Sorbonne Université
Master 1 de mathématiques et applications
Unité d'enseignement MU4MA056

Programmation en `C++` pour mathématiciens:

notes de cours

Damien Simon
Laboratoire de Probabilités, Statistique et Modélisation (LPSM)

Année universitaire 2019–2020

Table des matières

Introduction et organisation du cours

But du cours

Ce cours a pour but de vous donner une base solide de programmation en `C++` orientée vers les mathématiques appliquées, avec un fort accent mis sur le calcul intensif.

Étant donné le volume horaire restreint, de nombreuses bibliothèques ne seront pas présentées et nous ne pourrons pas aborder en détail le calcul parallèle ; en revanche, la structure du cours vise à donner suffisamment de bases sur ces questions pour que l'étudiant intéressé puisse s'y mettre par lui-même à la suite du cours, en M2 ou en stage.

Il faut savoir que le `C++` est un vieux langage (créé en 1980) qui a beaucoup évolué à partir du `C` et continue d'évoluer. Cela signifie qu'il existe de nombreuses manières d'écrire la même chose (à l'ancienne comme en `C`, sous une forme `C++` classique des années 1990-2000, en `C++` moderne post-2011) : même si en général il vaut mieux utiliser dans un nouveau projet le `C++` moderne, il est fréquent d'être confronté à de vieux codes et d'être capable de les modifier dans leur style. C'est pourquoi nous insisterons sur le fait de maîtriser toutes les syntaxes possibles pour une même opération.

Organisation du cours

Le côté pratique étant essentiel pour un cours de programmation, le nombre d'étudiants inscrits est limité à **100** étudiants. Il y a 4 groupes de TP et une répartition uniforme des étudiants sur ces 4 groupes est indispensable pour le bon fonctionnement des TP. L'effectif d'un groupe de TP est plafonné à **25**. L'inscription dans l'un des 4 groupes est **obligatoire et nécessaire** pour valider ce module.

La situation de pandémie actuelle perturbe le fonctionnement habituel de cours. Les modalités pour le début de ce semestre sont les suivantes :

- le cours a lieu *à distance* sur Zoom le lundi de 9h à 10h30. Les liens seront publiés sur Moodle chaque semaine.
- les TPs auront également lieu à distance avec une forte interaction étudiants-enseignants sur le site `https://repl.it`. Les liens seront également publiés sur Moodle.
- les TPs auront lieu les lundi, mardi, jeudi ou vendredi de 13h45 à 16h45.
- en cas d'ouverture du campus, si vous avez des problèmes matériels chez vous, il vous sera possible d'aller à l'UTES (à l'Atrium) pour vous connecter sur le site `https://repl.it` à partir des ordinateurs mis à disposition dans les salles publiques.

Calendrier prévisionnel pour l'année 2020–2021

- premier et deuxième cours : lundi 11 janvier 2021, 9h-10h30 et jeudi 14 janvier 2021, 14h15-16h ;
- cours suivants : les lundi de 9h à 10h30 ;
- début des TP : la semaine du 18 janvier 2021 ;
- premier TP noté : la semaine du 1er mars 2021 ;
- deuxième TP noté : la semaine du 12 avril 2021.

Vous trouverez sur le Moodle du cours :

https://moodle-sciences.upmc.fr/moodle-2020/course/view.php?id=149

tout le matériel nécessaire : support pdf du cours, vidéos des cours, brochure des TPs, corrigés des exercices, vidéos supplémentaires, liens Zoom et `repl.it`, etc.

Évaluation sous réserve d'évolution de la situation sanitaire).

Le calcul de la note finale procède selon la formule :

$$\text{note finale sur } 100 = \left(\frac{\text{note TP1 sur } 25 + \text{note TP2 sur } 25}{2}\right) + \text{note examen sur } 75$$

Vous remarquerez que cette règle ne contient pas de maximum car les TP sur machine sont extrêmement importants.

Si les règles de l'Université devaient évoluer vers du contrôle continu intégral, alors cette règle devrait évoluer vers des poids $(25, 25, 50)$ (au lieu de $(12.5, 12.5, 75)$). Mais, là encore, nous vous tiendrons au courant au fur et à mesure de l'évolution sanitaire.

Structure de l'ouvrage.

Le chapitre 1 ne traite pas du `C++` mais de l'*informatique en général*. Il vise à donner une culture générale sur les ordinateurs et l'algorithmique et fournit des conseils pour aborder sereinement un projet de programmation.

Le chapitre 2 présente le `C++` à travers des **exemples de programmes**, décrit leur structure générale et permet d'apprendre à les **compiler**.

Le chapitre 3 présente la **syntaxe de base** du `C++`, i.e. les mots-clefs et symboles à maîtriser pour écrire des programmes élémentaires. La plupart des notions sont classiques en programmation et se retrouvent dans de nombreux autres langages avec des syntaxes légèrement différentes.

Le chapitre 4 présente la **bibliothèque standard** du `C++`, qui permet de gérer en pratique de nombreux outils classiques tels les conteneurs, les algorithmes et les nombres pseudo-aléatoires.

Le chapitre 5 explique comment créer de nouveaux types en `C++` à l'aide de **classes** et comment exploiter la programmation orientée objet.

Le chapitre 6 décrit la **programmation générique** en `C++` qui permet d'écrire des morceaux de code réutilisables sur des objets de types très différents à l'aide de « **templates** » (modèles en français) de fonctions et de classes.

Le chapitre 7 décrit une spécificité majeure du `C` et du `C++` : la possibilité de gérer soi-même la mémoire à l'aide des **pointeurs et références** pour une plus grande efficacité du code.

Remerciements. Je tiens ici à remercier toutes les personnes qui ont participé à l'élaboration de ce polycopié. En premier lieu, je voudrais remercier Vincent Lemaire qui est à l'origine de ce cours et me l'a transmis et avec qui j'ai enseigné cette matière de nombreuses fois. Mes collègues de TP au fil du temps, Cédric Boutillier, Raphaël Roux, Nicolas Gilliers, Thibaut Lemoine et Daphné Giorgi ont été, au fil du temps et des évolutions de ce cours, d'une aide précieuse pour la relecture et la conception des sujets et source de nombreuses remarques constructives. Enfin, je voudrais remercier mes autres relecteurs, étudiants et collègues, qui ont corrigé de très nombreuses coquilles et mauvaises formulations : il doit malheureusement en rester encore...

Chapitre 1

Notions générales d'algorithmique et de programmation

1.1 Qu'est-ce que la programmation ?

1.1.1 Une première approche intuitive

Programmer consiste à écrire de manière détaillée une suite d'instructions élémentaires à effectuer pour pouvoir réaliser une tâche donnée. La connaissance de cette suite d'instructions permet ensuite à toute entité dotée de capacités de logique et d'arithmétique — autrement dit capable d'exécuter les instructions élémentaires — d'effectuer la tâche donnée : cette entité peut être un être humain ou un ordinateur.

L'avantage de ce dernier est sa capacité d'effectuer extrêmement rapidement les calculs logiques et arithmétiques ; son inconvénient — si cela en est un — est de suivre *aveuglément* les instructions qui lui ont été fournies et de ne pas mesurer la pertinence du résultat.

Les instructions doivent être écrites dans un *langage* que comprend l'ordinateur. Comme pour les langues humaines, il existe plusieurs langages mais ils décrivent tous les mêmes calculs mathématiques et logiques. Pour nous, ce langage sera le `C++`.

1.1.2 L'ordinateur

Pour bien comprendre les langages informatiques, il est nécessaire d'avoir une idée du fonctionnement d'un ordinateur. Nous n'entrerons pas dans les détails qui peuvent atteindre une très grande complexité et dont les informaticiens sont les spécialistes.

Un ordinateur est essentiellement composé :

— de mémoire vive (la RAM) : c'est là que sont écrites les données manipulées par le processeur. Y accéder pour lire ou écrire des données est très rapide. Lorsqu'un programme termine, toutes les données présentes dans la RAM qui dépendent de ce programme sont effacées et perdues. C'est une mémoire rapide mais temporaire.

— de mémoire morte (disque dur, CD, DVD, disquette, clef USB, etc.) : c'est là que sont écrites les données de manière durable, i.e. au-delà de la fin d'un programme. Y accéder est beaucoup plus lent que pour la RAM. Le processeur ne se permet d'y écrire que sur instruction explicite du programmeur.

— de processeurs : ce sont eux qui effectuent les calculs. Ils chargent quelques données, opèrent des opérations logiques et arithmétiques très élémentaires sur celles-ci et donnent le résultat.

— de périphériques, en particulier l'affichage : tout ce qui permet de faire l'interface

avec l'être humain ou d'autres machines (par exemple, l'écran, l'imprimante, des capteurs électroniques, une sortie Ethernet, une carte Wifi, etc).

Un langage de programmation comporte donc les types d'instructions suivants :

— des instructions de *calcul* qui permettent d'ordonner aux processeurs les opérations logiques et arithmétiques à effectuer et d'indiquer sur quelles données les effectuer,
— des instructions d'*affectation* qui permettent d'indiquer où stocker le résultat dans la RAM pour un usage futur,
— des instructions de *lecture* et *écriture* qui permettent d'aller lire des données ou d'écrire des résultats dans la mémoire morte ou de les afficher dans des périphériques, tels l'écran.

L'un des buts de ce cours est de maîtriser ces trois types de commandes dans le langage `C++`.

Toute mémoire, vive ou morte, est organisée de la manière suivante : c'est une suite *ordonnée* de valeurs 0 ou 1, appelées *bits*. Cette organisation unidimensionnelle a deux conséquences majeures.

D'une part, toute donnée un peu plus évoluée que 0 ou 1 doit être codée en termes d'une succession de 0 et 1. Par exemple, un nombre entier sera écrit en base deux ; les lettres de l'alphabet seront numérotées et leur numéro sera écrit en base deux. Dans les instructions de lecture et écriture, il faut donc faire attention à dire à l'ordinateur ce qu'il doit lire ! Par exemple la suite de *bits* `01101` peut coder aussi bien un tableau $(0, 1, 1, 0, 1)$ de cinq valeurs dans $\{0, 1\}$, l'entier $0 \times 1 + 1 \times 2 + 1 \times 4 + 0 \times 8 + 1 \times 16 = 22$, ou l'entier $1 \times 1 + 0 \times 2 + 1 \times 4 + 1 \times 8 + 0 \times 16 = 13$ écrit dans l'autre sens. Le *codage* de l'information a donc une grande importance : c'est à la base de la notion de *type* que nous étudierons en `C++`.

D'autre part, le caractère *ordonné* de la mémoire permet d'accéder à toute séquence écrite dans la mémoire, vive ou morte, très facilement en donnant au processeur le numéro du bit de début et le nombre de bits à lire. Cet *adressage* de la mémoire est à la base des notions de *pointeur* et *référence* essentielles en `C++`.

1.1.3 L'algorithmique

L'*algorithmique* est une discipline qui se focalise sur le découpage d'une méthode de traitement d'une tâche donnée en étapes élémentaires préalablement fixées et dont le but est de minimiser le nombre d'étapes élémentaires à utiliser. Cela permet d'avoir un programme qui s'exécute en un temps minimal avec le moins d'opérations possibles.

En particulier, l'algorithmique ne dépend que très peu du langage utilisé ; c'est la programmation des étapes élémentaires qui dépend du langage, non le choix du découpage en étapes élémentaires. Pour les questions mathématiques qui vont nous intéresser, la plupart des langages usuels offrent le même panel d'opérations élémentaires.

Il s'agit donc de bien réfléchir à la méthode de résolution d'un problème *avant* de commencer à écrire des lignes de programmation !

Un exemple extrêmement élémentaire de deux algorithmes qui réalisent la même tâche en des temps très différents est celui du calcul d'une puissance.

Exemple : calcul de a^{517}. Soit a un élément d'un anneau ; quel est le nombre minimal de multiplications à utiliser pour calculer a^{517} ?

Une première réponse naïve consiste à dire 516 : on commence par calculer a^2 (une multiplication), puis $a^3 = a^2 \times a$ (une autre multiplication), puis $a^4 = a^3 \times a$, etc., puis $a^{517} = a^{516} \times a$.

En fait, on peut en utiliser beaucoup moins ! Le nombre optimal ici est de 11 multiplications. Nous commençons par écrire $517 = 512 + 4 + 1 = 2^9 + 2^2 + 1$. On calcule a^2 (une multiplication) puis $a^4 = a^2 \times a^2$ (une multiplication), puis $a^8 = a^4 \times a^4$ (une multiplication), etc, puis $a^{2^9} = a^{2^8} \times a^{2^8}$ (une multiplication). Pour calculer a^{512}, nous avons utilisé 9 multiplications. De plus, pendant ce calcul, nous avons déjà calculé a^4. Il suffit donc de calculer $a^{517} = a^{512} \times a^4 \times a$ (deux multiplications). Au total, seules 11 multiplications ont été nécessaires !

1.1.4 La compilation ou l'interprétation

Le *code* d'un programme est la suite de lignes de programmation écrites dans un langage donné (`C++` dans ce cours) pour effectuer une tâche donnée. C'est la transcription dans ce langage `C++` des algorithmes expliqués dans la section précédente.

À ce stade, nous sommes encore loin de ce qui se passe réellement dans le cœur de l'ordinateur, où les opérations élémentaires « physiques » consistent à transformer des 0 en 1 aux bons endroits. Pour passer de l'un à l'autre, il faut convertir nos lignes de programmation en « instructions machine » : c'est l'étape de *compilation* ou *interprétation* du programme (voir en fin de section pour la différence entre les deux notions). C'est un autre programme qui est chargé de cela : le *compilateur*[1] ou l'*interpréteur*.

Un langage de programmation est ainsi inutile sans compilateur/interpréteurs associé. Pour un même langage, il peut exister différents compilateurs. Selon les processeurs utilisés et le système d'exploitation, les instructions machine peuvent varier : le travail du compilateur est de gérer tout cela. Nous n'entrerons pas dans les détails du fonctionnement du compilateur et des instructions machines dans ce cours.

Le compilateur produit un fichier *exécutable* qui contient les instructions machine. Il ne reste alors qu'à exécuter ce fichier pour que l'ordinateur effectue *réellement* les opérations que nous avions prévues dans notre algorithme de départ.

