3
4 int main()
5 {
6     vector<double> T;
7     T.resize(50);
8
9     int k;
10    for(k=0 ; k<T.size() ; k=k+1) {
11        T[k]= k*k;
12    }
13
14    return 0;
15 }
```

 [code/code-XIV-2-57.cpp]

fichier **exemple.c**

```

1 #include <stdlib.h>
2
3 int main()
4 {
5     double* T;
6     T = (double*) malloc(50*sizeof(
7         double));
8
9     int k;
10    for(k=0 ; k<T.size() ; k=k+1) {
11        T[k]= k*k;
12    }
13
14    free(T);
15    return 0;
16 }
```

 [code/code-XIV-2-58.cpp]

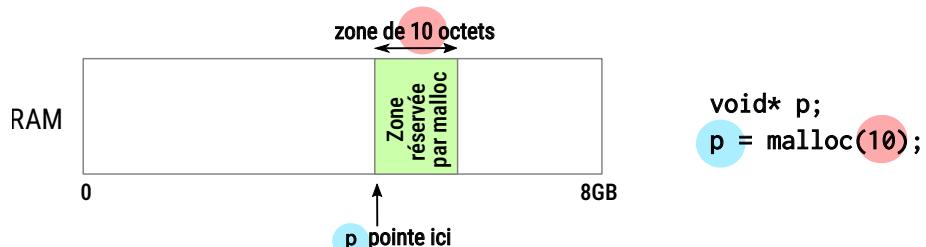
Sur cet exemple on a juste mis les carrés de **k** dans les cases du tableau, c'est juste pour faire un exemple, on aurait pu mettre n'importe quoi.

Les deux codes sont très similaires en apparence et au niveau du résultat ils font aussi exactement la même chose. Mais l'approche est différente et ça se voit bien dans la syntaxe. Décrivons :

- La chose qui marque le plus sont les lignes 6 et 7. En C/C++, on a déclaré un **vector<double>**, alors qu'en C standard, on crée un pointeur **double***. En effet les **vector** n'existent pas en C standard. Malgré tout, il n'est pas "fou" d'imaginer qu'un tableau puisse trouver une analogie avec une simple zone de mémoire (on a déjà vu ça), et donc puisse être représenté par un pointeur.
- Dans les deux cas, à la ligne 7, on attribue de la mémoire à **T** : en C/C++, par le **resize** et en indiquant le nombre de cases. Mais en C standard, les choses sont plus compliquées avec l'utilisation de **malloc**. Si on regarde la documentation du C standard, on trouve que **malloc** a pour prototype :

```
void* malloc(int nb_octets);
```

C'est une fonction qui prend en paramètre une quantité de mémoire voulue, en octets, et qui renvoie un pointeur sur le début de cette zone. Sur un exemple simple :



Dans le code qu'on a donné plus haut, on doit prendre en compte 2 choses supplémentaires :

- La zone de mémoire à réserver n'est pas de 10 octets mais de 50 **double**. Donc il faut en tout 50×8 octets, et il est plus clair de l'écrire, comme dans ce code, sous la forme **50*sizeof(double)** ("50 fois la taille d'un double").
- La mémoire réservée doit être vue comme une série de "double", et donc le type **void*** ne va pas nous convenir : c'est pourquoi on convertit le résultat de **malloc** en **double***, pour avoir le droit de faire l'affectation au pointeur **T**.
- Et puis il reste une chose surprenante : ligne 13, le **free(T)**. Cette ligne libère la mémoire réservée par **malloc**. Avec le **vector** du C/C++, la question ne se pose pas, elle est libérée automatiquement quand la variable **vector** est détruite. En C standard il faut faire plus attention.

Bien. Mais nous n'avons pas parlé du principal ! Il y a une chose qui est similaire mais qui est bizarre dans la version C standard. En effet, si on fait une analyse de type, à la ligne 5 on voit que **T** est un **double***, et que l'on a, ligne 10, la syntaxe **T[k]** comme avec un **vector** ? Si on avait suivi les règles de syntaxe apprises jusqu'ici, on aurait écrit à la place :

$*(T+k) = k*k;$

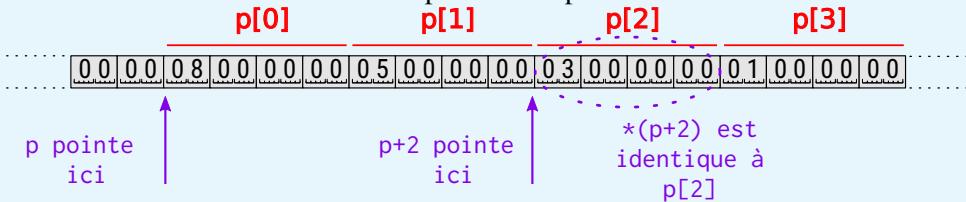
Et ça marcherait ! Mais ici on est devant une syntaxe liée aux pointeurs et dont on n'a jamais parlé.

Important

En C standard et en C/C++, les pointeurs supportent la notation **[]**. Il faut comprendre cette notation comme un équivalent de la notation *****. En effet, si **p** est un pointeur, on a toujours le droit de faire les équivalences :

$$\begin{aligned} (*p) &\equiv p[0] \\ *(p+k) &\equiv p[k] \end{aligned}$$

ce qui donne l'impression que **p** est une sorte de tableau plutôt qu'un pointeur, mais on sait très bien que ça n'est pas rigoureux. Ce qui est rigoureux c'est de réfléchir à cette notation en matière de déréférencement, comme avec l'opérateur *****, et de toujours penser à **p** comme à un pointeur, et **surtout pas** comme à un tableau ou à un **vector**. On peut se le représenter ainsi :



XIV.3 Allocation statique

En C/C++ ou en C standard, l'allocation statique est le type d'allocation que l'on fait quand on crée des variables. Autrement dit :

