Introduction au C++ pour les ingénieurs

Matthieu DURUT

1 Note préliminaire

Ces notes de cours ne constituent qu'une version préliminaire d'un cours toujours en écriture. Elles sont amenées à évoluer et à être modifiées, plus particulièrement en fonction des retours des élèves et relecteurs. Tout commentaire, agréable ou non, est le bienvenu.

Remerciements

Je souhaiterais remercier tous ceux qui m'ont aidé, de près ou de loin, lors de l'écriture de ce cours. David Sibaï tout d'abord, qui avait entamé les premiers chapitres de ce polycopié quand il était assistant informatique à l'ENSAE, et sans qui je ne me serais probablement pas lancé dans cette rédaction. Xavier Dupré, Pierre Jacob, Jérémie Jabukowicz, et David encore, pour toutes nos discussions au Gentleman ou ailleurs. Jérémie Bonnefoy, pour ses relectures attentives. Les étudiants de ce cours depuis 2009, pour leur intérêt, leurs questions, remarques, ou reproches. Ma femme Elodie, pour avoir supporté la rédaction en parallèle de ce document et de mon manuscrit de thèse.

Table des matières

— *It looked like a good thing to do.*

Dennis Ritchie, à propos de l'invention du langage C

— *The more general aim was to design a language in which I could write programs that were both efficient and elegant. Many languages force you to choose between those two alternatives.*

Bjarne Stroustrup, à propos de l'invention du langage C++

2

Notes sur le C++

Le langage C est apparu au cours de l'année 1972 dans les Laboratoires Bell. Il fut développé en même temps qu'UNIX par Dennis Ritchie et Ken Thompson. L'objectif était d'écrire un langage de programmation permettant le développement rapide de systèmes d'exploitation (OS). Dennis Ritchie fit évoluer un langage existant –le langage B– dans une nouvelle version suffisamment différente pour qu'elle soit appelée C. Le succès du C fut rapide, en raison de sa portabilité [1] et des performances du code écrit.

Dix ans plus tard, Bjarne Stroustrup, intéressé par la programmation orientée objet qui commençait à se répandre chez les universitaires, développa le langage C++, successeur du C, alors qu'il travaillait dans le laboratoire de recherche Bell d'AT&T. Il s'agissait en l'occurrence d'améliorer le langage C en lui ajoutant une composante objet (le C++ s'appelait alors C with classes). La proximité du C++ avec le C, couplée aux facilités qu'il offrait firent qu'il fut rapidement adopté.

Le C++ a été largement adopté par la communauté des développeurs et peut être considéré comme le langage de référence des années 80 et 90. Dans les années 2000, des langages plus faciles [2] mais aux performances comparables (Java et C# par exemple) sont

1. C'est-à-dire la possibilité d'utiliser un programme tapé sur un ordinateur donné et un système d'exploitation donné sur un autre ordinateur et un autre système d'exploitation.
2. tout est question de point de vue évidemment

préférés pour le développement de nouveaux projets. Cependant, le C++ reste le langage de référence : de très nombreux projets débutés dans les années 80/90 ont été codés en C++ et n'ont pas été portés vers des langages plus récents. En finance par exemple, il reste le langage le plus utilisé aujourd'hui (2012) car de très nombreuses bibliothèques toujours en production ont été développées en C++, même si les nouveaux projets ont tendance à regarder du côté de langages plus récents, objets ou fonctionnels. Plus proche de la machine que les langages plus récents, le C++ permet au prix d'une complexité plus forte un meilleur contrôle sur la machine et donc sur les performances [3].

Le C++ est un langage multi-paradigme, pouvant à la fois être procédural [4] et orienté objet, permettant de la programmation générique très poussée (jusqu'au template méta-programming en fait), mais aussi un début de programmation fonctionelle. Le C++ est un langage particulièrement difficile, suffisamment proche du langage machine pour que nous ayons besoin parfois de connaissances hardwares pour optimiser notre code, mais aussi suffisamment abstrait pour exprimer une réelle complexité dans les concepts manipulés.

Enfin, comme nous le verrons dans ce polycopié, le C++ est un langage bâtard. Pour qu'un langage soit cohérent, il est nécessaire que le principal de ses concepts clefs soit défini dès sa création, qu'une vision à long terme du langage soit explicitée dès le départ. Le C++ est né à une période où de nombreux concepts informatiques émergeaient. Construit "par dessus" le C, il reprend l'intégralité de la spec du C (pour convaincre les développeurs C de passer sous C++) et a ajouté au fur et à mesure de l'avancement des concepts, des couches supplémentaires au langage. L'enrichissement de ces concepts a mené à une sorte d'hydre, un langage puissant mais peu structuré qui s'est développé dans de nombreuses voies différentes. C'est donc à la fois un langage de référence et d'expérimentation.

A qui s'adresse le C++ ? Pourquoi enseigner et utiliser le C++ aujourd'hui ?

Le C++ est un langage très performant en vitesse d'execution (surtout lorsqu'il n'abuse pas des concepts introduits entre le C et le

3. sous condition d'une excellente maîtrise du langage

4. Plus par souci de compatibilité avec le C qu'autre chose d'ailleurs, je vous conseille vivement de ne pas coder en procédural...

C++), particulièrement adapté à la création de bibliothèques scientifiques ou d'applications temps réels comme dans le cadre de radars, de contrôleur de tableaux de bord dans les avions, de trading très haute fréquence, etc.

Cette haute performance a un prix : le C++ est un langage complexe, laborieux, peu adapté pour de nombreuses choses naturelles dans des langages plus récents comme le développement web par exemple. En dehors d'un contexte avec des contraintes énormes sur la performance, très peu de projets informatiques ambitieux ont donc de raisons valables de se lancer aujourd'hui dans ce langage.

L'intérêt pédagogique du C++ est soumis à débat : d'un côté, il est suffisamment difficile pour forcer l'étudiant à assimiler de nombreux concepts qu'il serait possible de contourner dans d'autres langages. De l'autre, il est actuellement souvent un mauvais choix de technologie pour monter un projet logiciel. En raison des contraintes industrielles actuelles, les jeunes diplômés de l'ENSAE se trouvent cependant amenés à travailler sur du code en C++, ou sur des langages pour lesquels la transition est assez aisée (notamment le Java et le C#). C'est pour cette raison que l'enseignement à l'ENSAE reste pour quelques années encore en C++. J'invite cependant les étudiants désireux de monter un ambitieux projet logiciel à préférer un des langages plus récents précédemment cités.

Ce document est la version écrite du cours d'introduction au C++ dispensé à l'ENSAE. Dans le cas de l'informatique, il est difficile de proposer le même contenu en cours magistral et dans un polycopié. La syntaxe du C++ ou la compilation peuvent être rapidement expliquées à l'oral par l'exemple, alors que la même explication dans ce document prend nécessairement des volumes plus considérables. Ceci a pour conséquence que les détails et les contraintes techniques ont mécaniquement dans ce document une place relative plus importante qu'en cours. C'est malheureusement au prix d'une place relative plus faible pour les concepts de programmation objet qui me semblent bien plus fondamentaux que la syntaxe du langage par exemple.

J'ai choisi d'écrire ce document en dépit de l'existence de centaines de documents similaires, disponibles en livres imprimés ou sur le web. A mes yeux, trop de ces documents n'étaient pas adaptés à l'enseignement du C++ pour des ingénieurs. En effet, il existe effectivement des "bibles", comme les ouvrages de Stourstrup [23] ou

[10], très complètes et techniques, mais qui s'adressent à des lecteurs déja familiers avec la programmation, ou alors des livres de "vulgarisation" trop peu ambitieux pour des ingénieurs. Avec ce document, j'espère fournir un cours qui soit adapté à la fois à vos attentes, mais aussi à votre vitesse de compréhension.

Dans la mesure du possible, j'ai consenti à de multiples ellipses dans les premiers chapitres, prenant le risque d'être imprécis, afin d'aller au plus vite au coeur du langage. Ces ellipses volontaires sont aussi un moyen de dépoussiérer l'apprentissage du C++ en choisissant de ne pas expliquer les concepts du C présents dans le C++ mais qui n'ont plus beaucoup de sens aujourd'hui ; certains sont discutables (je ne traiterai pas des structures), d'autres sont plus consensuels.

J'ai pris le parti d'organiser ce cours en deux grandes parties qui filent la métaphore des langues vivantes : je présente dans une première grande partie la grammaire et l'orthographe du C++. La deuxième partie, de loin la plus importante et sur laquelle vous serez jugés, sera celle du style. Il s'agit dans cette deuxième partie de comprendre comment architecturer votre code, de comprendre pourquoi certains designs sont bien supérieurs à d'autres.

Enfin, un langage de programmation ne s'apprend que d'une seule manière : par la *pratique*. Répétons-le : sans écrire de programme, il n'est pas possible de maîtriser un langage. Pour ce faire, des Travaux Dirigés, avec un corrigé détaillé vous sont fournis sur le wiki de l'école (Pamplemousse).

— A novice was trying to fix a broken Lisp machine by turning the power off and on. Knight, seeing what the student was doing spoke sternly : "You cannot fix a machine just by power-cycling it with no understanding of what is going wrong." Knight turned the machine off and on. The machine worked.

In [14], from "Al Koans", a collection of jokes popular at MIT in the 1980s

3

Quelques notions hardware

Ce chapitre peut être omis en première lecture.

Ce chapitre a pour vocation de fournir une très brève introduction aux différents composants d'un ordinateur commercial. Au fur et à mesure des développements de l'informatique, le développement logiciel a été suffisamment abstrait du hardware sur lequel les programmes étaient exécutés pour qu'il soit possible de développer une application sans connaissances particulières sur les machines exécutant le code[1]. Cependant, les applications intensives en "calcul" impliquent une utilisation efficace du hardware qui procède directement des connaissances du développeur ; les abstractions développées se révélant insuffisantes pour gérer de manière performante le hardware dans les cas critiques.

Depuis 30 ans, les progrès technologiques en terme de hardware ont été tels qu'il faut désormais des années pour acquérir une connaissance satisfaisante sur le fonctionnement d'un simple ordinateur commercial. C'est bien évidemment très au-delà de l'ambition de ce cours, mais certains points seront abordés dans le cours de troisième année. Nous renvoyons le lecteur curieux à une introduction en la matière très technique mais accessible aux plus motivés : [8]. Dans ce chapitre, nous présentons brièvement les différents composants d'une machine classique afin de donner quelques idées élémentaires.

1. C'est malheureusement de moins en moins vrai, que celà soit à cause des caches, des CPUs multi-coeurs, des OS embarqués, etc.

On peut définir la naissance des premiers ordinateurs (plutôt calculatrices géantes) dans les années 40, avec en tête Von Neumann. Pour les besoins en calculs du projet Manhattan à Los Alamos, Von Neumann élabore avec Mauchly et Eckert en s'inspirant des travaux de Turing[2] un premier calculateur, dont le design est depuis appelé (plutôt injustement) architecture de Von Neumann. C'est cette architecture qui reste présente dans la quasi-totalité des architectures actuelles[3]. Dans ces premiers ordinateurs, la machine avait très principalement vocation au calcul. Chacun des composants était fabriqué spécifiquement pour cette machine, par le même constructeur.

Ce n'est plus aujourd'hui le cas : les différents constructeurs de CPU, de RAM ou de cartes graphiques sont des sociétés distinctes. Pour donner quelques exemples, les constructeurs CPU comptent des sociétés comme Intel, AMD, etc. ; les constructeurs RAM sont représentés par exemple par Samsung, Toshiba, Rambus, etc. ; les constructeurs de carte graphique sont principalement Nvidia et ATI. Le résultat de cette spécialisation des constructeurs dans certains composants est qu'à la différence des premiers ordinateurs, certains des composants d'un ordinateur vont être très rapide par rapport aux autres, et ce sera les composants les plus lents qui ralentiront souvent la machine. Dans les paragraphes suivants, nous présentons les composants principaux d'une machine.

3.1 Le CPU

Le CPU (Central Processing Unit) est le composant principal de calcul. Les CPU qui nous intéressent sont construits sur un seul circuit imprimé (depuis les années 1970), on les appelle micro-processeurs. Un CPU moderne est constitué de millions de transistors, qui permettent de réaliser de très nombreuses opérations mathématiques par seconde.

Depuis une dizaine d'années, les processeurs sont progressivement devenus multi-coeurs, c'est à dire qu'ils contiennent chacun plusieurs entités de calcul appelées coeurs. L'organisation et la répartition des calculs sur ces différents coeurs est laissée à la charge des développeurs et du système d'exploitation (OS, pour Operating System).

2. Alan Turing est avec Von Neumann l'un des pères de l'informatique. Il s'est suicidé en croquant une pomme empoisonnée au cyanure, probablement en référence à Blanche-Neige, et le logo d'Apple est un hommage à Turing.

3. même si l'architecture interne des micro-processeurs depuis 10 ans s'en écarte continuellement

Sur les CPUs modernes[4], les différents composants internes de chaque coeur sont synchronisés sur une sorte d'horloge. A chaque modification de l'état interne du coeur, chacun des composants peut se retrouver dans un état électrique différent, et il importe que chacun de ces composants internes soit stabilisé avant de "regarder" à nouveau le coeur. L'horloge fournit un mécanisme de synchronisation en fournissant des "ticks" réguliers. A chaque tick, chacun des composants a eu le temps de se stabiliser (fin du régime transitoire) et le coeur se trouve donc dans un état cohérent. On voit donc que la performance d'un coeur va dépendre de la durée du régime transitoire entre deux états, mesurée par la durée entre deux ticks de l'horloge. Cette performance est donc mesurée en une fréquence, qui décrit le nombre de cycles d'horloge par seconde que peut réaliser le coeur. Actuellement, les processeurs modernes ont des coeurs qui tournent autour de 3 GHz, c'est à dire que chaque coeur réalise un peu plus de 3 milliards de cycles d'horloge par seconde.

Les composants internes principaux d'un CPU sont :

1. *l'unité arithmétique et logique (UAL, ou ALU en anglais)*, qui prend en charge les calculs arithmétiques et les tests logiques (comme les branchements IF).
2. *les registres*, qui sont des mémoires de très petite taille (quelques octets) et en très petit nombre. Ils sont cependant très rapides, et l'UAL peut les manipuler comme elle l'entend à chaque cycle d'horloge.
3. *L'horloge*, qui fournit un mécanisme de synchronisation interne des différents composants.
4. *L'unité de contrôle*, qui utilise l'impulsion fournie par l'horloge pour synchroniser les différents composants.
5. *L'unité d'entrée-sortie*, qui gère les communications avec les périphériques, extérieurs au CPU (RAM, carte graphique, écran, souris, clavier, disque dur, imprimante, ports USB, etc.).

Sur chaque coeur, nous pouvons faire une opération (multiplication ou addition) par cycle d'horloge[5], c'est à dire jusqu'à près de 3 milliards d'opérations sur des nombres réels par seconde (3 GFlops).

4. hormi partiellement pour certaines consoles de jeux
5. Sur les processeurs hyper-threadés, on peut en fait faire une multiplication ET une addition par cycle

3.2 La mémoire

La mémoire d'un ordinateur est répartie sur de multiples composants. Tous les composants d'un ordinateur stockent des données, au moins celles nécessaires à leur fonctionnement instantané. Certains composants (mémoire RAM, mémoire Cache, disque-dur) sont cependant entièrement dédiés à stocker les données et à les rendre disponibles pour les autres éléments. Ces composants de stockage se distinguent suivant trois critères : la persistence ou non des données lorsque la machine est éteinte/rallumée, la vitesse d'accès aux données stockées, et la taille de ces mémoires.

3.2.1 Disque dur (Hard disk drive : HDD)

Le disque dur est un système de stockage massif de données mais au débit relativement faible. Un disque dur commercial peut actuellement stocker 1 Téraoctet (To), c'est à dire mille Gigaoctets (Go). Le disque dur fournit un système de stockage dans lequel les données sont persistées après l'extinction de la machine. C'est sur le disque dur que sont stockées toutes vos données personnelles, les données de vos programmes, les données du système d'explotation, etc.

Le principal problème actuel des disques durs est la vitesse avec laquelle les données peuvent être lues et écrites. En effet, les disques-durs actuels [6] reposent sur un système de stockage magnétique sur des disques empilés les uns sur les autres. Tout comme pour les vieux lecteurs de disque 33 et 45 tours, des têtes de lecture se déplacent autour du disque pour aller lire les données [7]. En raison de problèmes de frottement d'air, les disques durs actuels ne peuvent faire beaucoup mieux que 7200 tours/minute. Cette restriction physique limite de deux manières la performance de lecture/écriture du disque-dur : tout d'abord, le débit d'un disque dur est très faible par rapport aux autres composants de stockage de mémoire ; ensuite, la latence entre la requête d'une donnée et l'obtention de cette donnée est élevée, du fait de la nécessité pour la tête de lecture de physiquement se déplacer pour accéder à l'emplacement de la donnée.

En conditions réelles, on dépasse très rarement les 100 Mo/sec pour les disques durs internes et les 40 Mo/sec pour les disques durs externes (c'est à dire extérieurs à l'ordinateur et juste reliés par un

6. La techno est en train d'être remplacée par des SSD mais devrait être encore dominante pour quelques années.

7. c'est souvent un des composant les plus fragiles : lorsque vous faites tomber votre laptop, les têtes de lecture viennent souvent perforer les disques, détruisant ainsi le disque dur.

USB ou un FireWire). Pour la latence c'est encore pire, transférer un octet depuis le disque-dur jusque le CPU met souvent entre 3 et 12 ms, soit plusieurs dizaines de millions de cycles d'horloge !

3.2.2 La mémoire RAM

La mémoire RAM (Random Access Memory) désigne un système de stockage dans lequel l'accès à une donnée est indépendant de l'endroit où la donnée est stockée (contrairement au disque dur où la position de la donnée par rapport aux têtes de lecture modifie sensiblement le temps de lecture/écriture). En première approximation, la mémoire RAM d'un ordinateur est effectivement en temps constant [8].

La mémoire RAM d'un ordinateur n'est pas persistante, c'est à dire qu'elle perd ses données lorsque l'ordinateur est éteint (pensez à tous ces documents que vous avez perdus parce que vous aviez oublié d'enregistrer...). Plusieurs technologies permettent de créer de la mémoire RAM, les plus connues étant la DRAM (dynamic RAM) et la SRAM (static RAM). La SRAM est beaucoup plus performante que la DRAM, mais elle coûte beaucoup plus cher à fabriquer et est donc principalement réservée à la mémoire Cache détaillée dans le paragraphe suivant.

La mémoire RAM DDR utilisée dans la plupart de nos ordinateurs permet de stocker moins de données qu'un disque-dur : environ 4Go. Cependant, la mémoire RAM est nettement plus rapide (environ 7Go /sec), et elle peut être accédée par le microprocesseur en quelques centaines de cycles d'horloge.

3.2.3 La mémoire Cache

Parfois aussi appelée antémémoire, c'est la roll's de la mémoire d'un ordinateur : très coûteuse et très performante, elle permet de rendre disponibles des données en quelques cycles d'horloge seulement. Généralement, la mémoire Cache est séparée en plusieurs niveaux (L1, L2, L3) de plus en plus volumineux mais de moins en moins rapides. Typiquement, le cache L1 contient quelques kilo-octets et peut être accédé en 2 ou 3 cycles d'horloge. Le cache L3,

8. Pour être tout à fait rigoureux, ceci n'est pas tout à fait exact dans le cas de la mémoire DRAM qui est communément utilisée comme mémoire RAM dans les ordinateurs récents. Cependant, on peut tout à fait omettre cette remarque en première approximation.

le moins rapide, contient jusque quelques Mo et peut être accédé en une cinquantaine de cycles d'horloge.

3.2.4 Memory swapping

L'OS est en charge de la gestion de la mémoire, c'est lui qui alloue la mémoire nécessaire à chaque programme. Par défaut, la mémoire nécessaire à l'exécution d'un programme est stockée dans la mémoire RAM et dans les caches. Lorsque cette mémoire vient à manquer (par exemple lors d'un monte carlo mal codé sur un très grand échantillon), diverses solutions peuvent être envisagées suivant l'architecture matérielle. Dans de nombreux systèmes, le dépassement de cette limite entraîne une erreur. Dans d'autre cas, l'OS va échanger (swap) la mémoire RAM manquante avec une partie du disque dur pour simuler la mémoire manquante. Ceci n'est en aucun cas une solution satisfaisante, étant donné que le débit en lecture/écriture d'un disque dur par rapport à de la mémoire DRAM est inférieur à un facteur 1/10. Devoir recourir à du memory swapping est donc souvent signe de performances dégradées d'un facteur 10 ou 100. Ceci se ressent lorsque vous essayez d'exécuter un programme sur des données trop volumineuses : votre programme se met à ralentir très sensiblement, vous pouvez entendre le bruit du disque dur qui tourne à plein régime, et votre programme meurt souvent dans d'horribles souffrances.

3.2.5 Prefetching

Les sections précédentes ont démontré les hiérarchies de mémoire, de la plus rapide à la plus lente : Cache L1, Cache L2, Cache L3, RAM. Comment un programme utilise-t-il ces différents espaces de stockage ? Toutes les données nécessaires à l'exécution d'un programme se trouvent dans la mémoire RAM. La mémoire Cache va servir de copie de la mémoire RAM et devenir une mémoire-tampon ; afin de limiter le coût d'accès à la mémoire RAM, la mémoire Cache va copier une partie des données nécessaires à l'exécution du programme. Lorsque la donnée est répliquée en cache, le CPU requêtera la mémoire Cache plutôt que la mémoire RAM, afin d'obtenir la donnée plus rapidement. Lorsque le CPU requête une donnée et que celle-ci se trouve dans le Cache, on parle de Cache Hit ; dans le cas contraire, on parle de Cache Miss.

Le jeu revient donc à choisir astucieusement quelles sont les données de la RAM qui doivent être copiées en Cache, et dans quel

Cache (L1, L2, L3) afin de maximiser le pourcentage des Cache Hit et minimiser ainsi celui des Cache Miss. Ce jeu, appelé Préfetching, n'est heureusement pas la tâche du développeur mais de l'OS et du hardware. En réalité, l'OS connait la liste des prochaines instructions qui seront exécutées[9] et peut donc anticiper quelles données seront nécessaires dans un futur proche, et s'il est nécessaire de ramener cette donnée à un endroit plus proche du CPU (RAM => L3, RAM=> L2, RAM => L1, L3=> L2, L3 => L1 ou L2=> L1).

Dans un programme idéal et bien codé (comme une multiplication matricielle par bloc ou un K-Means), une écrasante majorité des données nécessaires se trouvent dans le Cache L1 ou L2 au moment où elles doivent être lues ou écrites, c'est à dire que le CPU fait surtout des Cache Hit et celà entraine très peu de ralentissement du CPU. Dans d'autres cas, lorsque le préfetcheur échoue à "deviner" quelles seront les données qui seront accédées, le CPU connait beaucoup de Cache Miss, ce qui nuit sensiblement aux performances du programme : le CPU va passer son temps à attendre des données plutôt qu'à les traiter : on parle alors de "starvation" du CPU.

Dans la vie de tous les jours, un développeur ne peut pas faire grand chose pour éviter les Cache Miss car il ne maîtrise pas directement les stratégies de préfetching de la machine. Cependant, une conscience de ces problématiques peut parfois permettre de gagner un facteur 10 ou 100 sur la rapidité d'un programme en gardant une notion de localité des données (nous renvoyons le lecteur intéressé à [8]).

Sauf dans des cas extrêmes, il n'est pas nécessaire de se soucier des stratégies de préfetching, comme il n'est pas nécessaire d'avoir conscience de ces différentes mémoires. En effet, il ne faut se soucier de la performance que lorsque vous avez déterminé que vous aviez un problème de performance. S'il ne fallait citer qu'un homme, ce serait Donald Knuth : ***"Early optimization is the root of all Evil."***. Cette citation sera le moto de ce cours.

9. sauf lorsqu'il y a des instructions conditionnelles comme IF, auquel cas le préfetcheur va faire du branching...

3.3 Le GPU

La carte graphique, aussi connue sous le terme GPU (Graphic Processor Unit), est une sorte de processeur dédié. C'est en fait un composant matériel qui héberge en son sein plusieurs centaines de micro-CPU élémentaires. Ces micro-CPU possèdent beaucoup moins d'instructions que les CPU classiques et ont une cadence plus faible. Cependant, leur nombre fait des GPU des unités de calcul extrêmement puissantes dans certains cas.

A l'origine, les GPU étaient spécialisés dans le rendu graphique, notamment pour les jeux vidéos et les logiciels de dessin/retouche graphique. Les constructeurs de cartes graphiques, NVidia en tête, ont fourni depuis quelques années des librairies permettant d'utiliser les GPU pour d'autres utilisations (c.f. par exemple CUDA). Les GPU sont des processeurs extrêmement adaptés lorsqu'ils sont utilisés pour appliquer de manière indépendante une fonction à de multiples valeurs. Dans le cadre des jeux vidéos, à chaque pixel de l'écran est en effet appliqué une même transformation de l'espace. Depuis, d'autres utilisations ont été développées, regroupées sous le terme de GPGPU (General Purpose on Graphic Processor Unit) ; le pricing par Monte-Carlo est un excellent exemple d'utilisation de GPU pour réaliser des calculs plus rapidement que sur CPU.

Première partie

Éléments de syntaxe

4

Compilateurs, interpréteurs

Note préliminaire : le terme compilation désigne originellement une étape du processus de transformation de votre code source en un programme exécutable. Par synecdoque, il est également utilisé pour décrire l'intégralité de ce processus. Dans la suite de ce chapitre, nous utiliserons le terme dans ses deux entendements.

La compréhension théorique des différents concepts de la compilation revêt une importance pratique cruciale : c'est par cette compréhension que les étudiants peuvent résoudre les problèmes surgissant à la compilation ou à l'édition des liens, problèmes qui ne manqueront pas de surgir à la moindre tentative de compilation... Il faut donc relire ce chapitre jusqu'à l'avoir compris.

Qu'est ce que développer ? Développer ("écrire du code"), c'est projeter en pensée puis dans un langage de programmation un schéma d'exécution, une suite d'instructions (un "algorithme") qui réponde à l'objectif fixé.

Pour des raisons hardware, l'ordinateur ne comprend nativement qu'un jeu très faible d'instructions [1] (les opérations logiques OU, ET, NON, l'addition et la multiplication, etc...). Pour reprendre des éléments de comparaison avec les "vrais" langages, ce premier langage est trop peu expressif pour permettre de développer des

1. en première approximation, car le nombre d'instructions disponibles dans les CPUs modernes est en train de devenir monstrueux.

programmes ambitieux. Au fur et à mesure que la puissance des ordinateurs s'est élevée [2], il devenait possible de développer des programmes plus complexes, et la nécessité de créer des langages de développement plus riches, plus expressifs s'est alors fait ressentir. De nouveaux langages ont été développés, avec des règles qui leur sont propres.

Un langage informatique comporte un ensemble de règles qui peuvent être comparées aux "vrais" langages. De la même manière, il est composé d'une syntaxe, d'une grammaire, d'un vocabulaire, etc. Le langage informatique est un contrat passé entre le développeur et la machine, qui pose un cadre dans lequel tout code respectant les consignes du langage sera garanti de compiler et de pouvoir être exécuté.

Un programme est conçu comme un ensemble de fichiers textes comprenant la liste des instructions à exécuter. Cette liste peut être utilisée de deux manières : elle peut être transformée une fois pour toute en un exécutable, c'est à dire un ensemble d'instructions interprétables par la machine (on parle de *compilation*), ou alors cette liste peut être lue au moment de l'exécution, et chaque instruction est transformée à la volée en un jeu d'instructions compréhensibles par la machine (on parle d'*interprétation*). Même si tous les langages peuvent être interprétés et compilés, on a pour habitude de caractériser un langage par les implémentations qui en sont disponibles, c'est à dire s'il existe des compilateurs ou des interpréteurs pour ce langage. Les langages modernes peuvent donc abusivement se séparer en deux grandes catégories : les langages interprétés et les langages compilés [3]. Le C++ est un langage principalement compilé, mais il existe également des interpréteurs C++ [4]. Dans la suite de ce cours, nous considérons toujours une version compilée du C++.

2. La loi de Moore, qui n'avait pas été énoncée en ces termes par Moore, stipule que la puissance des ordinateurs, c'est à dire leur fréquence (le rapport entre fréquence et puissance des processeurs est détaillée dans la section 3) double tous les 18 mois, et cette loi s'est remarquablement vérifiée jusque 2007/2008.

3. En réalité, les choses sont plus complexes, puisqu'il existe de nombreux langages semi-interprétés, que certains langages ne sont partiellement compilés qu'à l'exécution (c'est le cas pour C# et Java avec la compilation Just In Time (JIT), que sont en développement des langages où la compilation se ferait en continu pendant l'exécution du programme (Compilation Continue), etc...

4. CINT, UnderC, ...

4.1 Déclaration/Définition

En C++, il vous faut à la fois déclarer et définir vos variables, vos fonctions et vos classes.

La ***déclaration*** d'une variable/fonction consiste en la mise en relation d'un nom et d'un type : par exemple, lorsque nous écrivons `int x`, nous spécifions que la variable désignée par le nom `"x"` sera de type entière. La ***définition*** d'une variable/fonction consiste en l'affectation d'une valeur pour cette variable/fonction.

Dans le cas des variables, nous pouvons effectuer ces deux opérations en une seule ligne[5] de la manière suivante :

```
1 int x = 2;
```

Nous pouvons également séparer en deux lignes différentes déclaration et définition, code dans l'exemple suivant :

```
1 int x;
2 x = 2;
```

Bonnes habitudes 1 (Déclaration des variables) *Préférez, quand celà est possible, la première forme pour les déclarations/définitions de variables.*

Dans le cas des fonctions, la déclaration consiste en la donnée des informations suivantes : *nom de la fonction*, *scope de la fonction* (nous y reviendrons), *type et nombre des arguments*, *type de retour*, et éventuellement d'autres informations pour caractériser l'usage de la fonction (mots-clef static/extern etc...). Voici un exemple de déclaration de fonction :

```
1 double Pow(double, int);
```

Nous *déclarons* ainsi une fonction Pow, qui prend deux arguments, l'un réel (type : double) et l'autre entier (type : int), et qui

5. en une seule ligne, mais pas de manière atomique. Ceci dépasse cependant le cadre de notre cours...

retourne un double. Nous pouvons maintenant *définir* cette fonction, c'est à dire instruire le compilateur de ce qu'elle fait exactement :

```
1 double Pow(double x, int n)
2 {
3     double result =1;
4     for (int i = 0 ; i < n; i++)
5     {
6         result *= x;
7     }
8     return result;
9 }
```

Pour les fonctions, les déclarations sont remplies dans des fichiers spécifiques appelés fichiers headers et portant l'extension : ".h", alors que la définition de ces mêmes fonctions sera effectuée dans des fichiers appelés fichiers source portant l'extension ".cpp" [6].

4.2 Les phases de la compilation

La compilation de notre code source en un exécutable est composée de différentes phases. Pour la plupart des langages, il existe un logiciel ou une suite de logiciels appelés environnement de développement (IDE) qui implémente différentes fonctionalités nécessaires au développement : un éditeur de texte, un compilateur, un éditeur de liens, un débugger, un profiler, etc. Dans la plupart des IDE, vous posséderez un bouton "Compiler" qui s'acquittera des différentes étapes automatiquement. C'est le cas par exemple dans Visual Studio (raccourci par défaut F6 ou Ctrl+Shift+B). Cependant, il vous est nécessaire de comprendre les différentes étapes logiques de la compilation pour résoudre les problèmes rencontrés par votre IDE lors de cette compilation.

4.3 Pré-compilation

La précompilation (ou pré-processing) désigne l'ensemble des instructions réalisées par l'IDE au niveau du texte représentant le code source. Avant la précompilation, votre projet est un ensemble de fichiers textes, après la pré-compilation également. Les instructions

6. Sauf dans le cas des fonctions inlinées ou templatées, mais nous y reviendrons.

de pré-processing commencent par le symbole #. Parmi les instructions du pré-processeur, nous parlerons des #include, des #define, des #ifndef, des #endif et nous parlerons des macros uniquement pour vous les interdire.

4.3.1 Les #include

Lorsque vous faites référence dans un fichier B à une fonction déclarée dans un fichier A.h, il est nécessaire que le compilateur sache où trouver cette fonction. Le compilateur n'a pas besoin dans un premier temps d'en connaître la définition, mais il doit pouvoir accéder à sa déclaration : vous vous devez donc d'informer le compilateur qu'il pourra aller chercher des déclarations à l'intérieur du fichier A.h. Ceci se réalise grâce à la commande #include qui sera insérée en en-tête du fichier B :

```
1 #include "A.h"
```

Au niveau de la précompilation, une opération est réalisée de telle sorte que le compilateur considère que le code du fichier A.h se trouve au début du fichier B, à l'endroit où se trouvait l'instruction #include, ce qui va permettre au compilateur de pouvoir accéder dans B à la déclaration de la fonction déclarée dans A.h. Tout se passe donc "comme si" le contenu du fichier A.h était copié dans le fichier B. Cette étape est représentée dans la figure 4.1.

