



Centre **I**nformatique pour les **L**ettres
et les **S**ciences **H**umaines

C++ : Leçon 1 Principes de base

1 - Du langage machine aux langages de haut niveau.....	2
2 - Programmer	3
3 - Codage de l'information en mémoire	4
La case mémoire vue comme un nombre entier sans signe	6
La case mémoire vue comme un nombre entier relatif	6
Autres façons de voir dans la mémoire une quantité numérique	6
Et pour représenter autre chose que des nombres ?	7
Récréation.....	8
4 - Traiter l'information	9
Accéder à l'information	10
Modifier l'information	11
5 - Bon, c'est gentil tout ça, mais ça fait déjà 11 pages. Qu'est-ce que je dois vraiment en retenir ?	12
6 - J'ai rien compris, est-ce que quelqu'un d'autre pourrait m'expliquer ça un peu plus clairement ?	12
Pour aller moins vite.....	12
Pour aller plus loin	13
7 - Pré-requis de la Leçon 2	13

1 - Du langage machine aux langages de haut niveau

Le fonctionnement d'un ordinateur repose sur l'exécution de programmes. Au sens propre, un programme est une liste d'instructions dont l'exécution produit un résultat utile ou agréable à l'utilisateur de la machine¹.

Pour qu'une liste d'instructions soit exécutable, il faut qu'elle ne se compose que d'instructions faisant partie du "vocabulaire" de base du processeur². Rédiger de telles listes présente deux inconvénients majeurs :

- Le vocabulaire de base d'un processeur ne comporte que des instructions correspondant à des opérations très élémentaires. Exprimer la décomposition d'une tâche humainement significative dans ce "**langage machine**" est très long et très difficile.
- Les différents processeurs disponibles sur le marché ont chacun leur vocabulaire de base. Ce qui constitue un programme pour l'un d'entre eux n'est qu'un charabia inutilisable pour les autres. Etant donné la difficulté d'écriture des programmes et la rapidité de l'évolution technologique des processeurs, la complexité maximale d'un programme pouvant être écrit pour un processeur donné avant que celui-ci ne cesse d'être fabriqué est sévèrement limitée.

Ces deux inconvénients ont, depuis longtemps, conduit à une conséquence très simple : on n'écrit plus que très exceptionnellement des programmes.

Comme, par ailleurs, le besoin de faire sans cesse effectuer de nouvelles tâches par les ordinateurs semble à peu près universellement répandu, il a fallu trouver un artifice rendant possible la production de programmes dans des conditions économiquement acceptables : les programmes ne sont plus écrits (par des humains) mais générés automatiquement (par les ordinateurs eux-mêmes).

Cette stratégie repose sur la mise au point de deux éléments essentiels : un **langage informatique** permettant de décrire facilement la tâche à effectuer, et un dispositif de **traduction automatique** permettant de produire un programme dont l'exécution effectue la tâche décrite. Ce double objectif (facilité de description des tâches / possibilité d'une traduction automatique) donne évidemment naissance à des contraintes contradictoires : le langage de programmation idéal devrait être aussi proche que possible du langage naturel, ce qui le rendrait facile à apprendre et permettrait de décrire aisément les traitements souhaités³, tout en évitant de trop s'éloigner du langage de la machine, parce que cela augmente la difficulté de la traduction automatique (ou la rend même irréalisable) et diminue généralement l'efficacité des programmes générés. La nécessité de trouver un compromis entre ces deux pôles, jointe à la diversité des tâches envisagées, a donné naissance à une pléiade de langages qui ont connu des fortunes diverses : Fortran, Cobol, Basic, Algol, Pascal, Lisp ou Ada ne sont que quelques-uns des plus connus.

Selon les estimations disponibles, plus de 1000 langages de programmation ont été mis au point (c'est à dire qu'on ne les a pas simplement conçus, mais qu'on a effectivement produit les outils de traduction automatique correspondants) au cours de la seconde moitié du XX^e siècle.

A l'heure actuelle, la technique de traduction automatique la plus souvent employée est la **compilation** : le programmeur écrit un **texte source** dans le langage de programmation de son choix, puis fait appel à un programme capable de traduire ce texte en une suite d'instructions exécutables par le processeur visé (ce qu'on appelle le **code objet**). Pour réellement donner naissance à un programme exécutable, ce code objet doit encore être adapté à son contexte d'utilisation (et, en particulier, aux us et coutumes imposés par le système d'exploitation en vigueur) et, éventuellement, combiné avec d'autres fragments de code objet préexistants, auxquels il peut sous-traiter certains aspects de sa tâche. Cette opération, connue sous le nom de **édition de liens**, est exécutée par un programme qu'on appelle un **linker**. L'ensemble du processus de création d'un programme comporte donc trois phases, et peut être résumé par le tableau suivant :

¹ Par "utilisateur de la machine", il faut entendre "celui qui est à l'origine de l'exécution du programme". Dans le cas d'un virus, par exemple, le résultat est rarement utile ou agréable à celui qui est assis en face de l'écran...

² Le processeur est le composant (ou le groupe de composants) électronique(s) qui, au sein d'un ordinateur, assure l'exécution des programmes.

³ Ce point mériterait sans doute d'être débattu, mais cela nous entraînerait un peu loin de notre sujet.

Nom de l'opération	Produit	Responsable	Objectif
Rédaction	Texte source	Programmeur (être humain)	Décrire la procédure à suivre pour parvenir au résultat souhaité
Compilation	Code objet	Compilateur (programme)	Traduire le texte source en une suite d'instructions exécutables par le processeur visé.
Edition de liens	Programme exécutable	Linker (programme)	Greffer sur le code objet tout ce qui lui manque pour devenir un programme exécutable dans le contexte visé.

Il est évident que, même si de nouveaux compilateurs et linkers peuvent souvent être écrits en faisant appel à des versions antérieures de ces programmes, il faut bien que, à un moment donné, quelqu'un se donne le mal d'écrire directement un programme (en langage machine). En fait, la création d'un compilateur n'est pas entreprise a posteriori, une fois le processeur fabriqué. C'est au contraire une opération étroitement liée à la conception même d'un nouveau processeur, car l'avenir commercial de celui-ci dépend de façon cruciale de la possibilité de réaliser des compilateurs efficaces.