```
int k;
double f;
matrice22 M; // si on a défini la structure matrice22
```

Ces variables sont réservées dans la pile, qui fait quelques Mo pour chaque application sur un PC habituel. Et donc, avant de créer un nombre de variables suffisant pour saturer la pile, il en faut vraiment beaucoup (calculez combien **int** de 4 octets on met dans 1 Mo par exemple ...) ! et en pratique ça arrive difficilement, très difficilement.

Mais il y a une chose que l'on n'a pas mentionné en C/C++ : sur la pile, on peut aussi réserver des zones de mémoire plus grandes qu'une simple variable. C'est dangereux, il faut éviter, mais on le trouve dans beaucoup de codes C standard, parce que la syntaxe est simple. Par exemple, on peut créer une zone de 100 `int`, et une zone de 50 `matrice22` de la façon suivante :

```
int tab[100];
matrice22 tab2[50];
```

A partir de là, pour accéder à une case, on écrit par exemple :

```
tab[3] = 8;
```

c'est à dire exactement la même syntaxe qu'avec un `vector`. Donc c'est assez simple.

Mais pourquoi ne pas en avoir parlé en C/C++ ? En effet, cela existe aussi, et ça semble simple ! Il y a deux raisons à cela, et toutes deux mènent à des situations dangereuses :

Raison 1 : la taille de la pile est limitée

En effet, prenons le petit code ci-dessous :

```
int main()
{
    int tab[1000000000];

    return 0;
}
```

 [code/code-XIV-3-59.cpp]

On obtient :

```
Erreur de segmentation (core dumped)
```

Autrement dit le programme a planté. Alors que :

```
int main()
{
    int tab[100000];

    return 0;
}
```

 [code/code-XIV-3-60.cpp]

N'indique juste rien, autrement dit il ne plante pas. Le plantage que l'on a vu est lié au fait que la pile est trop petite pour stocker un tableau de 10^8 `int`, soit 400Mo, qui est bien au delà de la mémoire contenue dans la pile (en général).

Raison 2 : les "tableaux" statiques ne sont pas vraiment des tableaux

C'est une raison plus subtile, qui demande à rentrer plus profondément dans ce que sont les tableaux statiques. Il faut d'abord savoir que quand on écrit :

```
int main()
{
    int tab[100000];

    return 0;
}
```

 [code/code-XIV-3-61.cpp]

on peut avoir l'impression que le symbole `tab` est un tableau. **C'est complètement faux !**. Pour le C ou pour le C/C++, le symbole `tab`, sur cet exemple, est un `int*`, mais un `int*` particulier, qui pointe sur une zone réservée de 100000 entiers, elle-même située dans la pile.



Important

Souvenez vous en ! Quand on crée un tableau statique :

```
float T[100];
```

Le symbole **T** n'est PAS un tableau de 100 **float**, mais un **pointeur** sur une zone de 100 float : **T** est un **float***, et à ce titre il dispose de la syntaxe [] ! ■

On peut le prouver sur un exemple simple :

```
#include <iostream>
using namespace std;

int main()
{
int tab[10]={8,5,3,1,2,6,8,7,-1,4};

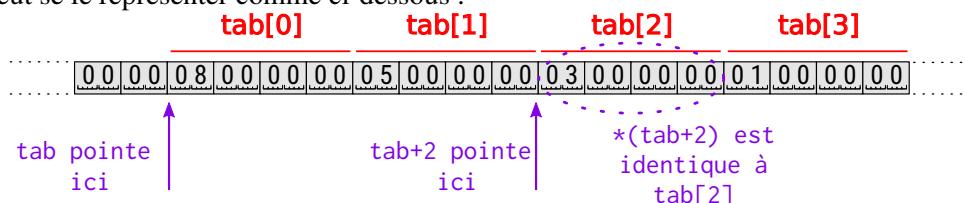
cout << tab[2] << "\n";
*(tab+2) = 7;
cout << tab[2] << "\n";

return 0;
}
```

 [code/code-XIV-3-62.cpp]

Le fait que **tab** supporte la notation *, sans que cela ne provoque d'erreur de compilation, signifie bien que **tab** est bien un pointeur.

On peut se le représenter comme ci-dessous :



Malheureusement, beaucoup de programmeurs débutants pensent que les tableaux statiques sont de vrais tableaux, comme le sont (mais pour de vrai) les **vector**. Or ce sont juste des pointeurs qui pointent sur des zones de mémoire réservées. Autrement dit, à part le fait que la mémoire a été réservée sur la pile, la situation est très similaire à ce que l'on a fait précédemment avec **malloc** : même si la syntaxe est assez différente, on aboutit à un pointeur sur une zone de mémoire. C'est la façon que le C standard a de décrire le concept de tableau.

XIV.4 Quelques Exemples

On écrit ici quelques codes qui sont similaires entre C/C++ modernisé et C standard.

Fonction qui prend un tableau en paramètre

fichier **exemple1.cpp**

```

1 #include <vector>
2 using namespace std;
3
4 float moyenne(vector<int> T);
5
6 int main()
7 {
8     vector<int> tab;
9     tab.resize(100);
10
11     int k;
12     for(k=0 ; k<tab.size() ; k=k+1)
13     {
14         tab[k]= k*k;
15     }
16
17     float m;
18     m = moyenne(tab);
19     return 0;
20 }
21
22 float moyenne(vector<int> T) {
23     int n;
24     float s=0;
25     for(n=0 ; n<T.size() ; n=n+1) {
26         s = s + T[n];
27     }
28
29     return (s/n);
30 }
```

 [code/code-XIV-4-63.cpp]

fichier **exemple1.c**