L'utilisation des guillemets spécifie au compilateur qu'il doit trouver le fichier A.h dans le répertoire courant de travail (c'est à dire le répertoire dans votre arborescence de fichier où se trouve votre projet). C'est en règle générale l'utilisation habituelle que vous en ferez. Cependant, lorsque vous utiliserez des fonctions définies non pas par vous mais dans le "noyau" du langage, vous utiliserez non plus des guillemets mais des <> pour spécifier au compilateur qu'il doit aller chercher ce fichier dans les répertoires standards de votre IDE. Exemple :

```
1 #include <iostream>
```

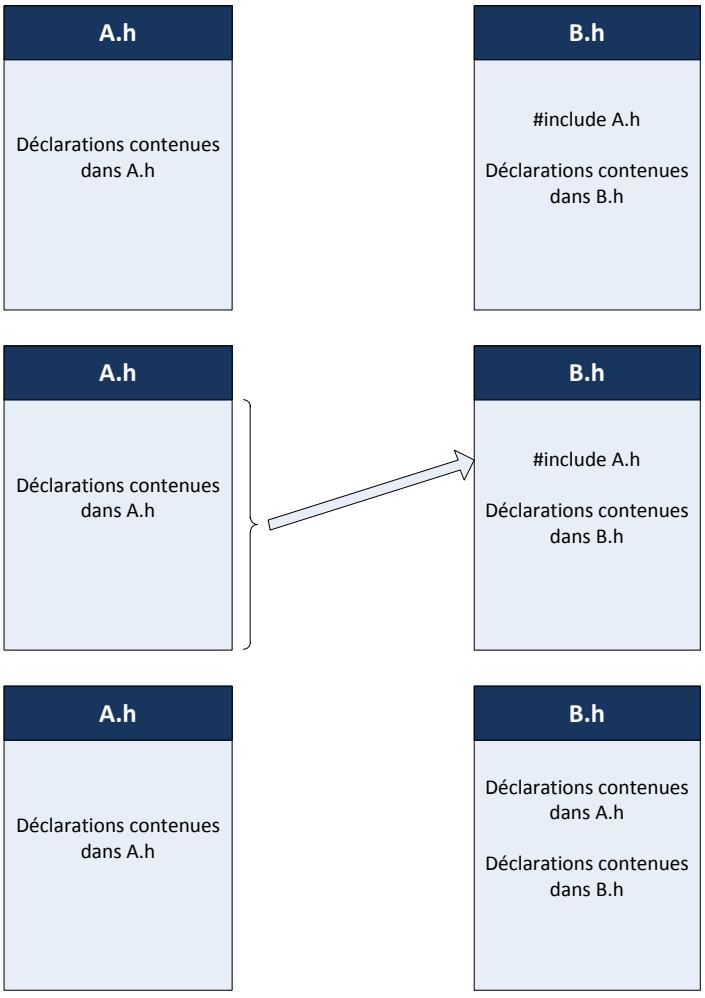

FIGURE 4.1 – Pendant l'étape de préprocessing, tout se passe comme si les déclarations contenues dans A.h étaient ajoutées en en-tête du fichier B.h

4.3.2 Les #define

La commande #define est initialement utilisée pour faire des substitutions de chaînes de caractères à l'intérieur du fichier dans lequel elle est écrite. Le pré-processeur va donc parcourir le fichier et remplacer toutes les occurences de la variable ainsi définie. Ainsi, la commande :

```
1 #define PI 3.141592653589793
```

aura pour effet de remplacer partout dans le fichier où cette commande est définie la chaîne de caractères PI par la chaîne de caractères 3.141592653589793. Il faut vraiment comprendre cette opération comme de la substitution de texte.

Bonnes habitudes 2 (Const et #define) *Sauf raison spécifique, le lecteur est encouragé à préférer l'emploi de const (défini plus bas) plutôt que de recourir à des #define. Comme le précise Scott Meyers dans [17], l'utilisation d'un #define devient invisible dès la fin de l'étape de préprocessing, ce qui peut rendre plus complexe le débugging dans certains cas de compilation. De plus, une variable est générée pour chaque référence dans le code à un #define, alors que l'utilisation d'un const ne génère la création que d'une seule variable.*

4.3.3 Les #ifndef, #endif

Il est possible en C++ de faire de la compilation conditionnelle. C'est à dire qu'une partie de votre code source peut à votre demande n'être compilée que sous certaines conditions. La compilation conditionnelle repose sur la définition de variables de pré-processing, variables qui n'existent qu'avant l'étape de pré-processing. Voici un exemple de telles variables :

```
1 #ifndef PI
2 #define PI 3.141592653589793
3 #endif
```

Grâce à ces variables[7], nous allons pouvoir faire de la compilation conditionnelle, c'est à dire nous assurer qu'une partie de notre

7. qui ne doivent pas être utilisées pour d'autres raisons d'ailleurs.

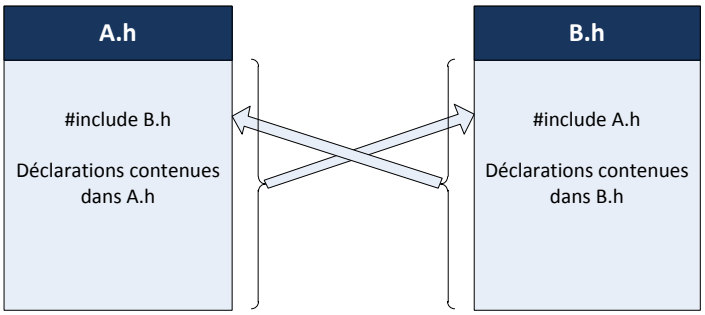

FIGURE 4.2 – Exemple d'inter-dépendance de deux fichiers headers. Si rien n'est fait pour l'éviter, le compilateur va "boucler" à l'infini en ajoutant dans chaque header le contenu de l'autre.

code ne sera compilée que si certaines conditions sont remplies. La compilation conditionnelle est principalement utilisée pour empêcher le compilateur de boucler à l'infini : reprenons notre exemple du fichier header B.h qui inclut le fichier A.h (via un include). Supposons maintenant que le fichier A.h inclut lui-même le fichier B.h. Que se passe-t-il par défaut ? Lorsque le pré-processeur lit l'instruction #include "A.h" dans le fichier B.h, il inclut le contenu de A.h dans B.h, ce faisant, il lit le contenu qu'il insère et le pré-processe. Dans A.h, l'instruction #include "B.h" est exécutée, et B.h est inclus dans A.h qui est inclus dans B.h, qui est inclus dans A.h, et ainsi de suite. Au bout d'un certain temps, votre compilateur rend l'âme et votre compilation échoue.

Pour éviter ceci, nous allons donc recourir à la compilation conditionnelle ; dans chaque fichier, nous adoptons un mécanisme qui s'assure que le code n'est inclus/compilé qu'une seule fois, même en cas d'inclusion multiple.

Le code du fichier A.h s'organise alors de la sorte :

```
1 #ifndef A_H //Following code run only if
2 #define A_H //A_H is not already defined
3
4 /* Content of A.h */
5
6 #endif
```

et le fichier B.h s'écrit de la même manière :

```
#ifndef B_H //Following code run only if
#define B_H //B_H is not already defined

/* Content of B.h */

#endif
```

Bonnes habitudes 3 (Compilation conditionnelle) *Tous les fichiers d'en-tête que vous créez doivent de la même manière contenir leur code entre les instructions #ifndef et #endif.* [8].

4.3.4 Les macros

Les macros sont des fonctions définies grâce à une instruction #define. Elles permettent de définir des fonctions qui n'existeront plus à la compilation : tout comme les variables définies par un #define sont remplacées avant la compilation par leur valeur, les fonctions définies par un #define sont remplacées par la liste de leurs instructions. Un intérêt que nous pourrions trouver à cette technique est de supprimer l'appel à une fonction (parfois coûteux). Nous proposons à titre illustratif un exemple de macro.

```
#define MULTIPLICATE(x,y) x*y
```

Partout où dans notre code un appel est fait à la macro MULTIPLICATE, le préprocesseur remplace l'appel à cet macro par le produit des deux arguments, sans appeler une fonction.

En réalité, les macros sont très dangereuses car particulièrement contre-intuitives. Puisqu'il ne s'agit pas d'appel à une fonction mais bien de substitution syntaxique du préprocesseur, elles génèrent souvent des comportements non attendus. L'utilisation de macros avait un sens il y a 20 ans, mais il n'en a plus aucun aujourd'hui [9]. A titre d'exemple seulement, nous présentons trois cas simples dont les résultats, erronés, doivent vous dissuader d'utiliser des macros :

8. Il est possible de remplacer cette syntaxe par un #pragma once, mais ceci n'est pas défini par la norme C++ : cette instruction est reconnue par Visual Studio mais pas par d'autres IDE, c'est pourquoi nous vous la déconseillons et n'en parlons pas plus avant.

9. On leur préfère maintenant des fonctions templatées inlinées, qui ont les mêmes avantages, mais pas leurs inconvénients.

```
#define MULTIPLICATE(x,y) x*y
```

Appliquons la macro MULTIPLICATE en a+1 et b+1 :

```
double y = MULTIPLICATE(a+1,b+1);
```

Nous obtenons en expression littérale :

```
y = a + 1*b + 1 = a + b + 1
```

Ce qui est tout à fait différent du résultat anticipé. Ceci est dû au fait que l'étape de préprocessing est uniquement une étape de substitution syntaxique. Dans le cas présent, il manque des parenthèses à notre macro, que nous corrigeons alors par :

```
#define MULTIPLICATE(x,y) (x)*(y)
```

Cette fois-ci, l'utilisation de la macro dans le cas précédent donne bien le résultat attendu. Cependant, dans le cas de l'expression suivante, la macro donne encore un résultat erroné :

```
6 / MULTIPLICATE(2,3)
```

Nous obtenons l'expression littérale :

```
6 / 2 * 3 = 3 * 3 = 9
```

Essayons alors la macro SQUARE :

```
#define SQUARE(x) ((x)*(x))
```

Nous vous laissons à titre d'exercice deviner pourquoi la variable y prend la valeur 6 dans le cas suivant :

```
1 int x = 1;
2 int y = SQUARE(x++);
```

La conclusion de ce paragraphe est donc la suivante :

Bonnes habitudes 4 (Utilisation des macros) *N'utilisez jamais de macros.*

4.3.5 Templates

C'est pendant l'étape de pré-processing que les classes et fonctions templatées sont créées. Selon l'usage des templates que vous aurez, cette phase pourra être réduite à sa plus simple expression ou au contraire se révéler très longue [10]. Nous reviendrons sur ce point dans le chapitre consacré aux templates.

4.4 La compilation

A la suite du pré-processing, vient la deuxième étape de la compilation dans laquelle le compilateur compile chaque fichier source (.cpp), c'est-à-dire qu'il transforme chacun de ces fichiers sources en fichiers binaires (.o ou .obj) contenant du code directement exécutable par la machine. Cette phase constitue la compilation proprement dite.

Bonnes habitudes 5 (Compilation régulière) *Compilez votre code dès que possible. Quoi qu'il arrive, compilez au moins une fois par heure. Lorsque vous débutez, compilez toutes les 5 minutes. Si la compilation échoue, réglez les erreurs de compilation avant de continuer à développer. Si vous travaillez dans un IDE correct, il vous montrera les problèmes de code avant même que vous ne compiliez ; réglez les dès que vous les voyez.*

Dans un environnement professionnel, vous disposerez d'intégration continue via un serveur de build [11]*. Veillez à ne pas casser le build.*

10. Le lecteur intéressé par le pré-processeur découvrira avec joies les délices du méta-programming, ou de la compilation avec Boost...

11. Si ce n'est pas le cas, changez de société.

4.5 L'édition des liens

Enfin, le linker (ou éditeur de liens en français) agrège chaque fichier .o ou .obj (avec éventuellement d'autres fichiers binaires si vous avez utilisé des librairies externes). Le linker va faire les liens entre les fichiers binaires générés en permettant de localiser le code correspondant aux appels de fonctions. Le linker vérifie en particulier que chaque fonction appelée dans le programme n'est pas seulement déclarée (ceci est fait lors de la compilation) mais aussi bien définie (chose qu'il n'avait pas vérifiée à ce stade). Il vérifie aussi qu'une fonction n'est pas implémentée dans plusieurs fichiers .o. A la fin de l'édition des liens, un exécutable est créé.

4.6 Un exemple éléméntaire

Nous disposons d'un projet contenant les fichiers main.cpp, A.cpp, B.cpp, A.h, B.h. Le fichier main.cpp fait référence au code contenu dans les fichiers A et B, le fichier A fait référence à B.

```
1  ////////////////////////main.cpp//////////////////////
2  #include "A.h"
3  #include "B.h"
4
5  void main()
6  {
7      functionInA();
8      functionInB();
9  }
```

```
1  ////////////////////////A.h//////////////////////////
2
3  #ifndef A_H
4  #define A_H
5
6  #include "B.h" //not useful here, just put to illustrate the compilation process
7
8  void functionInA(void);
9
10 #endif
```

```
//////////////////////B.h/////////////////////////////
#ifndef B_H
#define B_H

void functionInB(void);

#endif
```

```
//////////////////////////////////////A.cpp
    ////////////////////////////////
#include "A.h"
#include "B.h"

void functionInA(void)
{
    functionInB();
}
```

```
//////////////////////////////////////B.cpp
    ////////////////////////////////
#include "B.h"

void functionInB(void)
{
    //Do Nothing
}
```

Le préprocesseur copie :
- les déclarations de B.h dans A.h,
- les déclarations de A.h et B.h dans main.cpp,
- les déclarations de A.h et B.h dans A.cpp,
- les déclarations de B.h dans B.cpp

Le fait d'utiliser les #ifndef #endif nous permet d'éviter d'inclure deux fois les déclarations de B.h dans main.cpp et A.cpp, ce qui amènerait à des erreurs de compilation.

Une fois le preprocessing achevé, le compilateur convertit chaque fichier .cpp en un fichier binaire .o ou .obj (selon le compilateur). Lorsque les fichiers main.obj, A.obj et B.obj sont créés, l'éditeur des liens permet de matcher les fonctions, c'est à dire que l'éditeur des liens parcourt les fichiers .obj pour trouver la fonction functionInB() appelée dans la fonction functionInA(). Cette fonction est trouvée dans B.obj, et l'éditeur de lien indique que lorsque functionInA() ap-

pellera functionInB(), il faudra exécuter le code présent dans B.obj. L'éditeur de lien fait de même pour les fonctions functionInA() et functionInB() utilisées dans main.obj. Lorsque tous les appels à des méthodes extérieures sont résolus, l'éditeur de lien réassemble tous les fichiers .obj en un fichier exécutable (.exe) ou en une librairie (.dll).

4.7 Survivre à des messages d'erreurs incompréhensibles

4.7.1 Erreurs à la génération du projet

Lorsque vous voudrez compiler un projet, il y a fort à parier que la compilation échouera en raison d'erreurs. Votre IDE va vous fournir un descriptif d'une ou de plusieurs erreurs qu'il a rencontrées pendant la génération de votre projet. Même si ces erreurs sont multiples, seule la première des erreurs listées peut être considérée comme fiable, les autres erreurs étant sujettes à caution (en effet, un IDE est sensé fonctionner si vous fournissez un code sans erreur. Si vous lui demandez de compiler un code pour lequel il n'est pas sensé fonctionner, il est possible qu'il ne puisse pas identifier précisément tout ce qui l'empêche de fonctionner).

Tout débutant en informatique sait "coder", ce qui distingue un débutant autonome d'un débutant bloqué, c'est sa capacité à comprendre les messages sybillins d'erreurs de l'IDE. Dans un premier temps, il vous faudra comprendre si le premier message d'erreur retourné est un message du pré-processeur, du compilateur, ou de l'éditeur de lien. Dans le cas d'une erreur du compilateur, regardez le fichier et la ligne en cause : c'est probablement une erreur de syntaxe dans votre code, ou une utilisation d'une fonction dont le compilateur ne trouve pas la déclaration en raison d'un #include adéquat manquant en début de fichier. Dans le cas d'une erreur de l'éditeur des liens (ces erreurs commencent par LNK dans Visual Studio, pour "linker"), l'IDE a bien trouvé la déclaration de la méthode que vous utilisez, mais il n'a pas réussi à résoudre sa définition, c'est à dire qu'il a ou bien trouvé plusieurs fonctions de même nom et de même prototype et qu'il ne sait laquelle choisir, ou bien qu'il n'a trouvé aucune fonction qui convenait.

Bonnes habitudes 6 (Environnement de travail en anglais) *Efforcez-vous d'avoir un IDE entièrement en anglais. Tout d'abord,*

*puisque plus répandues, les versions anglophones des logiciels sont toujours moins buggées, ensuite parce que si vous avez un message d'erreur que vous n'arrivez pas à interpréter, une recherche dans google du message d'erreur en anglais vous mène souvent à une solution, alors que la même recherche sur le message d'erreur français vous apportera trop souvent : "Your search - ****** - did not match any documents."*

4.7.2 Divide and Conquer

Que vous ayez des problèmes à la compilation ou à l'exécution, si les messages d'erreur que vous obtenez ne sont pas suffisament explicites et que vous n'arrivez pas à diagnostiquer votre problème, adoptez une démarche dichotomique pour isoler la section de votre code fautive. Compilez/exécutez votre code en y enlevant certaines parties, et itérez ainsi afin de circonscrire la partie du code en défaut.

4.7.3 Stack Overflow

Une fois votre problème isolé, la résolution devrait vous paraître évidente. Dans le cas contraire, après vous être soigneusement assurés que vous ne pouviez trouver d'explications sur internet à votre problème isolé, vous pouvez le poster sur Stack Overflow (http ://stackoverflow.com) avec un maximum d'explications et en veillant à bien respecter les règles de rédaction [12].

Toutes les questions correctement formulées recoivent une réponse correctement formulée dans l'heure.

12. http ://stackoverflow.com/help/how-to-ask

"I spent a few weeks... trying to sort out the terminology of "strongly typed," "statically typed," "safe," etc., and found it amazingly difficult.... The usage of these terms is so various as to render them almost useless.", Benjamin C. Pierce, in Types and Programming Languages

5 Variables et Typage

Vous êtes familiers avec les mots clefs int, short, long, double et float, et avec la base 2 ? sautez ce chapitre en première lecture.

Remarque préliminaire : L'alphabet d'un ordinateur est limité aux signes 0 et 1, ce qui correspond à une porte électronique ouverte ou fermée, et cet état est représenté par un bit. Avec un ensemble de n bits, nous pouvons former 2^n combinaisons différentes, et donc stocker au plus 2^n valeurs différentes. Toutes les variables ont besoin de plusieurs bits pour définir leur valeur, et l'unité de référence pour la taille mémoire est l'octet (byte en anglais), c'est à dire un paquet de 8 bits consécutifs [1]. Dans ce chapitre, nous faisons un bref rappel des systèmes de base et introduisons les différents types primitifs du C++.

5.1 Représentation décimale

La représentation des nombres que nous utilisons dans la vie de tous les jours est une représentation en base 10 [2]. C'est à dire qu'une dizaine est consituée de dix unités, qu'une centaine est constituée de dix dizaines, etc. Les bases utilisées en informatique sont princi-

1. On prendra donc bien garde à ne pas confondre bit et byte (=8 bits). Certains providers internet abusent d'ailleurs de cette confusion en affichant les débits garantis en terme de Mbits/secs, c'est pourquoi vous avez une connection de 100 Mbits, mais que vous ne téléchargez pas à plus de quelques Mo/sec ...

2. Même si nous utilisons également des vieilles bases sexagécimales (60) d'origine babylonienne pour compter les minutes ou les secondes.

palement des bases binaires et hexadécimales (16), ceci étant lié à l'alphabet réduit à 2 caractères (0 ou 1).

5.2 Représentation en base 2

La représentation en mémoire des entiers (mais aussi des décimaux) va utiliser le principe de l'écriture en base 2. Sur tous les processeurs PC[3], le bit le plus à gauche est utilisé comme bit de poids faible, et correspond selon sa valeur à 0 ou 1 (2^0). Pour le bit à sa droite, celui-ci correspond à la valeur 0 ou 2 (2^1), celui d'encore à droite à la valeur 0 ou 4 (2^2).

Quelques exemples :

$$10110 => 1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3 + 0 * 2^4 => 13$$
$$011 => 0 * 2^0 + 1 * 2^1 + 1 * 2^2 => 6$$

5.3 les nombres négatifs

Comment représenter un nombre négatif? Comme il n'y a que deux possibilités pour un nombre (positif ou négatif), le signe d'un nombre peut être représenté sur un bit. Par convention, le bit de gauche d'un ensemble de bits est le bit de signe : sa valeur (0 ou 1) détermine le signe du nombre stocké dans les bits suivants. Celà diminue donc d'une unité le nombre de bits disponibles pour stocker le nombre lui-même. Lorsque l'on travaille uniquement avec des nombres positifs, il est possible de signifier à la machine que nous voulons faire l'économie du bit de signe pour pouvoir travailler sur des valeurs potentiellement plus grandes. Nous revenons sur ce point au paragraphe suivant.

5.4 les entiers

Il existe en C++ 6 types d'entiers différents, qui se répartissent en deux catégories, les entiers signés et les entiers non-signés. Les entiers non signés, qui ont renoncé au bit de signe, peuvent donc prendre deux fois plus de valeurs que leurs analogues signés.

3. et depuis récemment sous les machines Apple, le lecteur souhaitant développer sous des machines Mac un peu vieilles et utiliser des lectures disque est très vivement invité à s'enquérir du principe de Little et Big Endian.

Type	Minimum	Maximum
`unsigned int`	0	$2^{32}-1$
`int`	-2^{31}	$2^{31}-1$
`unsigned short`	0	$2^{16}-1$
`short`	-2^{15}	$2^{15}-1$
`unsigned long`	0	$2^{32}-1$
`long`	-2^{31}	$2^{31}-1$
`unsigned char`	0	255
`char`	-128	127

TABLE 5.1 – Bornes des différents types

- Les `short` stockent un entier sur 2 octets, ils prennent des valeurs entre -32768 et +32767
- Les `long` stockent un entier sur 4 octets, ils prennent des valeurs entre -2147843648 et +2147843647
- Les `int` n'ont pas une taille définie par la spec du langage. Ils ont sur la plupart des compilateurs/machines une taille de 4 octets, et ont donc le même comportement que les long, mais ce n'est pas systématique.
- Les `ushort` ou `unsigned short` stockent un entier sur 2 octets, ils prennent des valeurs entre 0 et $+2^{16}-1$
- Les `ulong` ou `unsigned long` stockent un entier sur 4 octets, ils prennent des valeurs entre 0 et $+2^{32}-1$
- Les `uint` ou `unsigned int` là encore, celà dépend des compilateurs, mais en règle générale, même cas que les unsigned long

Voici deux exemples de déclarations d'entiers.

```
1 unsigned int currentIndex;
2 int matchingCount;
```

Comme nous l'avons expliqué, signer ou non un entier a des conséquences sur l'intervalle des valeurs qu'il peut prendre ; ces intervalles sont précisés dans le tableau 5.1 [4].

Que se passe-t-il lorsque l'on ajoute un à une variable entière qui contient la valeur maximale ? Un tel phénomène est appelé Integer Overflow. Dans le cas d'entiers signés, le résultat est imprévisible. Dans le cas d'entiers non signés, le résultat est réduit modulo une puissance de 2. C'est parfois drôle [5], ca donne parfois l'occasion de

4. Les bornes pour les `int` ne sont valables que pour les architectures 32 bits.
5. ca donne des jolis Kill-Screen http ://en.wikipedia.org/wiki/Pac-Man#Split-screen

faire des bandes dessinées [6], mais ca donne des bugs très pénibles à isoler, et qui peuvent se révéler catastrophique (ca peut même crasher une Ariane 5 [7]).

Bonnes habitudes 7 (Typage des entiers) *Nous invitons le lecteur à prendre l'habitude de ne travailler uniquement qu'avec des int, mais c'est un parti pris qui ne fait pas l'unanimité. Ceci peut se révéler problématique dans certains cas, notamment puisque la taille des int est laissée à la discrétion du compilateur et qu'un même programme compilé sur une même machine mais avec deux compilateurs différents pourra avoir des comportements différents. Cependant, l'utilisation des int est en règle générale très suffisante. La vraie raison qui nous porte à faire cette proposition est la simplicité de votre code. Utiliser des short ou des long, c'est attirer l'attention de votre relecteur sur le fait que vous vous êtes passés des int, et c'est implicitement suggérer que ce changement est nécessaire à l'endroit où vous l'avez fait. Un bon code se doit d'être simple et banal aux points les plus simples, et n'attirer l'attention du lecteur que lorsque ceci est nécessaire.*

5.5 Les réels

Les nombres réels sont stockés autrement que les nombres entiers. Ils sont dits à virgule flottante. Les nombres à virgule flottante sont des nombres dans lesquels la position de la virgule en tant que séparateur entre partie entière et partie décimale n'est pas figée. La grandeur d'un tel nombre est donnée par un exposant de 10 adéquat. Par exemple, le nombre 27,6 peut être écrit sous les 3 formes :

$$2,76 * 10^1$$
$$0,276 * 10^2$$
$$276 * 10^{-1}$$

Dans la mémoire de la machine, un nombre réel est décomposé en un signe (+ ou -), en un exposant (ex : 10^1), et une mantisse (ex : 2,76). Comme nous avons plusieurs types pour distinguer les entiers selon les plages de valeurs que nous anticipons, nous avons également plusieurs types différents pour stocker un réel.

6. http ://xkcd.com/571/
7. http ://www.astrosurf.com/luxorion/astronautique-accident-ariane-v501.htm

- Les `float` : sur 4 octets, mantisse sur 23 bits, exposant sur 8 bits, signe sur 1 bit. Le float garantit une précision d'au moins 6 chiffres après la virgule.
- Les `double` : sur 8 octets, mantisse sur 52 bits, exposant sur 11 bits, signe sur 1 bit. Le double garantit une précision d'au moins 15 chiffres après la virgule.
- Les `long double` : sur 10 octets, mantisse sur 64 bits, exposant sur 15 bits. Le long double garantit une précision d'au moins 17 chiffres après la virgule.

Bonnes habitudes 8 (Typage des réels) *Là aussi, nous prenons le parti de vous conseiller d'utiliser uniquement des double. L'utilisation de float suggère à votre lecteur que vous êtes en train de faire des économies sur la mémoire pour faire des optimisations fines. En règle générale, étant donnés la taille des RAMs actuelles et le salaire horaire des bons développeurs, il vaut mieux avoir un programme un peu moins optimisé et éviter des bugs atroces qui peuvent coûter des semaines à détecter et fixer car vous avez voulu économiser un peu d'espace mémoire*[8].

5.6 Déclaration des variables

Il est crucial de comprendre la différence entre définition et déclaration. Nous renvoyons le lecteur au chapitre sur la compilation/interprétation si ce pas n'est pas encore clair[9].

Lorsque nous souhaitons déclarer une variable, nous le faisons de la manière suivante :

```
1 TypeDeLaVariable NomDeLaVariable;
```

La déclaration peut être effectuée n'importe où dans le code. Cependant, l'endroit où elle est déclarée est extrêmement important, puisqu'il définit le scope de la variable, c'est à dire sa portée, et pourra parfois empêcher la compilation. Nous reviendrons sur ce point lorsque nous aurons introduit les fonctions.

Bonnes habitudes 9 (Déclaration des variables) *Une bonne habitude est de déclarer les variables le plus tard possible dans le code,*

8. Repensez à la fusée Ariane!
9. Nous vous avions bien dit qu'il fallait lire ce chapitre jusqu'à l'avoir compris!

c'est à dire de minimiser leur scope, afin d'améliorer la lisibilité et d'aider le compilateur dans ses optimisations[10].

Bonnes habitudes 10 (Nom des variables) *Toutes vos variables, comme tous vos fichiers, vos méthodes et vos commentaires doivent être nommés en anglais.*

1. *L'anglais est souvent plus compact. Pouvoir exprimer une idée en moins de lettre est extrêmement appréciable puisque nous voulons limiter la taille des noms des variables.*
2. *Certains concepts n'ont pas de traduction adéquate et répandue dans notre langue.*
3. *Vous serez très probablement amenés à travailler en équipe, régulièrement avec des gens non-francophones. Imaginez-vous devoir lire du code écrit et commenté en russe...*

Bonnes habitudes 11 (Nom des variables (2)) *Trouvez des noms courts, expressifs, spécifiques et non-provoquants (On est jamais à l'abri d'un bug qui soulève un retour windows en expliquant que l'instance z de la classe fuckstring n'a pas été instanciée, devant un client/n+1/examinateur...). Une exception : pour coder une fonction mathématique, ces conseils sont différents, préférez des noms de variables très courts à des noms expressifs, du type x, y, z, dx, etc.*

Le listing suivant présente quelques exemples de déclaration de variables.

```
1 double threshold;
2 long   y;
3 int    length;
4 short  minValue;
5 float  mean;
6 char   firstLetter;
7 bool   isLocked;
```

Il est important de noter que les noms des variables sont sensibles à la casse (case sensitive)[11]. En d'autres termes, cela signifie que `Variable1` et `variable1` désignent deux variables différentes. **Nous attirons votre attention sur cette propriété, qui sera source dans vos premiers programmes des trois quarts des erreurs de compilation que vous rencontrerez.**

10. Il y a des contre-exemples bien sûr, mais celà dépasse le cadre de ce cours
11. C'est-à-dire à la distinction entre majuscules et minuscules.

Type	Description	Java	VBA
int	Entier	Integer	Integer
double	Decimal double précision	Double	Double
char	caractère	Char	char

TABLE 5.2 – Types de variables

Bonnes habitudes 12 (Usage des majuscules) *Par convention,*

- *les variables commencent par une minuscule.*
- *les fonctions commencent par une majuscule.*
- *les classes commencent par une majuscule.*
- *les constantes commencent par une majuscule.*
- *les constantes définies par un #define sont entièrement en majuscules.*
- *lorsqu'un nom est la concaténation de plusieurs mots, la première lettre de chaque mot en dehors du premier prend une majuscule (ex : PerfectRedWidget).*

Ces règles permettent de distinguer d'un seul coup d'oeil variables, classes et méthodes et sont indispensables[12]*. Elles permettent aussi de pouvoir recourir au type de syntaxe suivant, dans lequel le premier mot désigne le type et le deuxième mot désigne le nom de l'instance :*

```
1 Widget widget;
```

5.7 Les booléens

Les variables de type `booléen` contiennent un booléen, c'est à dire une des deux valeurs suivantes : *true* ou *false*. Par convention, la valeur false correspond à 0, la valeur true correspond à 1. Un booléen pourrait donc être stocké sur un unique bit. Cependant, puisque tous les autres types ont besoin de plusieurs octets, il aurait été malavisé de n'utiliser qu'un bit pour les booléens, puisque ceci aurait introduit des décalages. Les booléens sont donc stockés sur un octet entier.

12. Nous nous acharnerons sur vous en TD si vous ne les respectez pas.

5.8 Les caractères

Il y a malheureusement plusieurs standards différents dans l'encodage des caractères [13], tout comme il y a plusieurs types de clavier et plusieurs alphabets. C'est pourquoi vous récupérez parfois des mails avec des caractères étranges quand vous êtes sur des OS différents par exemple. Sur un octet, nous pouvons stocker 256 caractères différents. Un des standards les plus utilisés est le standard américain, ASCII. La variable caractère est désignée par le mot clef char, mais sauf cas de force majeur, utilisez plutôt des chaines de caractères représentées (string) que des tableaux de char.

5.9 Types définis par l'utilisateur

La déclaration/définition/instanciation de variables d'un type non primitif (c'est à dire défini par vous-même) s'effectue de la même manière. Par exemple :

```
1 Identifier identifier = myInstance.GetId();
```

Pour la définition de telles variables, nous approfondirons la question dans le chapitre dédié aux classes.

5.10 Constantes et énumérations

5.10.1 Constantes

Régulièrement lorsque nous écrivons un programme, nous avons besoin de définir des constantes, comme dans le listing 5.1.

```
1 #include <iostream.h>
2
3 int main()
4 {
5   double pi = 3.1415926538;
6
7   cout << pi/2;
8 }
```

Listing 5.1 – Nécessité d'une constante

13. c.f. par exemple http ://www.joelonsoftware.com/articles/Unicode.html

Dans le code précédent, rien n'interdit de redéfinir la valeur de cette "constante" au cours du programme. Il est possible de remédier à ce problème au moyen du mot clé `const`, qui indique qu'une variable est constante, et ne peut être modifiée.

```
1   const type instanceName = value;
```

Le listing 5.1 devient alors :

```
1  #include <iostream.h>
2
3  int main()
4  {
5    const double pi = 3.1415926538;
6
7    cout << pi/2;
8  }
```

Listing 5.2 – Nécessité d'une constante

5.10.2 Enumérations

Nous pouvons également avoir besoin d'une liste de constantes mais liées entre elles. Considérons le code du listing 5.3.

```
1  #include <iostream.h>
2
3  const int small = 0;
4  const int medium = 1;
5  const int large = 2;
6
7  void f(int size)
8  {
9    if (size == small)
10     cout <<"small";
11 }
12
13
14 int main()
15 {
16   f(small);
17 }
```

Listing 5.3 – Une série de constantes

Ce code présente plusieurs problèmes :

- Il est possible de passer une taille en dehors des valeurs de la liste de constantes. Par ailleurs, rien ne garantit que c'est bien une taille que nous allons passer ;
- Si nous voulons rajouter de nouvelles tailles, il faut gérer soi-même l'attribution de nouvelles valeurs (4, 5, 6, etc.).