Ces définitions étant posées, il nous faut bien reconnaître que le langage courant emploie presque systématiquement le mot *programme* pour désigner un texte source. Bien que ce terme soit clairement impropre (puisque aucun processeur n'est capable d'exécuter directement un texte écrit dans un langage de programmation), ce raccourci verbal est justifié par le fait qu'il est possible de traduire, de façon entièrement automatique, le texte source en un programme. La nuance n'a donc d'importance réelle que pour les programmeurs eux-mêmes (notamment parce qu'ils doivent mettre en œuvre le processus de traduction), et, sauf s'ils sont vraiment *extrêmement* débutants, il n'y a aucun risque de confusion dans leur esprit ! On considère habituellement que, dès la deuxième page d'un cours de programmation, le lecteur a atteint un niveau de compétence tel qu'il n'y a plus de risque d'erreur, et on s'autorise donc à lui parler normalement : le mot programme peut alors, selon le contexte, désigner soit du texte source, soit du code exécutable, et c'est seulement lorsque l'ambiguïté est réelle que la terminologie exacte est mobilisée.

2 - Programmer

Même si le recours à un langage de haut niveau nous isole de façon assez efficace du détail des instructions composant le "vocabulaire" d'un processeur, il n'est pas réellement possible d'écrire des programmes sans avoir conscience de certaines réalités basement matérielles.

La première de ces réalités incontournables est que la seule chose sur laquelle le processeur peut agir directement est la **mémoire** de l'ordinateur. Fondamentalement, toutes les instructions exécutables ne font que modifier le contenu de certaines parties de la mémoire, en fonction du contenu d'autres parties de cette même mémoire. L'écriture d'un programme traitant un problème quelconque exige donc de surmonter trois difficultés :

- Il faut représenter l'état initial de la situation sous forme de données contenues dans la mémoire.
- Il faut énumérer les opérations qui transforment les données représentant l'état initial en données représentant l'état souhaité (ce qu'on appelle les résultats).
- Il faut permettre à l'utilisateur du programme d'interpréter les résultats obtenus comme une représentation de la solution qu'il attend.

Dans un exemple très simple, ces difficultés sont surmontées si facilement qu'on risque d'avoir du mal à les percevoir. Ainsi, si le problème à résoudre est le calcul de la surface d'un rectangle à partir de sa largeur et de sa longueur, représenter l'état initial se résume à stocker ces deux valeurs en mémoire, et le traitement qui conduit au résultat désiré est une simple multiplication dont il suffit, pour satisfaire l'utilisateur, d'afficher le résultat.

Notons tout de même, au passage, que cette simplicité provient en grande partie du fait que la plupart des langages de programmation permettent de manipuler directement les quantités numériques (que ce soit pour les stocker en mémoire, les multiplier entre elles ou les représenter sur un écran en respectant les conventions habituelles). Les trois difficultés sont donc invisibles (car prises en charge par le langage lui-même) plutôt qu'absentes.

Si le problème est plus complexe, et en particulier si l'utilisateur ne l'envisage pas comme un problème numérique, les exigences de l'entreprise deviennent plus apparentes. Un être humain

qui joue aux échecs contre un ordinateur, par exemple, n'a aucune idée du type de représentation utilisé par la machine. Il serait notamment assez surpris d'apprendre que l'échiquier imaginaire sur lequel l'ordinateur effectue ses calculs comporte plus de 64 cases⁴. Lorsque le programmeur est conduit, pour représenter la situation initiale et décrire les traitements à lui appliquer, à s'écarter à ce point de la représentation sur laquelle s'appuie l'utilisateur, la troisième difficulté devient plus manifeste : il est clair qu'il faut retraduire le résultat du calcul (qui risque fort, dans notre exemple, de n'être pour le programmeur qu'une matrice de nombres) dans une forme plus intelligible (l'énoncé d'un coup à jouer, voir même une représentation graphique de l'échiquier).

Un autre point dont il est essentiel d'avoir conscience est l'interdépendance étroite qui lie la représentation de la situation sous forme de données et la logique des opérations susceptibles de conduire à la solution recherchée. L'explicitation de cette logique (ce qu'on appelle un **algorithme** du problème) dépend évidemment des instructions disponibles dans le langage utilisé⁵, mais, même dans un langage donné, un même problème peut donner lieu à plusieurs analyses, conduisant chacune à des représentations de l'état initial extrêmement différentes et à des traitements ayant fort peu de points communs.

Il est difficile de proposer un exemple de cette situation sans entrer dans des détails techniques un peu prématurés. A titre anecdotique, je me souviens avoir consacré un week-end pluvieux à l'écriture d'un programme résolvant le problème du solitaire (un plateau en forme de croix où sont placées des billes : elles peuvent se "prendre" les unes les autres un peu comme les pions du jeu de dames, et il faut faire en sorte qu'il n'en reste qu'une). Ce programme reposait sur une représentation du plateau et des billes restantes, et le texte source occupait une douzaine de pages. J'ai écrit, par la suite, un second programme, basé celui-ci sur l'inventaire des *mouvements possibles*, sans représentation explicite du plateau et des billes. Le texte source n'occupait plus qu'une page et demi, mais son écriture m'avait également coûté un week-end. Anecdote à l'intérieur de l'anecdote : la découverte d'une solution prenait presque deux fois plus longtemps à ce second programme qu'au premier.

Les débutants ont souvent spontanément tendance à ne voir dans un langage de programmation qu'une collection d'instructions qu'il s'agirait de mémoriser pour ensuite jouer à les combiner de façon astucieuse. Cette illusion est souvent renforcée par la pratique d'exercices dont les énoncés contiennent déjà, de façon plus ou moins explicite, le choix d'une certaine façon de représenter le problème sous forme de données. Ce choix détermine largement l'approche algorithmique qui va pouvoir être mise en œuvre, et l'ensemble réduit très efficacement l'activité de programmation à une simple démonstration de dextérité syntaxique. Il s'agit là d'une catastrophe didactique majeure, d'où l'étudiant ne peut ressortir qu'avec une illusion de compétence qui ne manquera pas de s'écrouler dès sa première confrontation avec un problème réel (c'est à dire un problème ne se présentant pas sous la forme d'un exercice dont l'énoncé fournit plus de 90% d'une des solutions possibles).