```

1 #include <stdlib.h>
2
3
4 float moyenne(int* T,int nb);
5
6 int main()
7 {
8     int* tab;
9     tab = (int*)malloc(100*sizeof(
10     int));
11
12     int k;
13     for(k=0 ; k< 100 ; k=k+1) {
14         tab[k]= k*k;
15     }
16
17     float m;
18     m = moyenne(tab,100);
19     return 0;
20 }
21
22 float moyenne(int* T, int nb) {
23     int n;
24     float s=0;
25     for(n=0 ; n < nb ; n=n+1) {
26         s = s + T[n];
27     }
28
29     return (s/n);
30 }
```

 [code/code-XIV-4-64.cpp]

Les codes sont très similaires, à part l'utilisation du `malloc` à la place du type `vector`. Mais il y a une différence importante malgré tout : la fonction `moyenne` a maintenant besoin de prendre, en C standard, la taille du tableau, soit `nb`, en plus du pointeur `T`. En effet, la fonction `size()` n'existe pas ici. Or, le pointeur ne représente que l'adresse de départ. Pour savoir où le tableau s'arrête, on a besoin de connaître sa taille. On le transmet en général sous la forme du nombre de cases qu'il contient : c'est indispensable.

On pourrait très bien écrire aussi ce code en allocation statique, puisque le tableau est raisonnablement petit :

fichier **exemple1b.c**

```

1 #include <stdlib.h>
2
3
4 float moyenne(int* T,int nb);
5
6 int main()
7 {
8     int tab[100];
9
10    int k;
```

```

11   for(k=0 ; k< 100 ; k=k+1) {
12     tab[k]= k*k;
13   }
14
15   float m;
16   m = moyenne(tab ,100);
17   return 0;
18 }
19
20
21 float moyenne(int* T, int nb) {
22   int n;
23   float s=0;
24   for(n=0 ; n < nb ; n=n+1) {
25     s = s + T[n];
26   }
27
28   return (s/n);
29 }
```

 [code/code-XIV-4-65.cpp]

On voit bien que l'on passe **tab** comme argument à la fonction **moyenne**. Or **tab** peut sembler être un tableau, alors que **tab** est bien un pointeur. C'est la raison pour laquelle on peut le passer comme premier paramètre à la fonction **moyenne**, dont le premier paramètre est un **float***.

Fonction qui renvoie un tableau

fichier **exemple2.cpp**

```

1 #include <vector>
2 using namespace std;
3
4 vector<float> hasard(int nb);
5
6 int main()
7 {
8   vector<float> tab;
9   tab = hasard(100);
10
11   return 0;
12 }
13
14 vector<float> hasard(int nb) {
15   vector<float> T;
16   T.resize(nb);
17   int x = 123456;
18   int k;
19   for(k=0; k<T.size(); k=k+1) {
20     x= (x*16807+1)%9781;
21     T[k] = x/9781.0;
22   }
23
24   return T;
25 }
```

 [code/code-XIV-4-66.cpp]

fichier **exemple2.c**

```

1 #include <stdlib.h>
2
3
4 float* hasard(int nb);
5
6 int main()
7 {
8   float* tab;
9   tab = hasard(100);
10
11   return 0;
12 }
13
14 float* hasard(int nb) {
15   float* T;
16   T = (float*) malloc(nb*sizeof(
17     float));
18   int x = 123456;
19   int k;
20   for(k=0; k<T.size(); k=k+1) {
21     x= (x*16807+1)%9781;
22     T[k] = x/9781.0;
23   }
24
25   return T;
26 }
```

 [code/code-XIV-4-67.cpp]

Donc là encore ça ne change pas grand chose. En revanche, il ne faut surtout pas faire cela en allocation

statique. **Le code ci-dessous est dangereux, il planter :**

```

1
2
3 float* hasard(int nb);
4
5 int main()
6 {
7     float* tab;
8     tab = hasard(100);
9
10    return 0;
11 }
12
13 float* hasard(int nb) {
14     float T[nb];
15
16     int x = 123456;
17     int k;
18     for(k=0; k<T.size(); k=k+1) {
19         x = (x*16807+1)%9781;
20         T[k] = x/9781.0;
21     }
22
23     return T;
24 }
25

```

 [code/code-XIV-4-68.cpp]

En effet, la fonction **hasard**, écrite comme ça, est dangereuse, et pour plusieurs raisons !

- La raison la plus importante, c'est que l'on fait un **return T** sur un tableau alloué comme variable statique. En effet : un tableau alloué par variable statique est automatiquement détruit lorsque l'on sort de la fonction. Or, **quand on écrit return T ici, ce que l'on renvoie, ce n'est pas le tableau, c'est le pointeur !**. Et c'est normal, on dit depuis le début que les tableaux en C standard n'existent pas vraiment, il n'existe que des pointeurs sur des zones de mémoire ! Et ici, on va renvoyer un pointeur sur un tableau détruit ! c'est très différent de ce que l'on a fait avec les **vector**, puisque quand on renvoie un **vector**, on ne renvoie pas un pointeur, mais bien un tableau ! et donc l'utilisation systématique des **vector** nous a évité ce genre de piège.
- La deuxième raison, ce qu'on l'on écrit une fonction avec **nb** comme paramètre. Or, **nb** peut être aussi grand que l'on veut, il pourrait valoir plusieurs millions. Et si on fait ça, on se retrouve à créer un tableau statique énorme, et donc on va dépasser l'espace de la pile, ce qui va faire planter le programme.



Important

On vient de voir deux bonnes raisons pour lesquelles il faut éviter d'utiliser l'allocation statique dès que l'on fait des choses un tout petit peu ambitieuses : renvoyer un tableau ou faire un grand tableau, ou un tableau dont on ne connaît pas la taille au moment où le code est écrit. ■

Tableaux à 2 dimensions

fichier **exemple3.cpp**