Le langage C++ fournit une méthode automatique pour résoudre ce problème, appellée énumération. Nous déclarons une énumération de la manière suivante :

```
1  enum NomEnumeration
2  {
3    premierElement,
4    deuxiemeElement,
5    troisiemeElement
6    ...
7  };
```

La numérotation [14] est automatique. En l'occurrence, notre enumération s'écrirait :

```
1  enum Size
2  {
3    Small,
4    Medium,
5    Big
6  }
```

Le listing 5.3 devient alors :

14. Ce n'est d'ailleurs pas forcément 1, 2, 3, etc.

```
#include <iostream.h>

enum Size
{
  SMALL,
  MEDIUM,
  BIG;
}

void f(Size size)
{
  if (size == SMALL)
    cout <<"small";
}


int main()
{
  f(Size.SMALL);
}
```

Listing 5.4 – Emploi d'une énumération

L'emploi d'une énumération a donc résolu nos problèmes :

- Un mécanisme garantit que c'est bien une valeur valable qui sera passée en argument de la fonction f.
- Nous pouvons rajouter de nouvelles valeurs sans nous préoccuper de la numérotation.
- Le code obtenu est nettement plus lisible.

5.11 Tableaux statiques

Il est possible en C++ de déclarer un tableau de variables. Cela se fait de la manière suivante :

```
Type arrayName[size];
```

Listing 5.5 – Déclaration d'un tableau

Par exemple,

```
double someArray[ 50 ];
```

permet de déclarer un tableau de 50 nombres réels en double précision.

Nous accèdons alors aux éléments du tableau à l'aide de l'opérateur `[ ]`. Pour créer un tableau de 10 éléments et y ranger les carrés des entiers de 1 à 10, nous écririons :

```
double someArray[ 10 ];

for( int i = 0 ; i < 10 ; i++)
{
    someArray[i]=(i+1)*(i+1); //All arrays start at index 0
}
```

Listing 5.6 – Utilisation d'un tableau

La syntaxe équivalente en Python ou en VBA serait :

Python

```
#c'est une liste  et
#non un tableau
tableau = []

for i in range(1, 11):
  tableau.append(i * i)
```

VBA

```
dim i as Integer
dim tableau(10) as Integer

for i=1 to 10
    tableau(i)=i*i
next i
```

Pour créer un tableau dont la taille n'est pas une constante écrite nommément dans le code, c'est beaucoup plus difficile ; nous reviendrons sur ce point dans le chapitre 15.

5.12 Opérations de conversion / casting

L'opération de convertir une variable d'un certain type en une variable d'un autre type s'appelle conversion de type, ou *type casting*. Nous utiliserons indifféremment les deux dénominations.

5.12.1 Conversions implicites

Les conversions implicites, comme leur nom l'indique, ne requiert aucune opération de votre part et sont réalisées automatiquement

par l'environnement. Ainsi le code suivant est correct :

```
float pi = 3.14;
double pii = pi;
```

Listing 5.7 – "Conversion implicite"

Il existe de nombreuses conversions implicites pour les types primitifs, comme les conversions suivantes : short => int, int => float, float=>double, double => int, etc.

Il existe aussi des conversions implicites qui sont beaucoup moins évidentes. Ainsi, si le type `B` possède un constructeur qui prenne en argument une instance de type `A`, le cast implicite suivant est légal :

```
class A {};
class B { public: B (A a) {} };

A a;
B b=a;
```

Listing 5.8 – "Conversion implicite par le constructeur"

C'est clairement une fonctionnalité WhatTheFuck du langage, mais elle reste assez agréable pour la déclaration de SmartPointer, cf chapitre 14.

5.12.2 Conversions explicites

Pour de nombreuses autres conversions, il est nécéssaire d'expliciter l'opération de conversion. Cette explicitation se réalise en mettant à gauche de la variable que l'on souhaite convertir (caster) le type vers lequel nous souhaitons convertir, encadré par des parenthèses. Ainsi, la conversion suivante est-elle réalisée explicitement :

```
short s = 200;
int b = (int) s;
```

Listing 5.9 – "Conversion explicite d'un short en int"

Deux fonctions ou deux opérateurs peuvent avoir le même nom, mais posséder des arguments de type différents, et avoir ou non le

même comportement. Ainsi, l'opérateur "/" (lire "divisé par") est-il défini différemment pour des entiers et pour des réels. Dans le cas où `a` et `b` sont des entiers, le résultat de `a/b` est le quotient dans la division euclidienne de `a` par `b`. Ainsi, si `a` vaut 16 et `b` vaut 17, le résultat de `a/b` vaut 0. Cependant, si `c` et `d` sont deux réels, alors le sens de `c/d` correspond au quotient réel, qui vaut une valeur proche de 1, même si `c` et `d` ont pour valeur un entier.

```
int a = 16;
int b = 17;
int q1 = a/b; //q1 = 0

double c = 16;
double d = 17;
double q2 = c/d // q2 = 0.94117647058
```

Listing 5.10 – "Divisions réelles et euclidiennes"

Ainsi, lorsque nous voudrons estimer un quotient de deux entiers, il faudra bien prendre garde que sauf mention du contraire, le quotient calculé sera celui de la division euclidienne. Si nous voulons obtenir le quotient réel, il nous faut caster au moins un des deux arguments en réel, comme dans le listing suivant, afin de faire comprendre au compilateur que nous ne voulons pas du quotient euclidien.

```
int a = 16;
int b = 17;
double q1 =((double) a)/b; //q = 0.94117647058
```

Listing 5.11 – "Cast d'un entier en double pour obtenir une division réelle"

5.12.3 Indécisions sur les cast

De manière semblable, définissons une fonction Power, qui prenne en argument deux réels `x` et `y` pour calculer x^y.

```
#include <math.h>
double Power(double x, double y)
{
    return exp(y*log(x));
}
```

Listing 5.12 – "Une première fonction Power"

Nous pouvons donner en argument à cette fonction un entier, l'environnement se chargeant de réaliser le cast implicitement.

```
1
2 void main()
3 {
4     int a = 3;
5     int b = 2;
6     double c = Power(a,b);
7 }
```

Listing 5.13 – "Un autre cast implicite"

Définissons ensuite la même fonction Power dont les arguments sont cette fois-ci des `float`.

```
1 #include <math.h>
2 float Power(float x, float y)
3 {
4     return exp(y*log(x));
5 }
```

Listing 5.14 – "Une deuxième fonction Power"

Nous ne pouvons plus maintenant réaliser de conversion implicite, car l'environnement ne sait s'il doit caster nos entiers en `double` et utiliser la première fonction, ou bien caster nos entiers en `float` et utiliser la deuxième fonction. Ainsi à la compilation, nous obtenons un message d'erreur semblable au suivant :

```
more than one instance of overloaded function "Power" matches the argument list : function "Power(double x, double y)" function "Power(float x, float y)" argument types are : (int, int)
```

Il nous faut donc ici réaliser un cast explicite pour lever l'ambiguïté.

```
1
2  void main()
3  {
4      int a = 3;
5      int b = 2;
6      double c = Power( (double) a, (double) b);
7  }
```

Listing 5.15 – "Le cast explicite est ici indispensable"

Bien que le C++ soit statiquement typé (cf paragraphe suivant) et que celà garantisse que de nombreuses erreurs de typage aient lieu à la compilation plutôt qu'à l'exécution (ce qui réduit très fortement leur potentiel de nuisance), il est possible de tromper le compilateur via toutes sortes d'horreurs résultant de casts explicites, et ainsi de passer les vérifications à la compilation et d'échouer seulement à l'exécution. Pour se prémunir au moins partiellement de ces problèmes, il existe différents opérateurs de cast supplémentaires (`dynamic_cast<T>`, `static_cast<T>`, `reinterpret_cast<T>` et `const_cast<T>`), mais ceci dépasse le cadre de notre cours.

5.13 Le typage du C++

Le C++ est un langage typé de manière statique. Cela signifie que les variables doivent être déclarées et leur types explicités, à la différence par exemple du python où une même variable peut contenir successivement un entier puis une chaîne de caractère ou un double. Le compromis qui se cache derrière ce choix est un compromis souplesse / performance et garantie. Devoir déterminer à l'exécution le type d'une variable plutôt que de l'avoir déterminé statiquement a un certain coût. Les types n'existent en C++ qu'à la compilation, c'est à dire qu'à l'execution il n'est plus possible de récupérer le type d'un objet ; cette limitation ne se retrouve pas dans des langages plus récents comme le C#. [15].

15. En C#, on peut par exemple récupérer le type d'un objet à l'exécution, parcourir l'ensemble des types chargés en mémoire, sélectionner les types qui héritent de telle classe et qui possèdent un constructeur vide, les instancier, etc.

6 Les pointeurs

6.1 Définition des pointeurs

Comme nous l'avons expliqué précédemment, le C++ permet à la fois de manipuler des abstractions puissantes mais aussi d'être très proche de la machine quand celà est nécessaire. Dans ce chapitre, nous nous plaçons à un niveau d'abstraction très bas pour présenter les pointeurs.

Chaque variable d'un programme est stockée dans une des mémoires de la machine qui exécute le code (mémoire Cache, mémoire RAM, disque dur, etc.). Nous pouvons pour le moment considérer la mémoire de notre machine comme un ensemble de cases mémoires contigües, chacune d'un octet. Chaque case est numérotée, afin de pouvoir rapidement accéder à une case précise. Lorsque nous écrivons :

```
1 unsigned int n=3;
```

nous pouvons accéder de deux manières différentes à la valeur de notre variable : la première en faisant appel au nom de la variable, c'est à dire en invoquant son nom -n-, la deuxième en allant chercher directement en mémoire à l'adresse de notre variable la valeur qui s'y trouve. Dans le cas qui nous intéresse ici, la variable n a été stockée sur 4 octets, par exemple de cette manière :

Case mémoire	0	1	2	3	4	5	6	7	8	9
Nom				n	n	n	n			
Valeur				110..000	00...	000...	000..			

Le C++ permet d'obtenir à l'exécution l'adresse d'une variable (qui va évidemment varier à chaque exécution puisque nous n'avons pas le contrôle sur l'endroit où nos variables sont stockées), grâce à l'opérateur &. Il nous suffit donc d'écrire &n pour obtenir l'adresse de n. Cette adresse correspond en fait à l'index de la première case mémoire dans laquelle est stockée notre variable (Dans notre exemple ce serait la case numéro 3). Naturellement, on pourrait penser que le type d'une adresse est donc un entier. En fait, il n'en est rien pour deux raisons :

- Il serait très facile de confondre un entier et son adresse si tous les deux étaient du même type.
- Pour retrouver une variable en mémoire, il ne suffit pas de connaître l'adresse de la première case mémoire qui lui est allouée, il faut également connaître son type. En effet, si la variable stockée est un int ou un double, elle ne sera pas encodée de la même manière et ne prendra pas le même nombre de cases mémoires.

Pour avoir une correspondance parfaite entre une variable et son équivalent en mémoire, il nous faut donc connaître à la fois le type de la variable, et l'index de la première case mémoire où cette variable est contenue. Partant de cette remarque, nous pouvons construire un nouveau type de variable, appelé pointeur, qui va contenir les informations nécessaires pour retrouver en mémoire la valeur de notre variable n. Nous déclarons et définissons notre pointeur sur n par l'instruction suivante :

```
1 int* pn= &n;
```

Dans l'instruction précédente, nous déclarons une variable pn de type pointeur sur entier (int*), et nous la définissons en lui affectant l'adresse de n, adresse récupérée par l'opérateur &. Le symbole * est ici utilisé pour indiquer que pn n'est pas un entier mais un pointeur sur entier. Pour chaque type T "de base", nous avons donc un type correspondant qui est le type T*, et des variables de type T* que nous appelons pointeurs sur une instance de type T, ou, par abus

de langage, simplement pointeur sur T.

Bonnes habitudes 13 (Nom des pointeurs) *Prenez pour habitude de préfixer vos pointeurs par la lettre p, celà simplifie beaucoup la relecture du code.*

Nous pourrions de la même manière définir un pointeur sur double :

```
1 double b = 3.2;
2 double* pb = &b;
```

6.2 Déréférenciation

Lorsque nous disposons d'un pointeur, nous pouvons souhaiter récupérer la valeur pointée, c'est à dire le *déréférencer*. Pour celà, nous utilisons l'opérateur * –dit opérateur de déréférencement– pour accéder à la valeur stockée à l'adresse du pointeur :

```
1 double b = 3.2;
2 double* pb = &b;
3 double c = *pb;
```

Nous venons de voir deux utilisations distinctes du symbole * :

1. lors de la déclaration d'un pointeur, il spécifie que la variable pb par exemple est un pointeur sur double.
2. lors de la déréférenciation, c'est l'opérateur * qui associe à un pointeur la variable vers laquelle le pointeur renvoie.

Il s'agit donc d'une homonymie de l'opérateur *, à laquelle il faut prendre garde lorsque l'on débute.

6.3 Opérateurs & et *

Les valeurs de a et de b sont égales à la suite des instructions suivantes :

```
1 double a = 3.2;
2 double b = *(&a);
```

Les valeurs de pa et pb sont égales à la suite des instructions suivantes :

```
1 double a = 3.2;
2 double* pa = &a;
3 double* pb = &(*pa);
```

Les instructions suivantes sont fausses car les types ne correspondent pas, et le compilateur retournera une erreur :

```
1 int a = 3;
2 double* pa = &a;
```

6.3.1 Formalisation des pointeurs

Notons E l'ensemble des variables stockées en mémoire d'un programme fini, et P l'ensemble des pointeurs qui pourraient être créés et qui pointeraient sur une de ces variables. On peut définir sur P une relation d'équivalence $\sim$ (réflexive, symétrique et transitive) par $px \sim py$ si et seulement si *px = *py. L'espace quotient de P par $\sim$ est en bijection avec E, et les opérateurs induits $\overline{*}$ et $\overline{\&}$ induits sont alors des bijections entre E et $\overline{P}$, inverses à gauche et à droite l'un de l'autre.

6.4 Usage des pointeurs

Les pointeurs sont omniprésents en C++. Des librairies comme la STL tendent cependant à diminuer leur utilisation directe par l'utilisateur, et aujourd'hui, il serait presque possible d'écrire en C++ sans utiliser de pointeurs. En effet, il est possible dans la plupart des cas d'écrire du code "plus haut niveau" où l'usage des pointeurs est caché. Néanmoins, il est fondamental que vous compreniez bien les mécaniques sous-jacentes. Les pointeurs restent souvent indispensables (de manière implicite ou explicite) pour le polymorphisme, lorsque nous manipulons des instances volumineuses, dans le cas de gestion dynamique de mémoire, dans la construction de la plupart des Design Pattern, ...

Les chapitres suivants montreront de nombreux exemples d'utilisation de pointeurs.

If 10 years from now, when you are doing something quick and dirty, you suddenly visualize that I am looking over your shoulders and say to yourself : 'Dijkstra would not have liked this', well that would be enough immortality for me.

Edsger Dijkstra

7 Fonctions et Scope

7.1 Fonctions

Intéressons nous d'abord à la fonction qui calcule le carré d'un entier.

```
1  int Square(int x)
2  {
3      return x*x;
4  }
```

Listing 7.1 – Fonction qui calcule le carré d'un nombre.

Dans cette définition de fonction, nous distinguons les éléments suivants :

int Square : Nom de la fonction, qui renvoie un *int* (Entier)

int x : paramètre de la fonction, de type *int* (Entier)

return x*x : valeur de retour de la fonction

Nous constatons plusieurs choses :

– Une fonction s'écrit comme une fonction mathématique : elle prend en argument des paramètres, et renvoie une valeur, également typée [1].

1. La comparaison s'arrête là, et on ne saurait trop mettre en garde notre public venant de

– Le début et la fin de la fonction sont indiqués par des accolades ouvrantes et fermantes.
– Les lignes d'instructions sont terminées par un point-virgule (;).

7.1.1 Prototypes des fonctions

Une fonction est déclarée en C++ de la manière suivante :

```
TypeDeRetour Nom(typeParam1 nomParam1, typeParam2 nomParam2, ...)
{
    /* Code */
    return someValue;
}
```

Listing 7.2 – Déclaration d'une fonction

Comme pour les noms de variables, les noms des fonctions tiennent compte de la casse (c'est à dire des majuscules et des minuscules).

Bonnes habitudes 14 (Conventions de nommages) *Il est important - faute de quoi on finit par s'y perdre - de décider de conventions de nommage des variables et fonctions, et de s'y tenir. Deux styles sont couramment utilisés :*

```
    int monNomDeVariableTropLong;
```

et

```
    int mon_nom_de_variable_trop_long;
```

Peu importe celui que vous choisissez, l'important est de ne pas en changer au sein d'un même projet. Nous nous fixons le premier style dans la suite de ce document.

Dans le cas particulier où la fonction ne renvoie rien (une `sub` en VBA), son type de retour est `void`. Nous en verrons un exemple un peu plus tard.

---

math spé contre la tentation certes naturelle de résumer l'informatique à un appel de fonctions mathématiques, et les boucles for à des $\sum$. Nous y reviendrons amplement, notamment quand nous introduirons l'objet.

Bonnes habitudes 15 (Du bon usage des fonctions) *Une fonction doit remplir au plus un objectif. Il faut faire des fonctions courtes, dont l'intégralité du code tient de préférence sur un écran. Si vous vous retrouvez face à des monstres de 100 lignes de code ou plus, c'est très probablement que vous avez fait une erreur de design.*

Bonnes habitudes 16 (Les 80 colonnes) *Une bonne habitude consiste à s'interdire de dépasser la 80ème colonne. Celà permet d'avoir un code lisible même sur des petits écrans sans avoir à scroller horizontalement. Celà permet de pouvoir mettre en vis-à-vis deux fichiers sur des écrans plus larges. Celà permet enfin de se contraindre à écrire du code plus concis, avec des noms de variables plus concis.*

7.1.2 La fonction `main`

Lorsque vous créez un nouveau projet, nous pouvez choisir de créer une application console (.exe) ou une bibliothèque de fonctions (.dll). La bibliothèque de fonctions n'a vocation qu'à être appelée par un code extérieur.

Si vous créez une application console, il est nécessaire de préciser le point d'entrée de votre programme, c'est à dire la première fonction à appeler lorsque vous lancerez votre programme. Cette fonction porte par convention le nom de `main`. Cela a une implication importante, qui est qu'*il ne peut y avoir qu'une seule fonction* `main` dans votre programme. Réciproquement, en l'absence de fonction main dans une application console, **l'édition des liens échouera**.

La fonction main accepte plusieurs prototypes : son type de retour est généralement void, mais il peut être entier (on peut utiliser ce type de retour entier pour retourner un code d'erreur si l'exécution du code a déclenché des erreurs). La fonction main peut également prendre des arguments, auquel cas ils sont de la forme suivante :

```
1 int main(int argc, char* argv[])
2 {
3   return 0;
4 }
```

Ces arguments, spécifiés par l'utilisateur au lancement du programme en mode ligne de commande, permettent d'intéragir avec le

programme. Par exemple, lorsque sous Linux nous utilisons le programme qui permet de lister tous les fichiers du répertoire courant, nous appelons le programme ls dans l'invite de commande, mais nous pouvons également utiliser ls -l, où la chaîne de caractères "-l" est passée au programme via la fonction main comme argument.

Sauf si vous souhaitez qu'il en soit autrement, nous vous conseillons d'adopter le prototype le plus simple [2] :

```
1 void main()
2 {
3   //code
4 }
```

7.2 Déclaration et définition de fonctions

Nous allons utiliser la fonction Square définie ci-dessus dans notre fonction main. Dans un projet vide, nous ajoutons le fichier main.cpp reproduit ci-dessous et compilons :

```
1
2 //Main.cpp
3
4 int Square(int x)
5 {
6   return x*x;
7 }
8
9 void main()
10 {
11   int a = 3;
12   int b = Square(a);
13 }
```

Lorsque nous inversons l'ordre des définitions des deux méthodes, nous obtenons une erreur de compilation :

2. attention cependant, ce prototype du main est parfois incompatible avec certaines librairies comme SDL...

```
1
2  //Main.cpp
3
4  void main()
5  {
6    int a = 3;
7    int b = Square(a);
8  }
9
10 int Square(int x)
11 {
12   return x*x;
13 }
```

erreur C3861 : "Square" : identificateur introuvable.

Le compilateur commence par compiler la fonction main, et il y trouve un appel à la fonction Square. Comme au moment où il compile main, il ne "connait" pas encore la fonction Square, il renvoie une erreur de compilation. Pour informer le compilateur de l'existence de la fonction Square, il est nécessaire de la déclarer avant la fonction main, comme dans l'exemple suivant :

```
1
2  //Main.cpp
3
4  int Square(int); //declaration de Square
5
6  void main()
7  {
8    int a = 3;
9    int b = Square(a);
10 }
11
12 int Square(int x) //definition de Square
13 {
14   return x*x;
15 }
```

Pour ne pas nuire à la lisibilité du code, nous allons isoler la déclaration de la fonction Square dans un fichier header (.h), que nous ajoutons à notre projet. Nous obtenons donc dans main.h :

```
1
2  //Main.h
3
4  int Square(int);
```

et dans main.cpp :

```
1
2  //Main.cpp
3  #include "main.h"
4
5  void main()
6  {
7    int a = 3;
8    int b = Square(a);
9  }
10
11 int Square(int x)
12 {
13   return x*x;
14 }
```

Nous retrouvons alors la séparation déclarations dans les fichiers headers et définitions dans les fichiers .cpp comme annoncée dans le chapitre interpréteur/compilateur.

Le lecteur déjà initié au C# ou au Java se demandera naturellement pourquoi il est nécessaire de déclarer les méthodes avant de les définir. C'est principalement pour des raisons historiques ; lorsque le C a été créé, et même lorsque le C++ a été créé, les machines ne possédaient pas assez de RAM pour pouvoir stocker tous les fichiers sources d'un programme en mémoire en même temps. Comme il est possible de compiler chaque fichier source séparément à condition d'avoir pour chaque source les déclarations nécessaires, ces langages ont décidé de séparer déclaration et définition. Ainsi en C++, et comme nous l'avons vu dans le chapitre sur la compilation, chaque fichier source est compilé indépendamment des autres. Il y a d'autres raisons qui peuvent être avancées, notamment l'accélération des temps de compilation, le fait de pouvoir faire référence à du code dont vous n'avez que la déclaration et pas la définition (et ainsi de pouvoir vendre des bibliothèques compilées sans dévoiler le code source), etc. Mais la vraie raison est qu'on ne pouvait faire autrement à l'époque.

7.3 Premier modèle mémoire, Stack et Scope des variables

Nous donnons à présent une première modélisation naïve de la mémoire. Puisque l'exécution d'un programme se décompose en l'exécution de multiples fonctions, l'environnement doit garder trace de l'enchaînement actuel des fonctions, ainsi que de l'état des données relatives à chacune de ces fonctions.

Lorsqu'une fonction est appelée, l'environnement lui construit un espace mémoire dédié, appelé *"Stack Frame"*, dans lequel sont stockés : les arguments de la fonction, les variables locales de la fonction, l'adresse de la ligne de code à exécuter lorsque l'appel à cette fonction sera terminé. Toutes ces données n'existent que pendant le temps d'exécution de la fonction : nous disons que la durée de vie (ou le *"Scope"*) de ces données sont les mêmes que ceux de la fonction.

Lorsqu'une fonction g appelée par une fonction f est terminée, l'environnement revient à la fonction f et restaure son état de telle sorte que les instructions de f qui faisaient suite à l'appel de g peuvent être exécutées.

Pour stocker toutes les données nécessaires à la bonne exécution d'une application, l'environnement fait donc appel à une structure de données spécialisée appelée : la *Stack*. Cette structure s'organise comme une pile d'assiettes sur laquelle les assiettes peuvent être empilées et dépilées selon l'ordre : la dernière arrivée est la première sortie (*"Last In First Out, ou LIFO"*).

A chaque fois qu'une fonction est appelée, une assiette (Stack Frame) est ajoutée sur la Stack, contenant toutes les informations et la mémoire nécessaire à la bonne exécution de la fonction. Dès que cette fonction est terminée, toutes ses variables deviennent inutiles, sa Stack Frame est alors ôtée du haut de la Stack, et l'exécution revient à la ligne de code suivante de la fonction appelante. La Stack Frame tout en haut de la Stack est donc celle correspondant à la fonction actuellement exécutée.

Considérons l'exemple suivant :

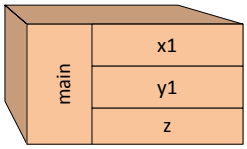

FIGURE 7.1 – Etat de la stack à l'entrée du main

```
1  double SquareError(double x, double y)
2  {
3      double s = x-y;
4      double result = s*s;
5      return result;
6
7  }
8
9  void main()
10 {
11     double x1 = 2.3;
12     double y1 = 3.2;
13     double z = SquareError(x1, y1);
14 }
```

Lorsque la fonction `main` est appelée, une première Stack Frame est ajoutée à la Stack, qui était vide. Nous nous retrouvons alors avec une mémoire dans l'état décrit par le graphique 7.1.

Lorsque la fonction `SquareError` est appelée, une Stack Frame est ajoutée en haut de la Stack, Frame contenant les variables locales `s` et `result`, les arguments `x` et `y`, ainsi que la ligne de code dans la fonction appelante (`main`) à exécuter lorsque SquareError sera terminée. Les deux Stack Frames sont schématisées sur le graphe 7.2.

Enfin, après l'exécution de la fonction `SquareError`, tout le contexte de la fonction devient caduque, et la Stack Frame correspondante est dépilée. En revenant dans la fonction `main`, nous nous retrouvons à nouveau dans l'état décrit par le graphe 7.1.

Dans cet exemple, les variables `x1` et `y1` sont des variables locales de la fonction `main`, tout comme les variables `x` et `y` sont des

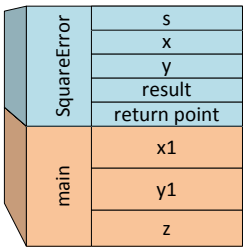

FIGURE 7.2 – Etat de la stack à l'entrée dans SquareError

arguments de la fonction `SquareError`. Dans les deux cas, ces variables sont des variables muettes, c'est à dire qu'elles servent à définir le sens des fonctions `SquareError` et `main`, mais qu'elles ne possèdent pas de sens en dehors, tout comme la variable k sert à définir la valeur de la fonction f dans $f(n) = \sum_{k=1}^{n} \frac{1}{k^2}$. Leur scope ne se rencontrant pas, il est donc possible d'utiliser les mêmes noms de variables dans chacune des fonctions sans que celà n'interfère avec le sens du code. Ainsi, le code suivant donne exactement les mêmes résultats :

```
1  double SquareError(double x, double y)
2  {
3      double s = x-y;
4      double result = s*s;
5      return result;
6
7  }
8
9  void main()
10 {
11     double x = 2.3;
12     double y = 3.2;
13     double z = SquareError(x, y);
14 }
```

Considérons un autre exemple :

```
1
2  #include <math.h>
3
4  double SquareError(double x, double y)
5  {
6    double s = x-y;
7    return s*s;
8  }
9
10 double ManhattanError(double x, double y)
11 {
12   double s = x-y;
13     double result = abs(s);
14   return result;
15 }
16
17 void main()
18 {
19   double a = 2.0;
20   double b = 1;
21   double c = 3.0;
22   double d = SquareError(a,b);
23   double e = ManhattanError(a,b);
24 }
```

Dans cet exemple, les valeurs `a,b,c,d,e` ont pour scope la fonction main. Une variable `s` est déclarée dans `SquareError`, et elle est détruite à la fin de la fonction. L'identifiant `s` est "recyclé" dans la fonction ManhattanError, et la variable `s` est également détruite à la fin de cette fonction. Le double usage de l'identifiant `s` ne pose pas de problème au compilateur, qui comprend que la variable est "muette" dans les deux cas.

A l'entrée de la fonction main, l'état de la Stack est représenté dans le graphique 7.3. Lorsque la fonction `SquareError` est appelée, la Stack se trouve dans l'état représenté par le graphique 7.4. Lorsque la fonction `SquareError` est achevée, l'état de la Stack revient à celui du graphique 7.3. Ensuite, la fonction `ManhattanError` est appelée, et la Stack passe alors dans l'état représenté dans le graphique 7.5. Enfin, juste avant la fin de la fonction `ManhattanError`, et avant de retourner dans le `main`, il est fait appel à la fonction `abs` qui vient ajouter alors une Stack Frame dans la Stack[3], comme il

3. Pour être tout à fait exact, le compilateur est capable, dans de nombreux cas comme celui-ci, de déterminer qu'il n'est pas nécessaire de garder trace de la Stack Frame de `ManhattanError` lorsque nous sommes dans la fonction `abs`, et qu'il peut optimiser les appels pour ne garder que les Stacks Frames de `abs` et `main` : c'est le principe du "Tail Call Optimization".

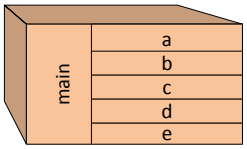

FIGURE 7.3 – Etat de la Stack à l'entrée du main

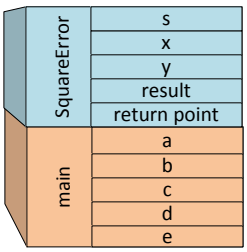

FIGURE 7.4 – Etat de la Stack à l'entrée dans SquareError

est présenté dans le graphe 7.6.

Notez que pour utiliser la fonction valeur absolue, fonction standard du langage, nous devons utiliser la librairie "math", dont les méthodes sont déclarées dans le fichier header `math.h` qui est stocké dans le répertoire de Visual Studio. Pour utiliser cette librairie, nous utilisons donc un include, mais substituons les symboles < et > en lieu et place des guillemets "" pour spécifier au compilateur que ce fichier `math.h` appartient à la bibliothèque standard et n'est pas un fichier de votre propre cru.

7.4 Passage d'arguments

Considérons une méthode Increment qui prenne un entier et l'incrémente. Nous voulons (à titre purement illustratif) que cette méthode ne retourne pas la valeur incrémentée, mais qu'elle modifie directement la valeur de l'argument qui lui est passé.

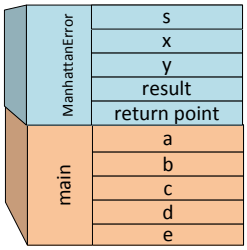

FIGURE 7.5 – Etat de la Stack à l'entrée dans ManhattanError

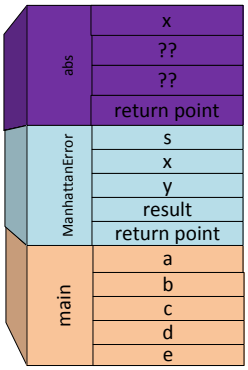

FIGURE 7.6 – Etat de la Stack à l'entrée dans abs

7.4.1 Passage d'argument par valeur

Pour une variable `int a`, les instructions `a=a+1;` et `a++` sont équivalentes. Une première idée serait donc d'implémententer la fonction Increment comme suit :

```
1  void Increment1(unsigned int a)
2  {
3      a++;
4  }
5
6  void main()
7  {
8      unsigned int i = 2;
9      Increment1(i);
10     unsigned int b = i;
11 }
```

Si nous exécutons ce code[4], nous constatons que lorsque nous appelons cette fonction dans notre main, nous n'obtenons pas les résultats attendus, la variable b prenant la valeur 2. La raison à cela est que c'est une *copie* de `i` qui est passée à la fonction (copie désignée par le symbole a), et non la variable i "elle-même" : le compilateur C++ va par défaut réaliser des copies des arguments que nous donnons à une fonction. Dans l'exemple précédent, une variable i est créée dans la fonction main. Lorsque nous appelons la fonction Increment1, une variable a est créée et le compilateur lui affecte la valeur de i. Cette valeur a est incrémentée dans la fonction Increment1, puis est détruite à la fin de la fonction. Lorsque nous retournons dans la fonction main après exécution de la fonction Increment1, nous avons donc incrémenté une variable a, que nous avons détruite à la fin de la fonction Increment1 et nous disposons maintenant de la variable i dont la valeur n'a jamais été modifiée.

Exemple d'allocation mémoire avant d'entrer dans la fonction Increment1 :

Case mémoire	0	1	2	3	4	5	6	7	8	9	10	11	12
Nom			i	i	i	i							
Valeur			0100...	000...	00...	000...							

En entrant dans la fonction Increment1 :

4. Il est fortement recommandé de le faire pour vous en persuader.

Case mémoire	0	1	2	3	4	5	6	7	8	9	10
Nom			i	i	i	i		a	a	a	a
Valeur			0100...	000...	00...	000..		0100...	000...	00...	000..

Juste avant de sortir de la fonction Increment1 :

Case mémoire	0	1	2	3	4	5	6	7	8	9	10
Nom			i	i	i	i		a	a	a	a
Valeur			0100...	000...	00...	000..		1100...	000...	00...	000..