Il est possible que le compilateur détecte des erreurs de syntaxe (parenthèses non fermées, instruction inconnue, tentative d'addition de vecteurs de dimensions différentes, etc.). Dans un tel cas, le compilateur indique la ligne du programme qu'il ne comprend pas et qu'il faut corriger. Il est important de garder en mémoire les deux choses suivantes :

- le compilateur ne prend aucune initiative : c'est donc au *programmeur* d'ajouter lui-même les parenthèses manquantes, même dans les cas où cela semble trivial. Pour éviter cela, il vaut mieux essayer d'être le plus rigoureux possible dès le début de la rédaction du code.
- le compilateur ne sait pas détecter une erreur de raisonnement : il produira toujours un programme exécutable si la syntaxe est correcte, même si ce programme effectue une toute autre opération que ce vous auriez souhaité. Il est de la responsabilité du programmeur de vérifier que son programme effectue bien la tâche imposée : personne ne peut le faire à part lui.

En résumé, pour un langage *compilé* – ce qui est le cas de `C++` —, les quatre étapes successives de programmation à respecter sont les suivantes : conception de l'algorithme (pas de langage à apprendre, uniquement des mathématiques), écriture du code, compilation du code pour produire le programme exécutable, exécution de ce programme.

Un langage *interprété* mélange les deux dernières étapes : au lieu de compiler une fois pour toute un code et de l'exécuter ensuite autant de fois que nécessaire, on lance directement l'interpréteur sur le code non-compilé et ce dernier transforme chaque ligne en instruction machine immédiatement exécutée.

1. Mais qui donc a compilé le premier compilateur ?...

1.1.5 Les différents langages et le choix de C++

Il existe une multitude de langages de programmation. L'immense majorité d'entre eux partage les mêmes fonctionnalités élémentaires. Ce qui les différencie est d'une part leur syntaxe, qui peut rendre la même opération agréable ou détestable à écrire, et d'autre part la présence de fonctionnalités spécifiques supplémentaires. Il est important de prendre conscience qu'*a priori* toute tâche est réalisable par tous les langages usuels. Le choix d'un langage est donc arbitraire.

Parmi les langages usuels, on peut citer Fortran, Pascal (anciens), Java, C, C++, Python (calcul), Perl, Ruby (traitement des chaînes de caractères), Haskell, C# , Caml, etc.

Les critères de choix usuels sont les suivants :

— habitude : en général, apprendre un nouveau langage requiert du temps donc on ne fait cet investissement en temps que sous obligation ou forte incitation ; sinon on continue à utiliser un langage que l'on connaît déjà ;

— intégration de fonctionnalités à des programmes préexistants ou travail en équipe : en général, on s'adapte à ce qui était déjà là quand on vient d'intégrer l'équipe de développement d'un projet ;

— choix d'une syntaxe adaptée aux outils que l'on souhaite utiliser : une même tâche peut prendre une seule ligne dans certains langages contre plusieurs dizaines de lignes dans un autre ;

— gestion de la mémoire et choix entre compilation et interprétation : pour de petits programmes, ce critère n'est pas important ; pour un gros programme de calcul intensif, l'utilisation d'un langage compilé, qui permet de gérer la mémoire soi-même, rend possible des optimisations à plus grande échelle lors de la transformation des lignes de code en instructions machine.

— ouverture du code et choix d'un langage connu : si le code du programme est public, plus le langage est largement connu, plus les chances augmentent que quelqu'un puisse vous aider.

— existence de bibliothèques qui réalisent déjà ce que l'on souhaite faire.

Le langage `C++` est un langage maintenant très classique et figure parmi les plus utilisés, en particulier en finance et en calcul intensif. D'une part, sa syntaxe est assez naturelle et, d'autre part, sa gestion directe de la mémoire permet une bonne vitesse d'exécution.

Le `C++` est réputé pour avoir une certaine lourdeur syntaxique, comparativement à Python par exemple : cette lourdeur se révèle être un avantage pour les gros programmes car la rigueur qu'elle impose évite l'accumulation de petites erreurs ou incompatibilités entre programmeurs. De plus, une fois qu'on maîtrise le `C++`, il est facile de passer à d'autres langages moins lourds et moins rigides : « Qui peut le plus peut le moins »...

1.2 Pourquoi programmer pour un mathématicien ?

Ce cours s'adresse à des mathématiciens et non à des informaticiens. C'est donc une certaine vision de la programmation qui domine ce cours. En particulier, nous ne traiterons absolument pas les questions d'interfaces (menus déroulant, messages clignotants, représentation graphique, etc), de protocoles TCP pour Internet, de communication par wifi, de hardware, etc.

Ce cours n'a pas pour but non plus de traiter le problème mathématique du contrôle des erreurs commises lors de discrétisation ou d'approximations : nous renvoyons pour cela à des cours de mathématiques.

L'emphase est mise sur l'implémentation de calculs mathématiques en `C++`, en particulier liés aux probabilités et aux statistiques en lien avec différentes applications.

1.2.1 Traiter efficacement des données numériques

Savoir résoudre formellement un problème est au cœur des mathématiques ; malheureusement, appliquer la formule de la solution dans un cas pratique peut parfois être long et pénible. Par exemple, le pivot de Gauss permet de résoudre formellement un système linéaire mais il est très fastidieux de résoudre « à la main » un système de 10 équations à 10 inconnues. Or, dans de nombreux cas pratiques, ce sont des systèmes de taille supérieure à 10000 qu'il faut résoudre. Il devient alors inévitable d'écrire un code qui réalise la méthode du pivot de Gauss.

De nombreux TP ce semestre ont pour but d'écrire les algorithmes et les lignes de code qui permettent de programmer des méthodes mathématiques que vous avez étudiées pendant vos années d'université : méthode de Monte-Carlo pour des estimations d'intégrales, algèbre linéaire, polynômes, chaînes de Markov, etc.

Avec le récent développement du « Big Data », les volumes statistiques à traiter sont énormes et requièrent de gros moyens de calcul programmés efficacement (calcul parallèle, etc) : le langage `C++` se révèle bien adapté pour cela.

1.2.2 Étudier un modèle à travers des simulations

De nombreuses situations pratiques issues de la physique, de la biologie ou de l'économie donnent lieu à des modèles mathématiques : équations différentielles, variables aléatoires de loi jointe prescrite, etc. Il est en général plus facile de prouver qu'une formule qu'on a préalablement devinée est vraie (par récurrence par exemple) plutôt que d'établir la formule sans savoir au préalable à quoi elle doit ressembler. Calculer des solutions numériques à un problème donné permet souvent de se faire souvent une bonne première idée de ce à quoi les solutions vont ressembler.

1.2.3 Ce que ne fait pas la programmation

Pour un mathématicien néophyte en programmation, certaines impossibilités peuvent parfois dérouter. Tout d'abord, sauf cas très particulier très récents (cf. le langage Coq par exemple), un programme ne prouve pas qu'un algorithme marche : il se contente de l'exécuter quelle que soit la pertinence du résultat. Si l'algorithme de départ est faux, le programme peut être syntaxiquement correct (i.e. le compilateur ne donne aucun message d'erreur) et le résultat faux. Autrement dit, l'écriture d'un programme vient *après* la formulation mathématique d'un problème et d'une méthode formelle de résolution.

Dans un tout autre registre, une autre impossibilité est déroutante : l'absence des *nombres réels* et de tout *infini* ! En effet, mathématiquement, un nombre réel requiert la connaissance de tous les chiffres de son développement binaire ou décimal. Il faudrait une mémoire infinie pour stocker ne serait-ce qu'un réel ; sur une machine réelle, on travaille avec des troncatures des nombres réels à une précision donnée. Cela a pour conséquence majeure l'accumulation des erreurs d'arrondis qui peuvent mener à des résultats aberrants.

Exemple : calcul de la série harmonique. Nous souhaitons calculer les valeurs de $H_n = \sum_{k=1}^{n} 1/k$. Nous savons que cette série diverge comme $\log n$ quand $n \to \infty$.

Néanmoins, calculons H_n selon deux méthodes différentes avec des variables de type `float` (réels avec une précision basse, cf. le chapitre suivant) :

— du grand vers le petit : on commence par 1, puis on ajoute $1/2$, puis $1/3$, etc, jusqu'à $1/n$;
— du petit vers le grand : on commence par $1/n$, puis on ajoute $1/(n-1)$, puis $1/(n-2)$, etc, jusqu'à $1/2$ puis 1 ;

Nous refaisons le même calcul avec des `double` (réels avec une plus grande précision) à la place des `float`. Nous obtenons alors les résultats suivants :

n	10^6	10^7	10^8	10^9
Du grand vers le petit (`float`)	14.3574	15.4037	15.4037	15.4037
Du petit vers le grand (`float`)	14.3927	16.686	18.8079	18.8079
Du grand vers le petit (`double`)	14.3927	16.6953	18.9979	21.3005
Du petit vers le grand (`double`)	14.3927	16.6953	18.9979	21.3005
Valeur attendue pour H_n	$\simeq 14.3927$	$\simeq 16,7$	$\simeq 19,0$	$\simeq 21,3$

L'analyse de ces résultats est la suivante. Tout d'abord, le passage de `float` à `double` augmente la précision des résultats : c'est normal puisque plus de bits sont utilisés pour stocker chaque nombre ; l'inconvénient est alors que le calcul prend plus de temps en `double` qu'en `float`. Même si cela n'apparaît pas ici, le type `double` a également ses limitations.

Au niveau des différentes méthodes, le deuxième est meilleure que la première car, dans la deuxième, chaque addition ne fait intervenir que des nombres *d'ordres de grandeur comparables* et cela limite les erreurs d'approximation. Dans la première, vers la fin du calcul, on ajoute des nombres très petits ($O(1/n)$) à des nombres très grands ($O(\log n)$) et les petits se retrouvent arrondis à zéro. À titre d'exemple, pour $n = 10^9$, la différence entre les deux méthodes pour des `double` est de $2,4 \times 10^{-12}$.

1.3 Programmer efficacement : quelques conseils

1.3.1 Aborder sereinement un projet de programmation

Le cerveau ayant des capacités limitées, il est complètement utopique — surtout pour des débutants — d'espérer être capable de penser simultanément aux mathématiques, à l'algorithmique, à tous les détails de syntaxe et à l'optimisation d'un code. De plus, il est assez difficile de *corriger* des lignes de code tapées rapidement et truffées d'erreur.

Le travail doit donc être divisé comme suit :

1. algorithmique : compréhension du calcul mathématique et décomposition en opérations élémentaires (on fera attention à n'oublier aucun cas ni aucun sous-cas, ni l'initialisation des récurrences), identification des fonctions à coder et des variables à manipuler ;
2. implémentation des algorithmes en lignes de code : on traduit les algorithmes en `C++` en écrivant précautionneusement chaque ligne pour laisser le moins possible d'erreurs de syntaxe.
3. compilation et correction des erreurs de syntaxe : les messages du compilateur ne sont pas des punitions et ils vous indiquent les lignes qu'il ne comprend pas. Ne corrigez qu'une erreur à la fois en commençant par la première puis recompilez [2].

2. une fois que le compilateur ne comprend pas une ligne, par exemple une parenthèse non fermée, il est possible qu'il perde ses repères et voie ensuite d'autres problèmes qui en fait n'existent plus une fois la première erreur corrigée.

4. test du programme sur des données pour lesquelles vous connaissez déjà les résultats attendus : ce n'est pas parce que la syntaxe est correcte que votre programme fait ce qu'il était prévu qu'il fasse ! Il suffit pour cela d'une mauvaise initialisation de récurrence par exemple.
5. utilisation du programme pour tout ce que vous voulez !

1.3.2 Bonnes pratiques de codage lors d'un travail en équipe

Selon que l'on travaille pour soi-même sur un petit programme ou dans une équipe de développeurs sur un grand logiciel, les habitudes de travail peuvent être différentes.

Lire du code est difficile ; lire le code d'autrui (ou son propre code quelques années plus tard) l'est encore plus. Il est important de *commenter* son code en expliquant avec des vraies phrases ce que fait chaque fonction et chaque morceau de code, i.e. de rédiger un documentation de son code (cf. par exemple le logiciel `doxygen`).

Il est fréquent de corriger ou améliorer du code, par exemple lorsqu'une erreur est découverte ou lorsqu'on souhaite utiliser une nouvelle méthode de calcul. Il ne faut pourtant pas tout réécrire à chaque fois. Pour éviter cela, il est important de morceler le plus possible le code et de respecter l'adage « une tâche, une fonction ». La maintenance du code se fait ensuite en modifiant seulement quelques lignes faciles à repérer.

Enfin, il est important de donner des noms de variables les plus explicites possibles et d'indenter le code proprement : une tabulation supplémentaire à chaque entrée dans un sous-bloc d'instructions, une instruction par ligne, des sauts de lignes entre les fonctions.

1.4 La notion de complexité

La complexité d'un programme peut être définie approximativement comme la quantité de ressources utilisées pour effectuer une tâche donnée. Les ressources peuvent être du temps ou de l'espace mémoire ou le nombre d'appels à une fonction donnée. À notre niveau, nous nous intéresserons au coût en temps, qui peut se ramener en général au nombre d'opérations élémentaires utilisées : en effet, une opération donnée prend en général toujours le même temps quelle que soit les valeurs génériques utilisées. Il faut néanmoins faire attention que certaines fonctions prennent plus de temps que d'autres : les calculs de fonctions analytiques cos, sin, log, exp, $\sqrt{\cdot}$, etc prennent plus de temps qu'une multiplication qui prend elle-même plus de temps qu'une addition (pour vous faire une idée, demandez vous comment vous feriez à la main : plus vous mettrez de temps, plus l'ordinateur mettra de temps lui aussi !).

Il n'est pas nécessaire de connaître le temps pris par chaque opération (cela dépend du processeur, du pourcentage d'occupation du CPU, etc) mais plutôt de savoir l'ordre de grandeur du nombre de fois où elles sont utilisées. En effet, c'est essentiellement le *répétition* un grand nombre de fois des opérations qui allonge le temps d'exécution.

En général, à un problème est généralement associée une (ou plusieurs) taille N. Par exemple, le tri d'un tableau dépend du nombre d'éléments (plutôt que de la nature de chaque élément), l'addition de deux vecteurs dépend de la dimension de l'espace, le calcul d'une puissance de réels dépend (entre autres) de l'exposant, etc. La complexité d'un algorithme qui résoud un problème est la donnée d'un comportement asymptotique $O(f(N))$ pour N grand du nombre d'opérations utilisées dans l'algorithme en fonction de la taille N du problème. C'est bien un O qui est utilisé car les constantes exactes dépendent du programme et de la machine.

Ce qui contribue à la complexité, ce sont généralement le nombre de boucles imbriquées et la taille des boucles utilisées. Prenons plusieurs exemples.

Puissances. Pour le calcul de la puissance, on a vu que seulement 11 multiplications étaient nécessaires pour calculer a^{517}. De manière générale, le calcul de a^N est de complexité $O(\log N)$ (il s'agit de trouver le plus grand chiffre en base 2 de N).