```

1 #include <vector>
2 using namespace std;
3
4
5 int main()
6 {
7     vector<vector<double>> M;
8     M.resize(10);
9     int ligne;
10    for(ligne = 0; ligne < M.size();
11        ligne = ligne +1) {
12        M[ligne].resize(15);
13    }
14
15    M[3][8] = 19.3;
16
17    return 0;
18 }
```

 [code/code-XIV-4-69.cpp]

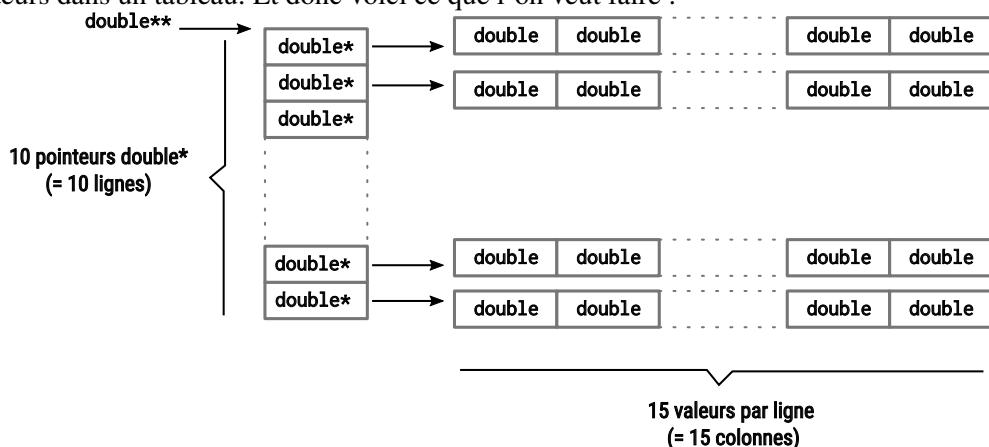
fichier **exemple3.c**

```

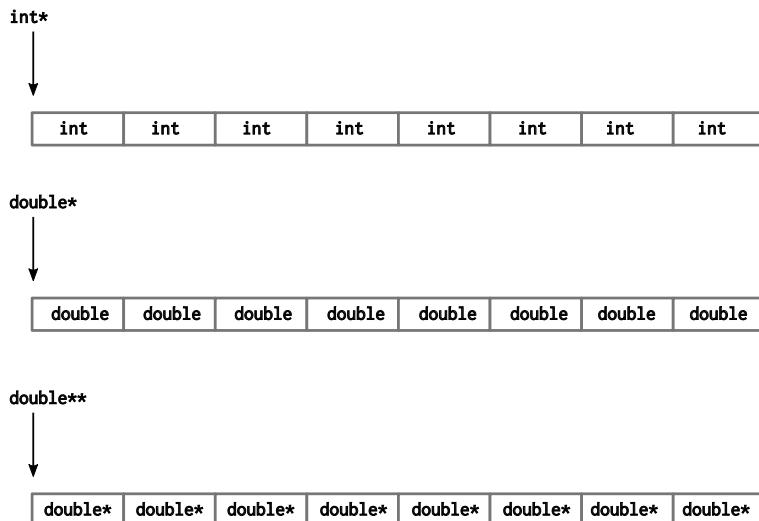
1 #include <stdlib.h>
2
3
4 int main()
5 {
6     double** M;
7     M = (double**) malloc(10*sizeof(
8         double*));
9     int ligne;
10    for(ligne = 0; ligne < M.size();
11        ligne = ligne +1) {
12        M[ligne] = (double*)malloc(15*
13            sizeof(double));
14    }
15
16    M[3][8] = 19.3;
17
18    return 0;
19 }
```

 [code/code-XIV-4-70.cpp]

Ici les codes sont assez similaires encore une fois, mais quelque chose peut surprendre : le **double****. Il faut d'abord comprendre ce que l'on fait : dans la boucle lignes 9 à 11, on crée 10 tableaux de 15 **double** chacun. Chacun de ces tableaux, comme tout tableau de **double** en C standard, est représenté par un **double***. Mais pour avoir 10 pointeurs **double***, il faudrait 10 variables de ce type. Pour 10 ça irait encore, mais imaginons qu'on en veuille 100 ou 1000 ? ça deviendrait fou à gérer, parce qu'on ne peut pas faire de boucle si on a autant de variables. On ne peut résoudre ce problème que l'en décidant de placer tous ces pointeurs dans un tableau. Et donc voici ce que l'on veut faire :



On veut donc créer un tableau pour stocker tous les pointeurs sur les lignes : ce sera donc un tableau dont chaque case est un **double**. Et pour cela, on fait exactement comme d'habitude : on ajoute une "*" au type du pointeur par rapport au type de la case. C'est ce que l'on fait depuis "toujours" en C standard, sans s'en rendre compte. Pour mieux le voir, voici sur quelques exemples :



Et donc il est bien normal que le pointeur sur les lignes soit de type `double**`.

❖ En résumé ...

Lorsque l'on travaille en C standard, il faut surtout se rappeler que le concept de "tableau" n'existe plus vraiment : tout est géré comme des zones de RAM pointées par des pointeurs, et c'est le seul outil que l'on a.

Il faut aussi se rappeler que l'allocation statique est potentiellement dangereuse et source d'erreurs, par rapport à l'allocation dynamique. Malgré tout, l'allocation dynamique existe la présence de bibliothèques, et donc d'au moins un minimum de système d'exploitation, alors que l'allocation statique n'a besoin de rien, elle fait partie du C lui-même. ■

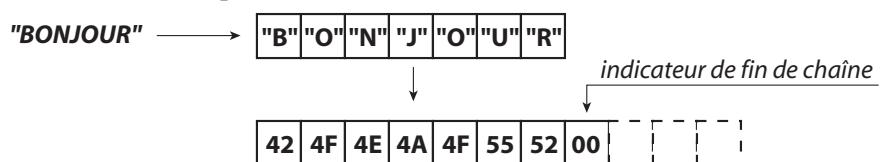
Chapitre XV – Gestion des chaînes de caractères

⌚ Motivations et objectifs

C'est une autre différence forte avec C++ : les chaînes de caractères. Les **string** n'existent plus, et il faut gérer les chaînes de caractères comme des zones de mémoire contenant des **char**. De même, **cout** et **cin** n'existent plus.

XV.1 Concept de chaîne de caractères en C standard

Une chaîne de caractères, en C standard, est simplement un tableau de **char** qui se termine par un 0. Entre le début de la chaîne et le 0, on a simplement une série de valeurs qui représentent le code ASCII des lettres stockées. Par exemple :



On n'a donc plus du tout les **string** comme en C++. Et ça change beaucoup de choses. Un petit exemple :