De retour dans le main après appel à Increment1 :

Case mémoire	0	1	2	3	4	5	6	7	8	9	10
Nom			i	i	i	i					
Valeur			0100...	000...	00...	000..					

Il nous faut donc spécifier en C++ au compilateur que nous ne voulons pas qu'il travaille avec une copie de la variable i, mais bien avec la variable i elle-même.

7.4.2 Passage d'argument par pointeur

Reprenons l'exemple précédent avec des pointeurs :

```
1  void Increment2(int* p)
2  {
3      (*p)++;
4  }
5
6  void main()
7  {
8      int i = 2;
9      int* pi = &n;
10     Increment2(pi);
11     int b = i;
12 }
```

Cette fois-ci, nous avons donné en argument non pas la variable i, mais un pointeur vers cette variable i via la variable pointeur pi. Le compilateur va créer une copie de la variable pi, p, qui prendra la même valeur que pi, c'est à dire qui pointera sur i également. Lorsque nous exécutons le code de la fonction Increment2, nous accédons donc bien à l'adresse mémoire de la variable i, qui est effectivement modifiée.

7.4.3 Passage d'argument par référence

Le C++ (à la différence du C), propose un sucre syntaxique pour abstraire la manipulation de pointeur : c'est le passage par référence. Plus précisément, il existe un équivalent du `ByRef` de Visual Basic, qui est le signe &. Lorsque nous passons une valeur par référence, nous adoptons la syntaxe du passage par valeur (au symbole & prêt) mais le compilateur va interpréter ceci comme un passage par pointeur. En voici un exemple :

```
1  void Increment3(int& a)
2  {
3      a++;
4  }
5  void main()
6  {
7      int a = 2;
8      Increment3(a);
9      int b = a;
10 }
```

Nous obtenons bien une valeur de 3 pour la variable `b` à la fin du main.

Il est important de noter que l'emploi du signe & à côté d'une déclaration de variable n'a *pas* le même sens que celui de l'opérateur qui récupère l'adresse mémoire d'un objet. Il y a donc 2 sens différents à la fois pour les opérateurs & et *.

Il y a deux raisons pour lesquelles nous pouvons souhaiter passer un argument par référence :

- La première, comme nous venons de le voir, pour modifier réellement la valeur de l'argument en dehors de la fonction appelée.
- La deuxième, pour éviter une copie (copie qui aurait des conséquences en termes de coût et de sens).

Dans le deuxième cas, puisque c'est vraiment l'objet et non pas une copie de lui-même qui est passé à la fonction, la fonction pourrait donc modifier par inadvertance l'objet donné en argument. Nous souhaiterions parfois pouvoir interdire la modification de l'objet manipulé, par mesure de sécurité, lorsque celui-ci est donné en argument par référence. Le C++ permet de faire ceci en précisant au niveau de l'argument de la fonction que celui-ci est constant (immutable) et qu'il ne devra pas être modifié :

```
void WidgetFunction(const int& a)
{
    //Do something
}
```

Dans notre cas d'incrémentation, nous ne souhaitons bien évidemment pas que la fonction Increment3 utilise un argument immutable comme dans l'exemple précédent, puisque précisément le but de la fonction est de modifier l'argument. Si cependant nous nous y essayions, le compilateur retournerait une erreur, car il est capable de détecter statiquement (c'est à dire à la compilation et non à l'exécution) que le code essaye de modifier une variable définie comme immutable.

7.5 Effets de bord

Le fait qu'une fonction agisse non pas par sa valeur de retour mais par le fait qu'elle modifie un de ses arguments, comme c'était le cas dans notre méthode Increment3 au dessus, est appelé effet de bord. En règle générale, les effets de bord sont peu intuitifs et doivent être utilisés avec circonspection.

Bonnes habitudes 17 (Méthodes à effet de bord) *Lorsqu'une méthode agit par effet de bord, il est de bon goût que cette méthode n'ait pas de type de retour. Ainsi, il est beaucoup plus aisé au relecteur du code de deviner que la méthode va agir par effet de bord. Inversement, une méthode qui a un type de retour n'est pas sensée avoir des effets de bord sur ses arguments.*

— Knowing C is a prerequisite for learning C++, right ?

Wrong. The common subset of C and C++ is easier to learn than C. There will be less type errors to catch manually (the C++ type system is stricter and more expressive), fewer tricks to learn (C++ allows you to express more things without circumlocution), and better libraries available. The best initial subset of C++ to learn is not "all of C". Bjarne Stroustrup, FAQ sur sa page personnelle

8

Autres éléments de syntaxe

8.1 Boucles et tests de condition

8.1.1 Tests

Tests simples

Un test est un ensemble d'instructions qui ne doivent Ãłtre exÃľcutÃľes que conditionnellement au fait qu'une certaine condition soit ou non vÃľrifiÃľe.

Comme dans les autres langages, nous voulons pouvoir faire des tests sur les valeurs des variables. La syntaxe d'un test ainsi qu'un exemple peuvent être lus ci-dessous :

```

if (condition)
{
    /*Code*/
}
else if(condition)
{
    /*Code*/
}
else
{
...
}
```

Listing 8.1 – Syntaxe d'un test

```

if (0 == x)
{
    x = x + 1;
}
else if (1 == x)
{
   x = x * 2;
}
else
{
   x = x + 3;
}
```

Listing 8.2 – Exemple de test

Nous pouvons également noter dans l'exemple l'utilisation de == pour effectuer un test d'égalité. Le test de différence se fait à l'aide de l'opérateur !=.

Le code suivant, bien que valide syntaxiquement, ne fait pas ce qui est attendu :

```
1 if (x = 0)
2 {
3     x = x + 1;
4 }
```

Listing 8.3 – Une erreur classique

En effet, il manque un symbole "=" dans le test du if. Le code en question va donc mettre la valeur 0 dans la variable x, avec des résultats imprévisibles pour la suite.

Bonnes habitudes 18 (Test) *Afin d'éviter l'erreur classique du listing 8.3, une bonne habitude est d'inverser les paramètres du test :*

```
1 if (0 == x)
2 {
3   x = x + 1;
4 }
```

En lieu et place de :

```
1 if (x == 0)
2 {
3   x = x + 1;
4 }
```

Enfin, nous remarquons à nouveau l'emploi des accolades pour signifier le début et la fin d'un bloc logique (comme en Java). Les habitués de Python noteront que l'indentation est libre - contrairement à Python -, ce qui justifie l'emploi des accolades [1].

1. Une petite précision est Ãă apporter : dans le cas où le code conditionnel Ãă exécuter fait une seule ligne, on peut se passer des accolades. Cependant, cette pratique est dangereuse (on rajoute une ligne après et l'on oublie de rajouter les accolades) et donc à éviter au moins dans un premier temps.

Bonnes habitudes 19 (Indentation) *Il est* très *important d'indenter son code correctement, faute de quoi il devient rapidement illisible.*

Tests en séries

Il arrive parfois que l'on cherche à effectuer une série de tests, comme sur le listing 8.4.

```
#include <iostream.h>


int main()
{
  char c;

  cout << "Veuillez entrer un caractere" << endl;
  cin >> c;

  if (c == 'A')
    cout << "Lettre A";
  else if (c == 'B')
    cout << "Lettre B";
  else if (c == 'C')
    cout << "Lettre C";
  else
    cout << "Autre lettre;

  return 0;
}
```

Listing 8.4 – Tests en série

Ecrire la série de test de cette manière n'est pas très élégante. Le langage C++ fournit une construction particulière, appelée `switch`. La syntaxe est la suivante :

```

switch (variable)
{
    case valeur1 :
      /*code*/
     break;
    case valeur2 :
      /*code*/
     break;
    /*autres cas*/
    default:
       /*code*/
      break;
}
```

Listing 8.5 – Syntaxe d'un switch

```
switch(c)
{
  case 'A':
    cout << ``lettre A'';
    break;
  case 'B':
    cout << ``lettre B'';
    break;
  default:
    cout <<``Autre lettre'';
       break;
}
```

Listing 8.6 – Exemple de tests multiples

- Le mot clé `switch` indique que nous allons opérer une séparation des cas.
- Chaque cas est ensuite traité à l'aide du mot clé `case`.
- Les cas non prévus explicitement sont traités par le cas `default`.
- Le mot clé `break` sert à indiquer la fin du cas. Si ce mot clé est absent, le cas en dessous sera *également exécuté*.

Le code du listing 8.4 devient donc :

```
#include <iostream.h>

int main()
{
  char c;

  cout << "Veuillez entrer un caractere" << endl;
  cin >> c;

  switch (c)
  {
    case 'A':
      cout << "Lettre A";
      break;

    case 'B':
      cout << "Lettre B";
      break;

    case 'C':
      cout << "Lettre C";
      break;

    default:
      cout << "Autre lettre";
      break;
  }
  return 0;
}
```

Listing 8.7 – Tests en série

Le code obtenu présente l'avantage d'être plus clair et élégant qu'avant.

8.1.2 Boucles

La boucle `for`

Fréquemment, nous nous retrouvons dans une situation où nous souhaitons parcourir un tableau ou une liste, ou encore parcourir le nombres de 1 à 100 pour pouvoir les afficher. On retrouve en C++ comme ailleurs la boucle `for`, dont la forme générale est :

```
for(variable1,variable2,...= valeurs de depart;
    condition de terminaison;
    operation sur les variables)
{
    /*Code*/
}
```

Listing 8.8 – Syntaxe d'une boucle `for`

Par exemple, pour énumérer et afficher les nombres de 1 à 100 nous écririons le code suivant :

```

for(int i = 1 ; i <= 100; i++) /*i++ signifie i=i+1*/
{
    cout << i << endl;
    //Note : endl signifie retour a la ligne
}
```

Listing 8.9 – Exemple de boucle `for`

La boucle `while`

L'autre type de boucle dont nous pouvons avoir besoin est une boucle signifiant *tant que*, que nous exprimons à l'aide du mot clé `while`. La condition est testée *avant* le bloc de code.

```
while (condition)
{
    /*code*/
}
```

L'affichage des nombres de 1 à 100 s'écrirait donc :

```
int i=1; /*on peut initialiser une variable*/

while ( i <= 100 )
{
    cout << i << endl;
    i++;
}
```

Ce type de boucle présente également une variante, dans laquelle la condition de sortie se trouve à la fin :

```
do
{
  /*code*/
}
while(condition);
```

Le listing 8.1.2 devient donc :

```
int i=0; /*on peut initialiser une variable*/

do
{
    i++;
    cout << i << endl;
}
while ( i < 100 )
```

Dans cette variante, l'intérieur des brackets est exécuté au moins une fois, quoi qu'il arrive.

`break` et `continue`

Nous avons parfois besoin d'interrompre l'exécution d'une boucle, ou de sauter une étape lors de son exécution. C'est le rôle des mots clés `break` et `continue`.

`break` sert à interrompre l'exécution d'une boucle, comme dans l'exemple suivant :

```
#include <iostream.h>



int main()
{
  for(int i = 0; i < 10 ; i++)
  {
    cout << i << endl;
    if (i>3)
      break;
  }
  return 0;
}
```

Listing 8.10 – examplebreak.cpp

`continue` sert à sauter l'étape courante de la boucle. Par exemple, pour afficher les nombres pairs entre 1 et 10, nous pourrions écrire :

```
#include <iostream.h>


int main()
{
  for (int i = 0; i<10; i++)
  {
    if (i % 2 == 1)
    {
      continue;
    }
    cout << i << endl;
  }
  return 0;
}
```

Listing 8.11 – examplecontinue.cpp

Deuxième partie

Programmation orientée objet

- - Ah, voilà enfin le roi de la classe ! L'homme trop bien sapé, Abitbol ! Alors comme ça tu as été élu l'homme le plus classe du monde ! Laisse-moi rire ! Style le grand playboy des fonds marins, [...]
- [...] Mais c'est pas ça qu'on appelle la classe. Je te dis ça en qualité d'homme le plus classe du monde.

- Eh je t'arrête tout de suite. La classe, c'est d'être chic dans sa manière de s'habiller. Rien de tel que d'aller chez Azzedine Alaia même de s'acheter des sous-pulls chez Yohji Yamamoto !

- Excuse-moi de te dire ça mon pauvre José, mais tu confonds un peu tout. Tu fais un amalgame entre la coquetterie et la classe. Tu es fou. Tu dépenses tout ton argent dans les habits et accessoires de mode mais tu es ridicule. Enfin si ça te plait... C'est toi qui les portes. Mais moi, si tu veux mon opinion, ça fait un peu... has been.

La classe américaine, José et Georges Abitbol

9 Introduction à l'objet : les classes

La complexité d'un code n'est pas linéaire avec sa taille : plus un projet est volumineux, plus l'ajout marginal de code s'avère complexe. Il devient rapidement (au delà de quelques centaines de lignes de code) nécessaire d'architecturer le code pour en faciliter la lecture, et donc la maintenance et le debuggage. Cette problématique est devenue centrale. En effet, depuis 30 ans la taille des projets a explosé : pour maintenir un projet, il faut donc penser son architecture pour l'organiser comme une synergie d'objets indépendants incarnant chacun un rôle/une fonctionnalité.

L'idée fondamentale, c'est d'avoir des entités logiques riches, le plus distinctes, réduites et découplées possible. Dans ce chapitre, nous introduisons ces entités logiques riches, appelées classes. En règle générale, une classe regroupe un ensemble de données et de fonctions. L'architecture générale d'une classe est de ne rendre disponible qu'une petite partie de ses fonctions (appelées méthodes) et données, celles réellement utiles de l'extérieur, et de cacher pour l'extérieur la majorité de son implémentation. Cette conception permet de découpler les différentes complexités d'un programme et de simplifier la lecture du code : pour comprendre comment utiliser une classe, il suffit de comprendre ce que font les méthodes "publiques" et non pas de comprendre tous les mécanismes sous-jacents.

La notion de classe est si centrale qu'on désigne souvent les langages qui permettent ce genre d'entités comme des langages "orien-

tés objet". Jusqu'à présent, nous avions présenté nos programmes sous une forme procédurale, c'est à dire qu'une méthode main appelait d'autres fonctions qui à leur tour en appelaient des autres, ... A présent, nous créerons des objets, instances de types plus complexes, où chaque type représentera un ensemble de méthodes et de données formant une unité logique. Les actions et intéractions de ces objets définissent le nouveau flux d'exécution.

A titre illustratif, nous utiliserons dans tout ce chapitre l'exemple d'une classe `Accumulator` dont la fonction sera de calculer la moyenne et la variance empirique de valeurs réelles.

9.1 Déclaration des classes

Commençons par déclarer cette classe `Accumulator`. Cette classe contiendra plusieurs variables, appelées des *champs*. Voici la déclaration de cette classe dans le fichier Accumulator.h correspondant :

```
1  //Accumulator.h
2  #ifndef ACCUMULATOR_H
3  #define ACCUMULATOR_H
4
5  #include <string.h>
6
7  class Accumulator
8  {
9  public:
10   int n;
11   double xSum;
12   double xSquareSum;
13 }; //ne pas oublier le ; ici !
14
15 #endif
```

Les champs `n`, `xSum` et `xSquareSum` sont intrinsèques à la classe. Chaque instance, c'est à dire chaque exemplaire de la classe `Accumulator` possèdera ces attributs.

Une fois notre classe déclarée, nous pouvons alors déclarer des instances de ce type ; Nous déclarons/définissons donc un `Accumulator` dans notre main de la manière suivante :

```
//main.cpp
#include "Accumulator.h"

void main()
{
  Accumulator acc;
}
```

Nous pouvons écrire des méthodes reliées à la classe `Accumulator`. Pour celà, nous ajoutons dans la déclaration de notre classe les prototypes des méthodes que nous souhaitons ajouter :

```
//Accumulator.h
#ifndef ACCUMULATOR_H
#define ACCUMULATOR_H

class Accumulator
{
    public:
      int n;
      double xSum;
      double xSquareSum;

      void Add(double);
};

#endif
```

Pour définir la méthode correspondante, nous procédons comme précédemment, mais précédons le nom de la méthode du nom de la classe suivi de " : :", afin de signifier au compilateur que la fonction est attachée à cette classe (nous parlons alors de *fonction membre* ou *méthode*).

```
//Accumulator.cpp

#include "Accumulator.h"

void Accumulator::Add(double x)
{
  n++;
    xSum += x;
    xSquareSum += x*x;
}
```

Lorsque nous souhaitons utiliser une méthode ou un champ d'une classe, la syntaxe que nous utiliserons sera la suivante :

```
instance.champ = ...
instance.Method()
```

En reprenant le code précédent, nous souhaitons maintenant calculer la moyenne des 10 premiers entiers.

Nous adaptons donc notre fonction main :

```
void main()
{
    //construction of object acc
    Accumulator acc;

    // Initialisation of the object
    acc.n=0;
    acc.xSum =0;
    acc.xSumQuare=0;

    for (int i = 0 ; i < 10;i++)
    {
        acc.Add( (double) i);
    }

    double mean = acc.xSum / acc.n;
}
```

9.2 Initialisation et constructeurs

9.2.1 Une première méthode d'initialisation

L'initialisation proposée dans la section précédente présente plusieurs inconvénients :

- elle est très verbeuse, puisqu'il faut une ligne de code par champ renseigné.
- il est possible (facile même) d'oublier un champ, et d'avoir alors un champ non initialisé, ce qui posera très vraissemblablement problème par la suite.
- nous n'avons pas d'initialisation par défaut de certains champs.

Créons donc une méthode d'initialisation de notre classe Accumulator en ajoutant une méthode Initialize. Nous updatons d'abord le fichier header par le prototype de cette méthode :

```
1  //User.h
2  #ifndef ACCUMULATOR_H
3  #define ACCUMULATOR_H
4
5  class Accumulator
6  {
7  public:
8    int n;
9    double xSum;
10   double xSquareSum;
11
12   void Add(double);
13   void Initialize(int, double, double);
14 };
15
16 #endif
```

De même, nous ajoutons le code suivant dans le fichier .cpp :

```
1  void Accumulator::Initialize(int nInit, double xSumInit, double
       xSquareInit)
2  {
3    n = nInit;
4    xSum = xSumInit;
5    xSquareSum = xSquareInit;
6  }
```

Il nous suffit alors de transformer notre main en :

```
void main()
{
    //construction of object acc
  Accumulator acc;

    // Initialisation of the object
  acc.Initialize(0,0,0);

    for (int i = 0 ; i < 10;i++)
    {
        acc.Add( (double) i);
    }

    double mean = acc.xSum / acc.n;
}
```

9.2.2 Constructeurs

Nous avons résolu une partie des problèmes énumérés précédemment et avons donc au final un type `Accumulator`, avec des *propriétés (champs ou fonctions membres)* bien définies. Cependant, un problème persiste : il faut obligatoirement appeler la méthode `Initialize`. C'est assez inélégant, car naturellement on voudrait que notre instance `acc`, soit créée "complètement" dès qu'elle est définie. Par exemple, au moyen d'un code ressemblant au listing ci-dessous :

```
Accumulator acc(0,0,0);
/* Initialize serait alors appelée automatiquement*/
```

Pour obtenir ce résultat en C++, il suffit de déclarer des fonctions spéciales, appelées *constructeurs*. Le constructeur est une méthode ayant comme nom le nom de l'objet, *sans type de retour*. Ajoutons donc un constructeur dans notre classe, qui remplacera notre fonction Initialize :

```
//Accumulator.h
#ifndef ACCUMULATOR_H
#define ACCUMULATOR_H

class Accumulator
{
public:
  int n;
  double xSum;
  double xSquareSum;

  Accumulator(int, double, double); // constructor
  void Add(double);
  void Initialize(int, double, double);
};
#endif
```

et le constructeur est défini dans Accumulator.cpp de la même manière que l'était la méthode Initialize :

```
Accumulator::Accumulator(int nInit, double xSumInit, double xSquareInit)
{
  n = nInit;
  xSum = xSumInit;
  xSquareSum = xSquareInit;
}
```

Notez que la définition du constructeur n'est précédée d'aucun type, même void.

Notre main devient donc :

```
void main()
{
  Accumulator acc(0,0,0);

    for (int i = 0 ; i < 10;i++)
    {
        acc.Add( (double) i);
    }

    double mean = acc.xSum / acc.n;
}
```

Notez que le code suivant ne compilera plus :

```
void main()
{
  Accumulator acc; //Does not work anymore since the constructor expects 3
      arguments.
  acc.Initialize(0,0,0);

    for (int i = 0 ; i < 10;i++)
    {
        acc.Add( (double) i);
    }

    double mean = acc.xSum / acc.n;
}
```

En effet, nous avons donné un seul prototype pour le constructeur : celui ci prend nécessairement 3 arguments, il n'est donc plus possible de construire une instance sans les préciser. Nous pourrions vouloir modifier cette situation, puisque dans la majorité des cas, les trois paramètres seront 0, et il semble un peu verbeux de devoir les respécifier impérativement. Nous pourrions par exemple *surcharger* le constructeur avec un autre prototype, c'est à dire ajouter un deuxième constructeur, de même nom, mais avec des arguments différents :

```
//Accumulator.h
#ifndef ACCUMULATOR_H
#define ACCUMULATOR_H

class Accumulator
{
public:
  int n;
  double xSum;
  double xSquareSum;

  Accumulator(int, double, double);
  Accumulator();
  ~Accumulator(void);
  void Add(double);
  void Initialize(int, double, double);
};
#endif
```

```
Accumulator::Accumulator(int nInit, double xSumInit, double xSquareInit)
{
  n = nInit;
  xSum = xSumInit;
  xSquareSum = xSquareInit;
}

Accumulator::Accumulator()
{
  n = 0;
  xSum = 0;
  xSquareSum = 0;
}
```

9.2.3 liste d'initialisation

A la construction d'une instance, il est possible de remplir les différents champs de la classe d'une manière différente, via la *liste d'initialisation*. Cette technique consiste à intercaler dans la définition du constructeur entre son prototype et les accolades { et } le nom de chaque champ, puis entre parenthèses la valeur qui doit lui être assignée. Ainsi, dans notre exemple précédent, celà reviendrait à :

```
Accumulator::Accumulator(int nInit, double xSumInit, double xSquareInit)
    : n(nInit), xSum(xSumInit), xSquareSum(xSquareInit)
{
}
```

Si les deux formes d'initialisation amènent en règle générale au même résultat, ce n'est cependant pas toujours le cas. Vous ne pouvez pas vous tromper en utilisant la liste d'initialisation, dans le sens où celle-ci est plus précise et générale que l'initialisation dans le constructeur, mais elle peut sembler aussi plus verbeuse. A votre niveau, vous pouvez utiliser indiférremment l'un ou l'autre.

9.2.4 Constructeur par défaut

Lorsque vous ne spécifiez pas de constructeur dans une classe, Visual Studio en crée un implicitement, sans argument et sans logique interne. Ainsi, ne pas mettre de constructeur dans la classe Accumulator revient à écrire :

```
//Accumulator.h
#ifndef ACCUMULATOR_H
#define ACCUMULATOR_H

class Accumulator
{
public:
  int n;
  double xSum;
  double xSquareSum;

  Accumulator();
  void Add(double);
  void Initialize(int, double, double);
};
#endif
```

```
Accumulator::Accumulator()
{
}
```

9.2.5 Destructeur

De la même manière qu'il existe une fonction spéciale pour construire des instances correctement initialisées, il existe une fonction spéciale, appelée destructeur, systématiquement appelée par l'environnement à la destruction de l'objet, par exemple quand son scope se termine. Pour les objets que nous avons considérés jusqu'à présent, le destructeur n'a besoin de rien faire de spécial. Comme pour le constructeur, il a un nom particulier : c'est le même que la classe, avec le symbole "∼" en préfixe :

```
//Accumulator.h
#ifndef ACCUMULATOR_H
#define ACCUMULATOR_H

class Accumulator
{
public:
  int n;
  double xSum;
  double xSquareSum;

  Accumulator(int, double, double);
  Accumulator();
  ~Accumulator(void); //Destructor
  void Add(double);
  void Initialize(int, double, double);
};

#endif
```

Et le fichier Accumulator.cpp est updaté avec la définition du destructeur :

```
Accumulator::~Accumulator(void)
{
}
```

Nous reviendrons plus en détail dans les chapitres suivants quant à l'usage des destructeurs.

9.3 Encapsulation

9.3.1 Notions de champs publics et champs privés

Dans la déclaration de notre classe Accumulator, nous avons fait précédé la déclaration des champs et des fonctions membres par le mot clef *public*. Par ce mot clef, nous rendons tous les champs et toutes les fonctions membres qui suivent ce mot clef accessibles "à l'extérieur de la classe". Ainsi, grâce à ce mot clef, nous avons pu dans la fonction main accéder directement au champ xSum par exemple :

```
//main.cpp
#include "Accumulator.h"

void main()
{
  Accumulator acc(0,0,0);

    for (int i = 0 ; i < 10;i++)
    {
        acc.Add( (double) i);
    }

    double mean = acc.xSum / acc.n;
}
```

Nous pourrions choisir de mettre les champs de cette classe en privé, en modifiant de manière adéquate le fichier header :

```
//Accumulator.h
#ifndef ACCUMULATOR_H
#define ACCUMULATOR_H

class Accumulator
{
private:
  int n;
  double xSum;
  double xSquareSum;

public:
  Accumulator(int, double, double);
  Accumulator();
  ~Accumulator(void);
  void Add(double);
  void Initialize(int, double, double);
};

#endif
```

Lorsqu'un champ d'une classe est déclaré en privé, il n'est pas accessible en dehors des fonctions membres de la classe. Ainsi, le compilateur refusera de compiler les instructions suivantes :

```
//main.cpp
#include "Accumulator.h"

void main()
{
  Accumulator acc;
  acc.Add(1);

  double mean = acc.xSum / acc.n; //ERROR: private fields xSum and n cannot
      be accessed outside the class Accumulator
}
```

Comment dès lors calculer la moyenne ? Il suffit de créer une méthode publique qui puisse manipuler les champs privés et qui nous retourne publiquement cette valeur :

```
double Accumulator::GetMean(void)
{
  return xSum / n;
}
```

Si cette méthode est publique, même si les champs xSum et n sont privés, il sera alors possible d'accéder à la moyenne, partout dans notre code. Ce faisant, nous avons contraint la manière dont un utilisateur peut accéder à la moyenne : sa seule solution est de passer par cette méthode publique, spécifiquement créée dans ce dessin. En architecture logicielle, le fait de contraindre un utilisateur dans sa manière d'utiliser un code est une excellente habitude, car c'est un moyen très efficace de l'empêcher de réimplémenter lui-même de la logique, et donc d'écrire du code qui pourrait se révéler défaillant ou difficile à maintenir. Au contraire, il faut s'efforcer que toute la logique relative à une classe se trouve implémentée par des méthodes de celle-ci, et que seuls les résultats de ces méthodes soient accessibles depuis l'extérieur de la classe.

```
1  //main.cpp
2  #include "Accumulator.h"
3
4  void main()
5  {
6    Accumulator acc;
7    acc.Add(1);
8
9    double mean = acc.GetMean();
10 }
```

Par convention, les champs privés d'une classe sont préfixés par "_". Nos fichiers header et source devenant alors :

```
1  //Accumulator.h
2  #ifndef ACCUMULATOR_H
3  #define ACCUMULATOR_H
4
5  class Accumulator
6  {
7  private:
8    int _n;
9    double _xSum;
10   double _xSquareSum;
11
12 public:
13   Accumulator(int, double, double);
14   Accumulator();
15   ~Accumulator(void);
16   void Add(double);
17
18   double GetMean(void);
19 };
20
21 #endif
```

```
#include "Accumulator.h"

Accumulator::Accumulator(int nInit, double xSumInit, double xSquareInit)
{
  _n = nInit;
  _xSum = xSumInit;
  _xSquareSum = xSquareInit;
}

Accumulator::Accumulator()
{
  _n = 0;
  _xSum = 0;
  _xSquareSum = 0;
}

Accumulator::~Accumulator(void)
{
}

void Accumulator::Add(double x)
{
  _n++;
    _xSum += x;
    _xSquareSum += x*x;
}

double Accumulator::GetMean(void)
{
  return _xSum / _n;
}
```

9.3.2 Accesseurs

Pour accéder à des champs privés depuis l'extérieur d'une classe, il est possible de définir des accesseurs, c'est à dire des méthodes publiques accédant en lecture ou en écriture à un champ privé spécifique. Par convention, un accesseur en lecture à un champ est préfixé par Get, et un accesseur en écriture est prefixé par Set. Ainsi, nous pouvons par exemple déclarer un accesseur en lecture pour notre champ privé _n, afin de connaître combien d'exemples ont été fournis à notre accumulator :

```
1 //in Accumulator.cpp :
2 int Accumulator::GetN(void)
3 {
4   return _n;
5 }
```

9.3.3 Philosophie de l'encapsulation

Pourquoi donc utiliser des accesseurs alors qu'il serait plus rapide de déclarer le champ public et de pouvoir le lire et le modifier plus facilement ? C'est le principe même de l'encapsulation, ébauché ci-dessus. Les champs d'une classe servent en règle générale à décrire un état interne de la classe, qui ne doit pas être accessible de l'extérieur.

Au contraire, il faut considérer qu'une classe bien conçue fournit un minimum de fonctions publiques, qui correspondent aux fonctionnalités de la classe. Les méthodes privées, ainsi que les champs(privés), servent d'intermédiaires aux méthodes publiques pour découper la logique générale exposée par ces dernières en des quantums de logique plus petits.

En adoptant un tel design, nous minimisons pour l'utilisateur de la classe Accumulator la quantité de compréhension qu'il doit posséder de cette classe pour l'utiliser. Sans connaître son implémentation interne, un utilisateur qui comprend ce que font les méthodes publiques d'une classe est capable de l'utiliser. Ce design permet ainsi de découpler de manière importante les dépendances sémantiques entre classes, et ainsi d'abstraire le code.

9.4 Méthodes et variables statiques

9.4.1 Champs statiques

Supposons que nous ayions plusieurs instances de la classe Accumulator. Nous souhaiterions savoir combien d'exemples au total ont été présentés à une instance quelconque de cette classe. Dans le cas où nous avons seulement deux exemplaires de la classe Accumulator, nous pourrions procéder de cette manière :

```
void main()
{
  Accumulator acc1;
    acc1.Add(1);
      acc1.Add(7);

      Accumulator acc2;
    acc2.Add(4);

      int n1 = acc1.GetN();
      int n2 = acc2.GetN();

      int n = n1+n2;
}
```

Cependant, lorsque nous ne connaissons pas le nombre exacts d'accumulateurs créés ni leur nom, la tâche devient plus difficile. Comment donc procéder ?

- Nous pourrions utiliser une variable globale que nous incrémenterions dans le constructeur de `User` et décrémenterions dans le destructeur. Cette solution est peu élégante car il s'agit fondamentalement d'une *propriété de la classe*. Par ailleurs, si nous voulions compter d'autres classes que `User`, nous nous retrouverions rapidement avec un grand nombre de variables, ce qui finirait par être peu pratique à manipuler. En règle générale, **il faut absolument s'interdire l'utilisation de variables globales.**

- Nous pourrions utiliser une des fonctionnalités du C++ : la *variable de classe* ou *variable statique*.

Une variable statique est tout simplement un attribut d'une classe, mais qui *n'est pas propre à une instance*. Au contraire, il est partagé par toutes les instances d'une même classe. Voici le prototype de la déclaration d'un membre statique :

```
static int userCount;
```

Utilisons donc un champ static entier pour compter combien d'instances ont été initialisées au total. La déclaration de notre classe Accumulator devient alors :

```
//Accumulator.h
#ifndef ACCUMULATOR_H
#define ACCUMULATOR_H

class Accumulator
{
private:
  int _n;
  double _xSum;
  double _xSquareSum;
  static int nTotal;

public:
  Accumulator(int, double, double);
  Accumulator();
  ~Accumulator(void);
  void Add(double);

  double GetMean(void);
  int GetN(void);
};

#endif
```

Et nous modifions la méthode Add pour que celle-ci incrémente également notre champ static :

```
void Accumulator::Add(double x)
{
  _n++;
    _xSum += x;
    _xSquareSum += x*x;
  nTotal++;
}
```

Ainsi, chaque fois que nous appellerons la méthode Add, que ce soit sur l'instance acc1 ou l'instance acc2, ou encore sur une autre instance, le champ static (et donc partagé) nTotal sera incrémenté.