Trouver le minimum d'un tableau de taille N. Pour cela, il faut parcourir tout le tableau une seule fois en comparant chaque élément au minimum des éléments précédents. La complexité est donc $O(N)$ (on a compté ici le nombre de comparaisons à effectuer).

Trier un tableau de taille N. Le résultat précédent donne une première piste : on cherche le minimum du tableau puis on le met en première position, on recommence alors avec les $N-1$ éléments restants. Il faut donc environ $\sum_{k=1}^{N} k = O(N^2)$ comparaisons. Vous verrez en TP que cet algorithme n'est pas le meilleur et cela peut être amélioré en $O(N \log N)$.

Une autre manière d'aborder la complexité est de regarder par combien de temps est multipliée la durée d'un programme lorsque la taille des données double. Par exemple, si N est remplacé par $2N$, le temps pour trouver le minimum d'un tableau est doublé et le temps de tri par la méthode naïve est multiplié par 4. Cette notion est donc extrêmement utile pour estimer le temps de calcul pour des problèmes de grande taille. En particulier, on peut ainsi deviner s'il faut attendre une réponse dans la seconde/heure/journée/année/siècle.

Chapitre 2

Structure et compilation d'un programme

2.1 Histoire du C++

À la base du `C++` avec qui il partage presque toute la syntaxe élémentaire présentée dans ce chapitre, il y a le langage C apparu en 1972. Au tout début des années 1970, au sein des laboratoires Bell aux États-Unis, est développé en assembleur (i.e. en langage machine) le système d'exploitation `UNIX` (le prédécesseur des futurs `Linux`, BSD et autres) par Kenneth Thompson. Face à l'ampleur de la tâche, Kenneth Thompson et Dennis Ritchie décident de créer un langage de programmation plus approprié afin de rendre l'écriture plus aisée. Après plusieurs essais, le langage `C` apparaît en 1972.

De cette histoire initiale liée à `UNIX`, le `C` hérite une particularité remarquable : il est très axé sur la manipulation directe de la mémoire vive avec un système efficace et omniprésent de pointeurs (cf. le chapitre 7). Cette possibilité de gestion directe de la mémoire par le programmeur permet ainsi au langage `C` d'être un langage bien adapté au calcul intensif, comme le Fortran et le Pascal à la même époque, et lui assure un important succès. L'idéologie dans ces langages est de casser tout objet évolué en briques élémentaires de mémoire et en nombreuses procédures qui manipulent ces briques de mémoire. Il ne reste alors de l'objet évolué qu'un long manuel d'utilisation qui décrit comment assembler soi-même des procédures qui se ressemblent toutes au niveau de la syntaxe.

Au début des années 1980, la programmation orientée objet (POO) commence à se répandre. L'idée est au contraire de manipuler un objet évolué par ses propriétés intrinsèques sans devoir se référer constamment aux briques élémentaires qui le constituent. Il faut donc une bibliothèque qui fait le lien entre ces propriétés intrinsèques et les briques de mémoire et les procédures élémentaires mais qu'on préfère ne pas avoir à lire. La personne qui manipule ensuite un tel objet évolué n'a plus à connaître le travail fait en amont et il lui est donc plus facile d'écrire ce qu'elle veut.

Face à ce nouveau paradigme, il fallait au début des années 1980 ajouter au langage `C` une couche supplémentaire de syntaxe permettant de penser les programmes en POO. Bjarne Stroustrup, là encore dans les laboratoires Bell, réalise cette tâche et le `C++` apparaît en 1983. Il s'impose assez vite comme un langage très efficace et polyvalent. Il est normalisé en 1998 puis subit des améliorations en 2003, 2011 et 2014, 2017 et 2020 et la prochaine révision est prévue pour 2023, ce qui témoigne qu'il s'agit d'un langage vivant et qui continue d'évoluer. Les révisions récentes les plus profondes sont celles de 2011 et 2020.

Pour un mathématicien, la programmation orientée objet apparaît comme une démarche

naturelle : en effet, toute définition d'une structure mathématique abstraite commence par le postulat de règles de manipulations d'objets. En pratique, les exemples de structures évoluées sont construits en combinant des structures plus élémentaires. C'est la même chose en `C++` : définir une classe (cf. le chapitre 5), c'est donner la liste des opérations qu'on peut faire sur les objets de cette classe ; coder une classe concrète, c'est décrire comment ces constructions peuvent être réalisées à partir d'objets concrets préexistants.

2.2 Quelques références indispensables

Il est hors de question, dans le cadre de ce cours, de maîtriser par cœur l'ensemble du `C++` et de sa bibliothèque standard. En revanche, les quatre points suivants sont importants :

1. maîtriser toute la syntaxe de base du langage (sans la bibliothèque standard),
2. maîtriser les conteneurs élémentaires tels que `std::vector` et les entrées-sorties de `std::iostream` ,
3. avoir une bonne idée des outils présents dans la bibliothèque standard pour savoir aller les chercher le jour où vous en aurez besoin,
4. savoir lire la documentation de la bibliothèque standard pour s'adapter rapidement à de nouveaux outils.

Pour cela, il y a quelques références indispensables :

— les deux sites internet :

https://en.cppreference.com/w/

et

http://www.cplusplus.com/reference/

Le premier est très complet et décrit en particulier en détail les nouveaux standards du langage C++14, C++17 et C++20 ; le second est moins à jour sur les standards au-delà de C++17 mais les exemples de la bibliothèque standard sont souvent plus abordables.

— le site internet de la norme officielle du langage `C++`

https://isocpp.org/

Toute l'actualité récente y est agrégée, en particulier les parutions des derniers livres de référence et de nombreux articles de blogs sur certains points de détails du langage ou les dernières nouveautés.

— les livres classiques que sont[1, 5, 6] sur les bases de langage mais qui ne sont pas nécessairement les plus à jour sur les dernières évolutions.

2.3 Exemples de programmes

2.3.1 Premier exemple

Considérons notre premier programme :

```
#include <iostream>
int main() {
        std::cout << "Bienvenue en 4M056 !";
        /*        Ceci est un commentaire
                sur plusieurs lignes        */
        std::cout << std::endl; //ceci est un commentaire en fin de ligne
```

```
        return 0;
}
```

Lors de l'exécution de ce programme dans un terminal de commande (cf. la section suivante pour comprendre comment le faire), ce programme affichera dans ce même terminal la phrase « Bienvenue en 4M056 ! », reviendra à la ligne puis s'arrêtera avec succès. Commentons-le ligne à ligne :

- ligne 1 : ce programme doit réaliser de l'affichage d'informations sur un écran par un terminal de commande : cette opération élaborée est préprogrammée dans une bibliothèque appelée `iostream` et cette ligne est là pour indiquer au programme d'aller chercher cette bibliothèque.
- ligne 2 : c'est le prototype qui débute le code de la fonction `main(void)` ; cette fonction principale (« main » en anglais) est la fonction lancée à l'exécution du programme.
- ligne 3 : nous sommes à présent à l'intérieur (délimité par les accolades `{ }`) du code la fonction `main()` . Cette ligne se termine par un point-virgule : c'est une *instruction*. L'objet `std::cout` est l'objet défini dans la bibliothèque `<iostream>` qui permet d'afficher dans le terminal de commande. L'opérateur `<<` dit à `std::cout` d'afficher le message voulu, placé entre guillemets.
- les lignes 4 et 5, délimitées par `/* ... */` sont ignorées par le compilateur et permettent de commenter un programme pour un autre être humain qui lirait ce code.
- ligne 6 : l'opérateur `std::endl` ordonne de passer à la ligne suivante (« endl » est l'abréviation de "end line"), là encore dans le terminal via `std::cout` . La fin de la ligne après `//` est un autre commentaire ignoré par le compilateur.
- ligne 7 : on indique à la fonction `main()` de terminer proprement en produisant la valeur nulle. L'accolade finale termine le code de la fonction `main()` .

Un programme en C++ est ainsi constitué d'une suite d'instructions terminées par des points-virgules et rassemblées dans des fonctions.

2.3.2 Un deuxième exemple avec une fonction auxiliaire

Considérons à présent un second exemple qui affiche l'aire d'un cercle dont le rayon est demandé à l'utilisateur du programme :

```
#include <iostream>
#include <cmath>
double circle_area(double r) {
        return M_PI*r*r;
}
int main() {
        std::cout << "Entrez le rayon du cercle:" << std::endl;
        double x;
        std::cin  >> x;
        std::cout << "L'aire du cercle est ";
        std::cout << circle_area(x);
        std::cout << std::endl;
        return 0; }
```

Nous retrouvons comme précédemment des instructions d'affichage de messages par l'intermédiaire de `std::cout` et `std::endl`. Les nouveautés sont les suivantes :

— les lignes 3, 4 et 5 définissent une fonction auxiliaire qui calcule l'aire d'un cercle de rayon r par la formule πr^2. Pour cela, il faut le symbole de multiplication `*` ainsi que le nombre π avec une grande précision. Pour cela, nous invoquons à la ligne 2 la bibliothèque `cmath` où sont définies de nombreuses constantes et fonctions mathématiques, dont le nombre π renommé ici `M_PI`.

— nous demandons le rayon du cercle à l'utilisateur en ligne 7 ; il faut donc lire sa réponse dans le terminal de commandes et stocker cette valeur dans la mémoire de l'ordinateur. Pour cela, nous créons une variable réelle `x` dans la mémoire avec la ligne 8 puis, ligne 9, `std::cin` permet de lire dans le terminal et `>>` met la valeur lue dans la variable `x`.

— enfin, la ligne 11 permet d'afficher dans le terminal le résultat de la fonction `circle_area` calculée en la valeur `x` donnée par l'utilisateur en ligne 9.

2.3.3 Un troisième exemple issu des mathématiques

Considérons un dernier exemple mathématique qui utilise une bibliothèque d'algèbre linéaire très efficace[1], appelée `Eigen`, pour calculer et afficher l'inverse de la matrice A de taille 4 donnée par :

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & -1 & 2 & -2 \\ 9 & 8 & 7 & 6 \\ 5 & 5 & 5 & 5 \end{pmatrix}$$

Le programme s'écrit :

```
#include <iostream>
#include <Eigen/Dense>
int main()
{
  Eigen::Matrix<double,4,4> A;
  A << 1, 2 , 3, 4 , 1,-1,1,-1,4,3,2,1,1,-1,0,0 ;
  std::cout << "La matrice A est: " << std::endl;
  std::cout << A << std::endl;
  std::cout << "Son determinant est: " << A. determinant() << std::endl;
  std::cout << "Son inverse est:" << std::endl ;
  std::cout << A.inverse() << std::endl;
}
```

Comme précédemment, la ligne 2 indique l'usage de la partie `Dense` de la bibliothèque `Eigen` que nous avons préalablement installée sur notre ordinateur. Elle donne ainsi accès à plusieurs opérations :

— à la ligne 5, un objet `A` qui décrit une matrice 4 par 4 contenant des nombres réels en double précision.

— à la ligne 6, un opérateur `<<` qui permet de remplir `A` à partir de la liste ordonnée de ses coefficients.

1. http://eigen.tuxfamily.org/

— à la ligne 9, une méthode `determinant()` qui calcule le déterminant de la matrice
— à la ligne 10, une méthode `inverse()` qui calcule l'inverse de la matrice.

Le résultat du programme tel qu'il s'affiche est le suivant :

```
La matrice A est:
 1  2  3  4
 1 -1  1 -1
 4  3  2  1
 1 -1  0  0
Son determinant est: 40
Son inverse est:
-0.075 -0.125  0.175    0.5
-0.075 -0.125  0.175   -0.5
 0.175  0.625 -0.075   -0.5
 0.175 -0.375 -0.075    0.5
```

Ce dernier programme semble magique puisqu'il calcule en une seule ligne l'inverse d'une matrice 4 par 4. Il est donc très utile pour celui qui ne veut pas connaître la formule pour inverser une telle matrice. En fait, tout le travail a été fait par des mathématiciens comme vous au sein de la bibliothèque `Eigen` . Nous souhaitons insister sur le double but de ce cours de programmation en `C++` :

— le premier but de ce cours est de vous apprendre à *utiliser* des méthodes de calcul déjà existantes dans des bibliothèques et à les combiner astucieusement pour résoudre de nouveaux problèmes ;
— le second but de ce cours est de vous apprendre à *concevoir* de nouvelles bibliothèques pour définir de nouveaux objets qui deviendront ensuite utilisables par vous-mêmes ou par d'autres.

2.4 Compilation de fichiers

2.4.1 Cas d'un fichier unique

Supposons que nous ayons pour l'instant dans notre répertoire de travail un unique fichier `essai1.cpp` que nous venons de créer avec un éditeur spécifique. Supposons également que ce fichier ne fasse appel à aucune bibliothèque spécifique. La compilation doit alors être faite dans le terminal par la commande

```
g++ essai1.cpp -o essai1.exe
```

Le nom `g++` est le nom du compilateur utilisé, `-o` est l'abbréviation de *output* et permet de donner au compilateur le nom du fichier exécutable que l'on souhaite choisir (ici `essai1.exe`). Après validation de cette ligne de commande, deux choses peuvent alors se passer :

— soit un message potentiellement très long apparaît pour décrire les erreurs de syntaxe que le compilateur a détecté. Le compilateur est bavard et essaie de vous aider : en général il donne la ligne à laquelle il pense avoir trouvé l'erreur [2] et la nature de l'erreur. Le message, en anglais avec un vocabulaire parfois abscons, peut

2. attention, cette ligne peut être fausse : s'il s'agit par exemple d'une parenthèse non fermée dans une ligne antérieure, le compilateur n'est pas capable de localiser ce que vous souhaitiez faire.

paraître déroutant au début mais ce sont toujours les mêmes erreurs qui reviennent et auxquelles on s'habitue. Il serait dommage de se priver de l'aide précieuse du compilateur pour trouver ses erreurs !

— soit aucun message n'apparaît et le terminal vous redonne une invite de commande : cela signifie que la compilation a réussi et un nouveau fichier, nommé `essai1.exe` s'est matérialisé dans votre répertoire de travail. Vous pouvez alors *exécuter* le programme par l'instruction

```
./essai1.exe
```

À chaque modification du fichier source `essai1.cpp`, il faudra bien évidemment compiler à nouveau le programme par la même commande et l'exécuter de nouveau.

Lorsque le programme utilise des bibliothèques à travers des `#include` , le compilateur ne sait pas toujours trouver ces bibliothèques dans l'ordinateur : cela dépend de la manière dont le compilateur a été installé et configuré. En général, il sait toujours où se trouve la bibliothèque standard et rien de particulier ne doit être fait pour utiliser `<iostream>` , `<fstream>` , `<cstdlib>` , `<vector>` , `<list>` , `<string>` , `<algorithm>` , etc. Pour les autres bibliothèques, il faut indiquer au compilateur où elles se trouvent : cette information se trouve dans la documentation de celles-ci ou par une recherche dans l'ordinateur.