```
1 int main() {  
2     char S[50] = "Hello World !\n";  
3  
4     return 0;  
5 }
```

🌐 [code/code-XV-1-71.cpp]

Ici on crée une chaîne de caractères. Elle utilise une zone de 50 char, mais ne remplit que le début, parce qu'il n'y a pas 50 lettres. Elle se termine quand on trouve une valeur 0 dans le tableau, qui, avec cette syntaxe, est automatiquement ajoutée lors de la déclaration. Si on veut réduire la chaîne, il faut faire la chose suivante :

```
1 int main() {  
2     char S[50] = "Hello World !\n";  
3     S[5] = 0;  
4  
5     return 0;  
6 }
```

🌐 [code/code-XV-1-72.cpp]

En effet, en plaçant un 0 à la case numéro 5, on va se retrouver avec une chaîne qui ne contient que "Hello". Cela ne signifie pas que "World" a disparu de la mémoire. Simplement, la convention est que c'est le premier 0 qui compte, donc la chaîne s'arrête maintenant à la case numéro 5.

💬 Remarque

En C Standard, une chaîne de caractères est donc représentée par un pointeur de type **char***. La zone mémoire pointée sur ce pointeur doit être suffisamment grande pour contenir la totalité des caractères utiles + 1, pour avoir le 0 de fin de chaîne.

XV.2 Y'a pas "cout"!

En C standard il n'y a plus **cout**. Voici ce que cela donne sur un exemple comparatif :
fichier **exemple4.cpp**

```

1 #include <iostream>
2 using namespace std;
3
4
5 int main()
6 {
7 // texte simple
8 cout << "Hello\n";
9
10 // afficher une chaîne de
11 // caractères
12 string str = "World !";
13 cout << "Hello " << str << "\n";
14
15 // afficher un résultat de calcul
16 // avec des entiers
17 int a=2,b=3,c;
18 c=a*b;
19 cout << "le résultat de la
20 // multiplication de " << a << "
21 // par " << b << " est " << c << "
22 // "\n";
23
24 // afficher un résultat de calcul
25 // avec des float
26 float A=80.5,B=4.5,C;
27 C = A/B;
28 cout << "le résultat du quotient de
29 // " << A << " par " << B << "
30 // est " << C << "\n";
31
32 return 0;
33 }
```



[code/code-XV-2-73.cpp]

fichier **exemple4.c**

```

1 #include <stdio.h>
2
3
4
5 int main()
6 {
7 // texte simple
8 printf("Hello\n");
9
10 // afficher une chaîne de
11 // caractères
12 char str[] = "World !";
13 printf("Hello %s\n",str);
14
15 // afficher un résultat de calcul
16 // avec des entiers
17 int a=2,b=3,c;
18 c=a*b;
19 printf("le résultat de la
20 // multiplication de %d par %d est
21 // %d\n",a,b,c);
22
23 // afficher un résultat de calcul
24 // avec des float
25 float A=2,B=3,C;
26 C = A/B;
27 printf("le résultat de la
28 // multiplication de %f par %f est
29 // %f\n",A,B,C);
30
31 return 0;
32 }
```



[code/code-XV-2-74.cpp]

On voit que la fonction du C standard pour afficher est la fonction **printf** de la bibliothèque **stdio.h**. De plus, on note que pour afficher un texte, comme pour **cout**, il suffit de donner le texte entre guillemets. Mais on voit aussi que pour les variables ça ne se passe pas comme pour **cout** : Cette fois ci, avec **printf**, on donne une séquence en **%** pour afficher une variable. On trouve autant de séquences en **%** qu'il y a de variables à afficher. On peut repérer dans les exemples donnés ci-dessus : la séquence **%s** ligne 12, les séquences en **%d** ligne 17, et les séquences en **%d** ligne 22.

Dans **printf**, il y a autant de **%** qu'il y a de variables à afficher. Par exemple, ligne 12, on veut afficher la variable **str** et uniquement la variable **str**, comme indiqué après la virgule. Et on a en effet une seule séquence en **%**. On peut faire la même remarque pour les lignes 17 et 22 : on a 3 commandes en **%** sur chaque exemple, et 3 variables (**a,b,c** puis **A,B,C**) à afficher.

Il reste une dernière chose à dire. La commande en % n'est pas la même à chaque fois : ligne 12 c'est %s, ligne 17 c'est %d, et ligne 22 c'est %f. En C standard, la commande diffère selon le type de variable à afficher. C'est ce que l'on résume dans le tableau ci-dessous :

Commande	type de la variable	exemple
%d	int	<pre>int a=3, b=2; printf("%d fois %d vaut %d\n", a, b, a*b); — résultat —</pre> <p>3 fois deux vaut 6</p>
%f	float	<pre>float pi=3.14159; printf("la valeur de pi est : %f", pi); — résultat —</pre> <p>la valeur de pi est 3.14159</p>
%c	char	<pre>char lettre1='A', lettre2=66; printf("les lettres sont %c et %c", lettre1, lettre2); — résultat —</pre> <p>les lettres sont A et B</p>
%s	char*	<pre>char str[50]="Bonjour a tous !"; str[3]=0; printf("le texte est : %s", str); — résultat —</pre> <p>le texte est : Bon</p>

XV.3 Y'a pas non plus "cin"!

Voici sur un petit exemple :

fichier **exemple5.cpp**

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
{
5
6 int age;
7 cout << "Quel est votre age ? ";
8 cin >> age;
9
10 float temperature;
11 cout << "Quelle température fait-il
12      aujourd'hui ? "
12 cin >> temperature
13
14 string nom;
15 cout << "Quel est votre nom ?"
16 cin >> nom;
17
18 cout << "\n Bonjour " << nom << " !
19     Vous avez " << age << " ans " et
20     la température aujourd'hui est
21     "<<temperature<< " degrés.\n";
22
23 return 0;
24 }
```