Comment devons-nous initialiser ce champ ? Puisque le champ static existe indépendemment des instances de la classe Accumulator, ce champ préexiste en fait même à la première instance de la classe Accumulator qui pourra être créée. Il faut donc l'initialiser en dehors d'un constructeur. La syntaxe pour initialiser un champ static d'une classe est la suivante (à ajouter dans le fichier Accumulator.cpp, en dehors de toute fonction) :

```
int Accumulator::nTotal=0;
```

L'initialisation des variables statiques peut poser problème. En effet, le compilateur agit "comme si" les variables étaient toutes déclarées, puis elles sont ensuite définies dans l'ordre dans lequel le compilateur lit les définitions dans le fichier source. Nous pouvons alors avoir des comportements étranges, comme en témoigne l'exemple suivant :

```
//A.h
#ifndef A_H
#define A_H

class A
{
  static int a;
  static int b;
};

#endif
```

```
//A.cpp

#include "A.h"

int A::a = b+1;
int A::b = a+1;
```

Les valeurs prises par les variables a et b sont respectivement 1 et 2, mais si nous modifions l'ordre dans lequel nous affectons ces variables statiques, leurs valeurs sont inversées : a vaut 2 et b vaut 1 :

```
//A.cpp

#include "A.h"

int A::b = a+1;
int A::a = b+1;
```

9.4.2 Méthodes statiques

Nous voudrions maintenant instancier un accumulator à la condition qu'il n'en existe pas déjà un. Pour faire ceci, nous voulons tout d'abord ajouter un compteur qui va dénombrer le nombre d'instances de la classe Accumulator qui ont été créées. Nous ajoutons donc un autre champ entier statique "accumulatorInstancesCreated" dans le fichier header, et l'initialisons dans le fichier Accumulator.cpp.

```
1  //Accumulator.h
2  #ifndef ACCUMULATOR_H
3  #define ACCUMULATOR_H
4
5  class Accumulator
6  {
7  private:
8    int _n;
9    double _xSum;
10   double _xSquareSum;
11   static int nTotal;
12   static int accumulatorInstancesCreated;
13
14 public:
15   Accumulator(int, double, double);
16   Accumulator();
17   ~Accumulator(void);
18   void Add(double);
19
20   double GetMean(void);
21   int GetN(void);
22 };
23
24 #endif
```

```
#include "Accumulator.h"

Accumulator::Accumulator(int nInit, double xSumInit, double xSquareInit)
{
  _n = nInit;
  _xSum = xSumInit;
  _xSquareSum = xSquareInit;
  accumulatorInstancesCreated++;
}

Accumulator::Accumulator()
{
  _n = 0;
  _xSum = 0;
  _xSquareSum = 0;
  accumulatorInstancesCreated++;
}

Accumulator::~Accumulator(void)
{
}

void Accumulator::Add(double x)
{
  _n++;
    _xSum += x;
    _xSquareSum += x*x;
  nTotal++;
}

double Accumulator::GetMean(void)
{
  return _xSum / _n;
}

int Accumulator::GetN(void)
{
  return _n;
}

int Accumulator::nTotal=0;
int Accumulator::accumulatorInstancesCreated=0;
```

Comment maintenant tester si une instance a déjà été créée ? Puisque le champ static "accumulatorInstancesCreated" est privé, il est nécessaire de construire une méthode GetInstancesCount() renvoyant cette valeur. Cependant, si aucune instance n'a encore été créée, comment pourrions nous écrire instance.GetInstancesCount() ?

Il est maintenant nécessaire d'ajouter un concept, celui de fonc-

tion statique. Une fonction est dite statique, si son comportement ne dépend pas de l'état interne de l'instance appelante, autrement dit, si son résultat et ses effets pourraient être appelés sur n'importe quelle instance de la classe avec à chaque fois le même effet. Puisqu'une telle fonction n'a pas besoin d'une instance spécifique pour être appelée, elle est dite statique et est attachée à la classe plutôt qu'à une instance spécifique.

```
1  //Accumulator.h
2  #ifndef ACCUMULATOR_H
3  #define ACCUMULATOR_H
4
5  class Accumulator
6  {
7  private:
8    int _n;
9    double _xSum;
10   double _xSquareSum;
11   static int nTotal;
12   static int accumulatorInstancesCreated;
13
14 public:
15   Accumulator(int, double, double);
16   Accumulator();
17   ~Accumulator(void);
18   void Add(double);
19   static int GetInstancesCreatedCount(void); //new static method just created
20
21   double GetMean(void);
22   int GetN(void);
23 };
24
25 #endif
```

```
1  //In Accumulator.cpp :
2
3  int Accumulator::GetInstancesCreatedCount(void)
4  {
5    return accumulatorInstancesCreated;
6  }
```

Les méthodes statiques publiques, tout comme les champs statiques publics, ne sont pas appelés via une instance mais directement via le nom de la classe suivi de : :. Ainsi, nous pouvons tester dans notre main par exemple s'il existe déjà une instance ou non :

```
void main()
{
    if (Accumulator::GetInstancesCreatedCount() >= 1)
    {
        // .....
    }
}
```

Remarque : une méthode statique n'a pas nécessairement besoin de champs statiques. C'est seulement une manière de signifier que son code est indépendant de l'instance appelante. Par exemple, nous pourrions concevoir une méthode statique Merge qui prendrait en argument deux instances de la classe Accumulator et qui en construirait une troisième. Cet exemple est laissé à titre d'exercice au lecteur.

Note : Il est tout indiqué de déclarer les méthodes comme statiques dès que possible. Lorsque nous écrivons "du code mathématique", nos méthodes sont contenues dans des classes mais ne dépendent bien souvent d'aucun des champs de la classe. C'est le cas idéal pour utiliser des méthodes statiques.

9.4.3 constructeur statique

En C++, il n'existe pas de constructeur statique comme le constructeur statique de C# ou les statics blocs de Java. Les constructeurs statiques peuvent être très utiles lorsque vous souhaitez effectuer une opération qui aura toujours le même résultat quelle que soit l'instance créée. L'utilisation d'un constructeur statique peut permettre de n'effectuer cette opération qu'une seule fois. Ceci n'est pas possible de cette manière en C++.

9.5 Constructeur-copie

Dans la section 7.4, nous avons expliqué que si nous ne spécifions pas qu'un argument doit être passé par référence, il est par défaut passé par valeur. Ainsi, dans le listing 9.1, l'instance `a` de type `A` est passée par valeur à la fonction `f` alors qu'elle est passée par référence à la fonction `g`. Nous nous trouvons donc à l'entrée de la fonction `f`, dans un état de la mémoire comparable à celui du graphique 9.1.

```
1
2  class A
3  {
4      public:
5          A(int c, int d);
6          ~A();
7
8      private:
9          int _c;
10         int _d;
11 }
12
13 A::A(int c, int d)
14 {
15     _c = c;
16     _d = d;
17 }
18
19 A::~A()
20 {
21 }
22
23 void f(A copy)
24 {
25
26 }
27
28 void g(A& a)
29 {
30
31 }
32
33 void main()
34 {
35     A a;
36     f(a);
37     g(a);
38 }
```

Listing 9.1 – passage par valeur et par référence d'un argument de type non-primitif

Dans le cas de la fonction `f`, comment l'environnement réalise-t-il une copie de l'instance `a` en `copy` ? Cette copie est réalisée par l'environnement qui appelle de manière invisible, juste avant d'entrer dans la fonction `f`, un constructeur spécial, appelé le **constructeur-copie**. Le constructeur-copie de la classe `A`, comme un constructeur classique, est une méthode membre de la classe correspondante (ici `A`), qui prend en argument une instance source de type `A` qu'il cherche à copier, et en crée une copie, également de type `A`.

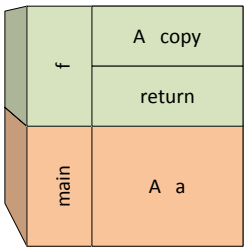

FIGURE 9.1 – Etat de la Stack en entrant dans la fonction f.

Tout comme l'environnement implémente par défaut implicitement un constructeur par défaut, il implémente également implicitement un constructeur-copie par défaut. Ce constructeur-copie par défaut se contente de copier octet par octet tous les champs (public, protected comme private) de l'instance source dans les champs correspondants de l'instance cible. Nous pouvons donc expliciter le constructeur-copie de la classe `A` de la sorte :

```
1
2  class A
3  {
4      public:
5          A(int c, int d);
6          A(const A& source);
7          ~A();
8
9      private:
10         int _c;
11         int _d;
12 }
13
14 A::A(int c, int d)
15 {
16     _c = c;
17     _d = d;
18 }
19
20 A::A(const A& source)
21 {
22     _c = source._c;
23     _d = source._d;
24 }
25
26 A::~A()
27 {
28 }
```

Listing 9.2 – Constructeur-copie de la classe A explicité

Notez bien que le constructeur-copie prend son argument par référence (sinon il faudrait appeler le constructeur-copie avant de rentrer dans le constructeur-copie, et il faudrait appeler le constructeur-copie avant de rentrer dans le constructeur-copie pour rentrer dans le constructeur-copie, et ainsi de suite.)

Dans l'exemple ci-dessus, nous avons explicité le constructeur-copie pour répliquer parfaitement le comportement implicite donné par défaut à ce dernier par l'environnement. Nous rencontrerons dans la chapitre 14 des exemples de cas dans lesquels il est nécessaire de donner un sens plus spécifique à ce constructeur-copie.

—Have you ever sat down and worked on a C++ project ? Here's what happens : First, I've put in enough pitfalls to make sure that only the most trivial projects will work first time. Take operator overloading. At the end of the project, almost every module has it, usually, because guys feel they really should do it, as it was in their training course. The same operator then means something totally different in every module.

Parodie d'interview de Bjarne Stroustrup

10

Opérateurs

10.1 Introduction

La surcharge d'opérateurs consiste à donner du sens à l'utilisation d'opérateurs comme +, *, (), etc. appliqués à des types non primitifs. Cette surcharge permet donc aux opérateurs du C++ d'avoir une signification particulière pour des types spécifiques.

Dans le chapitre précédent, nous avons écrit un objet `Accumulator`, qui permettait de collecter une grande quantité de nombres pour en calculer des statistiques élémentaires. Nous souhaiterions maintenant pouvoir assembler ("merger" dans la langue de Shakespeare) deux instances de cette classe en une troisiéme instance, qui serait dans le même état que si elle avait résumé tous les nombres présentés aux deux premières instances. Dans ce but, nous pouvons écrire une fonction `Merge` :

```
1  //Accumulator.h
2  #ifndef ACCUMULATOR_H
3  #define ACCUMULATOR_H
4
5  class Accumulator
6  {
7  private:
8    int _n;
9    double _xSum;
10   double _xSquareSum;
11   static int nTotal;
12   static int accumulatorInstancesCreated;
13
14 public:
15   Accumulator(int, double, double);
16   Accumulator();
17   ~Accumulator(void);
18   void Add(double);
19   static int GetInstancesCreatedCount(void); //new static method just created
20     Accumulator MergeWith(const Accumulator&);
21
22   double GetMean(void);
23   int GetN(void);
24 };
25
26 #endif
```

Listing 10.1 – Accumulator.h

```
1  //In Accumulator.cpp :
2
3  Accumulator Accumulator::MergeWith(const Accumulator& acc2)
4  {
5    Accumulator result;
6      result._n = _n + acc2._n;
7      result._xSum = _xSum + acc2._xSum;
8      result.xSquareSum = _xSquareSum + acc2._xSquareSum;
9      return result;
10 }
```

Listing 10.2 – Accumulator.cpp

Nous pouvons alors utiliser ce code de la sorte :

```
1  //In main.cpp :
2
3  #include "Accumulator.h"
4
5  void main()
6  {
7      Accumulator acc1;
8      Accumulator acc2;
9      acc1.Add(2);
10     acc1.Add(3);
11
12     acc2.Add(2.3);
13
14     Accumulator sum = acc1.MergeWith(acc2);
15 }
```

Listing 10.3 – main.cpp

Ce code a deux défauts mineurs. Tout d'abord, il pourrait être rendu plus compact. Plus gênant, il donne une impression d'assymétrie entre les instances `acc1` et `acc2`, qui ont pourtant des rôles parfaitement symétriques. Nous voudrions pouvoir remplacer cette syntaxe par quelquechose de la sorte :

```
1  //In main.cpp :
2
3  #include "Accumulator.h"
4
5  void main()
6  {
7      Accumulator acc1;
8      Accumulator acc2;
9      acc1.Add(2);
10     acc1.Add(3);
11
12     acc2.Add(2.3);
13
14     Accumulator sum = acc1 + acc2;
15 }
```

Listing 10.4 – main.cpp

Cette syntaxe est rendue possible en C++ par la surcharge de l'opérateur + pour la classe `Accumulator`. Il faut appréhender en première approche les opérateurs comme des fonctions membres, c'est à dire des méthodes. Nous allons donc définir notre opérateur comme s'il s'agissait d'une fonction membre à part entière, en ajou-

tant seulement le mot clef "operator" :

```
//Accumulator.h
#ifndef ACCUMULATOR_H
#define ACCUMULATOR_H

class Accumulator
{
private:
  int _n;
  double _xSum;
  double _xSquareSum;
  static int nTotal;
  static int accumulatorInstancesCreated;

public:
  Accumulator(int, double, double);
  Accumulator();
  ~Accumulator(void);
  void Add(double);
  static int GetInstancesCreatedCount(void);
  Accumulator operator+(const Accumulator&);

  double GetMean(void);
  int GetN(void);
};

#endif
```

Listing 10.5 – Accumulator.h

```
//In Accumulator.cpp :

Accumulator Accumulator::operator+(const Accumulator& acc2)
{
    Accumulator result;
    result._n = _n + acc2._n;
    result._xSum = _xSum + acc2._xSum;
    result.xSquareSum = _xSquareSum + acc2._xSquareSum;
    return result;
}
```

Listing 10.6 – Accumulator.cpp

Bonnes habitudes 20 (Quand surcharger un opérateur ?) *Pratiquement jamais. Il ne faut jamais surcharger un opérateur si le sens exact de l'opérateur n'est pas parfaitement limpide. Dans ce cas, il vaut mieux implémenter une fonction avec un nom suggestif. Dans la plupart des cas, implémenter un opérateur est une mauvaise idée, et vous devez*

vous abstenir de le faire sauf si c'est vraiment la solution évidente et indiscutable. Dès lors que le sens possible que pourrait avoir un opérateur soulève la plus petite ambigüité, ne l'implémentez-pas.

Bonnes habitudes 21 (Ne pas implémenter un seul opérateur)
Si après mûre réflexion, vous avez décidé d'implémenter un opérateur, implémenter les tous. Si a et b sont deux instances de type T, et que le type T implémente un opérateur $+$, alors nous pouvons écrire $a+b$. Dans ce cas, l'utilisateur de la classe T s'attend également à trouver l'opérateur $+=$. De la même manière, si l'opérateur "prefix increment" (non détaillé jusqu'à présent dans ce poly) $++a$ est implémenté, implémentez également l'opérateur "suffix increment" $a++$. Enfin, si vous surchargez l'opérateur $<$, surchargez également l'opérateur $>$.

Bonnes habitudes 22 (Ne pas oublier qu'un opérateur "coûte")
En dépit de la compacité du code d'utilisation d'un opérateur, il ne faut pas oublier que du point de vue de la machine, un opérateur revient à appeler une méthode, et s'accompagne donc d'un certain coût. Ainsi, lorsque nous étudierons la classe vector, vous verrez que l'opérateur [] est bien plus coûteux que l'opérateur [] pour les tableaux natifs.

10.2 Quelques exemples pertinents de surcharge d'opérateurs

Nous donnons pour illustrer notre propos quelques cas dans lesquels l'usage d'opérateurs est justifié. Comme nous n'avons pour l'instant pas découvert beaucoup de classes, ces exemples ne vous parleront pas pour le moment, mais vous pourrez y revenir en temps voulu.

- Opérateur [] pour la classe vector
- Opérateur -> pour les smart pointers
- Opérateur ++ pour les itérateurs
- Opérateur +,* pour les classes de type math (vecteur, matrices, big integer, etc.)
- Opérateur = pour toutes les classes avec de la gestion de ressource (cf chapitre allocation dynamique de mémoire)

10.3 Un autre exemple

Dans ce paragraphe, nous donnons un autre exemple d'utilisation d'opérateurs, dans un cas archi-classique, celui d'un code mathématique, en l'occurence l'implémentation (partielle) d'une classe Complex.

Il est fondamental en informatique de ne pas réinventer la roue. Probablement, le problème auquel vous devez faire face a déjà été rencontré et résolu par de nombreuses personnes avant vous. Outre le fait qu'elles sont probablement plus compétentes que vous, le code qu'elles ont produit a été éprouvé par le temps et l'usage, et il sera a priori moins sujet aux bugs que vous ne manquerez pas de rencontrer si vous optez pour une solution que vous développerez vous-même. En conséquence, **considérez toujours d'abord de réutiliser du code**[1] **plutôt que d'en écrire.** Dans notre cas, il est évident qu'il ne faut jamais réimplémenter une classe Complex, nous en donnons juste une implémentation partielle à titre pédagogique.

Notez que certains opérateurs sont implémentés comme des fonctions membres, alors que d'autres non. Celà respecte les suggestions fournies dans le paragraphe 10.5.

1. Ce qui ne vous dédouane pas de le tester bien sûr, par exemple unitairement.

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex
{
public:
    Complex(double,double);
  ~Complex(void);
  void operator += (const Complex&);
  void operator -= (const Complex&);

private:
  double _real;
  double _im;
};

//Non-Member functions
Complex operator + (const double&, const Complex&);
Complex operator + (const Complex&, const double&);
Complex operator + (const Complex&, const Complex&);

Complex operator - (const double&, const Complex&);
Complex operator - (const Complex&, const double&);
Complex operator - (const Complex&, const Complex&);

#endif
```

Listing 10.7 – Complex.h

```
#include "Complex.h"

Complex::Complex(double real, double imaginary)
{
  _real = real;
  _im = imaginary;
}

Complex::~Complex(void)
{
}

void Complex::operator+=(const Complex& c2)
{
  _real +=c2._real;
  _im += c2._im;
}

void Complex::operator+=(const double& real)
{
  _real +=real;
}

void Complex::operator-=(const Complex& c2)
{
  _real -=c2._real;
  _im -= c2._im;
}

void Complex::operator-=(const double& real)
{
  _real -= real;
}

Complex operator+(const Complex& c1, const double& real)
{
  Complex result = Complex::FromCartesian(0,0);
  result += c1;
  result += real;
  return result;
}

Complex operator+(const double& real, const Complex& c2)
{
  return c2 + real;
}

Complex operator-(const Complex& c1, const double& real)
{
  Complex result = Complex::FromCartesian(0,0);
  result += c1;
  result -= real;
  return result;
}

Complex operator-(const double& real, const Complex& c2)
{
  Complex result = Complex::FromCartesian(0,0);
  result += real;
  result -= c2;
  return result;
}
```

Listing 10.8 – Complex.cpp

10.3.1 Digression autour du "Named Constructor Idiom"

Dans notre classe Complex, il y a deux manières naturelles de construire une instance complexe :

- Par coordonnées cartésiennes, en donnant la partie réelle et la partie imaginaire du nombre.
- Par coordonnées polaires, en donnant le module et l'argument de notre complexe.

Malheureusement, ces deux manières de constuire un complexe font toutes les deux appels à deux arguments de type double, et il y a un risque qu'un utilisateur de notre classe construise un complexe en donnant de mauvais arguments. Par exemple, supposons qu'un utilisateur de notre classe Complex veuille constuire le nombre i en donnant son module et son argument $(1, \frac{\pi}{2})$. Il pourrait être tenté de le constuire de la sorte :

```
1 #include "Complex.h"
2 void main()
3 {
4     Complex c(1,Pi/2);
5 }
```

Listing 10.9 – main.cpp

Pour empêcher cette utilisation (qui mène au complexe $1 + \frac{\pi}{2}i$), nous voudrions avoir un nom plus spécifique pour le consructeur que simplement "Complex", afin de lever cette ambiguïté. Pour ce faire, nous recourons au design dit du "Named Constructor".

- Tout d'abord, nous mettons les constructeurs de notre classe en private, afin de nous assurer qu'ils ne pourront plus être utilisés directement.
- Ensuite, nous proposons des méthodes publiques static dont la fonction sera de construire une instance d'une manière spécifique.
- L'utilisateur peut et doit alors passer par ces méthodes pour construire un complexe.

```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex
{
public:
  static Complex FromCartesian(double, double);
  static Complex FromPolar(double,double);
  ~Complex(void);
  void operator += (const Complex&);
  void operator += (const double&);
  void operator -= (const Complex&);
  void operator -= (const double&);

private:
  double _real;
  double _im;
  Complex(double,double);
};

//Non-Member functions
Complex operator + (const double&, const Complex&);
Complex operator + (const Complex&, const double&);
Complex operator + (const Complex&, const Complex&);

Complex operator - (const double&, const Complex&);
Complex operator - (const Complex&, const double&);
Complex operator - (const Complex&, const Complex&);

#endif
```

Listing 10.10 – Complex.h

```
#include "Complex.h"
#include <cmath>

Complex Complex::FromCartesian(double real, double imaginary)
{
    return Complex(real, imaginary);
}

Complex Complex::FromPolar(double radius, double angle)
{
    return Complex(radius*std::cos(angle), radius*std::sin(angle));
}

Complex::Complex(double real, double imaginary)
{
  _real = real;
  _im = imaginary;
}

Complex::~Complex(void)
{
}

Complex operator+(const Complex& c1, const double& real)
{
  Complex result = Complex::FromCartesian(0,0);
  result += c1;
  result += real;
  return result;
}

Complex operator+(const double& real, const Complex& c2)
{
  return c2 + real;
}

Complex operator-(const Complex& c1, const double& real)
{
  Complex result = Complex::FromCartesian(0,0);
  result += c1;
  result -= real;
  return result;
}

Complex operator-(const double& real, const Complex& c2)
{
  Complex result = Complex::FromCartesian(0,0);
  result += real;
  result -= c2;
  return result;
}

void Complex::operator+=(const Complex& c2)
{
  _real +=c2._real;
  _im += c2._im;
}

void Complex::operator+=(const double& real)
{
  _real +=real;
}

void Complex::operator-=(const Complex& c2)
{
```

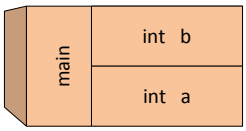

FIGURE 10.1 – Copie de b en a, cas des entiers.

```
#include "Complex.h"
void main()
{
    Complex c1 = Complex::FromPolar(1,Pi/2);
    Complex c2 = Complex::FromCartesian(0,1);
}
```

Listing 10.12 – main.cpp

10.4 Surcharge de l'opérateur d'affectation

Lorsque nous écrivons le code du listing 10.13, nous utilisons l'implémentation de l'opérateur = pour les entiers. Cette implémentation spécifie qu'après la ligne "a=b;", a et b sont deux instances d'entiers, désignant chacune un entier distinct, mais qui possèdent la même valeur. Ainsi, nous aurions comme représentation sur la Stack quelquechose d'analogue au graphique 10.1.

```
void main()
{
    int a = 3;
    int b = 2;
    b=a;
}
```

Listing 10.13 – affectation d'un entier par un autre

Lorsque nous voulons écrire la même chose pour des `Complex`, que se passe-t-il ?

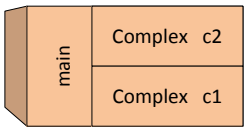

FIGURE 10.2 – Copie de b en a, cas des Complexs.

```
#include "Complex.h"
void main()
{
    Complex c1 = Complex::FromCartesian(1,0);
    Complex c2 = Complex::FromCartesian(0,1);
    c1=c2;
}
```

Listing 10.14 – affectation d'un Complex par un autre

Dans le listing 10.14, `c1` et `c2` sont deux instances de type Complex différentes, mais dont les champs `_real` et `_im` ont mêmes valeurs, *i.e.* `c1._real = c2._real` et `c1._im = c2._im` après exécution de l'opérateur =. Cet état est également décrit par le graphique 10.2. Que ce soit dans le cas des entiers ou dans le cas des Complexs, il est très important de noter et de retenir [2] que si nous écrivons à la suite de l'affectation `b=a;` (resp. `c1=c2;`) l'instruction : `a=1;` (resp. `c1 = Complex::FromCartesian(1,1)`), les valeurs de `b` et `c2` restent inchangées.

Comment le langage réalise-t-il donc l'affectation (opérateur =) d'une instance de type non primitif en une autre ? Par défaut, son comportement est très exactement le même que le constructeur-copie par défaut : l'environnement va copier octet par octet tous les champs de l'instance source dans l'instance cible.

En règle générale, c'est un comportement raisonnable, et nous pouvons dans la plupart des cas laisser cette implémentation par défaut de l'opérateur =. Cependant, nous verrons que lorsque l'un des champs à recopier est un pointeur, l'implémentation par défaut

2. Car c'est un comportement très différent de celui dans des langages comme le C# ou le Java par exemple.

de l'opérateur = peut se révéler très dangereuse, et nous voudrons dans ces cas-là la redéfinir nous-même explicitement. Afin de comprendre comment la redéfinir, nous donnons dans le listing 10.15, à titre d'exemple, une redéfinition explicite de l'opérateur = qui se contente de mimer fidèlement l'implémentation implicite par défaut, c'est à dire qui recopie les champs valeur par valeur.

```
1 void Complex::operator=(const Complex& source)
2 {
3   _real = source._real;
4   _im = source._im;
5 }
```

Listing 10.15 – première surcharge de l'opérateur d'affectation

Traditionnellement, et pour des raisons que nous ne détaillons pas ici [3], cet opérateur doit en réalité retourner la nouvelle valeur de l'instance modifiée. Pour faire celà, l'instance "appelante" doit donc se retourner elle-même. Cette opération est possible via l'utilisation du mot clef `this`, qui, appelé dans une méthode de classe, retourne un pointeur sur l'objet qui appelle la méthode. Nous pouvons donc dans une méthode récupérer l'objet "appelant" par "***this**", comme dans le listing 10.16.

```
1 Complex& Complex::operator=(const Complex& source)
2 {
3   _real = source._real;
4   _im = source._im;
5     return *this;
6 }
```

Listing 10.16 – surcharge canonique de l'opérateur d'affectation

10.5 Opérateurs internes, opérateurs externes

3. penser qu'on veut pouvoir écrire **c1=c2=c3** ;

Object-oriented programming is an exceptionally bad idea which could only have originated in California.

propos prêté faussement à Edsger Dijkstra, réel auteur inconnu

11 Les templates

Les templates sont un mécanisme puissant de factorisation de code, qui permettent d'écrire du code générique s'appliquant à des données, indépendamment de leur type. Plus précisément, ils permettent de produire en un seul fichier une famille de fonctions ou une famille de classes indicées par un type abstrait ou par un autre paramètre comme un entier.

11.1 Templating par un type, l'exemple des fonctions

Les templates sont issus originellement d'un souci de simplification de code par la factorisation, afin d'éviter les redondances. Prenons l'exemple d'une fonction Max qui prenne en argument un tableau de double et la taille de ce tableau :

```
1 double Max(double array[], int length)
2 {
3     double vmax = array[0];
4     for (int i = 1; i < length; i++)
5         if (array[i] > vmax)
6             vmax = array[i];
7     return vmax;
8 }
```

Si nous voulions définir la même fonction sur un tableau d'entiers, nous devrions alors produire le code suivant :

```
int Max(int array[], int length)
{
    int vmax = array[0];
    for (int i = 1; i < length; i++)
        if (array[i] > vmax)
            vmax = array[i];
    return vmax;
}
```

De la même manière, nous pourrions définir la fonction Max sur un tableau de float, de ushort, de uint, de long, etc... Dans le cadre d'exemples plus complexes, le dédoublement du code pour chaque type est très pénalisant : tout d'abord, il nuit à la clarté du code pour le lecteur, mais il entraine aussi un risque important de divergence des différentes versions du code. En effet, si la sémantique de la fonction Max est incorrecte, elle le sera à la fois pour la version du code manipulant des entiers comme pour celle manipulant des doubles. Si un développeur est amené à améliorer ou débugger une version de cette fonction, il court le risque d'oublier que d'autres versions de cette fonction demandent probablement les mêmes modifications.

Nous le concevons donc, **la redondance du code est à proscrire**[1]. Comment dans ces conditions créer une seule fonction Max qui permette de définir cette fonction pour des entiers, mais aussi pour des double, des uint, etc... ? Le C++ propose un mécanisme pour définir en une fois le code devant s'appliquer, quel que soit le type des arguments. Observons la syntaxe suivante :

```
template<typename T>
T Max(T array[], int length)
{
    T vmax = array[0];
    for (int i = 1; i < length; i++)
        if (array[i] > vmax)
            vmax = array[i];
    return vmax;
}
```

1. La règle 1 du développement pourrait être : "ne faites pas de copier/coller de code au sein d'un projet."

Dans cet exemple, la fonction Max devient paramétrée par un type abstrait T. Celui-ci est utilisé dans notre exemple à la fois pour définir le type du premier argument de la fonction, mais également pour définir son type de retour. Le préfixe template<typename T> indique au compilateur que le code qui suit sera paramétré par un type T. De manière équivalente, le mot clef *typename* peut être remplacé par *class*.

Lorsque dans notre code, nous voulons utiliser notre fonction Max, nous pouvons le faire de la sorte :

```
1 int values[]={ 16, 8, 3, 2, 11 };
2
3 cout << Max<int>(values, 5);
```

Lorsque le compilateur va lire l'appel à Max(values, 5), il va détecter qu'il s'agit d'utiliser la fonction templatée Max dans le cas où T = int. Le compilateur va alors générer le code correspondant et l'inclure dans la compilation. Bien évidemment, le compilateur ne génèrera la fonction Max que pour les types T pour lesquels il est fait appel quelque part à la fonction Max utilisée pour le type T : si nulle part dans notre code nous ne cherchons à déterminer le max d'un tableau de double, le code spécifique pour la fonction Max en le type double ne sera pas généré.

Le fait d'utiliser une fonction templatée en un type spécifique est appelé *spécialisation*. Pouvons-nous spécialiser la fonction Max en n'importe quel type ? L'approche du C++ sur la question est une approche optimiste (à la différence du C# par exemple) : par défaut, tout type est accepté. C'est uniquement à la compilation que le compilateur va tenter de générer le code nécessaire pour chacun des types en lesquels la fonction templatée a été spécialisée. Si notre fonction est spécialisée en un type T1 pour lequel l'opérateur < n'est pas défini, alors le compilateur échouera dans la génération du code spécialisé.

11.1.1 Templates et macro

Si nous reprenons la fonction templatée précédente appliquée à deux valeurs plutôt qu'à un tableau, nous pouvons produire le code suivant :

```
template<typename T>
const T & Max( const T & a, const T & b )
{
    return a > b ? a : b;
}
```

Cet exemple ressemble beaucoup avec la macro correspondante, comme détaillée dans le chapitre sur la compilation :

```
#define MAX(a, b) (((a) > (b)) ? (a) : (b))
```

En effet, dans chaque cas, nous avons une implémentation de la fonction max qui peut s'adapter à tous types d'objets. Le templating est la manière propre d'écrire des macros. Là où la macro est une simple substitution syntaxique, avec toutes les erreurs qui en découlent (4.3.4), le templating génère de réelles fonctions et permet donc d'obtenir de manière fiable le résultat.

Dans la comparaison qui vient d'être faite, il faut noter cependant qu'un avantage majeur de la macro par rapport à son homologue templatée est l'absence d'appel de fonction : puisque la macro ne crée pas de véritable fonction, il n'y a pas d'appel de fonction et donc pas de coût d'appel de fonction. Il est possible d'éviter ce coût également dans le cas d'une fonction templatée, en l'inlinant :

```
template<typename T>
inline const T & Max( const T & a, const T & b )
{
    return a > b ? a : b;
}
```

11.1.2 Fonctions membres templatées

Il est également possible de créer une fonction templatée au sein d'une classe non templatée. L'exemple suivant décrit une classe disposant d'une fonction membre templatée permettant d'afficher différente objets.