Un langage de programmation est, en fait, une façon de comprendre les problèmes et d'essayer de les résoudre. Son utilisation commence, bien avant l'écriture de la moindre ligne de texte source, lorsque se pose la question : "Ce problème pourrait-il être résolu par un ordinateur ?" La réflexion porte alors tout autant sur la description du problème en termes informatiques que sur la découverte des voies conduisant à sa solution, car les deux choses sont indissociables. On comprend donc bien qu'il ne saurait être question d'apprendre un langage de programmation sans se pencher très sérieusement sur les moyens qu'il emploie pour représenter l'information.

3 - Codage de l'information en mémoire

Du point de vue qui nous intéresse (c'est à dire celui d'un programmeur utilisant le langage C++), la mémoire d'un ordinateur peut être décrite comme une collection de dispositifs électroniques dont chacun se trouve, à tout moment, dans l'un de deux états possibles⁶. Ces dispositifs ne sont jamais manipulés individuellement : ils sont regroupés huit par huit pour

⁴ Dans le cas des jeux se déroulant sur un plateau, il est en effet parfois plus simple pour le programmeur d'envisager un plateau plus grand que les règles normales ne le prévoient, et d'ajouter des règles rendant impossible l'accès à cette surface supplémentaire.

⁵ Si les Djinns habitaient nos ordinateurs au lieu de préférer les lampes à huile, le seul langage de programmation dont nous aurions besoin ne comporterait qu'une seule instruction, qui serait aussi un algorithme de tous les problèmes envisageables : "Résous !" Mais il faudrait quand même, d'une façon ou d'une autre, exposer les données du problème.

⁶ Vous pouvez, par exemple, imaginer ces dispositifs comme autant d'interrupteurs, chacun d'entre eux pouvant être en position "marche" ou "arrêt".

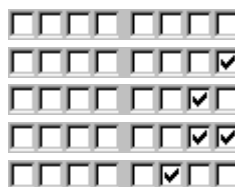
former ce qu'on peut appeler une **case mémoire**. L'usage du mot "mémoire" est justifié par le fait que ces cases présentent trois caractéristiques fondamentales : elles sont **stables**, **modifiables** et **consultables**.

- La notion de stabilité renvoie simplement au fait que l'état d'une case mémoire ne change pas spontanément, mais reste immuable tant que le programme en cours d'exécution ne le modifie pas.
- Le fait que l'exécution de certaines instructions d'un programme peut avoir pour effet de changer l'état d'une case mémoire constitue précisément la deuxième caractéristique de ce dispositif.
- L'ensemble du système resterait inutilisable s'il n'offrait pas également aux programmes la possibilité de consulter l'état d'une case mémoire pour, par exemple, entreprendre des traitements différents en fonction de cet état.

Par définition, un dispositif binaire (c'est à dire pouvant adopter l'un de deux états stables) contient une quantité d'information égale à un **bit**. Lorsqu'un système comporte plusieurs de ces dispositifs, la quantité d'information qui peut y être stockée est d'autant de bits, et elle peut être appréhendée en dénombrant les différents états que ce système peut adopter.

Pour pouvoir représenter les choses sur le papier, on peut décider de figurer les dispositifs binaires en question sous la forme de petits carrés dont les deux états sont "non coché" et "coché". Un système limité à 1 bit n'a que 2 états possibles, qui sont alors représentés par ☐ et ☒, alors qu'un système disposant de 2 bits a 4 états possibles : ☐☐, ☐☒, ☒☐ et ☒☒. D'une façon générale, l'ajout à un système d'un dispositif binaire supplémentaire multiplie par deux le nombre d'états possibles. En effet, tous les états antérieurement possibles existent ensuite en deux versions : l'une avec le nouveau dispositif dans l'un des états, l'autre avec le nouveau dispositif dans l'autre état. Un ensemble de n bits peut donc adopter 2^n états différents.

Les cases mémoires étant, nous l'avons vu, des groupes de 8 dispositifs binaires, elles peuvent contenir une quantité d'information égale à 8 bits (ce qu'on appelle un **octet**) et le nombre d'états différents possibles pour chacune d'entre elles est 2^8 , soit 256. Si l'on choisit de "remplir" les cases en commençant par la droite et en progressant systématiquement de façon à n'oublier aucune combinaison, on peut facilement retrouver ces 256 états différents :



et ainsi de suite (je n'ai ni le courage ni la place de les représenter tous !), jusqu'à



En dépit de sa simplicité extrême, la mémoire de l'ordinateur pose un sérieux problème de compréhension aux êtres humains normaux qui y sont confrontés pour la première fois. Ce problème tient au fait que cette mémoire contient de l'information pure, c'est à dire dénuée de toute signification. Pour que les choses prennent un sens (et donc un intérêt), il va nous falloir *attribuer* un sens à cette information. Cette attribution est parfaitement arbitraire (conventionnelle) et l'ordinateur lui-même y reste totalement étranger (rappel : l'ordinateur en question n'est jamais qu'un tas de ferraille dans lequel circule du courant électrique, il n'est donc pas question que l'information ait un *sens* pour lui).

Avant d'en venir au processus par lequel un programmeur attribue du sens à l'information stockée dans la mémoire de l'ordinateur, examinons quelques-unes des interprétations les plus simples et les plus classiques. Nous allons commencer par envisager différentes façons d'attribuer une signification numérique au contenu de la mémoire. Bien qu'elle ne soit pas plus légitime qu'une autre, cette interprétation est tellement courante qu'un grand nombre de gens oublient qu'il ne s'agit que d'une interprétation, et finissent par s'imaginer que les ordinateurs manipulent des nombres.

La case mémoire vue comme un nombre entier sans signe⁷

Il s'agit certainement de l'interprétation qui exige le moins d'effort d'imagination, puisqu'elle repose simplement sur la représentation des deux états binaires à l'aide des chiffres 0 et 1. Dans ces conditions, une case mémoire se trouvant dans l'état



sera représentée par

1 1 0 0 0 0 0 1

ce qui ressemble étrangement à un nombre. Comme les seuls chiffres qui peuvent y figurer sont 0 et 1, il est très tentant de poursuivre la métaphore en imaginant que le nombre en question n'est pas en base 10, mais en base 2, et c'est ce qu'on fait habituellement. La transcription décimale de ce nombre étant 193, on s'autorise à dire dans ce contexte que la case mémoire "contient la valeur 193".