 [code/code-XV-3-79.cpp]

fichier **exemple5.c**

```

1 #include <stdio.h>
2
3
4 int main()
{
5
6 int age;
7 printf("Quel est votre age ? ");
8 scanf("%d", &age);
9
10 float temperature;
11 printf("Quelle température fait-il
12      aujourd'hui ? ");
12 scanf("%f", &temperature);
13
14 char nom[50];
15 printf("Quel est votre nom ?");
16 scanf("%s", nom);
17
18 printf("\n Bonjour %s! Vous avez %d
19     ans et la température aujourd'
20     hui est %f degrés.\n", nom, age,
21     temperature);
22
23 return 0;
24 }
```

 [code/code-XV-3-80.cpp]

En C Standard, on va donc utiliser **scanf** au lieu de **cin**. Et on voit qu'il y a avec **scanf** le même système qu'avec **scanf** avec les commandes en %. Mais il reste une subtilité.

On le voit très bien lignes 8 et 12, **scanf utilise le passage par adresses**, on le voit par l'utilisation du **&**. En effet, la valeur saisie au clavier est forcément une sortie pour la fonction **scanf**, et on voit sur le code que ça ne passe pas par un mécanisme type **return**. Les choses sont assez simples pour le cas **int** et **float** (lignes 8 et 12). Mais cela semble moins intuitif pour la ligne 16 : il n'y a pas le **&**. Est-ce que **scanf** fait une exception pour les chaînes de caractères ? Absolument pas. L'analyse qu'il faut faire, c'est que la variable **nom** est déjà une adresse, puisque c'est un tableau statique sur cet exemple. Et on sait que dans ce cas, **nom** est du type **char***, c'est donc un pointeur.

Dernière différence : avec **scanf**, on doit prévoir un tableau suffisamment grand pour la saisie : ici, on sait qu'on ne peut pas se permettre une saisie de plus de 50 caractères, sinon on prend des risques, et il n'y a aucun mécanisme de vérification. C'est donc vraiment pas terrible et beaucoup moins bien que l'utilisation des **string** en C/C++.

Important

On voit très bien ici que le passage par adresses est un prérequis du programmeur en C standard : même une opération aussi simple que faire une saisie au clavier nécessite d'utiliser le passage par adresses ! ■

PARTIE IV

Mélanger du Python et du C/C++

Chapitre XVI – Exécuter un code C/C++ à partir de Python

XVI.1 Introduction

Le principe est le suivant :

1. créer une bibliothèque C/C++, c'est à dire un fichier **déjà compilé** qui contient des fonctions dont on va avoir besoin sous Python
2. importer la bibliothèque C/C++ dans un code Python
3. utiliser la bibliothèque C dans le code Python

Remarque

Il existe plusieurs façons pour faire des échanges entre Python et C/C++. La méthode que l'on propose ici n'est pas universelle, elle ne permet pas de "tout" faire, mais elle permet de faire déjà beaucoup de choses et a l'avantage d'être assez simple (comparativement aux autres).

De plus, ce qui est montré dessus ne fonctionne, de façon exacte, que sous Linux et probablement sur Mac, mais peut s'adapter assez facilement pour fonctionner sous Windows.

XVI.2 Le code que l'on a envie d'écrire en Python

Tout débute ici : il faut décider, à partir de Python, de comment on veut utiliser la fonction que l'on va créer en C/C++. Cette fonction doit permettre d'avoir les entrées utiles (c'est à dire U_0 et N , le nombre de termes à calculer), et fournir les valeurs en sortie.

Dans cette méthode, l'utilisation de **return** est limitée à un résultat avec un seul entier, on ne peut rien faire d'autre. Donc les résultats passeront par les paramètres, à la façon d'un passage par adresse, en C/C++. Autrement dit, on va passer à la fonction C/C++ (que l'on créera plus tard) des variables Python qu'elle devra modifier.