```
class SomeClass
{
    public SomeClass();
    public ~SomeClass();

    template<typename T>
    static void Display( const T & t )
    {
        cout << t;
    }
}

int main()
{
    SomeClass.Display<int>(2);
    SomeClass.Display<string>("Hello World");
    SomeClass.Display<double>(3.14);
}
```

11.1.3 Inférence automatique de type de spécialisation

Lorsque le compilateur est capable d'inférer le type en lequel est spécialisée une fonction templatée, il est superflu de spécifier explicitement en quel type la fonction est spécialisée. Dans le cas du code précédent par exemple, nous pourrions écrire :

```
int main()
{
    SomeClass.Display(2);
    SomeClass.Display("Hello World");
    SomeClass.Display(3.14);
}
```

Il est cependant des cas où une telle inférence n'est pas possible, notamment dans le cas d'ambiguïté que le compilateur ne peut pas lever lui-même. Ainsi, la fonction suivante doit être spécifiée explicitement :

```
template <typename T>
T Sum( T s1, T s2 )
{
    return s1 + s2;
}

int main()
{
    int s2 = 1;
    double s1 = 3.2;

    Sum( s1, s2 ); // Erreur : paramÃtre ambigÃ¨
    Sum<double>( s1, s2 ); // OK
}
```

11.1.4 Multi-templating

Il est possible de paramétrer une fonction par plusieurs arguments ; en voici un exemple :

```
template<typename T, typename U>
public T Add (const T& t, const U& u)
{
    return t+u;
}
```

11.2 Templating par un type, le cas des classes

De la même manière que nous avons défini le paramétrage d'une fonction par un type générique T, nous pouvons paramétrer une classe par un type générique. Le templating par des classes est principalement utilisé pour construire des "containers", c'est à dire des classes dont la fonction est de contenir un ensemble d'éléments d'un même type. Un exemple éclairant sera donné dans la section 15.3.1.

11.3 Templates et compilation

Un voile pudique est bien souvent jeté par les manuels d'introduction au C++ sur la compilation des templates. Ceux-ci répondant à des contraintes bien particulières (puisqu'il ne s'agit pas de code mais de "méta-code"), il n'est pas possible par défaut de déclarer

une fonction template dans un fichier .h puis de la définir dans un fichier .cpp. Par défaut, il vous faudra donc mélanger déclaration et définition dans un même fichier .h sous peine de vous exposer à des erreurs à l'édition des liens. Comme il est expliqué dans l'article d'Aurélien Regat-Barrel sur le site cpp.developpez.com, il existe néanmoins une astuce permettant de contourner le problème.

Cette astuce consiste à stocker la déclaration de votre classe templatée dans un fichier .h, de stocker votre définition dans un fichier texte (avec une extension différente de .cpp, comme .tpp par exemple), et d'inclure grâce à l'instruction #include le fichier .tpp à la fin du fichier header. Ainsi, nous obtenons par exemple quelque-chose de la forme :

```
// exemple.h

#ifndef EXEMPLE_H
#define EXEMPLE_H

template <typename T>
class Exemple
{
public:
    Exemple();
};

#include "exemple.tpp" // voici l'astuce
#endif

// exemple.tpp

template <typename T>
Exemple<T>::Exemple()
{
}
```

11.4 Templates et spécialisation

Il existe certains cas où nous voudrions faire des exceptions à la généricité, c'est à dire que pour certains types bien particuliers, une fonction templatée ait un comportement particulier, qui diffère du comportement général déjà défini. Prenons le cas de l'exponentiation de 2. Si y est un réel (double ou float), le calcul de 2^y demande de réécrire la formule en $e^{y.ln(2)}$ afin de l'évaluer. Nous pourrions

donc écrire :

```
1
2  template<typename T>
3  double TwoPow(T y)
4  {
5      return exp(y*ln(2);
6  }
```

Cette fonction fonctionnerait également si elle était spécifiée en deux entiers. Cependant, dans le cas où y est entier, il n'est pas nécessaire de passer par cette formule, il suffit alors de multiplier 2 par lui même y fois. Bien que la formule précédente soit exacte dans le cas où y est entier, elle entrainerait donc des calculs indûment longs. Pour améliorer cette situation, nous voudrions dire au compilateur : compile la fonction précédente pour tous les types nécessaires, SAUF dans le cas où y est entier, auquel cas contente toi de calculer directement la valeur de l'exponentiation. En C++, il est possible de préciser/redéfinir une spécialisation spécifique.

```
1
2  // Spécialisation pour les int
3  template <>
4  double TwoPow<int>( int i )
5  {
6      double q= (i >= 0) ? 2 : 0.5;
7      int iAbs = abs(i);
8      double r=1;
9
10     for (int j = 0 ; j < iAbs;j++)
11         r*=q;
12
13     return r;
14 }
```

11.4.1 Spécialisation partielle

Les templates peuvent être partiellement spécialisés, et la classe obtenue est alors encore un template. Cette spécialisation partielle intervient principalement dans le cas dans le cas d'un template paramétré par plusieurs types, pour lesquels seuls certains de ces types sont spécialisés, le résultat étant un template paramétré dans les types restants. Exemple :

```
template<typename T, typename U>
public double Pow(T x, U y)
{
    return exp(y*ln(x));
}

template<typename T>
public double Pow<T,int>(T x, int y)
{
    double q= (y >= 0) ? x : ((double)1)/x; //do not forget the explicit cast
        into double !
    int yAbs = abs(y);
    double r=1;

    for (int j = 0 ; j < yAbs;j++)
        r*=q;

    return r;
}
```

11.5 Templating par des entiers

Il est également possible de paramétrer une fonction par autre chose qu'un type. Notamment, il est possible de paramétrer une fonction par un entier. Ces mécanismes ne seront pas détaillés cette année, mais ils sont massivement utilisés dans de nombreuses librairies professionnelles, car ils permettent des optimisations très fines, notamment via le Template Meta-Programming. Les bonnes librairies de calcul scientifique en C++ par exemple reposent toutes massivement sur ce genre d'optimisation. Nous renvoyons le lecteur intéressé par exemple à [3].

In the fairy tales about heroes defeating evil villains there's always a dark forest of some kind. It could be a cave, a forest, another planet, just some place that everyone knows the hero shouldn't go. Of course, shortly after the villain is introduced you find out, yes, the hero has to go to that stupid forest to kill the bad guy. It seems the hero just keeps getting into situations that require him to risk his life in this evil forest.

You rarely read fairy tales about the heroes who are smart enough to just avoid the whole situation entirely. You never hear a hero say, "Wait a minute, if I leave to make my fortunes on the high seas leaving Buttercup behind I could die and then she'd have to marry some ugly prince named Humperdink. Humperdink! I think I'll stay here and start a Farm Boy for Rent business." If he did that there'd be no fire swamp, dying, reanimation, sword fights, rats, or anything else. Story really. Because of this, the forest in these stories seems to exist like a black hole that drags the hero in no matter what they do.

In object-oriented programming, Inheritance is the evil forest. Experienced programmers know to avoid this evil because they know that deep inside the Dark Forest of Inheritance is the Evil Queen Multiple Inheritance. She likes to eat software and programmers for many tasty teeth, chewing on the flesh of the fallen. But the forest is so powerful and so tempting that nearly every programmer has to go into it, and try to make it out alive with the Evil Queen's head before they can call themselves real programmers. You just can't resist the Inheritance forest, so you go in. After the adventure you learn to just stay out of that stupid forest and bring an army if you are ever forced to go in again.

http ://learnpythonthehardway.org/book/ex44.html

12 Heritage et Composition

12.1 Héritage simple

L'héritage est un des aspects fondamentaux de la programmation orientée-objet. Il permet de transmettre (*de faire hériter*)les propriétés d'une classe (méthodes et champs) à une autre classe, participant ainsi à la structuration des projets autour des classes.

12.1.1 Motivation

Par les vicissitudes d'une faille spatio-temporelle, vous vous retrouvez téléporté dans la peau d'un développeur pour une société de location de vélos et de voitures. Votre premier jet met en place les classes Bike et Car décrites dans les listings 12.1 et 12.2.

Pour ne pas perturber l'utilisateur des classes, ce sont les mêmes noms de variable qui sont utilisés dans les deux cas.

Nous faisons les remarques suivantes :
- Les vélos et les voitures sont tous deux des véhicules et partagent de nombreuses caractéristiques (prix, couleur, état), mais ont cependant des différences.
- Si nous voulons ajouter un autre type de véhicule (des motos, par exemple), il va nous falloir recréer une nouvelle classe presque identique aux précédentes.

```
#ifndef CAR_H
#define CAR_H

class Car
{
    public :
        Car(int state, int fuel, int color, int price);
        ~Car();

        void SetState(int state);
        int GetState(void);
        void SetFuel(int fuel);
        int GetFuel(void);
        void SetColor(int color);
        int GetColor(void);
        int GetPrice(void);
        void SetPrice(int);

    private:
        int _state; //is it second hand or new?
        int _fuel; // fuel remaining
        int _color;
        int _price;
};

#endif
```

Listing 12.1 – Car1.h

```
#ifndef BIKE_H
#define BIKE_H

class Bike
{
    public :
        Bike(int state, int color, int price);
        ~Bike();

        void SetState(int state);
        int GetState(void);
        void SetColor(int color);
        int GetColor(void);
        int GetPrice(void);
        void SetPrice(int);

    private:
        int _state; //is it second hand or new?
        int _color;
        int _price;
};

#endif
```

Listing 12.2 – Bike1.h

L'impression générale qui se dégage du code est une certaine lourdeur, liée aux redondances observées entre les classes Bike et Car. Dans la suite, nous cherchons à isoler ces redondances pour les factoriser et les écrire une seule fois.

12.1.2 Héritage simple et public

En informatique, cette factorisation est appelée *héritage*, et consiste à isoler le code commun pour créer une classe avec les propriétés communes, que nous appellerons *classe mère*. Les *classes filles* ou *classes dérivées*, dans notre exemple Car et Bike, vont hériter de cette classe mère et en possèderont toutes les propriétés.

```
#ifndef VEHICULE_H
#define VEHICULE_H

class Vehicule
{
    public :
        Vehicule(int state = 1,
      int color = 0,
      int price = 10000);
        ~Vehicule();

        void SetState(int state);
        int GetState();
        void SetColor(int color);
        int GetColor(int color);
        int GetPrice();
        void SetPrice(int price);

    private:
        int _state; //new?
        int _color;
        int _price;

};

#endif
```

Listing 12.3 – vehicule2.h

```
#ifndef CAR_H
#define CAR_H
#include "Vehicule2.h"

class Car : public Vehicule
{
    public:
        Car(int state = 1, int fuel = 100, int color = 0, int prix =
            10000);
        ~Car();

        void SetFuel(int fuel);
        int GetFuel();

    private:
        int _fuel;
};

#endif
```

Listing 12.4 – voiture2.h

```
#ifndef BIKE_H
#define BIKE_H
#include "vehicule2.h"

class Bike : public Vehicule
{
    public:
        Bike(int etat = 1, bool isPadlocked = false, int color = 0, int
            price = 10000);
        ~Bike();

        bool GetIsPadlocked();
        void SetIsPadlocked(bool isPadlocked);

    private:
        bool _isPadlocked;
};
#endif
```

Listing 12.5 – velo2.h

D'un point de vue syntaxique, la déclaration d'une classe dérivée est donc très simple. Il y a trois possibilités :

```
/*premiere possibilite */
class Fille : public class Mere
{
    /*nouveaux membres*/
};
/*seconde possibilite */
class Fille : protected class Mere
{
    /*nouveaux membres*/
};
/*troisieme possibilite */
class Fille : private class Mere
{
    /*nouveaux membres*/
};
```

Le lecteur attentif aura remarqué l'emploi des mots public, protected, ou private. Nous n'utilisons que la forme `public` pour le moment et parlons alors d'héritage publique. Les autres types d'héritage sont brièvement abordés dans la section suivante.

La notion d'héritage présente les intérêts suivants :

1. Le code factorisé est plus court : les propriétés communes ne sont pas écrites plusieurs fois.
2. le code factorisé est plus lisible : la structure d'héritage participe à une meilleur compréhension du code. En particulier, elle indique les structures de dépendances ou de similarités entre classes.
3. Le code factorisé est plus extensible : il suffit d'ajouter ou de modifier un champ ou une méthode dans la classe mère pour en faire bénéficier toutes les classes filles.
4. Le code factorisé est plus maintenable : lorsque nous voulons modifier le code d'une méthode non factorisée mais dupliquée au sein de plusieurs classes, il faut la modifier dans chacune des classes de la même manière exactement et dans toutes ces classes, faute de quoi les méthodes qui étaient les mêmes commencent à différer. Lorsque le code est factorisé, il suffit de le modifier une seule fois.

Nous pouvons appeler les méthodes de la classe mère `Vehicule` comme `double GetPrice(void)` depuis un objet `Bike` ou `Car` comme si ces méthodes étaient implémentées dans les classes filles. Par exemple, le code-ci dessous fonctionnera correctement :

```
1 Bike b;
2
3 cout << b.GetPrice();
```

12.1.3 Le mot clef "protected"

Lors de notre introduction à l'encapsulation, nous avions vu deux mots clés : `private` et `public`. `private` servait à interdire au reste du monde d'accéder à certains membres, et `public` servait à autoriser le reste du monde à accéder à certains membres.

Que se passe-t-il si nous voulons dans une méthode fille accéder à un champ `private` de la classe mère ? Supposons par exemple que nous ajoutions une méthode `void DisplayCost()` dans la classe Car, définie de la manière suivante :

Accès	public	protected	private
Membres de la classe	Oui	Oui	Oui
Membres des classes dérivées	Oui	Oui	Non
Reste du monde	Oui	Non	Non

TABLE 12.1 – Encapsulation : différents degrés de visibilité (cas de l'héritage publique)

```
1 void Car::DisplayCost(void)
2 {
3     cout << "Cost of this car is : " << _price << " euros.\n"
4 }
```

Si nous compilons le code présenté, le compilateur va nous donner une erreur :

```
'_price' : cannot access private member declared in class
'Vehicule'
```

Que signifie cette erreur ? Son origine est l'inaccessibilité/l'invisibilité de la variable `_price` dans les méthodes de la classe `Car`. Nous avons en effet implicitement supposé que nous avions accès dans nos classes dérivées aux membres privés de la classe mère. Ce n'est pas le cas.

Dans le cadre de l'héritage (publique) que nous traitons depuis le début, nous avons donc le comportement suivant :

- Les membres `private` de la classe mère ne sont *pas* accessibles à la classe dérivée.
- Les membres `public` de la classe mère sont accessibles à la classe dérivée.

Nous ajoutons maintenant un niveau intermédiaire dans lequel les champs de la classe mère sont accessibles à la fois dans les méthodes de la classe mère et des classes filles, mais qui ne sont pas accessibles de l'extérieur de ces classes (comme par exemple dans le main). Le mot clef qui permet de définir ce niveau intermédiaire est le mot `protected`.

L'ensemble de ces règles d'héritage est résumé dans le tableau 12.1.

Pour que nos classes dérivées aient accès au membre `_price`, nous avons donc deux possibilités :

- Utiliser l'accesseur `GetPrice` ;
- Mettre `_price` en `protected`.

Nous allons retenir la seconde solution, et notre classe `Vehicule` s'écrira alors :

```
#ifndef VEHICULE_H
#define VEHICULE_H

class Vehicule
{
    public :
        Vehicule(int state = 1,
      int color = 0,
      int price = 10000);
        ~Vehicule();

        void SetState(int state);
        int GetState();
        void SetColor(int color);
        int GetColor(int color);
        int GetPrice();
        void SetPrice(int price);

    protected:
        int _state; //new?
        int _color;
        int _price;

};

#endif
```

Listing 12.6 – vehicule3.h

12.1.4 Constructions et destructions d'objets filles

Dotons-nous tout d'abord d'une classe mère avec deux constructeurs distincts, pour lesquels une sortie console est affichée à chaque appel, afin de savoir quand et quel constructeur a été appelé.

```
#ifndef MOTHER_H
#define MOTHER_H

class Mother
{
  public :
    Mother();
    Mother(int);
    ~Mother();

  private:
    int _id;
}

#endif
```

Listing 12.7 – Mother.h

```
#include "Mother.h"
#include <iostream>
using namespace std;

Mother::Mother()
{
  cout << "Default mother constructor called \n";
}

Mother::Mother(int id)
{
  _id = id;
  cout << "1-argument mother constructor called \n";
}

Mother::~Mother()
{
  cout << "Mother destructor called \n";
}
```

Listing 12.8 – Mother.cpp

Nous créons alors une classe Child qui dérive publiquement de Mother.

```
#ifndef CHILD_H
#define CHILD_H

class Child : public Mother
{
  public :
    Child();
    ~Child();
}

#endif
```

Listing 12.9 – Child.h

```
#include "Child.h"
#include <iostream>
using namespace std;

Child::Child()
{
  cout << "Child empty constructor called \n";
}

Child::~Child()
{
  cout << "Child destructor called \n";
}
```

Listing 12.10 – Child.cpp

Lorsque nous voulons construire un objet fille dans notre main, via par exemple le code suivant :

```
#include "Child.h"

void f()
{
    Child c;
}

void main()
{
    f();
}
```

Nous pouvons lire après la fin de la fonction main dans la console les lignes suivantes :

```
  Default mother constructor called
Child empty constructor called
Child destructor called
Mother destructor called
```

Nous observons ainsi que la construction d'un objet fille appelle implicitement un constructeur mère avant d'appeler le constructeur de la classe fille. De même, à la destruction de notre objet, le destructeur de la classe mère est appelé après le destructeur de la classe fille.

A la construction de notre objet fille, quel est le constructeur mère appelé ? Si nous ne le spécifions pas, c'est le constructeur mère par défaut qui est appelé, comme nous le voyons par le message de la console. Il est cependant possible d'expliciter quel constructeur de la classe mère nous voulons utiliser au début de la définition de notre constructeur fille. Ainsi, nous pouvons modifier de la sorte le constructeur fille pour que ce soit le constructeur mère avec 1 argument qui soit appelé :

```
#include "Child.h"
#include <iostream>
using namespace std;

Child::Child(int id) : Mother(id)
{
  cout << "Child empty constructor called \n";
}

Child::~Child()
{
  cout << "Child destructor called \n";
}
```

Listing 12.11 – Child2.cpp

Si vous ne spécifiez pas explicitement le constructeur mère à appeler lorsque vous construisez une fille, nous avons vu que c'était le constructeur mère par défaut qui était appelé. Dans un tel cas, et si le constructeur mère par défaut n'existe pas (par exemple si vous avez déclaré un seul constructeur dans Mother.h qui prend en argument des paramètres), alors votre IDE échouera à compiler, vous spécifiant un message de la sorte :

`Error C2512: 'Mother' : no default appropriate constructor available in child.cpp (l5)`

12.2 Autres héritages

12.2.1 Héritage protected et private

Nous avions mentionné dans la section précédente qu'il était possible de déclarer une classe dérivée au moyen de trois mots clés différents : `private`, `protected`, et `public`. Jusqu'à présent, nous n'avons employé que le mot clé `public`. Que signifie-t-il précisément ?

En réalité, chacun des mots clefs (public, private ou protected) agit comme une sorte de filtre sur la visibilité des membres de la classe mère. Pour ce point, la manière la plus claire de définir ces différents filtres est d'en donner un exemple :

```
1  class Mother
2  {
3      public:
4          int x;
5      protected:
6          int y;
7      private:
8          int z;
9  };
10
11 class Child1 : public Mother
12 {
13     // x is public
14     // y is protected
15     // z is not accessible from Child1
16 };
17
18 class Child2 : protected Mother
19 {
20     // x is protected
21     // y is protected
22     // z is not accessible from Child2
23 };
24
25 class Child3 : private Mother
26 {
27     // x is private
28     // y is private
29     // z is not accessible from Child3
30 };
```

Quand utiliser de l'héritage protected ou private ?

Il y a encore débat aujourd'hui, mais une réponse assez communément acceptée est : jamais. L'héritage protected ou private sert à pouvoir utiliser quelquechose sans en donner directement accès de l'extérieur. Il est en celà très proche de la composition, détaillée ci-dessous. Nous recommendons de passer plutôt par de la composition que par un héritage protected ou private. Pour trouver des exemples pertinents d'héritage private, nous renvoyons le lecteur par exemple à la programmation par "Policies" d'Alexandrescù [3], mais c'est un exemple qui excède largement notre propos.

12.2.2 Héritage multiple et virtuel

Nous avons vu dans la section précédente que l'héritage permet de décrire une relation "est un" entre deux objets. Cependant, comment faire dans le cas où un objet "est un" X *et* un Y ? L'héritage multiple permet de résoudre ce problème. Avant de poursuivre, il nous semble important de préciser que l'héritage multiple est aussi *dangereux*[1] *que toxique*, pour des raisons qui deviendront claires par la suite mais qui ont déjà été esquissées dans la citation de début de chapitre.

Principe

L'idée est assez naturelle : nous allons faire hériter notre classe dérivée de *deux* classes mères. D'un point de vue syntaxique, on écrira :

```
1  class Child :
2    public|protected|private class Mother1,
3    public|protected|private class Mother2,
4    . . .
5    public|protected|private class MotherN
6  {
7  }
```

Par exemple, considérons les classes de véhicules précédentes, et ajoutons une nouvelle classe représentant un avion (ou tout autre

1. Il est considéré comme suffisamment dangereux pour être explicitement interdit dans les langages récents comme le JAVA ou le C#. Le manque d'interface en C++ ne laisse maheureusement parfois pas d'autre choix que d'utiliser un héritage multiple. Nous ne saurions trop vous inciter néanmoins à l'utiliser avec la plus grande circonspection

objet volant), qui va également dériver de `Vehicule`. Celui-ci aura une nouvelle propriété qui sera l'altitude maximum à laquelle le véhicule peut voler.

```
#include "vehicule4.h"
class FlyingVehicle : public virtual Vehicle
{
  public:
    FlyingVehicle();
    ~FlyingVehicle();

    void SetMaxAltitude(int maxAltitude);
    int GetMaxAltitude();

  private:
    int _maxAltitude;
}
```

Listing 12.12 – Véhicule volant

Supposons à présent que nous souhaitions manipuler une voiture volante. Une telle voiture est à la fois un véhicule volant et une voiture. Nous pouvons donc avoir recours à l'héritage multiple et écrire :

```
#include "voiture4.h"
#include "avion4.h"

class FlyingCar : public Car, public FlyingVehicle
{
  public:
    FlyingCar();
    ~FlyingCar();
}
```

Listing 12.13 – voiturevolante.h

Nous pouvons représenter l'ensemble des relations entre nos classes sur le diagramme de la figure 12.1.

Comme aurait pu dire Vinz dans La Haine, "Jusqu'ici, tout va bien", et les choses nous paraissent assez naturelles. Cependant, nous pouvons remarquer que `FlyingCar` dérive de `Car` et `FlyingVehicle`, qui dérivent elles-même de la même classe de base `Vehicle`. Si nous regardons le dessin formé par les boîtes décrivant les objets et les flèches les reliant (figure 12.1), nous constatons que l'ensemble forme - au moins approximativement - un losange, qui prend le nom de "Deadly Diamond of Death".

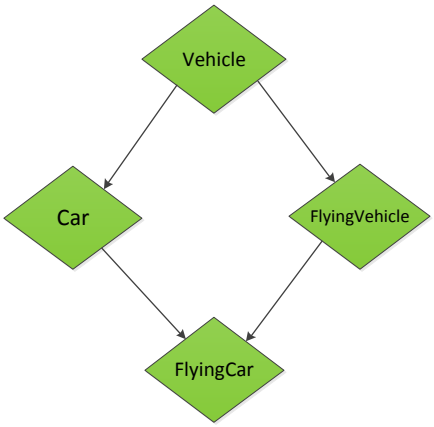

FIGURE 12.1 – Héritage multiple, le diamant

C'est là que se trouve la difficulté principale de l'héritage multiple, que nous explicitons à présent.

Deadly Diamond of Death

Le problème est le suivant : le champ `_color` a été déclaré dans la classe `Vehicle`, et il est donc hérité dans les classes `FlyingCar` et `Car`. Supposons maintenant que nous ajoutions une méthode à `FlyingCar` qui fasse appel à ce champ :

```
1 void VehiculeVolant::DisplayColor()
2 {
3   cout << "The color of our new brand flying car is " << \_color << ".\n"
        ;
4 }
```

À la compilation, cette méthode va provoquer une erreur. Pourquoi ? Le problème est que le champ `_color` qui a été défini dans les classes mères est hérité *deux fois* (une par l'héritage de `Car` et une par l'héritage de `FlyingVehicle`). L'appel à ce champ dans `FlyingCar` est donc ambigu. Le C++ nous fournit une technique permettant de lever cette ambigüité, technique qui porte le nom d'*héritage virtuel*.

Héritage virtuel

La technique consiste simplement à préciser au compilateur qu'il ne doit hériter des champs et des méthodes de la classe `Vehicle` qu'une seule fois, et non deux. Cela se fait au moyen du mot clé `virtual` que l'on écrira devant le nom de la classe dont on hérite :

```
class child : public virtual Mother
{
}
```

Les headers de nos classes `FlyingVehicle` et `Car` deviendront donc :

```
#include "vehicule4.h"
class FlyingVehicle : public virtual Vehicle
{
  public:
    FlyingVehicle();
    ~FlyingVehicle();

    void SetMaxAltitude(int maxAltitude);
    int GetMaxAltitude();

  private:
    int _maxAltitude;
}
```

Listing 12.14 – vehiculeVolant4.h

```
#include "vehicule5.h"

class Car : public virtual Vehicle
{
    public:
        Car(int state = 1, int fuel = 100,
    int color = 0, int price = 10000);
        ~Car();

        void SetFuel(int fuel);
        int GetFuel();

    private:
        int _fuel;
};
```

Listing 12.15 – voiture5.h

12.3 Composition et Aggrégation

12.3.1 Composition

Jusqu'à présent, toutes nos classes possédaient des champs publics ou privés de type primitif (comme des int, des double, des boolean). En règle générale, nous voulons avoir plus d'expressivité et pouvoir disposer de champs de tout type. Pour faciliter la construction de classes avancées à partir de classes plus élémentaires, le C++ permet justement de composer les objets, c'est-à-dire de fournir dans une classe des champs du type d'une autre classe. Ainsi, nous pourrons écrire :

```
1  class A
2  {
3      public:
4          A();
5          ~A();
6  };
7
8  class B
9  {
10     public:
11         B(A a);
12         ~B();
13
14     private:
15         A _a;
16 };
17
18 B::B(A a)
19 {
20     _a = a;
21 }
```

Si l'héritage exprime une relation de "Est un" entre la classe mère et la classe fille, la composition permet ainsi d'exprimer la relation "Possède un" entre deux classes.

Il est important de noter que l'objet possédant (`B` dans notre exemple) "détient" les objets possédés. Lorsque l'objet possédant (souvent désigné sous le terme d'objet composite) est construit, le paramètre a est passé par valeur, le constructeur-copie est appelé, et `B` détient alors une copie de `a`, notée `_a`. De la même manière,

lorsque l'objet possédant (`B`) est détruit, les objets possédés (`_a`) sont détruits simultanément.

12.3.2 Aggrégation

Un cas un peu plus subtil apparaît quand nous ne voulons pas de la notion de possession décrite dans le cas de la composition. Nous pouvons alors dans notre classe `B` posséder un pointeur sur `A`, plutôt qu'une instance de la classe `A`.

```
1  class A
2  {
3      public:
4          A();
5          ~A();
6  };
7
8  class B
9  {
10     public:
11         B(A* pa);
12         ~B();
13
14     private:
15         A* _pa;
16 };
17
18 B::B(A* pa)
19 {
20     _pa = pa;
21 }
```

Dans ce cas, l'instance `a` a une existence indépendamment de la construction, de l'existence, ou de la destruction des instances de classe `B`.

En règle générale, l'aggrégation est préférée à la composition pour éviter de copier des objets, ou lorsqu'il est manifeste que les deux types d'objets doivent avoir des existences qui ne sont pas subordonnées l'une à l'autre.

Il existe un cas où l'aggrégation est indispensable, c'est lorsqu'une classe `A` doit possèder des champs de type A. Dans ce contexte, une composition mènerait par récursion à une création infinie d'instances de type `A` (en construisant la première instance, il faudrait construire la deuxième instance qui est le champ de la première,

pour construire la deuxième instance il faudrait construire la troisième qui est le champ de la deuxième, etc.). Nous obtenons alors le code suivant :

```
class A
{
    public:
        A(A* pa);
        ~A();

    private:
        A* _inner;
};

A::A(A* pa)
{
    _inner = pa;
}
```

Donnons deux exemples plus concrets de cas d'utilisation.

Le premier exemple est une classe `User` correspondant aux informations stockées par utilisateur dans un réseau social. Une des raisons principales des réseaux sociaux étant de pouvoir déclarer au monde que nous sommes passés d'un statut de "single" à "it's complicated with `Trucmuche`", nous voulons pouvoir stocker dans chaque instance un pointeur vers `Trucmuche`. Le code `??` en donne un aperçu.

```
class FbUser
{
    public:
        FbUser(string name, int id, string status, FbUser* pa);
        ~FbUser();

    private:
        FbUser* _mate;
        string _name;
        string _status;
        int _id;
};
```

Le second exemple est la construction récursive d'un arbre binaire (schématisé dans le graphique 12.2). Dans ce premier exemple, chaque noeud de type `Node` détient deux pointeurs vers ses deux

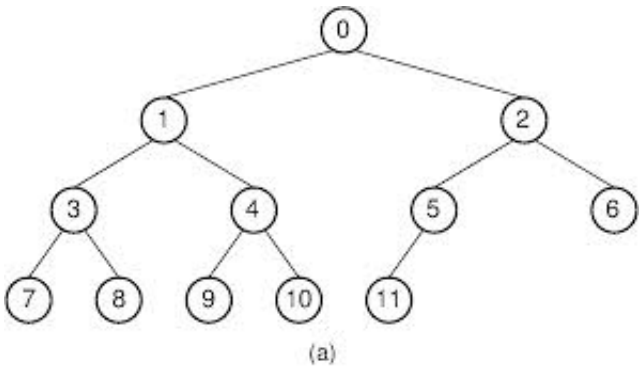

FIGURE 12.2 – Exemple d'arbre binaire

noeuds fils (`_left` et `_right`). Dans le cas d'un noeud sans fils (i.e. une feuille), les pointeurs valent `NULL`. Dans cet architecture, un arbre est donc une structure possédant un pointeur sur le premier noeud (la racine).

```
class Node
{
    public:
        Node(Node* left, Node* right);
        ~Node();

    private:
        Node* _left;
        Node* _right;
};
```

Par abus de langage, les concepts de Composition et d'Aggrégation sont souvent confondus sous le terme Composition.

12.3.3 Dépendances cycliques et Déclarations Forward

Il peut arriver qu'une classe `A` contienne un champ (ou un argument d'une méthode) de type classe `B` et que la classe `B` contienne elle aussi un champ (ou un argument d'une méthode) de type classe `A` [2]. Nous avons alors une dépendance cyclique. Si nous incluons A.h dans B.h et B.h dans A.h, nous avons un problème d'inclusion cyclique et la compilation échouera.

2. Nous pouvons complexifier à volonté ce design, par exemple avec `A` qui dépend de `B`, `B` qui dépend de `C` et `C` qui dépend de `A`

La solution à ce problème est d'utiliser des déclarations forward. Au lieu d'inclure le header déclarant `A` dans B.h ou dans B.cpp, nous déclarons seulement le type `A` pour indiquer au compilateur que ce type existe. Cette technique fonctionne en raison de la manière dont travaille le compilateur, mais c'est un "workaround" qui n'est plus nécessaire dans les nouveaux langages comme le Java ou le C#.