Par exemple, pour compiler le deuxième programme de la section précédente (calcul de l'aire du cercle) écrit dans un fichier `essai2.cpp`, il faudra taper

```
g++ essai2.cpp -o essai2.exe -lm
```

où `-lm` indique au compilateur d'aller chercher à l'endroit idoine le lien vers les fonctions mathématiques de `<cmath>` . Pour le troisième programme précédent (calcul de l'inverse d'une matrice) nommé `essai3.cpp`, il faut tout d'abord installer sur l'ordinateur la bibliothèque `Eigen` (facile sur un système Linux : il suffit d'installer le paquet `libeigen3-dev`), repérer où elle a été installée (ou se référer à la documentation) et enfin entrer dans le terminal la commande

```
g++ -I /usr/include/eigen3 essai3.cpp -o essai3.exe
```

De nombreuses autres options de compilations existent et leur invocation commence toujours par un tiret `-`. La figure 2.1 indique les options les plus fréquentes. Il suffit de taper `man g++` dans le terminal ou de se référer à la documentation complète de `g++` pour obtenir la liste exhaustive de toutes les options possibles.

2.4.2 Cas de plusieurs fichiers séparés

Lorsque la longueur d'un programme augmente nettement ou lorsque l'on souhaite réutiliser des morceaux de code pour différents programmes, il est indispensable de découper un programme en différents modules ou bibliothèques. Dans ce cas, un seul morceau du programme final contient la fonction `main()` et les autres morceaux contiennent des définitions de fonctions auxiliaires.

Étudions un exemple simple. Imaginons que nous ayons un programme en trois morceaux structurés de la manière suivante :

— le fichier `morceau1.cpp` contient le code de fonctions `int f1(void)` , `int f2(int)` et `void f3(void)` et utilise les bibliothèques `<iostream>` et `<cmath>` ,

— le fichier `morceau2.cpp` contient le code de deux fonctions `double g(int)` et `double h(double)` , utilise la bibliothèque `<fstream>` et la fonction `h()` fait appel à la fonction `f3()` du morceau précédent,

Option	Argument	Usage
`-std=`	`c++98`, `c++11`,...	Utilisation d'une version particulière du standard du langage `C++` (98, 03, 11, 14 et bientôt 17).
`-O1`, `-O2`, `-O3`, `-Os`		Optimisation du code par le compilateur selon différents niveaux.
`-Wall`		Affiche tous les avertissements sur des pratiques considérées comme douteuses.
`-pedantic`		Vérifie que le code est compatible avec toutes les normes ISO et est syntaxiquement parfait.
`-ggdb`		Produit des informations de débogage pour le programme GDB (non utilisé dans ce cours).
`-gp`		Produit des informations de profilage pour le programme `gprof` qui permet d'estimer le temps de calcul passé dans chaque fonction.
`-pthread`		Permet de paralléliser un programme en utilisant plusieurs cœurs d'un processeur et une bibliothèque idoine.

FIGURE 2.1 – Options de compilation de `g++` les plus fréquentes.

— le fichier `principal.cpp` contient le code de la fonction `main()` et utilise toutes les fonctions des morceaux précédents.

Il faut utiliser alors *cinq* fichiers, les trois précédents et deux fichiers d'en-tête. Les fichiers d'en-têtes sont des fichiers avec un suffixe en `.hpp` et contiennent toutes les définitions nécessaires pour assurer les compilations séparées des morceaux.

Nous pouvons à présent entrer dans le contenu des différents fichiers :

— le fichier d'en-tête `morceau1.hpp` contient :

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

#ifndef MORCEAU1
#define MORCEAU1
int f1(void);
int f2(int);
int f3(void);
#endif
```

Il y a la liste des bibliothèques nécessaires pour la compilation de `morceau1.cpp` et l'annonce des fonctions de ce même fichier, afin que celles-ci soient connues d'autres fichiers qui pourraient les utiliser.

— le fichier `morceau1.cpp` contient une première ligne `#include "morceau1.hpp"` pour connaître les bibliothèques requises puis le code des trois fonctions `f1()`,

`f2()` et `f3()`.

— le fichier `morceau2.hpp` contient

```
#include <fstream>
#include "morceau1.hpp"

#ifndef MORCEAU2
#define MORCEAU2
double g(int);
double h(double);
#endif
```

et sa deuxième ligne permet à `morceau2.cpp` de savoir que la fonction `f3()` existe quelque part dans un autre morceau.

— le fichier `morceau2.cpp` contient `#include "morceau2.hpp"` puis le code des fonctions `g()` et `h()`.

— le fichier `principal.cpp` contient

```
#include "morceau1.hpp"
#include "morceau2.hpp"

int main() {
        ...//tout le code de la fonction main()
};
```

car la fonction `main()` a besoin de connaître les prototypes de toutes les fonctions des morceaux qu'elle peut utiliser, ainsi que les bibliothèques nécessaires.

La compilation se fait ensuite en trois étapes distinctes, morceau par morceau :

```
g++ -c morceau1.cpp -lm
g++ -c morceau2.cpp
g++ morceau1.o morceau2.o principal.cpp -o principal.exe
```

L'option `-c` des deux premières lignes indique que le compilateur ne doit pas attendre de fonction `main()`, ni produire un fichier exécutable mais seulement produire des fichiers compilés : chacune de ces deux lignes produit un fichier `morceau1.o` ou `morceau2.o`. La dernière ligne compile le fichier `principal.cpp` et l'assemble avec les autres fichiers objets `.o` pour produire l'exécutable final. Le lancement de l'exécutable final se fait alors par la commande

```
./principal.exe
```

Si l'on modifie l'un des morceaux, il faut recompiler ce dernier ainsi que le fichier final. Cette procédure est assez lourde mais devient inévitable lorsque plusieurs personnes travaillent chacune sur un morceau d'un programme. Il nous reste à présent à expliquer les lignes qui commencent par des croisillons `#` : c'est le but de la section suivante.

Remarque : on ne compile jamais un fichier d'en-tête, d'extension `.hpp` ! Si vous le faites, cela peut produit des fichiers avec extension `.hpp.gch` qui ont potentiellement une taille énorme et nuiront à vos compilations suivantes.

Script de compilation. Lorsqu'il y a de très nombreux morceaux à compiler qui dépendent les uns des autres, il devient très vite nécessaire de recourir à des scripts de compilation qui (re)compilent automatiquement les morceaux nécessaires. Un outil puissant et simple d'utilisation est le logiciel `make`, basé sur des fichiers de configurations `Makefile` dont le fonctionnement est expliqué dans l'appendice, page 153.

2.4.3 Les directives de précompilation

Avant la compilation proprement dite qui transforme nos instructions en tâches réalisables par le processeur, `g++` a besoin de préparer le code et le programmeur peut influencer sur cette préparation par des directives de précompilation qui commencent systématiquement par un croisillon `#`.

La plus fréquente est la directive d'inclusion `#include` qui indique au compilateur quelle bibliothèque utiliser pour comprendre notre code. Les bibliothèques externes installées sur la machine doivent être appelées par leur nom entre chevrons `<nom_de_la_bibli>`. Pour des bibliothèques faites par l'utilisateur, leur chemin d'accès doit être spécifié entre guillemets `"nom_du_fichier_d_entete.hpp"` avec mention de l'extension `.hpp`.

Les directives de précompilation peuvent elles-mêmes contenir des tests logiques qui ont leur syntaxe propre. C'est en particulier utile pour des programmes réalisés en plusieurs morceaux qui ont chacun besoin d'une même bibliothèque. Reprenons l'exemple de la section précédente. Les fichiers `morceau2.hpp` et `principal.cpp` ont tous deux besoins de la bibliothèque `morceau1.hpp` et `principal.cpp` a elle-même besoin de la bibliothèque `morceau2.hpp` : le fichier `principal.cpp` a donc besoin deux fois de `morceau1.hpp`, une fois directement et une fois à travers `morceau2.hpp`. Ce n'est pas évident a priori mais cela pose un problème au compilateur ! En effet, il voit les mêmes fonctions déclarées plusieurs fois et, afin d'éviter toute collision malencontreuse, préfère afficher un message d'erreur. Il existe une solution simple pour éviter ce désagrément : insérer un test lors de la précompilation comme cela a été fait dans les fichiers `morceau1.hpp` et `morceau2.hpp` de l'exemple précédent :

- donner un nom à la partie du code concernée (ici la déclaration des fonctions de chaque fichier `.cpp`),
- vérifier si le morceau est déjà défini par `#ifndef LE_NOM` (**IF N**ot **DEF**ined),
- si ce n'est pas le cas, lui donner un nom pour le définir par `#define LE_NOM`,
- marquer la fin du bloc d'instruction par `#endif`.

2.4.4 Durée d'exécution d'un programme

Afin de tester l'efficacité d'un programme, il est facile de mesurer le temps d'exécution d'un programme sous Linux en utilisant la commande `time` lors de l'exécution du programme :

```
time ./NOM_DE_L_EXECUTABLE
```

Plusieurs informations s'affichent alors : seul le premier temps, exprimé en secondes, nous intéresse : il correspond au temps mesuré par le système d'exploitation entre le début et l'arrêt du programme.

Cet outil est utile pour mieux comprendre la notion de complexité : dans les différents exercices de TP, nous vous conseillons de mesurer la variation du temps d'exécution lorsque la taille des données à traiter varie. Une complexité en $O(N^\alpha)$ correspond à un temps d'exécution multiplié par environ 2^α lorsque la taille des données est multiplié par deux.

2.4.5 [hors programme] Compilation pour utilisation par d'autres langages de programmation : l'exemple de `swig`

Comme vous l'avez probablement découvert les années précédentes, il existe d'autres langages de programmation comme Python dont la syntaxe est bien plus légère, qui permet de faire l'affichage facilement mais que l'usage non-optimisé de la mémoire rend peu efficace pour des calculs numériques lourds. Il est donc souvent très intéressant de combiner la puissance des différents langages mais cela requiert d'écrire des interfaces pour passer de l'un à l'autre. En pratique, cela signifie qu'il y a besoin des briques suivantes :

- un programme `test.cpp` et son en-tête `test.hpp` en `C++` à convertir en une bibliothèque Python `test.py`
- des fichiers de description qui indiquent quelles options de compilation utiliser, quelles bibliothèques charger, etc.
- un logiciel qui fasse les bonnes compilations, produise une bibliothèque Python et écrive un programme qui permette la conversion systématique des objets d'un langage vers l'autre.

De tels logiciels existent et l'un de ceux les plus répandus est `swig` que nous allons utiliser avec la bibliothèque `distutils` de Python. Pour l'installer sous Linux, il suffit d'installer le paquet `swig` . Nous voyons ici comment l'utiliser sur un exemple.

Le fichier `RW.hpp` contient une classe en `C++11` qui permet de générer des marches aléatoires simples de paramètre p :

```
#include <random>
#include <vector>
class RW {
private:
        int state;
        unsigned _time;
        std::mt19937 RNG;
        std::bernoulli_distribution U;
public:
        RW(double p0=0.5, unsigned seed=1);
        void reset() { state=0; _time=0; };
        unsigned get_state() const {return state;}
        int iterate();
        std::vector<int> trajectory(unsigned n);
};
```

et il y a un fichier `RW.cpp` qui contient les codes des différentes méthodes.

Nous souhaitons à présent l'utiliser en Python. Pour cela, nous devons créer les fichiers suivants :

- un fichier `RW.i` , destiné à `swig` , qui explique à ce dernier qu'on utilise l'instanciation `std::vector<int>` de la STL (en effet, Python n'a ni *template*, ni STL mais `swig` sait transformer certains vecteurs de la STL du `C++` en objet Python via sa *propre* bibliothèque `std_vector.i`) :

```
%module RW
%{
#include "RW.hpp"
%}
%include "std_vector.i"
namespace std {
        %template(vectori) vector<int>;
};
%include "RW.hpp"
```

— un fichier `setup.py` qui explique à `swig` les caractéristiques de la compilation (utilisation de `C++11` , d'un éventuel parallélisme, etc) et de la bibliothèque Python à bâtir :

```
#!/usr/bin/env python
"""
setup.py file for SWIG example
"""
from distutils.core import setup, Extension
RW_module = Extension('_RW', sources=['RW_wrap.cxx', 'RW.cpp'],
                           extra_compile_args=['-std=c++11'])
setup (name = 'RW', version = '0.1', author = "Damien Simon",
       ext_modules = [RW_module], py_modules = ["RW"])
```

Ensuite, la magie de `swig` opère avec l'exécution successive dans le terminal des commandes suivantes :

```
swig -c++ -python RW.i
python setup.py build build_ext --inplace
```

La première ligne produit un nouveau fichier `RW_wrap.cxx` en `C++` qui contient le code nécessaire pour la conversion des différents objets entre Python et `C++`.

La seconde ligne est un script Python qui lance les compilations avec les options idoines sur les fichiers `RW.cpp` et `RW_wrap.cxx` puis les assemble en un fichier objet `_RW.so` utilisable par Python. Il produit également la bibliothèque `RW.py` pour Python.

L'utilisation en Python est alors simplissime. Considérons le programme `test.py` suivant

```
from RW import *
a= RW(0.5)
print a
print a.get_state()
u= a.trajectory(10)
print u
```

et produit la sortie :

```
<RW.RW; proxy of <Swig Object of type 'RW *' at 0x7f7b191bf7e0 > >
0
(0, -1, 0, -1, -2, -3, -4, -3, -4, -5)
```

que nous allons à présent commenter. La première ligne du programme charge la nouvelle bibliothèque. La déclaration d'un objet `a` de la classe `RW` en ligne 2 utilise le constructeur de la classe `RW` . L'affichage de `a` montre qu'en fait c'est un raccourci vers un objet `swig` sur un certain emplacement mémoire. L'utilisation des méthodes de la classe se fait en Python comme en `C++` en lignes 4 et 5. Le résultat de la ligne 5 est intéressante : elle utilise la méthode `trajectory` du `C++` qui produit en `C++` un vecteur de la STL ; la surcouche introduite par `RW_wrap.cxx` permet une conversion automatique vers le vecteur `u` en Python.

Il est très facile de généraliser l'exemple précédent à des bibliothèques beaucoup plus évoluées, tant que `swig` sait convertir les objets nécessaires de la STL vers d'autres langages.

Remarque : coût de la conversion. La solution présentée ici peut sembler magique et naturelle. Il faut néanmoins se méfier des coûts de calcul cachés dans la conversion. La conversion entre vecteur de la STL du `C++` et vecteur Python de la ligne 5 du programme précédent exige de parcourir le vecteur en entier (complexité en $O(N)$) : si les tailles sont grandes et que de trop nombreuses conversions sont utilisés, le programme peut passer plus de temps à réaliser l'interface entre langages qu'à calculer des résultats mathématiques !

Exercices

Exercice 2.1. Soient les fichiers suivants :

— fichier `a.hpp`