Si vous en éprouvez le besoin, vous pouvez vous reporter à l'[Annexe 0, "Combien avez-vous de doigts ?"](#), qui propose un petit rappel sur différentes façons de représenter les nombres entiers et, notamment, sur la façon de passer d'une représentation binaire à la représentation décimale habituelle.

Lorsqu'une case mémoire est interprétée de cette façon, ses 256 états possibles représentent donc les nombres entiers compris entre 0 et 255 (nombres dont les représentations binaires sont respectivement 0000 0000 et 1111 1111).

La case mémoire vue comme un nombre entier relatif

Etant donné que la case mémoire n'a que 256 états possibles, la représentation d'un certain nombre de valeurs négatives ne peut se faire qu'en sacrifiant la possibilité de représenter certaines des valeurs positives qui étaient autorisées dans l'hypothèse précédente. On peut imaginer toutes sortes de conventions, mais celle qui est habituellement adoptée⁸ est de représenter l'intervalle [-128, 127] de la façon suivante :

- Zéro et les nombres positifs sont représentés comme s'ils étaient sans signe. Ils utilisent donc les 128 patterns allant de 0000 0000 à 0111 1111
- Pour représenter un nombre négatif, on lui ajoute 256 et on représente le résultat comme si la case mémoire contenait un entier sans signe. Cette addition a simplement pour effet de "projeter" les valeurs de l'intervalle [-128, -1] sur les valeurs positives "interdites", c'est à dire l'intervalle [128, 255].

valeur à représenter	valeur + 256	représentation binaire
-128	128	1000 0000
-127	129	1000 0001
-126	130	1000 0010
...
-3	253	1111 1101
-2	254	1111 1110
-1	255	1111 1111

Notons dès à présent que la case mémoire se trouvant dans l'état



peut toujours être représentée par

1 1 0 0 0 0 0 1

mais qu'il n'est maintenant plus question de prétendre qu'elle "contient la valeur 193". Dans l'hypothèse où il s'agit de la représentation d'un entier relatif, nous sommes conduit à dire qu'elle "contient la valeur -63" (c'est -63 qui est représenté ainsi, car $-63 + 256 = 193$).

Autres façons de voir dans la mémoire une quantité numérique

Le fait qu'une case mémoire ne puisse adopter que 256 états différents limite sévèrement, nous venons de le constater, les valeurs numériques qu'on peut convenir d'y voir. Que se passe-t-il lorsqu'on a besoin de manipuler des nombres beaucoup plus grands ?

⁷ C'est à dire, de fait, positif

⁸ Ce choix présente en effet l'avantage déterminant de permettre d'additionner de la même façon les entiers, qu'ils soient positifs ou négatifs.

La réponse est très simple : il suffit d'utiliser plusieurs cases de mémoire. Si on mobilise une seconde case pour représenter un nombre entier positif, la représentation binaire de celui-ci s'étend sur 16 bits, et ses valeurs possibles vont de 0 à 65 535. Si cet intervalle ne suffit toujours pas, on peut bien entendu utiliser davantage de mémoire. Avec quatre cases mémoire (et donc 32 bits), la plage de variation accessible à un entier non signé⁹ s'étend de 0 à 4 294 967 295, ce qui devrait normalement vous suffire pour compter vos économies (même si vous les exprimez en centimes de Franc).

Il est également possible d'établir des conventions permettant de représenter des quantités non entières sous forme binaire, et donc de les stocker en mémoire. Le système le plus usité est la représentation dite "flottante", qui consiste à ramener la valeur du nombre à représenter entre 1 et 2 (par multiplication par une puissance de 2) et à stocker d'une part ce résultat, et d'autre part la puissance de 2 qui a été utilisée dans la multiplication. Les seules choses vraiment importantes à retenir à propos de ce système sont d'une part qu'il est pratiquement inutilisable pour des êtres humains¹⁰ et, d'autre part, qu'il offre une précision dépendant non seulement du nombre de cases mémoire utilisées, mais aussi de l'ordre de grandeur des nombres. En d'autres termes, si la représentation de

0.0000000001 et celle de

0.0000000002 sont bien différentes,


il n'est pas certain que celles de 123456789123456789.0000000001 et

123456789123456789.0000000002 le soient aussi.

Lorsqu'on utilise plusieurs cases de mémoire pour représenter une seule quantité, l'idée même que l'une de ces cases pourrait avoir un "contenu" (c'est à dire un sens) pose problème. Chacune des cases n'a qu'un "état", et ce n'est que collectivement qu'elles peuvent être interprétées, c'est à dire se voir attribuer un sens.

Et pour représenter autre chose que des nombres ?

Un certain nombre de choses a priori non numériques se laissent aisément représenter par des états de la mémoire. C'est notamment le cas pour tous les systèmes basés sur un ensemble de symboles distincts (les lettres de l'alphabet ou les notes de musique, par exemple). Il suffit d'attribuer à chacun de ces symboles un état différent de la mémoire, qui servira à le représenter. Chacun des états de la mémoire ainsi utilisés possède donc une interprétation unique, si l'on sait qu'il représente un de ces symboles.

Dans le cas des caractères alphabétiques et des autres symboles habituellement utilisés pour écrire (chiffres, signes de ponctuation, etc.), une des conventions les plus utilisées est celle connue sous le nom de "code ASCII". Il s'agit simplement d'une mise en correspondance arbitraire de ces symboles avec des patterns de bits. Le caractère 'A', par exemple, se trouve ainsi associé au pattern . Comme ce pattern peut également être interprété comme la représentation du nombre 65, on dit habituellement que le code ASCII de 'A' est 65, ce qui est clairement un abus de langage. Il vaudrait mieux dire que le caractère 'A' et le nombre 65 sont représentés selon des conventions différentes, qui conduisent l'une et l'autre au même pattern de bits. Le rapport entre la lettre 'A' et la quantité 65 est assez semblable à celui qui lie la quantité 4 à l'appareil qui sert à faire cuire les gâteaux : dans un certain système de codage, cet appareil est représenté par "four", et une pure coïncidence fait que, dans un autre système de codage (la langue anglaise) la quantité 4 se trouve, elle aussi, représentée par "four". Bien entendu, les langues "naturelles" telles que l'anglais ou le français disposent d'un nombre considérable de séquences de lettres susceptibles de représenter des mots, et la plupart d'entre elles restent en fait inutilisées, ce qui rend ce genre de coïncidences assez rares. Tous les états possibles de la mémoire sont, en revanche, susceptibles d'être lus comme des nombres entiers et la coïncidence, loin d'être rare, devient ici systématique. Comme nous le verrons au cours de la Leçon 2, ce phénomène permet au langage C++ de réduire les caractères typographiques à une simple façon un peu particulière d'afficher les valeurs entières.