```
1 class B;
2
3 class A
4 {
5     B* PtrB;
6 };
```

Listing 12.16 – A.h

```
1 #include "A.h"
2 #include "B.h"
3
4 // ...
```

Listing 12.17 – A.cpp

```
1 #include "A.h"
2
3 class B
4 {
5     A a;
6 };
```

Listing 12.18 – B.h

```
1 #include "B.h"
2
3 // ...
```

Listing 12.19 – B.cpp

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." James Whitcomb Riley.

"One issue with duck typing is that it forces the programmer to have a much wider understanding of the code he or she is working with at any given time. In a strongly and statically typed language that uses type hierarchies and parameter type checking, it's much harder to supply an unexpected object type to a class. For instance, in Python, you could easily create a class called Wine, which expects a class implementing the "press" attribute as an ingredient. However, a class called Trousers might also implement the press() method. With Duck Typing, in order to prevent strange, hard-to-detect errors, the developer needs to be aware of each potential use of the method "press", even when it's conceptually unrelated to what he or she is working on. In essence, the problem is that, "if it walks like a duck and quacks like a duck", it could be a dragon doing a duck impersonation. You may not always want to let dragons into a pond, even if they can impersonate a duck."

http ://en.wikipedia.org/wiki/Duck_typing

13 Polymorphisme

Le polymorphisme est une fonctionnalité d'un langage permettant de manipuler des méthodes non pas par elles-mêmes, mais par une abstraction qui leur est commune. L'importance du polymorphisme dans les langages récents est telle que la manière dont un langage appréhende ce concept est un bon indicateur de la philosophie de ce langage. Le C++ propose deux voies pour le polymorphisme : statique et dynamique. Historiquement, le polymorphisme dynamique a été implémenté dans la spec du C++ bien avant le polymorphisme statique. Plus simple à mettre en place, et plus intuitif, le polymorphisme dynamique est une bonne introduction au polymorphisme. Le polymorphisme statique, introduit après les templates, a souvent la faveur des développeurs avancés en C++, lorsque les performances sont un enjeu.

13.1 Polymorphisme dynamique

13.1.1 Factorisation de code

Nous possédons trois classes A, B, C chacune implémentant une méthode Display, de même prototype (void Display(void) par exemple), et dont le corps contient **les mêmes instructions**. Nous souhaitons implémenter une méthode f qui prenne en argument une instance d'une de ces classes, et qui appelle la méthode Display de cette instance. Nous voudrions pouvoir écrire quelquechose de la sorte :

```
1  void main(void)
2  {
3      A a;
4      B b;
5      C c;
6      f(a);
7      f(b);
8      f(c);
9  }
10
11 void f(? instance)
12 {
13     instance.Display();
14 }
```

Sous cette forme, notre fonction f doit pouvoir prendre en argument une instance de type A, de type B, ou de type C. Il serait possible d'implémenter 3 fonctions f (surcharge de fonctions) : la première qui prenne en argument une instance de type A, la seconde une instance de type B, la troisième une instance de type C. Ceci reviendrait à copier/coller du code, et c'est tout à fait inacceptable :

- Dupliquer du code, c'est courrir le risque qu'une implémentation soit modifiée et non les autres. Pour assurer une bonne maintenabilité du code, il ne FAUT PAS copier de code (il y a bien sûr des exceptions, notamment pour éviter certaines dépendances entre librairies, mais ceci dépasse le cadre de notre cours).

- Ceci augmenterait considérablement le volume du code, qui doit s'efforcer d'être le plus court possible.

- Il faudrait réitérer l'opération à chaque nouvelle classe possédant une méthode Display.

Nous devons donc établir un procédé pour que ce code puisse être appliqué à toute classe possédant la méthode Display, mais que ce code ne soit écrit qu'une seule fois, c'est à dire que nous voulons **factoriser** une partie du code, pour isoler la partie de la logique commune à chaque classe.

Nous avons vu au chapitre sur l'héritage que si nous avons deux types Derived et Base avec une structure d'héritage telle que Derived hérite de Base, alors nous pouvons considérer une instance de

type Derived comme une instance de type Base. Dans l'exemple précédent, il est donc possible d'écrire une classe Base possédant une méthode Display. Nous pouvons alors faire hériter publiquement les classes A, B, et C de la classe Base. Ce faisant, chaque instance de type A, B ou C possédera une méthode Display (héritée de la classe Base), alors que le code n'a été écrit qu'une seule fois : nous venons de factoriser du code.

Dans ces conditions, nous pouvons écrire notre méthode f comme une méthode prenant en argument une instance de type Base, et qui appelle la méthode Display de cette classe :

```
1  void main(void)
2  {
3      A a;
4      B b;
5      C c;
6      f(a);
7      f(b);
8      f(c);
9  }
10
11 void f(Base instance)
12 {
13     instance.Display();
14 }
```

13.1.2 Redéfinition de méthodes dans les classes filles et slicing

Cette partie doit être relue jusqu'à être parfaitement comprise.

Reprenons l'exemple précédent. Nous souhaitons maintenant pouvoir fournir des comportements différents pour la méthode Display selon le type de l'instance appelante. Par exemple, nous pouvons souhaiter que chaque instance retourne dans la console son type. Dans cet exemple classique, il est nécessaire que la méthode Display ait un comportement qui dépende du type dans lequelle elle est appelée. Nous proposons d'étudier le code suivant :

```
void Base::Display()
{
    cout << "I'm a Base instance. \n";
}

void A::Display()
{
    cout << "I'm a A instance. \n";
}

void B::Display()
{
    cout << "I'm a B instance. \n";
}
```

Reprenons alors le code proposé précédemment :

```
void main(void)
{
    A a;
    B b;
    Base c;
    f(a);
    f(b);
    f(c);
}

void f(Base instance)
{
    instance.Display();
}
```

Nous obtenons la sortie suivante :

```
I'm a Base instance.
I'm a Base instance.
I'm a Base instance.
```

Ce résultat est inattendu. En effet, nous nous attendions à ce que l'appel de la méthode Display sur les instances a et b de types respectifs A et B retournent des résultats différents. Ce résultat est en fait dû à une propriété assez pénible du C++ [1], qu'on appelle **le Slicing**. Lorsque nous passons l'instance a à notre méthode f, nous passons a par copie, c'est à dire qu'un constructeur-copie va être appelé pour réaliser une copie de l'instance a. Puisque f prend en argument une instance de type Base, il n'est pas possible dans le

1. On trouvera des personnes de mauvaise foi qui affirment encore aujourd'hui que c'est un comportement naturel, méfiez-vous de ces individus subversifs.

corps de f d'appeler une méthode de la classe A par exemple, qui ne serait définie ni dans Base ni dans B. Par économie de mémoire [2], le compilateur va donc appeler le constructeur-copie de la classe Base, et non de la classe A, ne conservant ainsi dans la copie de A en Base que les informations nécessaires à la construction de l'instance de type Base. Les informations supplémentaires que contenaient a, comme la redéfinition de la méthode Display sont perdues. C'est pour cette raison que nous obtenons une sortie dans la console en contradiction avec nos attentes.

13.1.3 Passage par pointeur

Pour lutter contre ce phénomène de slicing, nous allons être plus précautionneux, et éviter le passage d'argument par copie. Nous modifions donc le prototype de notre méthode f, pour qu'elle prenne en argument non plus une instance de type Base, mais un pointeur vers une instance de type Base.

```
1  void main(void)
2  {
3      A a;
4      B b;
5      Base c;
6      f(\&a);
7      f(\&b);
8      f(\&c);
9  }
10
11 void f(Base* pInstance)
12 {
13     pInstance->Display();
14 }
```

La substitution que nous venons de réaliser empêche la création de nouvelles instances, qui seraient tronquées en classe Base. Cependant, la sortie que nous obtenons dans notre console reste la même :

```
1  I'm a Base instance.
2  I'm a Base instance.
3  I'm a Base instance.
```

Que s'est-il passé ? Le premier pointeur que vous avons fourni à f (&a), pointe bien sur une instance de type A, et non uniquement Base. Nous pouvons donc accéder à la fois par cette instance à la

2. Le débat est assez technique, mais cette position se justifie en partie, pour optimiser les caches L1d et L2d des processeurs...

méthode Display de la classe mère et de la classe fille A. Pourquoi alors la méthode utilisée est-elle celle de la classe Base ?

Il faut revenir à la manière dont fonctionne le compilateur et l'éditeur de liens. Lors de la compilation et de l'édition des liens, l'environnement détermine que la méthode f appelle une méthode définie dans l'instance pointée par le pointeur passé en argument. Ensuite, l'éditeur des liens va s'arranger pour que la bonne méthode soit appelée. Problème : l'éditeur des liens est appelé à la compilation et non à l'exécution, il ne peut donc pas faire varier son comportement en fonction de l'état de certaines variables. Comme nous passons en argument un pointeur Base*, l'environnement ne voit qu'un pointeur de type Base*. Lorsque nous donnons &a comme argument, le compilateur réalise une conversion implicite de A* vers Base* (ce qui est demandé dans le prototype de la méthode f). L'éditeur des liens ne voit donc en &a uniquement qu'un pointeur vers Base, il ne possède pas l'information évidente pour nous que le pointeur pointe en réalité vers une instance de type A. Que peut alors faire l'éditeur des liens ? Il dispose d'un pointeur vers Base, qui peut être un pointeur vers Base, mais aussi vers A ou vers B, mais il n'en sait rien. La seule réponse raisonnable qu'il peut alors fournir est de considérer que la méthode à appeler est celle de la classe mère, qui sera quoi qu'il arrive disponible dans l'instance pointée. **Le problème vient donc du fait que la résolution de la méthode à appeler est réalisé statiquement (compilation), alors que le type exact de la méthode qu'il faudrait appeler ne peut être connu par la machine qu'à l'exécution**.

13.1.4 Virtualité

Nous introduisons maintenant le mot-clef du C++ réservé à ce problème : virtual. Le mot clef virtual informe l'environnement que la résolution de la méthode Display (c'est à dire le choix de la méthode Display entre les 3 disponibles) doit être repoussé au moment de l'exécution du code, et non pas de la compilation. Ce mot clef est à placer entre l'indicateur de portée de la méthode (public, protected, private) et le type de retour de la méthode. Voici notre exemple achevé :

```
class Base
{
    public Base();
    public ~Base();
    public virtual void Display(void);
};

class A : public Base
{
    public A();
    public ~A();
    public void Display(void);
};

class B : public Base
{
    public B();
    public ~B();
    public void Display(void);
};
```

Nous obtenons alors par l'appel successif proposé dans les exemples précédents :

```
I'm a A instance.
I'm a B instance.
I'm a Base instance.
```

Conclusion : dans l'exemple précédemment traité, nous ne voulions écrire qu'une fonction f, prenant en argument un pointeur vers une instance de type Base, A ou B. Pour avoir une unique fonction, nous avons défini f comme prenant un argument de type Base*, et avons alors casté (implicitement) le pointeur &a vers un pointeur de type Base*, afin que celui-ci soit compatible avec le prototype de notre méthode void f(Base* pBase). Ce cast implique que la connaissance de la méthode qu'il faut réellement appeler est perdue à la compilation (l'environnement ne voit qu'un pointeur sur Base* là où nous pointons par exemple sur un A), et que cette connaissance n'est effectivement récupérée qu'à l'exécution, quand le pointeur est déréférencé, et que nous observons le type réel de l'instance pointée. Pour obtenir de l'environnement qu'il repousse la résolution de la méthode à appelé à l'exécution, nous ajoutons dans le prototype de la classe-mère le mot clef virtual.

13.1.5 Virtualité Pure, classes abstraites et interfaces

Reprenons le problème du début de chapitre, avec l'éclairage du polymorphisme dynamique que nous avons déjà étudié. Nous somme dans le cadre d'un projet où nous voulons faire un jeu vidéo dans l'univers de Star Wars. Nous disposons de deux classes Wookie et MilleniumFalcon chacune implémentant une méthode void Display(void), de logiques différentes.

```
class Wookie
{
    public Wookie();
    public void Display(void);
};

class MilleniumFalcon
{
    public MilleniumFalcon();
    public void Display(void);
};
```

Pour factoriser ces deux classes et pouvoir considérer les deux classes comme d'un seul type, nous introduisons une classe Sprite, possédant également une méthode Display (virtuelle) :

```
class Sprite
{
    public virtual void Display();
};
```

Pour définir la méthode Display de la classe Sprite, nous nous retrouvons face à un problème : un sprite n'a pas par défaut une manière canonique de s'afficher à l'écran. Il n'y a donc pas de sens à implémenter cette méthode pour la classe mère ; nous avons besoin de l'existence de cette méthode dans la classe mère (pour signifier au compilateur que les classes filles posséderont cette méthode), mais nous n'avons pas besoin de sa logique même. Le C++ propose une solution : la méthode virtuelle pure. La syntaxe pour déclarer une méthode virtuelle comme pure est de placer à la fin de sa déclaration le signe =0.

```
class Sprite
{
    public virtual void Display()=0;
};
```

Lorsqu'une méthode est déclarée comme virtuelle pure, il n'est plus nécessaire d'implémenter le corps de la fonction. Cependant, une classe possédant au moins une méthode virtuelle pure ne peut pas être instanciée : puisqu'au moins une méthode de cette classe est virtuelle pure, c'est qu'aucun sens ne peut être donné à une instance de la classe mère, qui n'est donc pas instanciable : on parle alors de classe abstraite. Dans notre cas présent, tout élément à afficher est d'un certain type, il n'y a pas d'objet à afficher qui ne soit pas wookie, ou milleniumFalcon.

En C++, le langage n'offre pas la possibilité de créer des interfaces, mais la création de classes abstraites les remplace souvent. En créant des classes dont certaines méthodes sont virtuelles pures, nous assurons que toute classe héritant de cette classe mère et pouvant être instanciée implémente les méthodes dont le prototype est donné dans la classe abstraite.

Remarque : si une classe A possède des méthodes virtuelles pures, une classe B en héritant a deux alternatives :

1. implémenter toutes les méthodes virtuelles pures de la classe A et pouvoir être instanciée
2. ne pas implémenter toutes les méthodes virtuelles pures de la classe A, et attendre qu'une classe C hérite de B et implémente les méthodes qui ne l'étaient pas encore. Dans ce cas, seule la classe C pourra être instanciée, les classes A et B restant abstraites.

Remarque : comme une classe abstraite ne peut pas être instanciée, si la classe A est abstraite il ne sera pas possible d'écrire :

```
1 void f(A a)
2 {
3 }
```

En effet, l'argument de f étant passé par copie, il faudrait appeler le constructeur-copie d'une classe abstraite pour l'instancier, ce qui n'est pas possible. Nous passerons donc toujours par pointeur :

```
1 void f(A* pa)
2 {
3 }
```

13.1.6 Coût de la virtualité

Le polymorphise dynamique du C++ a été délaissé dans les librairies scientifiques depuis une dizaine d'années au profit du polymorphisme statique que nous aborderons à la section suivante en raison de ses performances moindres. La raison de cette performance moindre est triple, nous présentons ces raisons de la moins sérieuse à la plus importante.

Virtual table pointer

Si une classe possède une méthode virtuelle, alors des données supplémentaires sont ajoutées dans la classe. Ces données permettent au programme exécuté de savoir à quelles méthodes il doit faire appel dans le cas d'un héritage. Il s'agit du vptr (virtual table pointer). C'est une table virtuelle qui contient des pointeurs vers les fonctions virtuelles de la classe. La taille de ce pointeur est de 32 bits pour un processeur 32 bits, etc. La taille de la classe sera alourdie de la taille de ce pointeur. Cette taille est minime, mais imaginez que votre classe ne contienne qu'un entier, une méthode dynamique va doubler son poids en mémoire.

Indirection

A chaque appel d'une méthode virtuel, le programme doit résoudre la méthode à appeler. Cette résolution est appelé indirection. L'indirection étant réalisée au run-time à chaque appel de la méthode et non à la compilation, le programme peut s'en trouver légèrement ralenti.

Inlining

Nous pouvons lire dans beaucoup de manuels que le principal défaut du polymorphisme dynamique est le coût de la résolution au run-time de la méthode exacte à appeler. Ce coût (très variable et complexe à estimer) est en général assez faible, et bien souvent négligeable devant le temps d'exécution de la méthode en elle-même, surtout lorsque la méthode appelée comporte au moins une dizaine

d'instructions. Cependant, il y a un vrai défaut du polymorphisme dynamique : c'est l'impossibilité de recourir à l'inlining. Lorsque vous utilisez de la virtualité, la détermination de la méthode à appeler se faisant au run-time, il est impossible au précompilateur de recourir à l'inlining de votre méthode. Cette impossibilité d'inlining dans le cas de polymorphisme dynamique est particulièrement dommageable dans le cas de méthodes très courtes.

Dans la vie de tous les jours, ceci s'observe régulièrement :

1. Dans le cas d'un pricer, on peut souvent lire dans des projets débutants la création d'une classe BaseOption possédant une méthode Payoff virtuelle pure, et une dizaine de classes héritant de BaseOption (EuropeanCall, AsiaticPut, LookBack, Barrier, ...) et implémentant chacune le payoff correspondant. Par l'usage de la virtualité, vous vous privez de la possibilité d'inliner ces méthodes payoff. Le gain à passer par du polymorphisme statique et de l'inlining est ici très conséquent, mais ne doit être mis en place dans un véritable projet que si contrainte de performance il y a.

2. Dans le cas d'algorithmes de datamining, comme un programme de recherche de proches voisins (KNN) dans un jeu de données dans $\mathbb{R}^D$, nous pouvons être tentés d'utiliser de la virtualité pour manipuler différentes métriques : une classe BaseMetric, abstraite, et différentes classes en héritant et implémentant des métriques L_∞, L_1,L_2 ... Là encore, dans beaucoup d'algorithmes ces calculs de métriques vont être utilisés de très nombreuses fois, pour représenter une charge importante des calculs. La virtualité est ici à proscrire.[3]

La virtualité étant parfois source de bugs pénibles (le lecteur peut s'imaginer les longues soirées d'hiver à traquer une méthode non déclarée comme virtuelle qui est mal appelée), certains langages récents ont choisi de mettre par défaut toutes les méthodes de toutes les classes en virtuel (c'est le cas de Java), optimisant le compilateur (ou plutôt la JVM dans le cas de Java), pour compenser partiellement cette perte de performance. D'autres langages, comme le C#, n'ont pas fait ce choix.

3. Nous renvoyons le lecteur intéressé par des benchmarks sur le sujet à :
`http://matthieudurut.com/2013/06/19/cost-of-dynamic-polymorphism-in-c-the-neighbors-search-example/`

13.1.7 Virtualité et Destructeurs

Il est nécessaire de rendre le destructeur d'une classe de base virtuel quand celle-ci est destinée à être détruite polymorphiquement, c'est à dire dès lorsqu'un pointeur de type pointeur sur classe mère sera utilisé dans un delete pour détruire une instance de classe fille. Dans l'exemple suivant, l'appel de delete sur pB entraine l'appel du destructeur de la classe A au lieu d'appeler le destructeur de la classe B.

```
1  class A
2  {
3      public A();
4      public ~A();
5  };
6
7  class B : public A
8  {
9      public B();
10     public ~B();
11 };
12
13 void main()
14 {
15     B* pB = new B();
16     delete pB; //Le destructeur de A est appelé, en lieu et place du destructeur
           de B, car le destructeur de A n'a pas été marqué comme virtuel
17 }
```

Une solution simple consisterait à rendre les destructeurs de chaque classe virtuels. Cependant, ce serait une erreur, tant d'un point de vue performance (cf paragraphe précédent, et notamment quand la classe en question est une petite structure de donnée destinée à être instanciée/détruite de nombreuses fois), qu'en terme de sémantique (si la classe n'a aucune classe mère et classe fille, celà n'a pas de sens d'utiliser la virtualité).

Nous proposons une solution communément admise :

- Toute classe ayant au moins une fonction virtuelle doit déclarer son destructeur virtuel.
- Lorsqu'une classe n'a aucune classe fille et n'hérite d'aucune classe, laisser son destructeur non-virtuel.

Cette solution n'est pas parfaite, car le code que vous écrivez est destiné à être utilisé par d'autres personnes que vous, et même si vous ne souhaitez pas dériver de cette classe, d'autres personnes

peuvent le souhaiter. Dans des langages plus récents, il est possible de spécifier qu'il est interdit de dériver d'une classe (mot clef sealed du C#, mot clef final du Java, ...), interdisant des héritages malheureux d'une personne tierce (mais aussi permettant certaines optimisations du compilateur). En C++, un tel mot clef n'est pas disponible. Pour hériter d'une classe, il vous faut donc vérifier que la classe que vous souhaitez dériver a son destructeur virtuel (ce n'est pas le cas des principales classes de la STL : nous ne pouvons donc pas faire dériver les classes list, vector, string ou map par exemple).

13.2 Polymorphisme statique

13.2.1 Position du polymorphisme statique

L'introduction des templates a permis de résoudre le problème pré-cité d'une manière différente : au lieu de n'écrire qu'une fonction f, écrivons une seule fonction f templatée, dont le comportement va varier selon le type en lequel elle est spécifiée ; ainsi, nous n'écrivons le code qu'une fois (c'était la contrainte), mais la logique va être générée par le pré-compilateur autant de fois que de types différents seront utilisés pour la fonction f.

13.2.2 Implémentation

Nous l'avons vu, le polymorphisme dynamique vient avec un léger coût en performance, mais aussi en complexité du code :

- Il est nécessaire d'introduire une classe mère.
- Toute classe impliquée doit en hériter, impliquant parfois de multiples héritages
- Les fonctions surchargées doivent être déclarées virtuelles dans la classe mère.

Dans le cas du polymorphisme statique, il n'est pas nécessaire de rendre le code plus complexe. Il s'agit réellement d'un essai optimiste de polymorphisme.

```
1
2  template <class T>
3  public void f(const T& t)
4  {
5      t.Display();
6  }
7
8  class A
9  {
10     public void Display(){};
11 };
12
13 class B
14 {
15     public void Display(){};
16 };
17
18 class C
19 {
20     public void AnotherMethod(){};
21 };
22
23 int main()
24 {
25     A a;
26     B b;
27     C c;
28
29     f(a); //will compile
30     f(b); //will compile
31     f(c); //will break compilation.
32 }
```

Traditionnellement, le polymorphisme statique, pour les raisons évoquées dans la section 13.1.6, est considéré comme plus performant. Il est important cependant de bien comprendre que les questions de performances sont trop complexes pour pouvoir être résolues si naïvement. En particulier, une fonction templatée est générée autant de fois que le nombre de classes en lesquels elle est spécifiée. Cette génération multiple peut parfois accroitre significativement la taille des programmes générés, et ainsi saturer des caches de mémoire alloués aux instructions processeurs. Il n'est donc pas possible de répondre en toute généralité à cette question.

Troisième partie

Gestion de la mémoire

14
Allocation dynamique

Dans ce chapitre, nous raffinons le modèle mémoire déjà ébauché en section 7.3 en introduisant une nouvelle forme de mémoire, la *Heap*, à la fois plus souple et plus complexe que la "*Stack*".

14.1 Motivation

Comme nous l'avons vu dans la section 7.3, la zone mémoire dédiée à chaque fonction, la "*Stack Frame*", est ajoutée à la *Stack* lorsque nous entrons dans la fonction. De même, cette zone mémoire est automatiquement détruite lorsque nous terminons la fonction.

Il existe deux cas de figure dans lesquels nous ne pouvons/souhaitons pas utiliser la Stack pour stocker de la mémoire :

1. Lorsque nous souhaitons créer un objet dont le scope excède celui de la fonction dans laquelle il est créé. Si l'objet est stocké dans une Stack Frame, il sera détruit en même temps que celle-ci, il va donc falloir le stocker dans un autre espace pour le maintenir en vie plus longtemps.
2. Lorsque nous souhaitons créer un tableau dont la taille n'est pas connue à la compilation, voir même n'est pas connue à l'entrée de la fonction. Ce tableau ne peut donc être contenu dans une Stack Frame, puisqu'à la construction de cette Stack

Frame, sa taille est encore inconnue.

Ces deux exemples sont fondamentaux, et ils font l'objet de ce chapitre.

14.2 Allocation dynamique : le cas des instances simples

Mettons-nous tout d'abord dans la première situation, dans laquelle nous voulons créer des objets qui survivent à la fonction qui les crée, en les allouant sur la *Heap*. Cette allocation s'effectue via l'opérateur `new`. Considérons le code suivant :

```
1  class A
2  {
3      public:
4          A();
5          ~A();
6  };
7
8  void f()
9  {
10     A* pa = new A();
11 }
12
13 void main()
14 {
15     int b = 3;
16     f();
17 }
```

Listing 14.1 – Exemple d' appel à `new`

Dans la fonction `f`, l'usage du mot clef `new` nous permet d'instancier une variable de type A qui sera stockée dans la Heap et non pas sur la Stack. La création d'une instance de type A sur la Heap via le mot clef new retourne un pointeur de type A*, pointeur stocké dans la Stack Frame de `f`. Ce pointeur permet d'accéder à cette instance de type A nouvellement créée. Juste avant la sortie de la fonction f, notre mémoire est donc dans l'état du graphique 14.1.

Comme A est un type non primitif, la construction d'un A nécessite d'appeler un constructeur. Dans le cas présent, c'est le constructeur par défaut qui est appelé (en raison de l'absence d'argument). Si nous voulions procéder à l'instanciation d'un objet de type A via

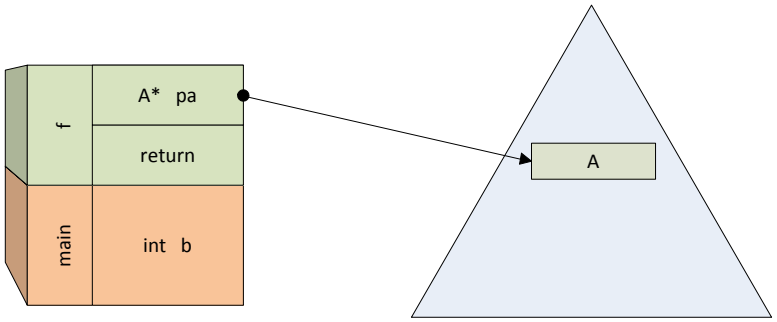

FIGURE 14.1 – Etat de la Stack et de la Heap dans la fonction f, après l'allocation dynamique

un autre constructeur, nous pourrions par exemple écrire le code du listing 14.2.

```
1  class A
2  {
3      public:
4          A(int i);
5          ~A();
6          void Display();
7  };
8
9  void f()
10 {
11     int n = 8;
12     A* pa = new A(n);
13 }
14
15 void main()
16 {
17     int b = 3;
18     f();
19 }
```

Listing 14.2 – Exemple d' appel à `new` en donnant un argument au constructeur-copie

Des exemples non triviaux de cas où l'allocation dynamique est nécessaire sont donnés en fin de chapitre.

14.3 Les 3 problèmes de l'allocation dynamique de mémoire

14.3.1 Memory Leaks

Nous savons à présent allouer de la mémoire selon nos besoins, en cours d'exécution du programme. Qu'advient-il de cette mémoire allouée lorsque notre code ne l'utilise plus ? La réponse est simple : elle reste allouée. Pour cela, nous devons détailler plus avant le processus d'allocation de la mémoire.

L'OS (système d'exploitation) d'un ordinateur est un "programme" qui tourne en permanence, et qui se charge (entre autres) de faire cohabiter les différents programmes tournant simultanément les uns avec les autres. En particulier, lui seul dispose d'un contrôle total sur la mémoire de l'ordinateur. En pratique, chaque fois qu'un programme demande de la mémoire au moyen d'une instruction `new` ou similaire, c'est en fait au système d'exploitation qu'il la demande. Celui-ci réserve alors une zone, et la marque comme appartenant au programme demandant la mémoire.

Pendant l'exécution du programme, l'OS n'a *aucune* raison de désallouer la mémoire qui a été demandée, et ce quelle que soit la situation. On pourrait très bien imaginer des cas dans lesquels un utilisateur a lancé une trentaine de programme simultanément qui consomment tous beaucoup de mémoire, et où l'OS aurait intérêt, voire besoin de libérer de la mémoire. Cependant, il ne le fera jamais. Certains langages (Java, Python, Pascal, etc.) fournissent un mécanisme de nettoyage automatique [1] de la mémoire qui n'est plus utilisée, mais ce n'est pas le cas du C++ [2].

Afin d'éviter ce genre de situation, la mémoire allouée dans un programme doit *toujours* être libérée dès qu'elle n'est plus utilisée, faute de quoi elle encombrera la mémoire jusqu'à ce que le programme se termine ou meurt, un encombrement appelé `Memory Leak` ou `Fuite Mémoire`.

A la différence de la mémoire allouée sur la Stack qui est automatiquement libérée par l'environnement, la mémoire allouée sur la Heap doit donc être libérée manuellement et explicitement afin d'éviter ces Memory Leaks. Pour ce faire, nous notifions donc ex-

1. appelé Garbage-Collector, ou *ramasse miette en Français*
2. Au moins à notre niveau

plicitement l'environnement lorsque nous souhaitons qu'il détruise une telle instance. Cette notification est réalisée au moyen de l'instruction `delete`. `delete` est un opérateur qui prend en argument un pointeur pointant sur une instance allouée sur la Heap, qui déréférence ce pointeur, détruit l'instance pointée en appelant le destructeur correspondant, puis libère la mémoire correspondante de la Heap. L'instruction prend donc la forme suivante :

```
1 void f()
2 {
3     int n = 8;
4     A* pa = new A(n);
5     delete pa;
6 }
```

Listing 14.3 – Exemple d' appel à `delete`

Cet exemple est bien entendu idiot, puisque nous désallouons ici la mémoire dans la même fonction que nous l'allouons, ce qui supprime tout l'intérêt d'allouer dans la Heap plutôt que dans la Stack.

Bonnes habitudes 23 (`new`/`delete`) *À chaque fois que nous écrions un* ***`new`****, il faut écrire immédiatement le* ***`delete`*** *associé. Cela permet d'être certain de ne pas l'oublier, et donc d'éviter bien des problèmes plus tard.*

14.3.2 Segmentation Fault

Lorsque nous appelons `delete` sur le pointeur `pa`, l'instance pointée par pa est détruite, mais nous disposons toujours du pointeur, qui lui ne sera détruit que lorsque la Stack Frame dans lequel il est contenu sera détruite. Ainsi, par erreur, nous pourrions désallouer par deux fois une même zone mémoire de la Heap en appelant deux fois `delete` sur `pa`.

```
1 void f()
2 {
3     int n = 8;
4     A* pa = new A(n);
5     delete pa;
6     delete pa;
7 }
```

Listing 14.4 – Exemple d' appel à `new`

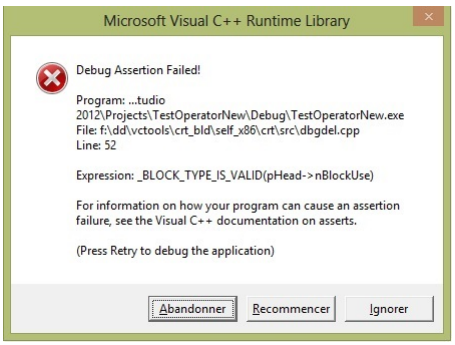

FIGURE 14.2 – Screenshot d'une fenêtre d'erreur générée par une Segmentation Fault.

Si nous exécutons ce code, nous obtenons une erreur au RunTime qui tue notre application, et affiche une fenêtre proche de celle affichée dans le graphe 14.2.

Les erreurs de Segmentation Fault sont particulièrement coûteuses, car elles sont complexes à répliquer et à isoler. Elles participent largement à faire du C++ un langage complexe et laborieux.

Donnons maintenant un cas classique où la Segmentation Fault apparaît naturellement. Nous construisons par composition un objet `B` qui possède un objet `A` construit dynamiquement dans le constructeur de `B`. Pour lutter contre les problèmes de Memory Leak, il semble assez naturel d'assurer la libération de la mémoire de la Heap dans le destructeur de `B`. En effet, puisque l'instance de type `A` n'est pointée que par `B`, elle devient superflue à la mort de `B`, et peut donc être détruite dans le destructeur de `B`. Ce design semble une excellente pratique, puisque nous n'avons plus à nous soucier de la libération de `A` : c'est à la destruction automatique de `B` que `delete` sera appelé et que le destructeur de `A` sera appelé subséquemment et automatiquement.

```
1  class A
2  {
3  public:
4    A(void);
5    ~A(void);
6  };
7
8  class B
9  {
10 public:
11   B(void);
12   ~B(void);
13
14 private:
15   A* _pa;
16 };
17
18 B::B(void)
19 {
20   _pa = new A();
21 }
22
23
24 B::~B(void)
25 {
26   delete _pa;
27 }
```

Listing 14.5 – Un cas classique de SegFault

Cependant, si dans le code précédent nous passons par valeur l'instance de type `B` en argument d'une fonction `f`, le constructeur-copie va recopier l'instance de type `B` en une autre, et ces deux instances possèderont chacune un pointeur `_pa` possédant la même valeur, c'est à dire pointant sur le même objet de la Heap. Le code correspondant est décrit dans le listing 14.6 et l'état de la mémoire en entrant dans la fonction `f` est schémarisé dans le graphique 14.3.

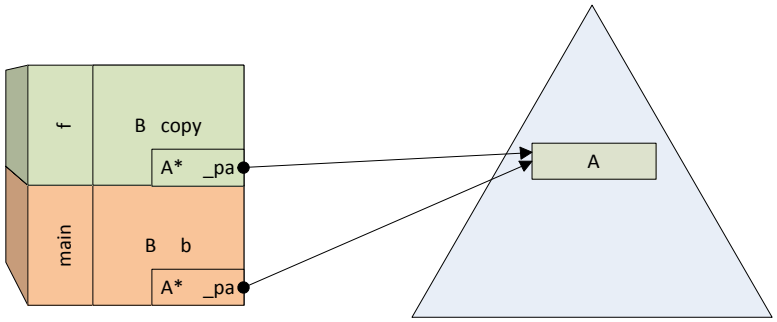

FIGURE 14.3 – b et copy possèdent chacun un pointeur qui référence l'instance de type A.

```
1
2  void f(B copy)
3  {
4
5  }
6
7  void main()
8  {
9      B b;
10     f(b);
11 }
```

Listing 14.6 – Un cas classique de SegFault

Lorsque la fonction `f` s'achève, sa Stack Frame est détruite, le destructeur de `copy` est appelé automatiquement, celui-ci déréférence le pointeur `_pa` de `copy` et désalloue l'instance pointée, c'est à dire l'instance de type `A` présente sur la Heap. Lorsque la fonction `main` s'achève à son tour, l'instance `b` est à son tour détruite automatiquement, son destructeur est appelé, et celui-ci tente d'appeler `delete` sur `b._pa`, alors que l'instance de type `A` a déjà été détruite, produisant ainsi une Segmentation Fault.

14.3.3 Dangling Pointers

Enfin, le troisième risque que comporte la gestion dynamique de mémoire consiste à tenter de déréférencer un pointeur pointant vers une instance déjà détruite. Par exemple, le code du listing 14.7 mène

à un comportement indéfini à l'exécution.

```
1 void f()
2 {
3     int n = 8;
4     A* pa = new A(n);
5     delete pa;
6
7     pa->Display();
8 }
```

Listing 14.7 – Exemple d' appel à `new`

14.4 Smart Pointers

Nous voudrions donc nous doter d'un pointeur "intelligent", appelé Smart Pointer dans ce qui suit, pour lequel nous avons le comportement d'un pointeur classique (possibilité de déréférencer le Smart Pointer via l'opérateur `*`, possibilité d'accéder aux champs et méthodes de l'instance pointée directement par l'opérateur `->`), mais pour lequel nous n'avons pas à gérer l'appel au `delete`, c'est à dire que ce Smart Pointer gère lui-même l'allocation et la désallocation dynamique de mémoire, sans créer de Memory Leaks ou de SegFault.

Nous commençons par écrire par écrire la partie "Wrapper" du Smart Pointer, c'est à dire les opérateurs `*` et `->`, comme dans le listing 14.8 :

```
template < typename T > class SmartPointer
{
    public:
        SmartPointer(T* pValue)
        {
            _pInner = pValue;
        }

        ~SmartPointer(){}

        T& operator* ()
        {
            return *_pInner;
        }

        T* operator-> ()
        {
            return _pInner;
        }

    private:
        T*    _pInner; // Generic pointer to be stored
};
```

Listing 14.8 – Première implémentation d'un SmartPointer

Dans cet exemple, nous retrouvons les problèmes de Memory Leaks détaillés dans la section 14.3.1. Nous pouvons donc ajouter un appel à l'opérateur `delete` dans le destructeur de notre Smart Pointer, comme il a été fait dans la section 14.2, transformant ainsi notre code en celui du listing 14.9.