```
#include <_______>
#ifndef AAA
#define AAA
void f(int);
int add(int);
#endif
```

— fichier `a.cpp`

```
#include "a.hpp"
void f(int x) { std::cout << "Valeur: " << x <<std::endl;}
int add(int x) { return x+1 ;}
```

— fichier `b.hpp`

```
#include <cmath>
#include <random>//attention c'est du C++11
#include _____
#ifndef BBB
#define BBB
int g();
double h(double,int);
#endif
```

— fichier `b.cpp`

```
#include "b.hpp"
#include <ctime>
std::mt19937 G(time(NULL));
int g() { return std::binomial_distribution<int>(10,0.5)(G); }
double h(double x,int i) { return add(i)*x; }
```

— fichier `c.hpp`

```
#include <Eigen/Dense>
#ifndef CCC
#define CCC
int r(const Eigen::Matrix<int,4,4> &);
#endif
```

— fichier `c.cpp`

```
#include "c.hpp"
int r(const Eigen::Matrix<int,4,4> & A)
{       return 2*A.trace(); }
```

— fichier `d.cpp`

```
--------
--------
--------
#include <Eigen/Dense>
int main() {
        std::cout << h(4.,2.) << std::endl;
        Eigen::Matrix<int,4,4> A;
        for(int i=0; i<4;i++)
                for(int j=0; j<4; j++)
                        A(i,j)= g();
        f( r(A));
}
```

Compléter les en-têtes des fichiers puis écrire toutes les commandes de compilation pour obtenir un fichier de sortie intitulé `yes_we_can.exe` et l'exécuter plusieurs fois. Essayer de comprendre l'affichage du programme.

Chapitre 3

La syntaxe du C++

3.1 Les types

3.1.1 Généralités

Définition d'une variable. L'objet élémentaire de la programmation est la *variable* qui permet de stocker dans la mémoire un objet mathématique. La notion de *variable* en informatique est plus évoluée que la notion mathématique de *variable*. En effet, dans un langage dit « fortement typé » comme le `C++`, une *variable* est définie comme *un espace réservé en mémoire décrit par quatre qualités* :

1. son **nom** qui permet de l'identifier et la manipuler dans un programme,
2. son **type** qui est la convention de codage choisie pour la stocker dans la mémoire,
3. son **adresse** qui permet de décrire précisément l'endroit de la mémoire où elle est stockée
4. sa **valeur** qui est la séquence de bits écrite à son adresse dans la mémoire.

Le **nom** est choisi par le programmeur et n'a aucune incidence sur l'exécution du programme : il est donc important de choisir des noms les plus explicites possible afin d'être capable de lire efficacement des lignes de code.

La **valeur** est remplie directement par le programmeur par une *affectation* soit d'une valeur choisie au préalable, soit du résultat d'une fonction.

L'**adresse** est le plus souvent choisie directement lors de l'exécution du programme par le système d'exploitation qui cherche un espace disponible dans la mémoire et le réserve temporairement pour notre programme.

Le **type** est choisi par le programmeur lorsqu'il donne pour la première fois le nom d'une variable. Il est important de prendre conscience que la notion de type dépasse la description mathématique d'un objet et reste intimement liée à la gestion de la mémoire. Prenons l'exemple d'un entier $n = 19 = 16 + 2 + 1 = 2^4 + 2^1 + 1 = \overline{10011}_2$ en base 2 : la mémoire de l'ordinateur est faite de bits prenant les valeurs 0 et 1 et le codage en base 2 est le plus naturel. Encore faut-il choisir une convention pour savoir si l'on écrit les chiffres de gauche à droite ou à l'envers, une autre convention pour savoir où on écrit le signe de l'entier, etc. La même question se pose pour les réels afin de distinguer des écritures du type $10413 = 10,413 \times 10^3 = 1,0413 \times 10^4$.

Il est important de noter que le seul codage ne suffit pas pour avoir un type facilement manipulable : il faut également décrire les opérations élémentaires nécessaires (par exemple l'addition et le produit pour les entiers).

Il y a ainsi trois aspects à maîtriser :

— apprendre à utiliser la plupart des types usuels du C++ pour un usage mathématique élémentaire (cette section a pour but de les présenter),
— apprendre à utiliser les types un peu plus élaborés de la bibliothèque standard du C++, comme `std::vector` et `std::list` par exemple (c'est le but du chapitre 4),
— apprendre à créer des nouveaux types à partir des types élémentaires (c'est le but des chapitres 5 et 6).

Nous vous conseillons de bien travailler les deux premiers aspects avant de passer vous-même à la création de nouveaux objets.

Utilisation d'une variable. Avant toute utilisation, une variable doit être *déclarée* de la manière suivante :

```
int x;
int z;
char y;
```

en écrivant son type (ici deux entiers puis un caractère) puis le nom de la variable puis un point-virgule qui marque en C++ la fin d'une instruction.

L'*affectation* d'une valeur à une variable se fait par l'intermédiaire du signe `=` :

```
z= 19;
x= z+3;
y= 'u';/* les ' ' font la différence entre une variable u
 et la lettre u        */
```

Contrairement à l'usage mathématique, le signe `=` n'est pas un test d'égalité et n'est pas symétrique mais permet seulement de recopier la *valeur* du membre de droite dans la *variable* du membre de gauche. Ainsi, `19=z;` conduit à une erreur de compilation. D'autre part, il est important qu'à gauche et à droite d'un signe `=`, les deux objets aient le même type.

Il est possible de déclarer et initialiser une variable simultanément en écrivant :

```
int z=19;
```

Durée de vie et effacement. Une variable a une *durée de vie* et n'est définie que dans le bloc d'instruction où elle est déclarée, un bloc d'instructions étant délimité par des accolades `{...}`. Après la fin de ce bloc, l'espace mémoire où elle était stockée est effacé et son nom oublié : il est donc important de déclarer les variables aux bons endroits afin de ne pas perdre leur contenu en terminant un bloc d'instructions. La question de la durée de vie est la source de nombreuses erreurs de programmation.

3.1.2 Les types simples

Les entiers

Tout d'abord, une première remarque s'impose : il n'existe pas de type qui puisse gérer tous les entiers de $\mathbb{N}$ ou $\mathbb{Z}$ car la mémoire de l'ordinateur est finie et ne peut donc pas

stocker des nombres entiers arbitrairement grands. Il existe différents types d'entiers selon la taille maximale que l'on souhaite pouvoir atteindre :

— `short int` : codé sur 2 octets (16 bits, un pour le signe et ensuite 15 bits pour le codage du nombre en base deux : on peut donc atteindre $2^{15} = 32768$, c'est peu),
— `int` : codé sur 4 octets (32 bits, un pour le signe et ensuite 31 bits pour le codage du nombre en base deux, valeur maximale 2147483648)
— `long int` : codé sur 4 ou 8 octets
— `long long int` : codé sur 8 octets depuis la norme de 1998.

Plus le stockage est gros, plus les opérations seront lentes : c'est le prix à payer pour atteindre de grandes tailles.

Pour affecter des valeurs à ces variables, on peut les entrer directement :

— soit en base 10 :

```
int n=534;
```

— soit en base 8 en faisant précéder les chiffres par un zéro `0` muet :

```
int n=01026;
```

— soit en base 16 en faisant précéder les chiffres par un `0x` muet :

```
int n=0x216;
```

Si l'on souhaite n'utiliser que des entiers positifs, il est possible de récupérer le bit de signe pour multiplier par deux les valeurs maximales en déclarant les entiers par le type `unsigned` :

```
unsigned int x=34;
unsigned long int x=67;
```

Enfin, il existe une dernière manière de préciser la longueur du codage en utilisant `int_Nt` (ou `uint_Nt` pour la version sans signe) avec le symbole `N` remplacé par 8, 16, 32 ou 64 selon le nombre de bits utilisés.

Les opérations arithmétiques élémentaires disponibles sur les entiers sont les suivantes :

— l'addition et la soustraction par les symboles `+` et `-`

```
int x=4, y=7, z, w;
z= x+y;                 // z vaut alors 11
w= x-y;                 // w vaut alors -3
```

— le passage à l'opposé par le signe `-` :

```
int x=4,z;
z=-x;         // z vaut alors -4
```

— la multiplication par le symbole * (qui doit impérativement être écrit)

```
int x=4, y=7, z;
z= x*y;                 //z vaut alors 28
```

— le reste modulo m par le symbole % et le quotient euclidien par m par le symbole /

```
int x=37, m=5, z, w;
z= x % m;         //z vaut alors 2 car 32=5*7+2;
w= x / m;         //w vaut alors 7 car 32=5*7+2;
```

Dans les exemples précédents, nous avons rempli la valeur d'une troisième variable par opération sur deux autres variables. Très souvent, on souhaite seulement actualiser la valeur d'une variable en la combinant avec une autre. Il y a alors deux solutions et les deux codes suivants sont équivalents :

```
int x=4, y=7, z=5;
x = x+y;
z = z*y;
```

```
int x=4, y=7, z=5;
x += y;
z *= y;
```

Il n'y a (presque) aucune différence d'efficacité entre ces deux syntaxes et l'on peut choisir ce que l'on préfère. Il faut néanmoins être capable de comprendre les deux pour les circonstances où l'on doit lire le code d'autrui. Ces opérateurs existent bien évidemment également pour la soustraction, le reste euclidien et le quotient euclidien avec `-=`, `/=` et `%=`.

Enfin, lors du parcours d'une boucle, il est très fréquent de devoir incrémenter un compteur de 1 ; il existe quatre manières équivalentes de le faire :

```
n = n+1;
n += 1;
n++;
++n;
```

et, bien évidemment, la même syntaxe marche avec le signe `-` en lieu et place du signe `+`. Il existe une subtilité entre les deux derniers opérateurs et, pour éviter les pièges, nous déconseillons leur utilisation autrement qu'isolément.

Les « flottants »

Les problèmes de taille de stockage rencontrés pour les entiers se révèlent encore plus problématiques pour les nombres réels. De manière générale, il est raisonnable de considérer — du point de vue du programmeur — qu'il n'existe pas de nombres réels en informatique. La notion la plus proche est celle de « flottants » qui correspond à l'écriture d'un nombre réel sous la forme

$$\begin{aligned} 254,36 &= 2,5436 \times 10^2 = \overline{11111110,010111}_2 \\ &= \overline{1.11111110010111}_2 \times 2^7 = \overline{1.11111110010111}_2 \times 2^{\overline{111}_2} \end{aligned}$$

où $\overline{\bullet}_2$ désigne l'écriture en base 2. Dans la mémoire, un premier bit est réservé pour le signe, plusieurs octets sont utilisés pour représenter le nombre entier présent dans la puissance de 10 et enfin les bits restants pour la mantisse, i.e. le nombre sans virgule figurant devant la puissance de 10. Le nom de « flottant » provient du fait que l'écriture précédente s'appelle écriture en virgule flottante.

Les différents types de « flottants » sont ainsi caractérisés par le nombre de bits alloués à l'exposant et à la mantisse (tous deux écrits en base 2) :

- `float` : codé sur 4 octets, i.e. 32 bits (1 bit de signe, 8 bits pour l'exposant, 23 bits pour la mantisse), cela permet de couvrir des réels tronqués au sixième chiffre après la virgule et des exposants variant de -38 à 38.
- `double` : codé sur 8 octets, i.e. 64 bits (1 bit de signe, 11 bits d'exposants et 52 bits pour la mantisse), cela permet de couvrir des réels tronqués au quinzième chiffre après la virgule et des exposants variant de -308 à 308.

Il existe également un type `long double` mais nous ne l'utiliserons pas car les erreurs d'arrondi sont le plus souvent causées par des problèmes de calculs qui ne relèvent pas du choix des `double`. En général, nous choisirons de manière quasi-systématique les `double`.

Les opérations arithmétiques usuelles sont disponibles avec la même syntaxe que pour les entiers. Il est important de remarquer que la division avec l'opérateur `/` réalise pour, les flottants, la division réelle (multiplication par l'inverse) et non la division euclidienne, celle du cas entier. Un piège fréquent consiste à utiliser l'une à la place de l'autre. Le résultat de `11/3` vaut 3 alors que `11/3.0` vaut $3,66666\ldots$.

Les booléens

Les tests logiques n'ont que deux valeurs possibles, comme en mathématique : vrai et faux. Sur ces valeurs, il existe des opérations logiques élémentaires comme « et », « ou », « non », etc. Tout cela est géré en `C++` par le type `bool`.

Une variable de type `bool` ne prend que deux valeurs possibles, `true` (écrit aussi `1`) et `false` (écrit aussi `0`). Le tableau suivant résume les opérations logiques élémentaires :

Opération logique	Opérateur en `C++`	exemple
ET	`&&`	`a && b ;`
OU	`||`	`a || b ;`
NON	`!`	`!a ;`

Nous présentons à présent quelques opérateurs à valeurs dans les booléens. Un *test logique* est la vérification de la présence d'une relation entre deux objets. Cela ne modifie pas les objets soumis au test, ni n'alloue de mémoire. Le résultat d'un test logique est vrai ou faux et est donc un booléen.

Le test d'égalité entre deux variables de même type `a` et `b` s'écrit `(a == b)`. Il est important de ne pas le confondre avec le signe `=` qui efface l'ancienne valeur de `a` pour lui

substituer la valeur de `b` . Il y a également tous les opérateurs de comparaisons usuels `!=` (différent de), `<=` (inférieur ou égal), `>=` (supérieur ou égal), `<` (strictement inférieur) et `>` strictement supérieur.

Pour trois variables entières n, m et p de valeurs prescrites, on peut tester si $4 \neq n < m \leq p$ avec la syntaxe

```
(4 != n) && (n<m) && (m<=p)
```

Les énumérations

Imaginons que nous souhaitions décrire les couleurs d'objets parmi une palette finie de couleur. Quel type de variable utilisé ? Un premier choix consiste à associer un entier ou une lettre à chaque couleur et à bien documenté ce codage pour l'utilisateur du type ; c'est long et en général cela génère de nombreuses erreurs d'étourderies. Une seconde option consiste à utiliser une chaîne de caractères avec le nom de la couleur : c'est mieux du point de vue des erreurs potentielles mais cela demande d'alourdir énormément le code avec la gestion des chaînes de caractères et rend certaines énumérations difficiles. La construction de types avec un nombre fini de valeurs auxquelles on associe un nom se fait par :