Il n'y a guère plus de difficultés dans le cas des phénomènes mesurables : s'il semble, par exemple, difficilement envisageable de représenter une *température*, il n'y a en revanche aucun problème pour en représenter une *mesure* (puisque celle-ci s'exprime sous forme numérique), et ceci suffit pour traiter un grand nombre de questions faisant intervenir la température.

⁹ Le principe de représentation des nombres négatifs décrit plus haut sur un seul octet peut, sans difficultés particulières, être généralisé aux représentations faisant intervenir plusieurs cases de mémoire.

¹⁰ Les opérations nécessaires pour passer de la représentation décimale à une représentation flottante (et inversement) sont assez lourdes, et les calculs usuels (addition, multiplication...) ne sont pas particulièrement faciles à effectuer sur des quantités représentées en notation flottante.

Un cas voisin, quoique légèrement plus complexe, est celui des phénomènes qui peuvent être décrits par *plusieurs* mesures. Une couleur, par exemple, peut être représentée de façon satisfaisante sous la forme des quantités relatives de rouge, vert et bleu qui la composent.

Cette façon de représenter les couleurs a une pertinence particulière en micro-informatique car elle correspond directement à la technologie de l'écran cathodique. D'une certaine façon, le rouge, le vert et le bleu sont des réalités concrètes accessibles à l'ordinateur (dans le contexte de l'affichage à l'écran).

Certains phénomènes posent un autre type de problème : une image, par exemple, peut être représentée sous la forme d'un (vraiment) très grand nombre de points. La répétition de la mesure donne ici la clé de la représentation, et cette technique n'est envisageable que parce que la quantité d'information stockable en mémoire est considérable. Il est, par ailleurs, possible que la mesure qui se trouve répétée soit elle-même composée : si chacun des points est décrit en termes de rouge, de vert et de bleu, l'image représentée est en couleurs.

Tout état de la mémoire peut être interprété comme représentant une quantité numérique¹¹. Il est donc clair que, si un état d'un phénomène est représenté dans une section de la mémoire d'un ordinateur, cet état du phénomène se trouve, par-là même, associé à une représentation numérique, c'est à dire mesuré. Cette mesure peut être multidimensionnelle (comme dans l'exemple de la couleur) ou, à l'autre extrémité de l'échelle du raffinement, n'être que purement nominale (comme dans le cas des lettres de l'alphabet), mais il s'agit bien, dans tous les cas, d'une mesure. Nous sommes donc implacablement conduits à conclure qu'un phénomène n'est représentable en mémoire que pour autant qu'il est mesurable.

Même si ce type de représentation ne permet de capturer que fragmentairement la réalité, ses limitations sont moins drastiques qu'on pourrait le croire a priori. Il y a quelques années, la conclusion à laquelle nous venons d'aboutir sonnait, dans les oreilles du public, comme une sentence reléguant les ordinateurs dans les laboratoires scientifiques et autres services de la comptabilité. Maintenant que le téléphone, la hi-fi, la vidéo et les bibliothèques sont "numériques", l'illusion inverse devient plus fréquente, selon laquelle tout serait parfaitement mesurable (ou, pour employer l'euphémisme à la mode, "numérisable").

La représentation d'états du monde un peu complexes fait le plus souvent appel à la description d'objets dont une seule mesure ne peut rendre compte d'une façon suffisante à la résolution du problème envisagé. Le programmeur est donc conduit à mettre en place des conventions assez élaborées, qui lient certains états de la mémoire à certaines propriétés mesurables des objets manipulés. L'exemple de la couleur, représentée par une mesure sur chacune de ses composantes rouge, verte et bleue, illustre de façon simple cette façon de procéder. Faire en sorte que toutes les opérations effectuées sur les cases mémoire décrivant les objets restent compatibles avec les conventions de représentation adoptées est une des tâches du programmeur (et ce n'est pas la plus facile). Dans l'exemple des couleurs, il semble évident qu'une opération telle que le mélange de deux couleurs pour en produire une troisième ne pourra être définie qu'en respectant le sens attribué à chacun des trois nombres représentant chaque couleur¹². Un traitement qui multiplierait, par exemple, la mesure du rouge dans l'une des couleurs par la mesure du vert dans l'autre aurait peu de chances de correspondre à une opération interprétable dans le cadre du système adopté pour représenter les couleurs. Une des caractéristiques majeures du langage C++ est l'ensemble de moyens qu'il propose au programmeur pour l'aider à assurer cette compatibilité entre conventions de représentation et traitements, mais l'étude de ces moyens nécessite un examen préalable de la façon dont un processeur opère des traitements.

Récréation

Avant que nous ne nous engageons dans cet examen, il serait peut être utile d'essayer de concrétiser un peu les notions que nous venons d'introduire. Le programme [Visuram](#) va nous permettre de jouer directement avec la mémoire, et de voir comment ses différents états peuvent être interprétés. L'affichage proposé par Visuram présente trois zones :