Compte tenu de cette remarque, on va utiliser un code Python qui ressemble au code ci-dessous pour faire appel à la fonction C/C++ que l'on créera ensuite :

```
U0 = 27
N = 120
sortie = np.zeros((N,), dtype=np.uint32)
syraccuse_du_c(U0,N,sortie)
```

 [code/code-XVI-2-3.py]

Ca n'est qu'une "maquette", pas un code définitif, mais il sert à situer les choses. En Python, on fournira donc à la fonction **syraccuse_du_c** la valeur de U_0 puis N , qui sont des entrées. Le paramètre suivant est un **np.array** Python de N valeurs de type **uint32** (qui correspond au **unsigned long** ou **unsigned int** du C/C++), que la bibliothèque C/C++ devra remplir avec les valeurs de la suite de Syracuse. Voyons maintenant les choses côté C/C++.

XVI.3 Créer la fonction C/C++

On va partir de l'exemple de la conjecture de Syracuse. On donne un exemple de code C/C++ qui calcule les valeurs :

```

#include <vector>
#include <fstream>
using namespace std;

// calcule N elements de la suite de syracuse, debutant a U0
int syracuse(unsigned int U0, unsigned int N, unsigned int* output)
{
    unsigned int U;
    unsigned int k;

    U = U0;
    k = 0;
    output[k]=U;
    k++;

    while (k < N)
    {
        if (U % 2 == 0)
        {
            U = U / 2;
        }
        else
        {
            U = 3*U+1;
        }

        output[k]=U;
        k = k+1;
    }
    return 0; // toujours renvoyer 0 avec cette methode
}

```

 [code/code-XVI-3-81.cpp]

Commentons :

- On a créé une fonction **syracuse** qui prend en paramètres des entiers pour U_0 et N
- Pour faire sortir les valeurs, comme on l'a montré en Python, on a un 3ieme paramètre de type **unsigned int***, qui est un pointeur qui va pointer sur les valeurs du **np.array** créé dans Python.

XVI.4 Créer la bibliothèque C/C++

Pour que cette fonction soit "comprise" comme une bibliothèque C/C++, il faut ajouter encore des petites choses :

```

1 #include <vector>
2 #include <fstream>
3 using namespace std;
4
5
6 // force l'utilisation de la convention d'appel C
7 extern "C" {
8 int syracuse(unsigned int U0, unsigned int N, unsigned int* output);
9 }
10
11
12 // calcule N elements de la suite de syracuse, debutant a U0

```

```

13 int syracuse(unsigned int U0, unsigned int N, unsigned int* output)
14 {
15
16     unsigned int U;
17     unsigned int k;
18
19     U = U0;
20     k = 0;
21     output[k]=U;
22     k++;
23
24     while (k < N)
25     {
26         if (U % 2 == 0)
27         {
28             U = U / 2;
29         }
30         else
31         {
32             U = 3*U+1;
33         }
34
35         output[k]=U;
36         k = k+1;
37     }
38     return 0; // corresponds to python assumed return type by default
39 }
```

 [code/code-XVI-4-82.cpp]

On a juste ajouté les lignes 6 à 9, qui forcent la "convention d'appel" type "C" plutôt que "C++". Ca n'est pas facile à décrire rapidement. Disons pour faire simple que le protocole qui permet de faire appel à une fonction en C n'est pas identique à celui que l'on utilise en C++ (en apparence sur le code que l'on écrit il n'y a pas de différence, mais le compilateur fait les choses différemment). Et pour une bibliothèque, on doit utiliser le protocole type "C" pour les fonctions que Python va voir.

Maintenant, si on compile de code de la même façon que d'habitude, on n'obtiendra pas une bibliothèque¹. Il faut demander au compilateur de ne pas générer un exécutable mais une bibliothèque, et cela se fait en ajoutant simplement quelques options. Voici :

```
g++ -fPIC -shared syracuse.cpp -o syracuse.so
```

Remarque : Ceci est valable sous Linux et fonctionne sûrement sur Mac. Sous Windows il faudrait adapter un peu.

On obtient alors le fichier **syracuse.so**, de type **.so**, qui est le format des bibliothèques pour Linux (et sûrement compréhensible pour un Mac).

XVI.5 Importer la bibliothèque dans Python

On va maintenant importer la bibliothèque dans Python. La difficulté principale est que C/C++ est type explicitement (c'est à dire qu'on doit déclarer les variables), alors que Python non. On va donc devoir utiliser un certain protocole pour indiquer à Python quels sont les types à utiliser avec nos fonctions C/C++. Voici comment cela s'écrit :

```

1 import ctypes
2 from ctypes import c_uint, POINTER, cast
3 import numpy as np
4
5 # chargement de la bibliotheque
```

1. En fait on n'obtiendra rien parce qu'il manquerait un **main**

```

6 lib = ctypes.cdll.LoadLibrary('./syraccuse.so')
7 # on recupere la fonction "syraccuse" de la bibliotheque
8 syraccuse_du_c = lib.syraccuse #
9 # on declare a Python les parametres de la fonction C
10 # soit 3 parameters : 2 uint and 1 uint* (uint = unsigned int pour ctypes)
11 syraccuse_du_c.argtypes=[c_uint,c_uint,POINTER(c_uint)]

```

 [code/code-XVI-5-4.py]

A partir de là, on a fait le travail qu'il fallait faire : on a importé la bibliothèque, récupéré la fonction C, et décrit à Python les paramètres. A partir de maintenant, on peut utiliser `syraccuse_du_c` comme une fonction Python.