```
enum TYPE { valeur1, valeur2, ..., valeurN};
```

Par exemple, la description des mois de l'année ou des couleurs de l'arc-en-ciel se fera par :

```
enum Mois {janvier, fevrier, mars, avril, mai, juin,
        juillet, aout, septembre, octobre, novembre, decembre};

enum Arc_en_ciel {rouge,orange,jaune,vert,bleu,indigo,violet};
```

La déclaration d'une variable de ce type se fait alors simplement par :

```
Mois rentree_scolaire=septembre;
Arc_en_ciel  espoir=bleu;
```

Les énumération `enum` sont très souvent utilisées avec des `switch` (cf. ci-dessous) qui permettent de tester des nombres finis de valeurs.

En interne, le compilateur associe un entier à chaque valeur et remplace à chaque apparation d'une valeur l'entier correspondant. L'entier est par défaut calculé en assignant 0 à la première valeur de la liste puis 1 à la deuxième, etc. Cela permet en particulier une conversion automatique vers les entiers de ces variables :

```
std::cout << rentree_scolaire;//affichera 8
```

Dans le cas des mois ci-dessus, il n'est pas malin de conserver le codage d'origine. Pour cela il suffit d'attribuer une valeur entière lors de la définition de l'énumération par :

```
enum Mois {janvier=1, fevrier, mars, avril, mai, juin,
        juillet, aout, septembre, octobre, novembre, decembre};

enum Test { cedric = 2, damien, raphael=7, nicolas, thibaut };
```

Dans le premier cas, la valeur 1 est associée à janvier puis on continue par incrémentation de 1 à chaque nouvelle valeur : on recouvre ainsi la numérotation naturelle des mois de l'année. Dans le deuxième cas, `cedric` est associé à 2, puis `damien` à 3, puis `raphael` à 7, `nicolas` à 8 et `thibaut` à 9.

Conversion implicite entre types

Mathématiquement, l'inclusion naturelle des nombres entiers dans les réels fait qu'on ne distingue pas l'entier 2 et le réel 2. En informatique, c'est différent puisque, comme nous l'avons vu, le codage en mémoire varie suivant la nature de l'objet considéré : il faut ainsi spécifier dans un programme les règles de conversion que l'on souhaite utiliser.

Heureusement, pour les types simples vus précédemment, il existe des conversions naturelles déjà programmées en `C++` mais il faut se méfier car cela est source de nombreux pièges. La seule conversion à connaître est celle des entiers vers les réels.

Si `x` est une variable de type entier (`int` , `unsigned` , `long int` , etc.), alors la syntaxe `double(x)` ou `(double)x` permet d'utiliser `x` à un endroit où un `double` est attendu.

Une utilisation fréquente de cette syntaxe est le choix de la division comme en témoigne l'exemple suivant :

```
int x=5, y=2;
std::cout << x/y ;                  // affiche 2 (division euclidienne)
std::cout << x/double(y);           // affiche 2.5 (division réelle)
std::cout << x/(double) y;           // affiche 2.5 (division réelle)
```

3.1.3 Les tableaux statiques

À partir d'un type élémentaire, il est possible de définir des tableaux de variables qui ont toutes ce type. Cela correspond en mathématiques à considérer des n-uplets. La déclaration d'un tableau statique se fait par la syntaxe :

```
int t[10];          // tableau de 10 entier au format int
double u[5]; // tableau de 5 réels au format double
TYPE  nom_du_tableau[TAILLE] ;        //syntaxe generique
```

où `TYPE` est le nom d'un type préexistant et `TAILLE` est un entier non-signé **constant** [1] qui donne le nombre de cases du tableau (i.e. le nombre d'éléments n du n-uplet).

1. Pour des questions de rétro-copatibilité avec des programmes écrits en langage C, des entiers déclarés non-constants sont tolérés mais sont déconseillés par la norme du langage. On préférera toujours utiliser les conteneurs `std::array<TYPE,TAILLE>` de la bibliothèque standard `<array>` .

Attention, dans un tableau de taille N :
les cases **ne** sont **pas** numérotées de 1 à N mais **de** 0 **à** $N-1$.
C'est la source de nombreux pièges !

On accède au contenu d'une case par l'opérateur `[]` appelé sur le numéro de la case :

```
int t[10];
double u[5];
t[0]=17;        // pour affecter 17 à la première case du tableau t
std::cout << u[4];  // pour afficher la dernière case du tableau u
```

Une fois déclarée, la taille d'un tableau est *statique* et ne peut être modifiée : il faut donc bien réfléchir lors de la déclaration. Le chapitre 7 et la bibliothèque standard présentée au chapitre 6 fourniront d'autres structures qui permettent de contourner cette restriction.

Il est possible de créer et utiliser des tableaux de tableaux (des matrices) par la syntaxe :

```
double M[10][15];
M[7][13]= 3.14159 ;
```

Néanmoins, dans de nombreux cas pratiques, il est souvent plus efficace de ne créer qu'un seul tableau de taille le produit des dimensions et d'accéder à la bonne case par un simple calcul arithmétique. L'équivalent de la syntaxe précédente est alors :

```
double M[10*15];
M[7*15+13]= 3.14159 ;
```

3.1.4 Les structures

Nous avons vu dans la section précédente comment assembler N variables de même type dans un nouveau type appelé tableau. Il est également possible de regrouper plusieurs variables de types différents dans une même variable, dont le type est appelée *structure*. Cette notion est fondamentale et est l'exemple le plus élémentaire de *classes*, telles que nous les définirons dans le chapitre 5.

Définir une structure revient à définir un nouveau type par la syntaxe suivante

```
struct NOM_DE_LA_STRUCTURE {
        type1       nom_d_acces_1;
        type2       nom_d_acces_2;
        ...
        typeN       nom_d_acces_N;
};
```

qui doit être placée le plus tôt possible dans le programme, avant tout usage d'une variable de ce nouveau type. Par exemple, pour décrire une plaque d'immatriculation française de la forme "AA 435 BZ" (i.e. deux lettres, un entier entre 1 et 999 et à nouveau deux lettres), on peut définir une structure du type :

```
struct Immatriculation {
        char debut1, debut2;
        int milieu;
        char fin1, fin2;
};
```

La déclaration d'une variable de ce type se fait ensuite par :

```
Immatriculation plaque1;
```

On utilise ensuite ce nouveau type et ces variables comme des variables usuelles. Il ne reste donc qu'à spécifier la méthode d'accès aux champs de la structure : cela se fait simplement par

```
nom_de_la_variable.nom_d_acces_i
```

Cette syntaxe avec un `.` est commune à la plupart des langages de programmation orientée objet comme le Python ou le Java par exemple.

Dans l'exemple précédent, on peut remplir et utiliser la variable `plaque1` de la manière suivante pour décrire la plaque « AC 782 ZT » :

```
Immat plaque;        // ou bien: struct Immatriculation plaque;
plaque.milieu= 782;
plaque.debut1= 'A';
plaque.debut2= 'B';
plaque.fin1= 'Z';
plaque.fin2= 'T';
// Pour l'affichage classique:
std::cout << plaque.debut1 << plaque.debut2 << " ";
std::cout << plaque.milieu << " ";
std::cout << plaque.fin1 << plaque.fin2 << " ";
```

Une structure peut également contenir comme champ une autre structure. Par exemple, pour décrire un véhicule, on peut souhaiter également noter son année de fabrication et la date du dernier passage chez le garagiste :

```
struct Date {
        short int jour, mois, annee;
}; //definition de la date par trois entiers
struct Voiture {
        Date fabrication;
        Date derniere_visite;
        Immat plaque;
};
```

En pratique, une structure contient peu de champs car il est souvent assez long de manipuler tous les champs dans un programme. De plus, il n'y a pas d'opération naturelle sur une structure, contrairement aux types simples vus précédemment. Cette notion était la seule présente dans le langage `C` et l'un des principaux apports du `C++` est de généraliser la notion de structure en programmation orientée objet : c'est pourquoi, dès que l'on souhaitera manipuler des objets plus complexes et plus évolués, on préférera utiliser des classes, décrites en détail au chapitre 5.

3.1.5 Les références

La plupart des types décrits précédemment correspondent à des objets mathématiques relativement bien définis. Toutefois, nous avons vu au début de cette section que, en informatique, une *variable* est un objet plus évolué qui comporte trois autres caractéristiques au delà de sa simple valeur (son nom, son type et son adresse) qui permettent de gérer la mémoire. Il faut donc définir des types spéciaux de variables dont les valeurs permettent de manipuler efficacement la mémoire. Ce sont les *pointeurs* d'une part et une sorte d'équivalent plus évolué en `C++`, les *références*. Les pointeurs, les références et leurs propriétés font l'objet de tout le chapitre 7 et permettent une gestion avancée de la mémoire.

Une *référence* est une variable dont la valeur est le nom (ou identifiant) d'une d'autre variable. Elle a son propre nom et, bien évidemment, sa propre adresse. Si la variable qu'elle identifie a pour type `int`, `double`, etc, alors la référence a pour type `int &`, `double &`, etc.

La référence identifie donc en mémoire la même zone mémoire que la variable qu'elle identifie. La syntaxe

```
double x=2.71;
double & ref=x; // une syntaxe alternative est: double &ref(x);
```

permet ainsi d'avoir en mémoire le nombre 2.71 écrit une seule fois, mais on peut y accéder par `x` ou bien par `ref`. Si l'on continue le code précédent par :

```
std::cout << ref << "\n";
ref += 1.05;
std::cout << x << "\n";
```

on observe que la ligne 1 affiche `2.71` puisque `ref` identifie `x`, puis la ligne 3 affiche `3.86`. En effet, le nombre `1.05` de la ligne 2 n'a pas été ajouté à `ref` (qui n'est pas un nombre réel mais un identifiant) mais à la variable que `ref` identifie (ici `x`).

On remarquera également que `int &ref=3;` n'a aucun sens. En effet, 3 est une valeur et n'identifie pas de variable particulière.

Non-réaffectation d'une référence. Une référence vers une variable ne peut pas être redirigée vers une autre variable. Regardons le résultat du code suivant :

```
int n=63;
int m=15;
int & ref=n;
ref=m;
n++;
std::cout << "Valeur de n: " << n << std::endl ;
std::cout << "Valeur identifiee par ref: " << ref << std::endl;
```

La ligne 4 ne change pas la référence `ref` vers `m` en remplacement de `n` car sinon la ligne 7 afficherait `78`. La ligne 4 met la valeur de `m` dans la variable identifiée par `ref`, i.e la variable `n` : à la fin de cette ligne, `n` vaut donc 15 et `ref` pointe toujours vers `n`. Les lignes 6 et 7 affichent donc toutes les deux `16`.

Utilité des références. De prime abord, cette notion semble inutile ; elle est en effet absente de nombreux langages de programmation. Son utilité en `C++` apparaîtra dans ce chapitre à la section 3.3 puis au chapitre 5 quand nous utiliserons l'héritage entre classes. Pour l'instant, il faut simplement retenir qu'une référence est une autre manière d'identifier une variable sans faire allusion à sa valeur mais seulement à sa place dans la mémoire.

3.2 Les tests et les boucles

Le cœur de l'algorithmique consiste à donner à l'ordinateur une liste d'instructions ; la liste elle-même peut différer selon les valeurs des variables. Le langage doit ainsi être capable de distinguer des cas et effectuer des actions en conséquences.

3.2.1 Le test `if (...) {...} else {...}`

Il est possible de choisir des instructions à exécuter selon la véracité d'un test logique avec la syntaxe :

```
if ( test logique )
{
        ...//instructions à réaliser si le test est vrai
}
else
```

```
{
        ...//instructions à réaliser si le test est faux
}
```

La partie en `else {...}` est optionnelle s'il n'y a rien à faire en cas de test faux.

Si nous souhaitons écrire le maximum de deux nombres réels `a` et `b` dans une variable `c`, il suffit d'écrire la portion de code suivante

```
if (a >= b)
{
        c=a;
}
else
{
        c=b;
}
```

Lorsque les instructions à réaliser sont uniques, il est possible d'alléger le code en se débarrassant des accolades qui ne sont là que pour délimiter les blocs d'instructions multiples. Le code précédent s'allège alors :

```
if (a >= b) c=a;
else c=b;
```

Il existe également une version encore plus concise de cette syntaxe lorsque les instructions sont des affectations comme dans le précédent calcul d'un maximum. Il existe un opérateur ternaire `(test)?(val1):(val2)` qui produit la valeur `val1` si le test est vrai et `val2` si le test est faux. Le code précédent devient ainsi :

```
c = (a>=b)?a:b;
```

Le choix entre les différentes syntaxes précédentes se fait selon la longueur des expressions à écrire : dans tous les cas, la lisibilité du code doit prévaloir.

Remarque : il est fréquent d'écrire par étourderie `=` au lieu `==` dans le test d'un `if` ; le compilateur ne signale aucune erreur et il se passe la chose suivante : l'affectation demandée par `=` a lieu (même si elle est à l'intérieur du test) et le test est en général considéré comme étant vrai. Si vous observez des valeurs incorrectes dans un code comprenant des tests d'égalité, c'est la première chose à vérifier.

3.2.2 Le test `switch`

L'usage du `if` ne permet que deux issues possibles puisqu'elle dépend de la valeur d'un booléen. Il est possible d'enrichir ce test selon les valeurs d'une variable par la syntaxe :

```
switch (VAL) {
case VAL1:
        ...//instructions1
        (eventuellement: break; )
case VAL2:
        ...//instructions2
        (eventuellement: break; )
...//autres cas et autres instructions
default:
        ...//instructions
};
```

L'action exacte de ce code n'est pas si immédiate à comprendre :

- le programme cherche dans la liste `VAL1` , `VAL2` , etc, la *première* occurrence de la valeur de `VAL` .
- S'il n'en rencontre aucune, il exécute les instructions du bloc après `default` si ce bloc est présent et ne fait rien si ce bloc est absent. S'il rencontre la première occurrence de `VAL` en, disons, `VALi` , il effectue toutes les instructions après la ligne `case VALi:` jusqu'à rencontrer l'instruction `break;` (même au-delà des lignes `case VALj:` suivantes s'il y en a).

Prenons un exemple :