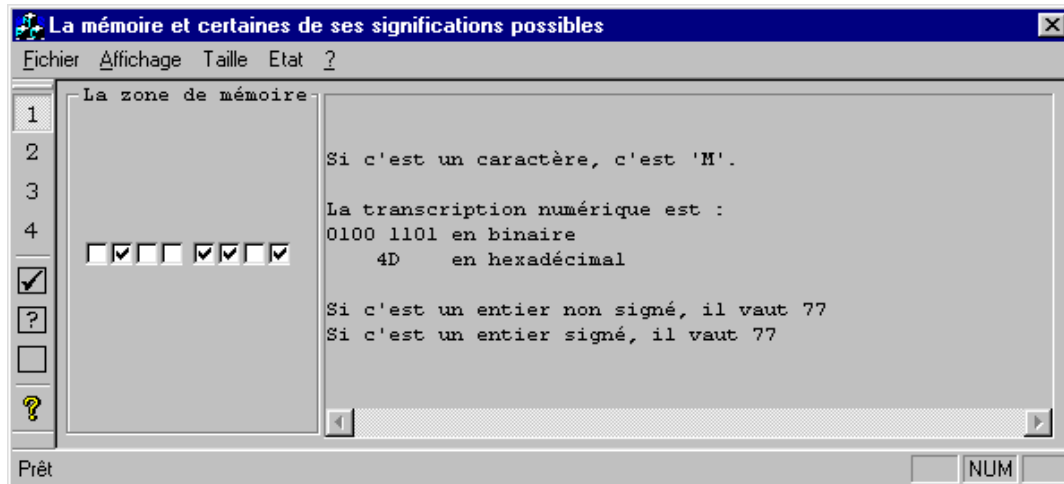
- une barre d'outils (initialement présentée verticalement, à gauche de la fenêtre)
- une représentation d'une zone de mémoire sous la forme de petits carrés
- une zone de texte où s'affichent certaines interprétations possibles de l'état de la zone de mémoire considérée.

¹¹ Nous avons même vu qu'il existait **plusieurs** façons de faire une telle interprétation.

¹² Il est en fait assez délicat de mélanger de façon réaliste deux couleurs ainsi représentées, mais peu importe.

La zone de mémoire est représentée à l'aide du symbolisme que nous avons utilisé dans les paragraphes précédents : les deux états des dispositifs binaires sont figurés par ☐ et ☒. Pour changer l'état d'un de ces dispositifs, il suffit de cliquer sur le carré qui le représente.

La barre d'outils comporte quatre boutons qui permettent de fixer la taille de la zone de mémoire considérée (1, 2, 3 ou 4 cases). Elle comporte également un bouton qui permet de mettre toute la zone de mémoire dans l'état ☒, un bouton (☐) qui met la zone de mémoire dans un état fixé au hasard et un bouton qui permet de mettre toute la zone de mémoire dans l'état ☐.



La fenêtre principale de [Visuram](#)

Visuram est également un exemple du genre de programmes que vous serez, d'ici quelques semaines, en mesure de créer (même si on peut supposer que vos propres créations ne s'intéresseront guère à la représentation en mémoire des quantités numériques !).

Quelques suggestions d'exercices :

Commencez avec une zone de mémoire réduite à une seule case :

- Essayez de retrouver les représentations des quatre premiers entiers positifs.
- Essayez de faire de même pour les quatre premiers entiers négatifs.
- Trouvez la représentation du nombre - 128.
- Trouvez la représentation du caractère 'C' (vous connaissez la représentation de 'A', et vous pouvez tirer parti du fait que les lettres sont représentées dans l'ordre alphabétique).
- Vérifiez que vous comprenez comment sont obtenues les transcriptions binaires et hexadécimales des différents états.

Passez ensuite à une zone de mémoire comportant deux cases :

- Essayez de retrouver les représentations des quatre premiers entiers positifs.
- Trouvez la représentation du nombre 256.
- Trouvez la représentation du nombre - 32 768.
- Vérifiez que vous comprenez comment sont obtenues les transcriptions binaires et hexadécimales des différents états (en particulier, comprenez vous ce que veut dire "little endian" ?).

Avec une zone comportant trois cases :

- Trouvez un rouge, un vert et un bleu vifs.
- Essayez de trouver un jaune vif.

Avec une zone comportant quatre cases :

- Jouez un peu avec l'état de la mémoire en observant l'effet de vos manipulations sur l'interprétation décimale, mais
- N'essayez surtout pas de retrouver la représentation du nombre 3.14

4 - Traiter l'information

Si la mémoire contient des représentations de l'état actuel de certains objets, un des premiers problèmes qui se posent est de pouvoir, lorsqu'on écrit un programme, spécifier sur lequel de ces objets les opérations décrites doivent être opérées.

Accéder à l'information

Fondamentalement, le processeur ne dispose que d'un seul moyen pour distinguer les cases mémoires les unes des autres : elles sont numérotées et chacune d'entre elles peut donc être désignée sans ambiguïté par le numéro qu'elle porte. On a l'habitude d'appeler ces numéros des **adresses**, et il est commode de se représenter les cases mémoires comme autant d'immeubles alignés le long d'une unique et très longue avenue. Dans ces conditions, une adresse (au sens que donne un chauffeur de taxi à ce mot) se réduit effectivement à un numéro qui permet de retrouver l'immeuble désigné, tout comme l'adresse informatique permet de retrouver la case mémoire désignée.

Cette façon de procéder conduit le processeur à accepter d'exécuter des ordres du genre :

```
Place la case mémoire d'adresse 9692 dans l'état représenté en binaire par 0000 0001
```

Le problème posé par cette façon de s'exprimer est qu'elle reste totalement étrangère à la signification attribuée au contenu de la case mémoire 9692. Si le programmeur a décidé d'utiliser cette case mémoire pour y stocker le nombre de pions blancs restant sur un damier, par exemple, l'instruction précédente signifie en fait qu'il n'y a plus qu'un pion blanc. Ecrire les programmes de cette façon entraîne de graves difficultés.

La première de ces difficultés est d'ordre purement mnémonique : si notre programme manipule de très nombreuses quantités, il va être très pénible de devoir se souvenir des adresses correspondant à chacune d'entre elles, et le risque d'erreur atteint très rapidement la certitude. Un premier confort qui va nous être offert par l'usage d'un langage de programmation est la possibilité d'attribuer aux zones de mémoire qui nous intéressent des **noms** qui vont nous aider à nous souvenir de la signification de ce que nous y avons stocké. Dans notre exemple, cela nous permettrait de désigner la case 9692 en utilisant des sobriquets plus ou moins imagés, tels que "nbBlancs" ou "effectifDesForcesDuMal".