```
template < typename T > class SmartPointer
{
    public:
        SmartPointer(T* pValue)
        {
            _pInner = pValue;
        }

        ~SmartPointer()
        {
            delete _pInner;
        }

        T& operator* ()
        {
            return *_pInner;
        }

        T* operator-> ()
        {
            return _pInner;
        }

    private:
        T*    _pInner; // Generic pointer to be stored
};
```

Listing 14.9 – Seconde implémentation d'un SmartPointer

Cependant, nous nous retrouvons avec les risques de Segmentation Fault expliqués dans la même section 14.3.2. Pour régler cette difficulté, il nous faut remarquer que le problème de Segmentation Fault rencontré ici provient du fait que plusieurs Smart Pointers possèdent un pointeur vers le même objet dans la Heap, et que l'un de ces Smart Pointer supprime l'objet de la Heap alors que d'autres vont essayer de le faire plus tard. Il conviendrait donc de construire un mécanisme dans lequel c'est le dernier Smart Pointer qui pointe sur un objet et lui seul, qui va se charger de la désallocation. Pour réaliser ce mécanisme, nous allons donc construire un comptage du nombre de Smart Pointers "pointant" vers un objet de la Heap donné.

Dans les listings 14.10 et 14.11, nous construisons une classe `Count` qui va recenser le nombre de Smart Pointers pointant sur une instance donnée. A la construction, la construction-copie ou l'affectation via l'opérateur =, ce compte est incrémenté. A la destruction d'un Smart Pointer, ce compte est décrémenté. Lorsque le compte

atteint 0 après une décrémentation dans le destructeur, plus aucun Smart Pointer ne pointe sur l'instance de la Heap, et nous pouvons alors nettoyer sans danger cette instance. Cette technique prend le nom de *Reference Counting*.

Le Reference Counting s'inscrit dans un cadre plus large de gestion des ressources, et peut se résumer par un principe empirique de la "Rule of Three" : **si vous êtes amenés à expliciter pour une classe le code du destructeur, du constructeur-copie ou de la surcharge de l'opérateur =, alors probablement vous devriez expliciter le code pour chacune de ces 3 methodes.**.

```
1
2  template <typename T> class Counter
3  {
4      public :
5
6          void Decrement()
7          {
8              _count--;
9          }
10
11         void Increment()
12         {
13             _count++;
14         }
15
16         int GetValue()
17         {
18             return _count;
19         }
20
21         Counter()
22         {
23             _count = 1;
24         }
25
26     private :
27         int _count;
28 }
```

Listing 14.10 – Implémentation d'un SmartPointer avec un Reference Counting (Partie1)

```
1
2  template <typename T> class SmartPointer
3  {
4      public:
5          SmartPointer(T* pValue)
6          {
7              _pInner = pValue;
8              _pCount = new Count();
9          }
10
11         ~SmartPointer()
12         {
13             _pCount->Decrement();
14             if ( 0 == _pCount->GetValue())
15             {
16                 delete _pInner;
17                 delete _pCount;
18             }
19         }
20
21         SmartPointer(const SmartPointer<T>& spSource)
22         {
23             _pInner = spSource._pInner;
24             _pCount = spSource._pCount;
25             _pCount->Increment();
26         }
27
28         SmartPointer<T>& operator = (const SmartPointer<T>& spSource)
29         {
30             // Avoid self assignment
31             if (this != &spSource)
32             {
33                 // Decrement the old reference count and remove old data if
                       necessary
34                 _pCount->Decrement();
35                 if( 0 == _Count->GetValue())
36                 {
37                     delete _pInner;
38                     delete _pCount;
39                 }
40
41                 // Copy the data and counter then increment counter
42                 _pInner = spSource._pInner;
43                 _pCount = spSource._pCount;
44                 _pCount->Increment();
45             }
46             return *this;
47         }
48
49         T& operator* ()
50         {
51             return *_pInner;
52         }
53
54         T* operator-> ()
55         {
56             return _pInner;
57         }
58
59     private:
60         T* _pInner;
61         Count* _pCount;
62 };
```

Listing 14.11 – Implémentation d'un SmartPointer avec un Reference Counting

La différence entre le constructeur-copie et la surchage de l'opérateur = tient dans la possibilité d'écrire par erreur des choses semblables à celles du listing 14.4. Nous n'entrons pas plus dans le détail ici.

```
1 a=a;
```

Les propos développés dans cette section nous amènent à comprendre pourquoi la gestion de mémoire est difficile. Dans la vie réelle, la classe SmartPointer est déjà définie et disponible, il s'agit de la classe "`shared_ptr`" de la librairie standard, disponible via l'inclusion du header "<memory>". Nous invitons bien évidemment le lecteur à utiliser cette classe plutôt qu'à réimplémenter lui-même cette classe.

Il est important de noter que les algorithmes de Smart Pointers présentés dans cette section ne permettent pas de nettoyer parfaitement la mémoire. En effet, dans le cas de dépendances cycliques entre objets, ceux-ci ne pourront jamais être libérés via un mécanisme de Reference Counting.

14.5 Allocation dynamique : le cas des tableaux

Supposons à présent que nous souhaitions écrire un programme qui demande à l'utilisateur de rentrer un nombre n. Le programme en question demande ensuite à l'utilisateur de rentrer n nombres successifs, pour ensuite tous les réafficher.

14.5.1 Allocation dynamique

Spontanément, nous voudrions créer un tableau de taille n au moyen du code suivant :

```
#include <iostream.h>

int main()
{
   int n;

   cout << "Veuillez entrer un nombre : ";
   cin >> n;

   int tableau[n]; /*ERREUR*/

   return 0;
}
```

Listing 14.12 – Tableau de taille variable

Malheureusement, ce code est erronné. En effet, comme nous l'avons vu précédemment, chaque fois que nous déclarons une variable (ou un tableau), l'environnement C++ réserve des cases mémoires dans la Stack Frame correspondante. Mais pour les réserver correctement, il faut qu'il sache *à la compilation* combien de case il doit réserver, *ce qui n'est pas le cas ici*. Il nous faut donc trouver une autre solution.

Le problème se pose en ces termes : comment dire au compilateur " Je veux réserver n cases mémoires, *pendant l'exécution et non pas à la compilation* du programme " ?

De la même manière que nous avons utilisé le mot clef **new** pour allouer dynamiquement une instance sur la Heap, nous pouvons utiliser ce même mot clef suivi d'un nombre entre crochets pour allouer dynamiquement un tableau d'un type donné à l'exécution. Dans le cas de la déclaration dynamique d'un tableau, l'instruction **new** renvoie un pointeur vers le début du tableau, c'est à dire vers son premier élement. La syntaxe (dans notre cas) est la suivante :

```
/*The following line allocates 10 integers onto the Heap*/
int* someArray = NULL;
someArray = new int [10];
```

Listing 14.13 – Exemple d' appel à `new`

14.5.2 Arithmétique des pointeurs

Une fois la mémoire réservée, nous voulons pouvoir accéder au bloc mémoire en question. Le langage C++ permet de traiter la zone mémoire ainsi réservée exactement comme un tableau standard. Par exemple, le code suivant réalise ce que nous voulions faire au début (rentrer des nombres, les stocker, et les réafficher) :

```
#include <iostream.h>


int main()
{
   int n, i;
   int* someArray = NULL;

   cout << "Give a size for the array : ";
   cin >> n;

   someArray = new int [ n ];

   for(i=0 ; i<n ; i++)
      cin >> someArray [ i ];
   for(i=0 ; i<n ; i++)
      cout << someArray [ i ];

   /*warning, something missing*/
   return 0;
}
```

Listing 14.14 – tableautaillevariable.cpp

Cependant, que se passe-t-il en réalité ? Lorsque l'on écrit

```
someArray[i]=30;
```

c'est en fait l'instruction suivante qui est exécutée :

```
*(someArray + i)=30;
```

En d'autre termes, someArray contient bien l'adresse du début du bloc mémoire. En ajoutant i, on se déplace[3] à l'adresse du i-ème élément, dont on modifie la valeur à l'aide de l'opérateur de déréférentiation *. Le schéma ci-dessous représente la situation en mémoire lorsque l'on a un tableau de taille 5 :

Case mémoire	0	1	2	3	4	5	6	7	8	9
Nom			someArray		mémoire réservée pour le tableau					
Valeur			case mémoire 4							

3. Attention ! Il y a ici une subtilité. Saurez-vous la voir ?

Etudions le déroulement du code suivant :

```
1  int* tableau;
2
3
4  tableau = new int [ 5 ];
5
6
7  tableau[2] = 37;
8
9
10 *(tableau + 2) = 42;
```

Après l'étape 1, nous sommes dans la situation ci-dessous :

Case mémoire	0	1	2	3	4	5	6	7	8	9
Nom			tableau		mémoire réservée pour tableau					
Valeur			Case mémoire 4							

La mémoire réservée n'est *pas* initialisée à 0 (ou tout autre valeur).

Après l'étape 2,

Case mémoire	0	1	2	3	4	5	6	7	8	9
Nom			someArray		mémoire réservée pour tableau					
Valeur			Case mémoire 4				37			

le troisième élément du tableau a été modifié.

Après l'étape 3,

Case mémoire	0	1	2	3	4	5	6	7	8	9
Nom			someArray		mémoire réservée pour tableau					
Valeur			Case mémoire 4				42			

le même élément a été modifié, mais en écrivant explicitement où il se trouvait en mémoire.

Il est intéressant de remarquer que `*tableau = *(tableau + 0) = tableau[0]` correspond au premier élément du tableau. C'est une des raisons pour lesquelles les tableaux en C++ commencent à l'indice 0 et non 1.

Une subtilité s'est cependant glissée ici, qui a été passée sous silence pour des raisons de simplicité. En effet, suivant son type, une variable n'occupe pas le même nombre de cases mémoires : un `int` va généralement [4] occuper 4 cases, un `char` 1 case, et un `double` 8 cases :

4. Le cas des `int` est un peu particulier, puisqu'il dépend du type de processeur/compilateur utilisé. Sur les processeurs 32 bits, ce sera 32 bits soit 4 octets (cases).

Case mémoire	0	1	2	3	4	5	6	7	8	9	10	11	12
Nom	char c	int n				double d							
Valeur													

Pourquoi, lorsque l'on écrit

```
1 *(tableau + 2) = 42;
```

se trouve-t-on à l'adresse `tableau + 2* sizeof(int)` [5] et non `tableau + 2` (c'est à dire au milieu d'un entier) ? Pour des raisons de confort, lorsque l'on manipule des pointeurs, le compilateur C++ ajoute automatiquement le `sizeof` nécessaire. Cette arithmétique un peu particulière -on ajoute 2 *en apparence* - porte le nom d'arithmétique des pointeurs.

14.6 Gestion de la mémoire pour les tableaux

La création dynamique de tableaux souffre des mêmes difficultés que l'allocation dynamique de variable, difficultés recensées en sections 14.3.1, 14.3.2 et 14.3.3. Afin de supprimer la mémoire, il faut recourir à l'opérateur `delete[]` qui permet de supprimer un tableau dynamique. Tout comme en section 14.3.1, ne pas appeler l'opérateur `delete[]`, c'est créer des Memory Leaks, et l'appeler deux fois c'est faire une Segmentation Fault.

Nous appuyant sur les considérations précédentes de ce chapitre et la Rule of Three, nous pouvons donc à présent construire une classe Vector, dont le constructeur prend en argument un entier n, construit un tableau dynamique de taille n, se comporte comme un tableau (i.e. possède un opérateur []), et évite à la fois les Memory Leaks et les SegFaults. Le code de cette classe Vector est présenté dans le chapitre suivant.

14.7 Quelques précautions

Le langage C++ est un langage qui peut-être considéré comme de *bas-niveau* -par opposition à des langages comme Java, Python, Pascal, etc., qui sont dits de haut niveau- dans la mesure où il ne dissimule pas au programmeur ce qui se passe dans l'ordinateur. Cela a des avantages (contrôle très fin sur le déroulement du programme,

5. `sizeof` est une fonction particulière qui renvoie la taille en mémoire (le nombre de cases) d'une variable.

optimisations plus faciles à effectuer pour le compilateur, meilleures performances[6]), et des inconvénients (obligation de penser "comme la machine", plus grande difficulté à écrire du code robuste, augmentation du temps de développement)

Il est important de noter que plus de 50% des bogues rencontrés lors du développement d'application sont dus à :

- des dépassements d'indices dans des tableaux
- de la mémoire libérée plusieurs fois
- des pointeurs non initialisés ou pointant sur des zones mémoires non-initialisées.

Il convient donc, lorsque l'on écrit du code manipulant de la mémoire, de le faire avec la plus grande prudence.

Récapitulatif :

- Un tableau est déclaré en C++ de la manière suivante :

```
1 typeDeVariable nomTableau[nbElements];
```

- Si l'on a besoin d'un tableau de taille variable, ou inconnue à la compilation, il faut allouer dynamiquement de la mémoire, c'est-à-dire la demander au système au moyen de l'opérateur `new`. Celle-ci renvoie un pointeur sur une zone mémoire, qui peut être manipulée comme un tableau. Par exemple,

```
1 int* tableau = new int[40];
2 tableau[27] = 42;
```

- Tout objet (tableau ou non) construit via l'opérateur new doit être détruit manuellement via l'opérateur delete (ou delete[] dans le cas d'un tableau). Cette destruction manuelle peut être déléguée dans un destructeur, mais il faut alors prendre garde aux constructeurs-copie et à la surcharge de l'opérateur =.

6. Ceci est de plus en plus discutable, cf les derniers benchmarks http ://shootout.alioth.debian.org/ ; il faut de tout manière être très prudent lorsque l'on parle de performances

—C makes it easy to shoot yourself in the foot ; C++ makes it harder, but when you do it blows your whole leg off.

Bjarne Stroustrup

15 Gestion de la mémoire en C++

15.1 Un exemple simple

Considérons le cas d'une classe vecteur simple, analogue à celle que nous avions écrit dans le chapitre 4 :

```
1  class Vecteur
2  {
3      public :
4          Vecteur(unsigned int n);
5          ~Vecteur()
6
7          unsigned int GetLength();
8        double & operator[](int i);
9
10     private:
11
12         unsigned int _length;
13         double* _data;
14 }
```

Listing 15.1 – Vecteur.h

```
1  \#include "Vecteur.h"
2
3  Vecteur::Vecteur(unsigned int length)
4  {
5      _length = length;
6      data = new double[_length];
7  }
8
9  void Vecteur::~Vecteur()
10 {
```

```
11      dimension=0;
12      delete[] data;
13  }
14
15  unsigned int Vecteur::GetLength()
16  {
17      return dimension;
18  }
```

Listing 15.2 – Vecteur.cpp

15.1.1 Problème à l'affectation

Considérons maintenant le programme suivant :

```
1
2  #include "Vecteur.h"
3  #include <iostream.h>
4
5  int main(int argc, char **argv)
6  {
7      Vecteur u(50), v(50);
8
9      u = v;
10 }
```

Listing 15.3 – main.cpp

Si nous exécutons le programme d'exemple, il va *planter*[1]. Pourquoi ? Il faut nous intéresser au mécanisme de copie d'un objet. Lorsque l'on copie un objet, on copie tous ses membres 1 à 1. En particulier, cela veut dire que dans notre cas les opérations suivantes vont avoir lieu (de manière transparente bien entendu) :

```
1
2  u.data = v.data
3  u.dimension = v.dimension
```

Il s'agit bien d'une égalité membre à membre. En revanche, une petite subtilité s'est glissée ici : lorsque l'on écrit

```
1  u.data = v.data
```

on fait une copie de *pointeurs* et non des *données* pointées. Plus visuellement, avant la copie on se trouve dans la situation figure **??**. Les deux pointeurs de données pointent sur deux zones mémoires différentes qui ont été allouées. Après la copie, on se trouve dans la situation figure **??** : les deux vecteurs sont maintenant associés *au même bloc mémoire.*

Ce que nous aurions souhaité, c'était un comportement de copie ressemblant à celui-ci (une copie terme à terme, sous réserve que la mémoire pour u soit déjà allouée) :

1. Il est important de taper ce programme et de constater qu'il plante.

```
  u.dimension = v.dimension
  for(int i = 0; i < v.dimension; i++)
    u[ i ] = v[ i ];
```

Jusqu'ici, en dehors d'un problème de copie qui ne s'est pas déroulée comme nous le supposions, nous ne voyons pas pourquoi notre programme devrait s'arrêter. Nous avons en fait oublié le destructeur de la classe `vecteur` : en effet, celui-ci est automatiquement appelé lors de la destruction des objets `U` et `V`. Supposons que `V` soit détruit en premier. Alors la zone mémoire correspondante va être libérée, comme sur la figure **??**. Nous constatons alors que

- `U` ne pointe plus sur une zone mémoire allouée, puisque celle-ci vient d'être détruite.
- La zone mémoire allouée pour `U` initialement n'a plus de pointeur associé. En d'autre terme, elle est perdue, et c'est une fuite mémoire. Ce comportement est ennuyeux, car si une application comporte beaucoup de fuites mémoires, sa consommation mémoire va augmenter, et elle va finir par ralentir tout le système.

Le premier point est particulièrement gênant, puisque le destructeur de `U` va être appelé à son tour. Il va alors tenter de libérer la mémoire pointée, mémoire qui a déjà été libérée, et le programme va planter.

Nous venons de voir, au travers de cet exemple simple, une des difficultés majeures du C++ : la gestion de la mémoire. Afin de remédier aux problèmes posés par les copies, deux fonctionnalités du langage nous seront utiles, pour résoudre les deux cas dans lesquels on fait des copies (passages par valeur et affectation).

- La surcharge de l'opérateur d'égalité
- Le constructeur de copie

15.1.2 Le cas de la copie d'instance

```

#include "Vecteur.h"
#include <iostream.h>

void DoNothing(Vecteur v)
{
    
}

int main(int argc, char **argv)
{
    Vecteur u(50);

    DoNothing(u);
```

```
15 }
```

Listing 15.4 – main.cpp

Nous obtenons également ici une erreur au run-time. Comme notre méthode DoNothing possède un argument Vecteur passé par valeur et non pas référence, l'environnement procède à une copie de u, copie qui est passée en argument à DoNothing. Lors de la copie de u en u', un mécanisme très similaire à celui expliqué dans le paragraphe précédent a lieu : le pointeur data de u' va pointer sur la même chose que le pointeur data de u, et lorsque la méthode DoNothing prendra fin, le scope de l'instance copiée u' prendra fin, l'instance sera détruite, et la zone mémoire vers laquelle pointe u'.data sera désallouée. Lorsque nous retournons dans notre main, notre instance u possède donc un pointeur pointant vers une zone mémoire déjà désallouée. De plus, lorsque nous détruirons u, nous obtenons de nouveau une erreur, puisque nous essayons de supprimer une zone mémoire déjà désallouée.

15.2 Le constructeur copie

L'exemple précédent nous a montré les difficultés qui pouvaient surgir lors d'une simple copie d'instances. Celà vient du fait que la sémantique de copie peut se révéler arbitrairement complexe. Tout comme l'environnement fournit par défaut un constructeur et un destructeur sans arguments, il fournit également une méthode par défaut pour copier une instance, cette méthode s'appelle le constructeur-copie. Le prototype du constructeur-copie d'une classe T est toujours :

```
1 T(const T& instance);
```

La sémantique du constructeur-copie par défaut est la copie membre à membre, ce qui nous amène dans notre cas à :

```
1 u'._data = u._data
2 u'._length = u._length
```

,

La norme C++ spécifie que si un constructeur est redéfini par l'utilisateur (plutôt que d'utiliser la version par défaut), plus aucun constructeur (copie ou non) n'est généré par défaut. Visual Studio s'est affranchi de cette spécification, afin de simplifier la vie des développeurs débutants, et le constructeur copie est généré automatiquement, même si le constructeur de notre classe Vecteur a été redéfini.

Nous voulons modifier la sémantique du constructeur-copie par défaut, et allons donc redéfinir ce constructeur. On ajoutera donc

au header :

```
Vecteur(const Vecteur& v);
```

Listing 15.5 – Vecteur.h

et au fichier source :

```
Vecteur::Vecteur(const Vecteur& v)
{
  _length = v._length;


  if (_data != NULL)
    delete[] _data;

  _data = new double [_length];
  for(int i = 0; i < _length ; i++)
    _data[i] = v._data[i];
}
```

Listing 15.6 – Vecteur.cpp

15.3 Surcharge de l'opérateur =

Nous avons vu au chapitre 10 qu'il était possible de surcharger des opérateurs, et en particulier l'opérateur d'égalité. C'est ce que nous allons faire. Nous allons donc rajouter une méthode dans le header :

```
  Vecteur& operator=(const Vecteur &v);
```

Listing 15.7 – fichier.h

Ici, l'opérateur = prend en argument un vecteur passé par référence constante (const Vecteur &v). Nous pourrions donner en type de retour le type vide, puisque l'opération a = b modifie la valeur de a par effet de bord. Cependant, nous voulons pouvoir enchaîner les affectations et écrire :

```
  v1 = v2= v3;
```

Listing 15.8 – fichier.h

C'est pour cette raison que nous donnons comme type de retour un vecteur. Cette valeur de retour est passée en référence, pour limiter le nombre de copies créées.

```
Vecteur& Vecteur::operateur=(const Vecteur &v)
{
  _length = v._length;
  if (_data != NULL)
    delete[] _data;

```

```
7   data = new double [_length];
8   for(int i = 0; i < _length ; i++)
9     _data[ i ] = v._data[ i ];
10
11  return (*this);
12 }
```

Listing 15.9 – fichier.cpp

Le listing précédent présente tout de même un problème subtil, qui peut être la source de problèmes assez difficiles à détecter. Que se passe-t-il si l'on écrit le programme suivant ?

```
1  int main()
2  {
3    Vecteur v;
4
5    v = v;
6    return 0;
7  }
```

Le code que nous avons écrit commence par nettoyer la mémoire au cas où. Cependant, dans le cas où l'on fait un *self-assignment*, c'est-à-dire une affectation vers soi-même, il va commencer par nettoyer sa propre mémoire, et la ligne

```
1   data[ i ] = v.data[ i ];
```

n'aura plus de sens ! Il nous faut donc traiter ce cas, en testant si l'on n'affecte pas l'objet à lui même :

```
1  Vecteur& Vecteur::operateur=(const Vecteur &v)
2  {
3    if (&v == this)
4      return (*this);
5
6    _length = v._length;
7    if (_data != NULL)
8      delete[] _data;
9
10   _data = new double [_length];
11   for(int i = 0; i < _length ; i++)
12     _data[i] = v._data[i];
13
14   return (*this);
15 }
```

Listing 15.10 – fichier.cpp

15.3.1 Code complet de notre exemple

Le code complet de notre objet `vecteur` s'écrit alors :

```
class Vecteur
{
  public:
    Vecteur(unsigned int n);
    ~Vecteur();

    /*operateur d' egalite*/
    Vecteur& operator=(const Vecteur &v);
    /*constructeur de copie*/
    Vecteur(const Vecteur &v);

    /*operateur pour acceder aux composantes*/
    double & operator[](unsigned int i);

    unsigned int getDimension();

  private:
    unsigned int dimension;
    double *data;
};
```

Listing 15.11 – vecteur4.h

```cpp
#include "vecteur4.h"

Vecteur::Vecteur(unsigned int n) : dimension(n)
{
  data = new double[n];
}

Vecteur::~Vecteur()
{
  delete[] data;
}

unsigned int Vecteur::getDimension()
{
  return dimension;
}


Vecteur& Vecteur::operator=(const Vecteur &v)
{
  if (&v == this)
    return (*this);

  dimension = v.dimension;
  if (data != 0)
    delete[] data;

  data = new double [dimension];
  for(int i = 0; i < dimension ; i++)
    data[ i ] = v.data[ i ];

  return (*this);
}

Vecteur::Vecteur(const Vecteur &v)
{
  dimension = v.dimension;

  if (data != 0)
    delete[] data;

  data = new double [dimension];
  for(int i = 0; i < dimension ; i++)
    data[ i ] = v.data[ i ];
}

double & Vecteur::operator[](unsigned int i)
{
  return data[ i ];
}
```

Listing 15.12 – vecteur4.cpp

On peut vérifier que l'ensemble fonctionne bien à l'aide du programme suivant :

```
#include "vecteur4.h"
#include <iostream>
using namespace std;


/*Passage par valeur d'un objet*/
double sommeComposantes(Vecteur v)
{
  double total = 0;
  unsigned int i;

  for(i = 0; i < v.getDimension(); i++)
    total += v[i];

  return total;
}


int main()
{
  Vecteur v1(5), v2(6);

  for(int i = 0; i < v1.getDimension(); i++)
    v1[i] = 3;

  /*copie, v2 change de taille au passage*/
  v2 = v1;

  cout << "La Somme des composantes du vecteur vaut " ;
  cout << sommeComposantes(v2);

  return 0;
}
```

Listing 15.13 – Affection et passage par valeur

Bonnes habitudes 24 *De manière générale, il faut éviter les passages d'objets par valeur, leur préférer des passages par const &. Il est en fait possible d'interdire le passage de paramètres par valeur pour une classe donnée en rendant le constructeur de copie* `private`.

15.4 Quelques subtilités

15.5 Les shared et smart pointers

Les pointeurs soulèvent à l'emploi de nombreux problèmes, et le débuggage de programme présentant des segfaults[2] est particulièrement hardu et lent. Des classes ont été développées pour pouvoir transformer les pointeurs en de nouveaux types. [3]

2. cf le chapitre sur la gestion de la mémoire

Quatrième partie

D'autres considérations

16

Coûts algorithmiques et containers

16.1 Rappels sur les notations de Landau

Soit $\mathbb{E}$ l'ensemble des fonctions g continues de $\mathbb{R}$ dans $\mathbb{R}$, telles que $g > 0$ au voisinage de $+\infty$. Soient f et g sont deux fonctions de $\mathbb{E}$.

Si la fonction f/g est majorée par une constante au voisinage de $+\infty$, on écrit $f = O(g)$, que l'on lit "f est un grand o de g au voisinage de $+\infty$" , ou même par abus "f est un grand O de g".

Si la fonction f/g tend vers 0 en $+\infty$, on écrit $f = o(g)$, que l'on lit "f est négligeable devant g au voisinage de $+\infty$" , "f est un petit o de g au voisinage de $+\infty$" ou même par abus "f est un petit o de g".

— I think [making computer languages easier for average people] would be misguided. The idea of programming as a semiskilled task, practiced by people with a few months' training, is dangerous. We wouldn't tolerate plumbers or accountants that poorly educated. We don't have as an aim that architecture (of buildings) and engineering (of bridges and trains) should become more accessible to people with progressively less training. Indeed, one serious problem is that currently, too many software developers are undereducated and undertrained.

Bjarne Stroustrup

17

Pour aller plus loin

Il n'était ni possible ni nécessaire d'aborder dans ce cours d'introduction de nombreux points essentiels pour les étudiants qui seraient intéressés par une carrière de développeur. Nombre de ces points seront abordés dans le cours avancé, dispensé en troisième année. Dans ce chapitre, je donne quelques références que le lecteur curieux pourra décrouvrir avec profit.

Le C++ plus en détails : Pour une introduction plus en détail et en précision du C++, je vous recommande l'ouvrage de Bruce Eckel : *Thinking in C++* ([10]). Vous pouvez également lire l'ouvrage du père fondateur du langage ([23]), mais que je trouve moins éclairant.

Subtilités du C++ : L'ouvrage de Scott Meyers ([17]) détaille de nombreux points techniques du C++, notamment de nombreux pièges du langage. C'est un ouvrage essentiel pour les gens désireux d'en acquérir une meilleure maîtrise.

Design Pattern : De nombreux problèmes de design se rencontrent régulièrement dans la conception de logiciel. Le livre du gang of four ([12]) référence nombre de ces problèmes et en fournit des solutions élégantes et génériques.

Sur la compilation : Le dragon book ([2]) introduit aux concepts de la compilation. Du fait de son âge, il n'aborde pas de nombreux

concepts récents, notamment la compilation Just In Time ou la compilation continue, mais reste l'ouvrage de référence pour une introduction à ces techniques.

Design Pattern en finance : Si malgré tout, vous voulez coder des pricers, l'ouvrage de Marc Joshi ([15]) est une excellente introduction en la matière.

Algorithmie : L'algorithmie est un pan bien particulier du développement, qui est souvent utile aux ingénieurs. L'ouvrage de Cormen, Leiserson, Rivest et Stein ([6]) fournit une introduction très abordable à cette discipline. Un peu plus théorique et technique, l'ouvrage de Korte et Vygen ([16]) décrit certains algorithmes plus avancés, notamment des algorithmes d'optimization sous contrainte convexe.

Programmation parallèle : La programmation parallèle est un sujet majeur pour tous les ingénieurs qui seraient amenés à travailler sur des projets calculatoires. Bien que parfois un peu trop théorique pour notre propos, l'ouvrage de Shavit et Herlihy ([14]) présente une excellente introduction aux concepts de la programmation parallèle, extrêmement plaisante à lire. Si la programmation multi-thread est extrêmement peu adaptée à un langage non garbage-collecté comme le C++ ([4]), vous pouvez quand même vous y essayer via Boost ([19] ou [7]). Plus à la mode, la programmation sur carte graphique (GPGPU) peut dans certains cas apporter de gros gains de performance. Le lecteur intéressé pourra découvrir à ce propos le livre de Sanders et Kandrot ([18]).

Gestion des caches et de la mémoire : Dans son document phare ([8]), Ulrich Drepper présente une incursion très technique dans le domaine de la RAM et des caches.

Sur le métier de développeur : Joel Spolsky, co-fondateur du site Stack Overflow, est aussi bien connu pour ses 2 ouvrages à propos du métier de développeur ([21] et [22]).

Si vous vous heurtez à des problèmes insolubles : D'autres ont très probablement déjà eu ce problème, et vous trouverez toujours la solution sur Stack Overflow ([20]).

Templates et Metaprogramming : Le livre d'Alexandrescu ([3]) est un ouvrage de référence à propos des templates et du design

par "policy". Dans cet ouvrage, une courte introduction au Metaprogramming est également abordée. Le lecteur plus intéressé par ce sujet pourra lire avec profit l'ouvrage d'Abrahams et Gurtovoy ([1]).

Versionning de code : Le versionning de code est indispensable pour tout projet informatique ou même pour la rédaction d'un document. Si l'ENSAE met à disposition de tous ses étudiants Ankh SVN qui est un client Subversion ([11]), beaucoup de projets informatiques récents préfèrent aujourd'hui Git ([5]).

L'intégration continue : Est également une pratique indispensable dans une société logicielle. On peut en trouver une introduction par exemple dans [9].

Tests unitaires.
Machine learning :

Listings

Bibliographie

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming : Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series).* Addison-Wesley Professional, 2004.

[2] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers : Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] Andrei Alexandrescu. *Modern C++ Design : Generic Programming and Design Patterns Applied.* Addison-Wesley Professional, February 2001.

[4] H.J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6) :261–268, June 2005.

[5] S. Chacon. *Pro Git.* Apress, Berkely, CA, USA, 1st edition, 2009.

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, 2nd edition, September 2001.

[7] R. Demming and D.J. Duffy. *Introduction to the Boost C++ Libraries ; Volume I - Foundations.* Datasim Education BV, The Netherlands, 2010.

[8] U. Drepper. What every programmer should know about memory, 2007.

[9] P. Duvall, S. Matyas, and A. Glover. *Continuous integration : improving software quality and reducing risk.* Addison-Wesley Professional, first edition, 2007.

[10] B. Eckel. *Thinking in C++, Volume 1 : Introduction to Standard C++ (2nd Edition).* Prentice Hall, April 2000.

[11] B. Fitzpatrick, B. Collins-Sussman, and C.M. Pilato. *Gestion de projets avec Subversion.* O'Reilly, 2004.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns : elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning : Data Mining, Inference, and Prediction.* Springer, corrected edition, August 2003.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[15] M. S. Joshi. *C++ Design Patterns and Derivatives Pricing.* Cambridge University Press, New York, NY, USA, 2nd edition, 2008.

[16] B. Korte and J. Vygen. *Combinatorial Optimization : Theory and Algorithms.* Springer Publishing Company, Incorporated, 5th edition, 2012.

[17] S. Meyers. *Effective C++.* Addison-Wesley, 2004.

[18] J. Sanders and E. Kandrot. *CUDA by Example : An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 1st edition, 2010.

[19] B. Schling. *The Boost C++ Libraries.* XML Press, 2011.

[20] J. Spalsky and J. Atwood. `http://www.codinghorror.com`.

[21] J. Spolsky. *Joel on Software : Selected Essays.* APress, 2004.

[22] J. Spolsky. *More Joel on Software : Further Thoughts on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and ... Luck, Work with Them in Some Capacity.* Apress, Berkely, CA, USA, 1 edition, 2008.

[23] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Professional, third edition, February 2000.