```
switch (n) {
case 1:         std::cout << "A";
                 break;
case 2:         std::cout << "B";
case 3:         std::cout << "C";
case 4:         std::cout << "D";
                 break;
case 5:         std::cout << "E";
default:
                 std::cout << "F";
};
```

Nous avons alors l'affichage suivant selon les valeurs de `n` :

- `n=1` : le programme affiche seulement `A` ,
- `n=2` : le programme affiche `BCD` ,
- `n=3` : le programme affiche `CD` ,
- `n=4` : le programme affiche `D` ,
- `n=5` : le programme affiche `EF` ,
- `n=6` (ou plus) : le programme affiche `F` ,

C'est une instruction dangereuse à utiliser car il faut faire attention à disposer les `break` aux endroits idoines et nous éviterons autant que possible d'utiliser le `switch` .

3.2.3 Les boucles

La dernière notion algorithmique dont nous aurons besoin est celle de *boucle* qui permet d'itérer un cycle d'instructions tant qu'une certaine condition est vraie. Les boucles sont de trois sortes en `C++` mais ces trois sortes peuvent être utilisées les unes à la place des autres à condition de procéder à certains ajustements.

La boucle `for`. La première sorte correspond à la boucle `for` qui permet d'exécuter un cycle d'instructions selon la variation d'un paramètre :

```
for(initialisation ; test ; incrementation) {
        ...//instructions
}
```

où `initialisation` et `incrementation` sont des instructions, `test` est un test logique et `instructions` une succession d'instructions. La boucle `for` fonctionne alors ainsi :

- le programme exécute toujours `initialisation`
- tant que le résultat de `test` est vrai, le programme exécute le bloc `instructions` puis l'instruction `incrementation`.

Parmi les utilisations fréquentes, les boucles `for` sont utiles pour parcourir des tableaux. Par exemple, pour remplir un tableau `t` d'entiers avec les carrés des entiers naturels de `0` à `99`, nous pouvons écrire :

```
const int N=100;
int t[N];
for ( unsigned int i=0; i<N ; i++) {
        t[i]=i*i;
}
```

Remarque importante : l'erreur la plus classique est de ne pas bien écrire le bloc d'instructions ou la condition d'incrémentation, de sorte que la condition `test` ne devient jamais fausse : le programme reste alors éternellement bloqué à l'intérieur de la boucle.

La boucle `while`. Le `for` est pratique pour les incrémentations faisant intervenir des entiers ou des pointeurs (grâce à l'opérateur `++`) ; néanmoins, ce n'est pas toujours le cas en pratique. La boucle `while` permet plus de liberté par la syntaxe :

```
while( test ){
        instructions;
}
```

Tant que la valeur de `test` est vraie, le programme exécute le bloc `instructions`. Ici,

le programme commence toujours par vérifier que le `test` est vrai *avant* de commencer le bloc d'instructions. Celui-ci peut donc ne jamais être effectué si le test est faux d'emblée.

L'initialisation du tableau vue précédemment aurait alors pu se faire par :

```
int t[100];
int i=0;
while ( i<100 ) {
        t[i]=i*i;
        i++;
}
```

Là encore, il faut faire attention à ne pas oublier d'écrire `i++` sinon le compteur `i` reste éternellement à zéro et le programme réécrit indéfiniment la valeur `0` dans `t[0]` .

La boucle `do {...} while ()` . Il existe également la version suivante :

```
do {
        ...//instructions
} while (test);
```

Cette fois-ci, le programme exécute le bloc d'instruction puis vérifie la condition de `test` pour décider s'il recommence : le test a lieu *après* le bloc (contrairement au `while` précédent). Une conséquence est quele bloc d'instructions est toujours parcouru au moins une fois quelle que soit la valeur du test.

3.3 Les fonctions

3.3.1 Description générale

Nous avons déjà vu dans les exemples introductifs la subdivision du travail en fonctions auxiliaires : dans l'exemple de la page 12, nous avons défini une fonction `circle_area` qui prend un `double` en argument (le rayon du cercle) et produit un `double` qui est l'aire du cercle. Nous avons vu également que tous les programmes exécutables contenaient une fonction principale `main()` lancée directement à l'exécution du programme.

Une *fonction* en `C++` est un ensemble d'instructions (le corps de la fonction) qui peut être exécuté à plusieurs reprises dans un programme avec des paramètres (les arguments de la fonction) qui varient à chaque exécution et qui produit (éventuellement) une valeur de type donné.

Le *prototype* d'une fonction est la donnée

— du type du résultat (ici `TYPE_SORTIE`)

— du nom de la fonction (ici `NOM`)

— d'une liste de types pour chaque argument de la fonction (ici `TYPE1` , `TYPE2` ,..., `TYPEn`)

sous la forme suivante :

```
TYPE_SORTIE  NOM( TYPE1, TYPE2, ... , TYPEn );
```

En général, les prototypes des fonctions sans leur code sont annoncés dans les en-têtes des programmes, i.e. les fichiers `.hpp` , afin qu'elles puissent être utilisées indifféremment par plusieurs morceaux du programme. C'est l'équivalent en mathématiques de se donner les domaines et codomaines d'une fonction $f : E \times F \to G$.

Le *code* de la fonction est la suite des instructions qu'elle exécute. C'est l'équivalent en mathématiques de se donner une formule de calcul pour $f(x, y)$. Il est en général écrit dans les fichiers `.cpp` qui sont ceux qui sont compilés. Le code se présente sous la forme suivante :

```
TYPE_SORTIE  NOM( TYPE1 a1, TYPE2 a2, ... , TYPEn an) {
        ...//instructions
        return RESULTAT;
};
```

Autrement dit, on répète le prototype en attribuant des noms aux arguments afin de pouvoir les utiliser comme des variables dans les instructions. Le bloc d'instructions doit absolument se terminer par `return` qui détermine le résultat produit par la fonction. La variable `RESULTAT` , en général définie et calculée dans `instructions` , doit nécessairement être du type `TYPE_SORTIE` , sous peine de produire une erreur de compilation.

Il est possible qu'il n'y ait aucun résultat : cela est spécifié par le mot-clef `void` qui doit être utilisé à la place du type de sortie. Dans le cas d'un résultat absent, la fonction se termine plutôt par un simple `return;` . L'absence d'argument est indiquée par une liste vide `TYPE_SORTIE NOM()` : on peut également utiliser le mot-clef `void` comme en `C` mais ce n'est plus le standard.

Protection et copie des arguments et le rôle des références. Une spécificité du `C++` est qu'une fonction **ne peut jamais modifier la valeur de ses arguments**. Dans la syntaxe précédente, cela signifie qu'avant ou après un appel à une fonction, les variables mises en argument conservent toujours la même valeur (même si l'on écrit `a1=autre_valeur;` dans le code de la fonction). Cette règle impose de nombreuses contraintes mais permet aussi d'éviter de nombreuses erreurs, surtout quand on utilise des fonctions dont on ne connaît pas le fonctionnement interne précis.

Au niveau de la mémoire, cela fonctionne de la manière suivante : à chaque appel de la fonction, des *copies* des variables données en argument sont créées et la fonction travaille sur ces copies en laissant les originaux inchangés. Ces copies sont simplement effacées à la fin de l'appel correspondant [2].

Cela pose alors deux problèmes. D'une part, il y a des situations où l'on souhaiterait qu'une fonction puisse avoir un effet sur ses arguments et, d'autre part, la copie des

2. Pour les classes que nous définirons au chapitre 5, le destructeur est appelé.

arguments à chaque appel peut consommer inutilement beaucoup de temps et de mémoire, en particulier lorsque les données sont grosses. C'est alors qu'entrent en jeu les *références*, mentionnées en section 3.1.5, qui résolvent simultanément ces deux problèmes.

Changer dans le prototype d'une fonction un argument de `TYPE1 a1` à `TYPE1 & a1`, i.e. passer d'une variable à une référence vers cette variable, résout le problème de copie : en effet, la valeur d'une référence est seulement un identifiant — de petite taille — d'une variable et non sa valeur — potentiellement de grande taille— et cela permet un gain de temps et de mémoire significatif sur de grands tableaux.

D'autre part, la règle de non-modification d'un argument s'applique à présent à la référence dont la valeur est un identifiant. Cela n'est pas un problème car on ne peut pas changer un identifiant et de plus cela permet de changer la valeur identifiée par l'identifiant car elle n'est plus protégée par la règle de non-modification.

Dans la situation où l'on voudrait éviter une copie dans la mémoire *et* éviter la modification de la valeur identifiée par la référence, il existe le mot-clef `const` qui, écrit juste avant le type de l'argument, permet de préciser que les valeurs identifiées ne changent pas.

Prenons plusieurs exemples pour illustrer ce qui précède.

Exemple : affichage d'une structure et retour `void`. Supposons que nous voulions afficher une plaque d'immatriculation comme définie page 31. Taper les lignes d'affichage 8, 9 et 10 de l'exemple en changeant le nom de la variable `plaque` est long et c'est aussi une source d'erreur. Il vaut mieux ainsi faire une fonction que l'on réutilisera pour chaque affichage ou écriture, dans un fichier ou dans un terminal. On écrira ainsi :

```
void ecrire_immat( std::ostream & flux, struct Immatriculation p) {
        flux << p.debut1 << p.debut2 << " ";
        flux << p.milieu << " ";
        flux << p.fin1 << p.fin2;
        return;
}
```

et on peut ensuite remplacer les lignes 8, 9 et 10 de la page 31 par la simple ligne `ecrire_immat(std::cout, plaque)` pour écrire dans le terminal. On remarquera la présence d'une référence dans le premier argument : en effet, l'affichage modifie l'état du *flux* donc le passage par référence est nécessaire.

Dans cet exemple, le deuxième argument `p` est recopié dans la mémoire. Ce n'est pas très grave car c'est une petite structure mais, pour éviter cela, nous aurions tout aussi pu choisir le prototype suivant pour la fonction d'écriture précédente :

```
void ecrire_immat( std::ostream & flux, const struct Immatriculation &p);
```

Exemple : échanger deux variables. Supposons que nous ayons deux variables réelles `a` et `b` et que nous souhaitions faire une fonction qui échange leurs valeurs. Une telle fonction prendra ces deux variables comme arguments mais elle modifiera leurs valeurs. Il

faut donc des références et nous avons le code suivant :

```
void echange(double & a, double & b) {
        double c=a;
        a=b;
        b=c;
        return;
};
```

Exemple : trouver le nombre d'occurences d'une valeur dans un vecteur d'entiers. Nous verrons dans la section 4.4.1 qu'il existe une structure de tableau un peu plus évoluée, appelée `std::vector`. Soit `n` une valeur entière dont nous souhaitons trouver le nombre d'apparitions dans un tel vecteur d'entier. Cela se fait par l'exemple suivant :

```
unsigned int occurences(const vector<int> & V, int n) {
        unsigned compteur=0;
        for( int i=0; i<V.size() ; i++) {
                compteur += (V[i]==n)?1:0;
        }
        return compteur;
}
```

On remarquera l'usage d'une référence étiquetée `const` pour le premier argument : en effet, un tableau peut être extrêmement grand et, en général, il vaut mieux éviter de le recopier lors de l'appel d'une fonction sur ce tableau ; ici, en revanche, aucune valeur n'est modifiée et il vaut mieux alors étiqueter l'argument par `const`.

Cela est encore plus utile lorsque l'on travaille à plusieurs sur un même projet : dans ce cas, on ne souhaite en général pas lire en détail les `.cpp` écrits par autrui mais on souhaite savoir comment utiliser les fonctions écrites par autrui ; en particulier, il est nécessaire de savoir si ces fonctions modifient les valeurs derrière des références ou non. L'utilisation du `const` signale cette non-modification et permet au compilateur de vérifier qu'on ne tente pas de modifier des valeurs interdites.

Remarque sur les variables locales. Nous avons déjà mentionné au début de la section 3.1 la notion de durée de vie d'une variable. Les exemples précédents montrent que le code d'une fonction comporte des déclarations de variables. Il est important de prendre conscience des faits suivants.

Tout d'abord, une variable définie dans une fonction donnée est une variable *locale* et est inutilisable dans une autre fonction. Dans la pratique, il faut s'en tenir à la règle suivante : le code d'une fonction n'a le droit d'utiliser que les arguments de la fonction et des nouvelles variables qui lui sont internes.

De plus, toutes les variables locales déclarées dans une fonction donnée sont créées à chaque appel de la fonction et effacées à la fin de l'appel correspondant. A priori, il n'y a, d'un appel à l'autre, aucune valeur réutilisable.

Cette dernière règle peut être contournée en faisant précéder la déclaration une variable locale du mot-clef `static` . Cela permet à une variable locale de ne pas être effacée à la fin d'un appel et d'être ensuite réutilisée directement avec sa valeur obtenue à la fin de l'appel précédent. Un exemple classique est la génération d'une variable gaussienne par la méthode de Box-Müller qui génère simultanément deux réalisations indépendantes d'une telle variable aléatoire.

Valeurs par défaut d'arguments. Il est possible de donner des valeurs par défauts à une fonction lorsque l'on sait que la fonction sera utilisée souvent avec les mêmes arguments. Seuls les *derniers* arguments peuvent recevoir des valeurs par défaut et cela se fait lors de la déclaration du prototype de la fonction. Par exemple, supposons que nous souhaitions écrire une fonction qui réinitialise toutes les cases d'un vecteur d'entier à une valeur donnée et que, en général, cette valeur soit nulle. On peut alors écrire :

```
void reinit( vector<int> & V, int p=0) {
        for( int i =0 ; i < V.size() ; i++) {
                V[i]=p;
        };
        return;
}
```

Un appel `reinit(U,3);` fixe toutes les cases de `U` à la valeur 3, alors que `reinit(U);` fixe toutes les cases à la valeur 0.

3.3.2 Les λ-fonctions à partir du standard C++11.

Tout ce qui est décrit dans cette section a été introduit dans le standard C++11 et perfectionné en C++14 : il faut donc compiler les programmes avec l'option `-std=c++11` , `-std=c++14` ou tout autre standard supérieur.

Lorsqu'on utilise la bibliothèque standard `<algorithm>` décrite en section 4.6.3 ou lorsqu'on définit des classes, il arrive fréquemment que l'on ait besoin de définir des fonctions « jetables » dont le code est cours et que l'on utilise qu'une ou deux fois dans le programme.

Avant le standard C++11, il fallait nécessairement déclarer ces fonctions dans l'en-tête du programme ; cela avait le quadruple inconvénient de surcharger les en-têtes, de devoir écrire des prototypes compliqués, de ne pas pouvoir utiliser facilement des variables locales lors de l'appel de la fonction et de séparer les définitions de leurs quelques appels. Les λ-fonctions du standard C++11 résolvent simultanément tous ces problèmes !

La définition d'une λ-fonction se fait, *n'importe où dans le code*, de la manière suivante :