La seconde difficulté est un peu plus technique, car elle concerne la façon dont est codée la valeur qui est stockée en mémoire. Supposons que le nombre de pions blancs puisse, par moment, excéder 255. Le programmeur sera alors immanquablement conduit à ne pas stocker ce nombre dans une, mais dans plusieurs cases mémoire. S'il choisit d'en utiliser deux, y stocker la valeur 1 nécessite de donner au processeur deux ordres successifs :

```
Place la case mémoire d'adresse 9692 dans l'état représenté en binaire par 0000 0001
Place la case mémoire d'adresse 9693 dans l'état représenté en binaire par 0000 0000
```

Le problème devient encore plus complexe si un système de représentation moins élémentaire (comme par exemple celui permettant de représenter des nombres décimaux) est utilisé pour représenter la valeur. L'usage d'un langage de programmation permet en fait de décharger le programmeur qui utilise une zone de mémoire (celle représentant le nombre de pions blancs, dans notre exemple) de la corvée consistant à se rappeler combien de cases mémoires sont effectivement impliquées et comment l'information y est codée. Le procédé qui offre ce confort est l'attribution d'un **type** à la zone mémoire utilisée. Une fois qu'une zone mémoire s'est vue attribuer un type, le langage prend en charge les détails techniques relatifs au codage de l'information. Ainsi, dans notre exemple, le simple fait que le programmeur décide que la zone mémoire nommée `effectifDesForcesDuMal` est du type "entier sur deux octets" permet au langage de savoir qu'un ordre comme :

```
mettre à 1 l'effectifDesForcesDuMal
```

se traduit, pour le processeur, par les deux instructions successives décrites ci-dessus. Si, en revanche, le programmeur décide que la zone mémoire nommée `effectifDesForcesDuMal` est du type "entier sur un seul octet", le même ordre

```
mettre à 1 l'effectifDesForcesDuMal
```

sera traduit par le compilateur au moyen d'une seule instruction¹³. De même, si la zone mémoire nommée `effectifDesForcesDuMal` est déclarée comme étant du type "nombre décimal", c'est le compilateur qui se chargera de retrouver quel est l'état de la mémoire qui signifie 1 dans ce contexte.

¹³ Cette explication n'est pas à prendre au pied de la lettre, car les processeurs actuels n'exigent pas deux instructions pour modifier deux cases de mémoire d'adresses consécutives. Du point de vue de notre exposé, la prise en compte de ce genre de détails technologiques ne ferait toutefois qu'obscurcir le raisonnement, sans rien y apporter d'intéressant.

C'est donc grâce à la notion de type que nous pouvons nous offrir le luxe d'ignorer le détail de la représentation en mémoire des nombres décimaux. Remerciez-la au passage, car elle vous a déjà épargné plusieurs pages d'explications encore plus ennuyeuses.

Donner un nom qui évoque la signification de l'information stockée et choisir un type convenant aux valeurs susceptibles d'être utilisées ne devrait pas s'avérer trop difficile. Mais comment le programmeur décide-t-il d'utiliser telles cases mémoire plutôt que telles autres ? Dans la plupart des cas, il ne s'en occupe en fait pas du tout, et c'est le compilateur lui-même qui se charge de choisir la zone de mémoire où sera stockée l'information. La création d'une variable se résume donc au choix d'un type et d'un nom, et nous pouvons nous contenter de retenir que :

Lorsqu'une zone de mémoire se voit attribuer un **type** et un **nom**, cette zone de mémoire devient une **variable**.

Le fait qu'une zone de mémoire soit considérée comme ayant type et un nom (et soit donc une variable) ne change évidemment rien au fait qu'il s'agit toujours d'une zone de mémoire. En particulier, les cases mémoires concernées ont toujours chacune une adresse, qui permet au processeur de les retrouver (les noms et les types n'existent que dans le langage de programmation que vous utilisez, le processeur en ignore tout). Lorsqu'une variable correspond à plusieurs cases de mémoire, l'adresse de la première de ces cases est aussi considérée comme étant **l'adresse de la variable**. Ceci permet aux variables de n'avoir qu'une seule adresse, même lorsqu'elles s'étendent sur plusieurs cases de mémoire.

Modifier l'information

Nous venons de voir comment la notion de type permet au langage de prendre en charge les choix de conventions de représentation qui sont effectués par le programmeur. Cette prise en charge ne concerne pas simplement le codage et le décodage de l'information lorsqu'on veut "lire" ou remplacer le contenu d'une variable, elle s'étend également au contrôle des traitements qui sont effectués sur les variables.

Un premier contrôle concerne la légitimité des traitements envisagés. Prenons un exemple simple : l'action qui consiste à ajouter 1.

Du point de vue d'un processeur, il y a fort à parier que cette action correspond à l'un des mots de son vocabulaire de base¹⁴ et, à partir de là, aucun obstacle ne s'oppose à son exécution sur n'importe laquelle des cases mémoires accessibles. Il semble clair que le *sens* de cette action, pour sa part, dépend de ce que la case mémoire est sensée représenter et que, donc, le processeur ne peut pas en tenir compte. L'absence d'erreur dans le programme¹⁵ n'est donc garantie que par l'infailibilité du programmeur (ce qui revient à dire que la présence d'erreurs, elle, est absolument garantie).

Du point de vue du langage de programmation, en revanche, la notion de type permet de mettre en place certains garde-fous. Ainsi, si une variable est de type "couleur", il est clair que tenter de lui "ajouter 1" n'a aucun sens, et le compilateur peut signaler l'erreur bien avant que qu'il essaie d'exécuter le programme.

Même lorsqu'une action peut être considérée comme légitime, il reste que la façon dont elle doit modifier l'état de la mémoire est étroitement liée aux conventions établies pour donner un sens à la mémoire en question. Si l'on considère les opérations qui doivent être effectuées sur l'état de la mémoire, ajouter 1 à un entier est très différent d'ajouter 1 à un nombre décimal. L'existence des types permet au programmeur de ne pas se soucier de ce genre de problèmes. Il peut se contenter d'exprimer l'idée qu'il faut "ajouter 1 au contenu de la variable", et le compilateur se chargera de faire ce qu'il faut¹⁶, étant donné le type de la variable impliquée.

¹⁴ En toute rigueur, dire que l'addition fait partie du répertoire d'actions du processeur est déjà un abus. Ce qui fait partie du répertoire d'actions du processeur, c'est une certaine façon de modifier l'état d'une case mémoire. Le fait que cette modification produit un état qui peut, si la case mémoire représente un nombre entier, être interprété comme le résultat d'une addition n'est certainement pas un hasard. Il n'en reste pas moins qu'il ne s'agit là que d'une *interprétation* : rien ne garantit que la case mémoire représentait effectivement un entier et, si ce n'est pas le cas, de quelle addition pourrait-il bien s'agir ?