XVI.6 Utiliser la fonction C dans Python

On revient à quelque chose de très proche de ce que l'on avait écrit au tout début de ce chapitre : on va indiquer à la fonction importée les paramètres à prendre, et utiliser les résultats. Cela démarre comme on l'a déjà indiqué :

```

1 import ctypes
2 from ctypes import c_uint,POINTER,cast
3 import numpy as np
4
5 # chargement de la bibliotheque
6 lib = ctypes.cdll.LoadLibrary('./syraccuse.so')
7 # on recupere la fonction "syraccuse" de la bibliotheque
8 syraccuse_du_c = lib.syraccuse #
9 # on declare a Python les parametres de la fonction C
10 # soit 3 parameters : 2 uint and 1 uint* (uint = unsigned int pour ctypes)
11 syraccuse_du_c.argtypes=[c_uint,c_uint,POINTER(c_uint)]
12
13
14 # on prepare l'appel a la fonction : on definit les valeurs des parametres
15 U0 = 27
16 N = 120
17 resultats = np.zeros((N,), dtype=np.uint32)
18 # et voici l'appel lui meme
19 syraccuse_du_c(U0,N,cast(resultats.ctypes.data,POINTER(c_uint)))
20
21
22 # et on affiche le resultat
23 import matplotlib.pyplot as plt
24 plt.plot(resultats,'r.')
25 plt.show()

```

 [code/code-XVI-6-5.py]

Jusqu'à la ligne 12, c'est exactement ce que l'on a déjà écrit pour importer la fonction. Les lignes 15 à 19 donnent des valeurs aux paramètres que l'on va utiliser dans la fonction.

La ligne 20 est plus complexe à comprendre : elle fait appel à la fonction `syraccuse_du_c` en fournissant 3 paramètres. Pour les deux premiers, `U0` et `N`, c'est simple. Mais le 3ième vaut :

`cast(resultats.ctypes.data,POINTER(c_uint))`

Et ça semble très obscur ! En fait, cette ligne consiste d'abord à récupérer l'adresse des valeurs du `np.array` appelé `resultat` : c'est le sens à donner à `resultats.ctypes.data`. Mais cette adresse n'est pas du bon type : elle est par défaut de type C `void*`. Et il faut donc la convertir en `unsigned int*` pour que cela soit accepté par la fonction. Et c'est le sens à donner à `cast` et à `POINTER(c_uint)`. En résumé, tout ce paramètre s'écrirait, en C, `(unsigned int*)(resultats.ctypes.data)`, sauf qu'on est en Python. On voit bien ici l'aspect orienté "bas niveau" du C/C++, qui décrit facilement les adresses et les pointeurs, et la complexité que

l'on a à décrire cela avec Python, qui n'a, la plupart du temps, rien à faire avec le concept d'adresse ou de pointeur.

Au final, on trace dans un `plot` les données.

XVI.7 Une écriture plus élégante

Si on doit faire plusieurs fois des appels à la fonction `syraccuse_du_c`, on va devoir réécrire à chaque fois la conversion du pointeur, et c'est pénible. Alors, à la place, on peut réécrire le code ci-dessus comme ceci :

```

1 import ctypes
2 from ctypes import c_uint, POINTER, cast
3 import numpy as np
4
5 # chargement de la bibliotheque
6 lib = ctypes.cdll.LoadLibrary('./syraccuse.so')
7 # on recupere la fonction "syraccuse" de la bibliotheque
8 syraccuse_du_c = lib.syraccuse #
9 # on declare a Python les parametres de la fonction C
10 # soit 3 parameters : 2 uint and 1 uint* (uint = unsigned int pour ctypes)
11 syraccuse_du_c.argtypes=[c_uint,c_uint,POINTER(c_uint)]
12
13
14 # definition d'une fonction Python qui utilise syraccuse_du_c :
15 def syraccuse(U0,N):
16     r = np.zeros((N,), dtype=np.uint32)
17     syraccuse_du_c(U0,N,cast(r.ctypes.data,POINTER(c_uint)))
18     return r
19
20 # maintenant on peut ecrire facilement les appels a syraccuse_du_c :
21 resultats = syraccuse(27,120)
22
23 # et on affiche le resultat
24 import matplotlib.pyplot as plt
25 plt.plot(resultats,'r.')
26 plt.show()

```



On a juste créé une fonction Python qui gère le protocole d'utilisation de `syraccuse_du_c`

Remarque

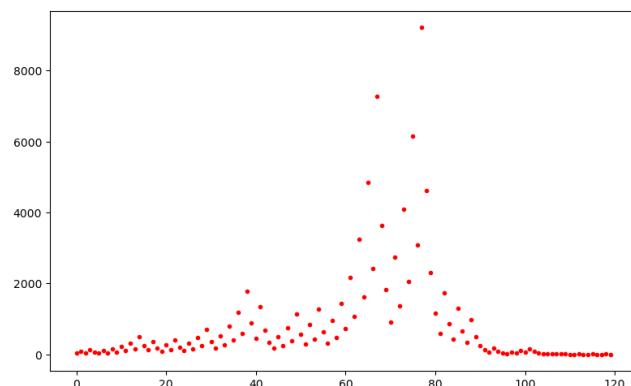
A ce stade si vous êtes un peu perdus, ce qu'il vous faut faire pour faire fonctionner ça :

- Récupérer le code de §XVI.4, le coller dans un fichier nommé par exemple `syraccuse.cpp`, puis le compiler comme bibliothèque avec le terminal, en faisant :

`g++ -fPIC -shared syraccuse.cpp -o syraccuse.so`
- Récupérer le code de §XVI.7, le collé dans un fichier nommé par exemple `main.py`, puis l'exécuter en tapant dans le terminal :

`python3 main.py`

Et vous obtiendrez :



Written with open-source software only



LINUX



the LATEX Project



INKSCAPE



KRITA