¹⁵ Il ne s'agit pas ici d'erreurs d'exécution qui seraient dues à un mauvais fonctionnement du processeur, mais d'erreurs logiques présentes dans le programme lui-même, comme, par exemple, ajouter 1 à une case mémoire dont la signification ne serait pas numérique.

¹⁶ C'est à dire qu'il se chargera d'insérer, dans le code exécutable qu'il va générer, les instructions qui produiront effectivement l'effet spécifié par le programmeur.

La notion de type n'est pas le seul confort apporté par l'utilisation d'un langage de haut niveau. L'écriture d'un programme correct est grandement simplifiée par la possibilité de segmenter le texte source en petites parties pouvant être mises au point et testées relativement indépendamment les unes des autres. En C++, le plus petit fragment de code ayant une réelle autonomie est la **fonction**. Tout comme les variables, les fonctions portent des noms, et le choix de ces noms incombe généralement au programmeur.

Qu'il s'agisse d'une variable ou d'une fonction, le choix d'un nom doit être fait avec soin, car la précision avec laquelle il évoque la nature de l'information représentée (dans le cas d'une variable) ou du traitement effectué (dans le cas d'une fonction) exerce une grande influence sur la difficulté de mise au point du programme.

5 - Bon, c'est gentil tout ça, mais ça fait déjà 11 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) Les ordinateurs exécutent des programmes.
- 2) Les programmes modifient l'état de la mémoire.
- 3) Le processeur distingue les cases de mémoire les unes des autres grâce à leurs adresses.
- 4) Les programmeurs s'imaginent que l'état de la mémoire a une signification.
- 5) Il y a un tas de façons cohérentes et respectables de projeter une signification sur un état de la mémoire.
- 6) Lorsqu'ils sont doués, les programmeurs écrivent des programmes qui ne modifient l'état de la mémoire que d'une façon qui respecte la signification qu'ils ont attribuée à celle-ci.
- 7) Le langage C++ aide les programmeurs à donner l'impression qu'ils sont doués en leur proposant d'organiser la mémoire en variables ayant chacune un nom et un type.
- 8) En C++, les programmes sont divisés en unités appelées des fonctions.

6 - J'ai rien compris, est-ce que quelqu'un d'autre pourrait m'expliquer ça un peu plus clairement ?

La littérature recommandable sur ce sujet est malheureusement assez rare. La plupart des ouvrages d'initiation à la programmation s'en tiennent, au mieux, à une présentation très allusive des idées exposées dans cette première Leçon. Un certain nombre d'entre eux contiennent même des erreurs grossières, de nature à perturber durablement la compréhension d'un lecteur innocent (il est plus facile de se faire des idées fausses que de s'en débarrasser ensuite). Il est vrai que, dans une logique purement commerciale, il vaut sans doute mieux caresser le lecteur dans le sens du poil en le laissant tout de suite jouer avec les boutons plutôt que d'essayer de lui fournir des bases conceptuelles dont il ne peut évidemment pas mesurer immédiatement l'importance.

Si l'on considère l'exemple de notre manuel, la situation est claire : Horton n'évoque tout simplement pas ces questions, et semble, à certains moments, considérer que leur maîtrise est inutile (cf. l'introduction, toute en douceur, de la notion de variable), pour ensuite considérer qu'elle est certainement acquise (aucun commentaire sur l'usage systématique de l'hexadécimal dans le debugger, par exemple). Cette valse-hésitation est d'ailleurs payée au prix fort au moment de l'introduction des pointeurs, qui est, comme par hasard, l'un des plus douloureux points faibles de la présentation du langage C++ proposée dans cet ouvrage.

Pour aller moins vite

Steve Heller : [Who's Afraid of C++ ?](#), Academic Press, 1996 (ISBN 0-12-339097-4)

Plus de 150 pages de discussion sur la vraie nature des ordinateurs et de la programmation avant d'en venir au langage C++ proprement dit ! Le tout expliqué très lentement, en prenant en compte les protestations de Susan, qui a un peu de mal à suivre. Deux problèmes, toutefois : c'est en anglais, et le style n'est, somme toute, pas très différent du mien (si vos difficultés proviennent d'un manque d'atomes crochus avec l'auteur, le simple fait que la présentation faite par Steve aille moins vite que celle proposée ici pourrait bien ne pas suffire à vous réconcilier avec le sujet).

Pour aller plus loin

Paolo Zanella & Yves Ligier : Architecture et technologie des ordinateurs. Dunod, 1998 (ISBN 2-10-003801-X)

Des circuits logiques aux systèmes d'exploitation, en 450 pages étonnamment abordables étant donné leur niveau de technicité. Vous ne comprendrez peut être pas tout dès la première lecture, mais si cette technologie vous intéresse, il y a peu de risques que vous perdiez votre temps avec cet ouvrage.

Charles Petzold : Code. Microsoft Press, 1999 (ISBN 0-7356-0505-X)

Par rapport au Zanella & Ligier, le Petzold présente l'inconvénient évident de ne pas (encore ?) avoir été traduit en français. Il couvre quasiment le même sujet, mais dans un style franchement "grand public" plutôt que "universitaire", sans jamais sombrer pour autant l'approximation vulgarisatrice outrancière. Apprenez l'anglais et faites-le-vous offrir.

7 - Pré-requis de la Leçon 2

La Leçon 1 doit avoir été étudiée sérieusement, même si la parfaite maîtrise de toutes les notions qui y sont présentées n'est pas indispensable.

Étant donné la nature exceptionnellement abstraite du contenu de la Leçon 1, sa maîtrise parfaite ne peut qu'être un objectif à long terme. C'est par leur utilisation répétée au cours des Leçons suivantes que les notions présentées ici prendront vraiment toute leur signification.

Il n'est pas nécessaire que le TD 1 ("Premier dialogue") ait été fait.

Attention toutefois : le TD1 comporte des exercices obligatoires qui peuvent vous occuper assez longtemps. Prendre trop de retard maintenant risque de vous rendre les TD suivants difficiles.