```
auto NOM = [capture de variables](TYPE1 arg1,TYPE2 arg2,...) -> TYPE {
        //code...
        return ...;
        };
```

Dans ce prototype, il y a à la fois des choses classiques et des nouveautés.

Les arguments (dans les parenthèses) sont déclarés comme d'habitude par leur type et leur nom, les uns à la suite des autres. Le type de retour `TYPE` est, quant à lui, placé *après* les parenthèses et précédé du symbole `->`. Il est important de remarquer que `-> TYPE` peut être parfois omis lorsque le type de retour est détectable par le compilateur. Enfin, le code de la fonction est, comme d'habitude, placé entre accolades `{...}`.

Décrivons à présent les nouveautés. Tout d'abord, le nom de la fonction est cette fois-ci placé dès le début, juste avant le signe `=`. Il y a le point-virgule final de la ligne 3. Il est en effet nécessaire car la syntaxe précédente définit `NOM` comme un *objet* (autrement c'est l'équivalent fonctionnel de `int n=4;` où `4` est remplacé par la définition de la fonction).

Enfin, comme toute variable, la définition est détruite à la fin du bloc où est définie la λ-fonction.

Le mot-clef `auto`. qui attribue un type automatique à l'objet `NOM`. En effet, l'idée des λ-fonctions est de simplifier le code pour des fonctions *jetables* donc il serait dommage de devoir nous-même attribuer un type à `NOM` sachant que ce type ne serait utilisé qu'une ou deux fois seulement. Pour cela, le compilateur attribue à `NOM` un type qu'il va utiliser en interne sans que nous ayons à nous en préoccuper.

La capture de variables. Comme dit précédemment, le code de ces fonctions n'est plus dans les en-têtes mais à *l'intérieur* du code d'une autre fonction au milieu des variables temporaires utilisées par cette fonction. Les λ-fonctions peuvent *capturer* ces variables soit par copie, soit par référence. On a le choix entre les syntaxes suivantes :

- `[=]` pour capturer toutes les variables nécessaires par copie,
- `[&]` pour capturer toutes les variables nécessaires par référence,
- `[a,b,&c]` pour capturer les trois variables `a`, `b` et `c`, les deux premières par copie et la troisième par référence,
- `[=,&c]` pour tout capturer par copie sauf `c` par référence,
- `[&,c]` pour tout capturer par référence sauf `c` par copie.

Exemple 1. À tout endroit du code, si l'on observe que l'on utilise fréquemment `x*x*x*x`, `y*y*y*y`, etc. pour de nombreuses variables, et que l'on souhaite mettre en évidence le côté puissance, alors, sans aucune perte d'efficacité mais avec un énorme gain de lecture pour le programmeur, on peut définir :

```
auto fourth_power = [](double x) { return x*x*x*x;};
```

et remplacer chaque apparition de `u*u*u*u` par `fourth_power(u)`.

Cela permet également au programmeur astucieux qui veut économiser une multiplication de changer uniquement la ligne précédente en :

```
auto fourth_power = [](double x) { double x2=x*x; return x2*x2; };
```

et il n'y a plus à changer chaque occurence de `u*u*u*u`.

Exemple 2. Un exemple de capture par référence est le suivant. Supposons que nous ayons un grand ensemble de chaînes de caractères `ES` décrits par `std::set<std::string>`, ainsi qu'un deuxième ensemble `FS` de chaînes de caractères. Nous souhaitons savoir si

tous les éléments de `FS` sont présents dans l'ensemble `ES` . Pour cela, nous écrivons le programme suivant :

```
std::set<std::string> ES;
std::set<std::string> FS;
// remplissage de ES et FS...
bool b= std::all_of(FS.begin(),FS.end(),
        [&ES](std::string & s) { return ES.count(s);}
        );
```

De tels exemples seront présentés en plus grand détail dans la section 4.6.3.

Exercices

Exercice 3.1. Soit le programme `overflow.cpp` suivant

```
#include <iostream>
int main() {
        unsigned int a = 8;
        unsigned int b = 13;
        std::cout << "Une soustraction étrange: " << a-b << std::endl;
        int u=100000;
        int v=u*u;
        std::cout << "Une multiplication étrange: " << v << std::endl;
        long long int uu=100000;
        long long int vv=uu*uu;
        std::cout << "Une multiplication étrange: " << vv << std::endl;
}
```

Comprendre le résultat de l'affichage. En tirer des conclusions sur les erreurs dues aux sorties de domaines de valeurs.

Exercice 3.2. (entrée/sortie et arithmétique)

1. Développer un court programme qui écrive dans un fichier `nombrescarres.dat` la liste des carrés des nombres entre 0 et 100 avec sur chaque ligne la phrase "Le carré de 0 est 0", puis "Le carré de 1 est 1", etc.
2. Développer un programme qui écrive dans un fichier `nombrespremiers.dat` la liste des nombres premiers entre 2 et 1000 avec sur chaque ligne la phrase "Le nombre premier numero 1 est 2", puis "Le nombre premier numero 2 est 3", etc.

   Compiler et exécuter votre programme. Vérifiez ensuite le contenu du fichier `nombrespremiers.dat` (la dernière ligne doit correspondre à 997).

Exercice 3.3. Soit le programme suivant :

```
#include <iostream>
#include <cmath> // pour la definition du nombre Pi comme var M_PI
double f(double x) { x += M_PI; return x;}
double g(double & x) { x += M_PI; return x;}
double h(const double & x) { x += M_PI; return x;}
int main() {
        double a=1.0, b=1.0, c=1.0;
        std::cout << f(a) << std::endl;
        std::cout << a    << std::endl;
        std::cout << g(b) << std::endl;
        std::cout << b    << std::endl;
        std::cout << h(c) << std::endl;
```

```
        std::cout << c << std::endl;
}
```

1. Le code de l'une des trois fonctions donne une erreur de compilation. Lequel ?
2. Supprimer la fonction problématique, exécuter le programme et comprendre le résultat.

Exercice 3.4. Soit la fonction `main()` suivante :

```
int main() {
        int a=0;
        if (a=2) std::cout << "Oui" << std::endl;
        else std::cout << "Non" << std::endl;
        std::cout << a << std::endl;
        return 0;
}
```

Inclure les bibliothèques nécessaires puis comprendre le résultat du programme. Compiler avec et sans l'option `-Wall` . Qu'aurait-il fallu utiliser dans le test pour obtenir `Oui` et pour que `a` ne change pas ?

Chapitre 4

La bibliothèque standard (STL)

4.1 Présentation de la bibliothèque standard

La notion de standard. Le standard `C++` consiste en une liste de spécification des mots-clefs et symboles du langage, des règles de syntaxe d'écriture de programme que doivent respecter les programmeurs et que les compilateurs doivent être capables de transformer en programme exécutable, ainsi que certains critères d'efficacité ou de complexité. En revanche, le standard ne spécifie pas tout le détail de l'implémentation et de la compilation.

Les chapitres précédents ont présenté quelques types élémentaires, comme les entiers et les doubles, ainsi que la syntaxe des fonctions. On pourrait tout à fait s'arrêter là et laisser chaque programmeur ou chaque bibliothèque « réinventer sa propre roue » pour un certain nombre d'algorithmes et de structures fréquemment utilisées. Le `C++` va plus loin en proposant *directement dans le standard* une liste d'algorithmes et de structures classiques, avec des noms, des fonctionnalités choisies au préalable et des critères de complexité, pour lesquels tout compilateur digne de ce nom doit proposer une implémentation. Ces algorithmes et ces structures sont alors regroupées dans ce que l'on appelle la *bibliothèque standard*. Toutes ces définitions sont basées sur des *templates* — notions présentées au chapitre 6 —, d'où le nom anglais de *Standard Template Library*, STL en abrégé.

Avantages et inconvénients. On peut tout d'abord se poser la question : pourquoi ne pas tout standardiser en une bibliothèque unique ? Tout simplement parce que, pour de nombreux algorithmes, à complexité optimale donnée, il n'existe pas d'implémentation idéale, mais plutôt plusieurs implémentations possibles avec chacune leur efficacité selon les conditions d'utilisation : c'est alors à l'utilisateur de choisir celle qu'il préfère. Ainsi, chaque implémentation de la STL fournit un candidat parmi les possibles, qui est toujours optimal en terme de complexité mais avec des variantes d'implémentations.

Il faut donc retenir que la STL n'est pas la solution à tout et ne contient pas tout mais c'est un candidat à toujours privilégier en première approche.

L'avantage de la standardisation est d'avoir une syntaxe *harmonisée* que tous les programmeurs connaissent et cela facilite la lecture des programmes. Un autre avantage est que la STL est *sûre et optimisée* : les algorithmes sont toujours les meilleurs connus en terme de complexité et l'utilisation très large de la STL fait que tout éventuel bug a déjà été trouvé et corrigé ; on peut ainsi avoir une confiance presque totale (ce n'est pas le cas du tout pour des bibliothèques à faible nombre d'utilisateurs...). Sauf en cas de recherche personnelle d'optimisations particulières, même pour des codes très courts, il est *préférable d'utiliser les outils de STL* plutôt que de réécrire sa propre version, avec tous les risques d'étourderie que cela comporte.

Pour être présent dans la STL, un algorithme et une structure doit être largement utilisée, avoir des algorithmes canoniques clairement définis, être efficaces numériquement et fonctionner sur tout type de machines et d'architectures. L'un des inconvénients est donc que Les nouveaux outils mettent ainsi du temps à être intégrés à la STL. En particulier, la bibliothèque standard ne permet pas de :

— décrire des structures de données en arbre,
— faire du calcul parallèle sur carte graphique,
— faire du calcul parallèle distribué,
— faire du calcul matriciel.

La documentation. La STL est immense et il est hors de question pour un débutant de tout connaître ! Néanmoins, il faut être conscient de l'existence de cette pléthore d'outils et savoir où aller chercher la documentation correspondante. Essentiellement, nous vous conseillons deux sites :

http://www.cplusplus.com/reference/
et
http://en.cppreference.com/w/cpp.

Le premier est souvent plus simple à prendre en main mais ne couvre pas les standards les plus récents (à partir de `C++` 14) ; le deuxième est plus récent et très complet.

La suite de ce chapitre présente différents outils de la STL avec des exemples de fonctionnalités et d'utilisations mais, en aucun cas, ne contient de documentation détaillée et **il est important de le lire et de travailler avec l'un des deux sites précédents ouverts devant vous** pour avoir tous les détails techniques et les syntaxes précises.

Notre sélection d'outils. Nous n'utiliserons dans ce cours que les éléments de la STL soit en lien avec les entrées-sorties, soit utiles pour les mathématiques appliquées de niveau M1.

La section 4.2 explique comment lire et écrire des données dans le terminal ou dans des fichiers.

Les sections 4.3, 4.4, 4.5 présentent des conteneurs fréquemment utilisés (qui correspondent en mathématiques à des n-uplets, des ensembles ou des fonctions sur un ensemble).

La section 4.6 est très importante en pratique et explique comment utiliser de manière *unifiée* toute une classe d'algorithmes élémentaires sur de très nombreux conteneurs, à travers la notion fondamentale d'itérateur.

Enfin, la section 4.7 présente toute la partie de génération de nombres aléatoires de la STL, sur laquelle nous baserons toutes nos simulations probabilistes.

La figure 4.1 présente une vue plus synthétique des principaux outils mathématiques de la STL.

Le préfixe `std::`. Tous les types et fonctions de la STL font partie de l'espace de nom `std`. Cela signifie en pratique que, pour les invoquer, le nom du type ou de la fonction doit commencer par `std::`. Cette nécessité provient du fait que plusieurs bibliothèques peuvent contenir des fonctions différentes avec le même nom et il est nécessaire pour les distinguer d'associer un espace de nom à chaque bibliothèque et de préfixer les noms des fonctions.

Il peut néanmoins être désagréable d'écrire partout — parfois plusieurs fois par ligne !—, le préfixe `std::`. Pour cela, au début d'une fonction ou même au début d'un fichier `.cpp` (mais jamais dans un fichier d'en-tête `.hpp` !), on peut utiliser

```
using namespace std;
```

qui permet, dans toute la suite du bloc correspondant, de se dispenser du préfixe `std::` pour les appels de types ou de fonctions.

4.2 Les entrées-sorties vers le terminal et les fichiers

Documentation. Une documentation complète est disponible aux adresses :

http://www.cplusplus.com/reference/iostream/ et
http://www.cplusplus.com/reference/fstream/

4.2.1 Lecture et écriture de données

De nombreux exemples déjà présentés font appel à `std::cout` , `std::endl` et `<<` pour permettre l'affichage dans le terminal. Comme l'espace de nom le précise, ces objets appartiennent à la STL et possèdent de nombreux analogues pour la lecture et l'écriture dans des fichiers. L'écriture et la lecture sont rassemblées dans deux en-têtes :

```
#include <iostream>        // lecture et écriture dans le terminal
#include <fstream>        // lecture et écriture dans des fichiers
```

Le mot `stream` correspond à « flux », `io` à *Input/Output*, i.e. « entrée/sortie », et la lettre `f` abrège *file*, i.e. « fichier ».

4.2.2 Écriture dans le terminal.

L'objet `std::cout` correspond à la sortie par défaut dans le terminal, i.e. la fenêtre de lignes de commande où le programme est exécuté. Toute instruction du type `std::cout << x` où `x` est un type reconnu par `<<` affiche la valeur de `x` dans le terminal, sans espace ni saut de ligne ni avant ni après.

L'affichage d'une chaîne de caractères donnée se fait par l'usage de `"abcdef"` entre guillemets anglais. Le saut de ligne se fait par l'objet `std::endl` , abréviation de *end of line*.

Ces objets sont définis dans `<iostream>` .

4.2.3 Lecture dans le terminal.

L'objet `std::cin` permet la lecture dans le terminal avec l'opérateur d'extraction `>>` suivi d'une référence de variable :

```
int i;
std::cout << "Entrez un entier et pressez la touche Entrée:" << std::endl;
std::cin >> i;
```

À l'exécution, la phrase de la ligne 2 apparaît dans le terminal puis le programme attend que l'utilisateur entre une chaîne de caractères et tape sur la touche "Entrée". Une fois cela réalisé, la ligne 3 convertit la chaîne tapée en une valeur entière stockée dans l'entier `i` .

Type	Objets ou fonctions	Bibliothèque à inclure	But
Nombres complexes	`std::complex<T>`	`<complex>`	Nombres complexes avec précision à choisir avec `T`
Entrée-sortie	`std::cout`, `std::cin`, `std::ofstream`, etc.	`<iostream>`, `<fstream>`	Lecture et écriture de données
Chaînes de caractères	`std::string`	`<string>`	Chaînes de caractères avec opérations de base.
Conteneurs	`std::vector<T>`, `std::set<T>`, `std::map<T1,T2>`, etc.	`<vector>`, `<set>`, `<map>`, etc.	Stockage structuré de données
Algorithmes	`std::count_if`, `std::all_of`, `std::any_of`, `std::replace`, etc.	`<algorithm>` (parfois `<numeric>`)	Algorithmes classiques sur les conteneurs.
Aléatoire	`std::mt19937_64`, `std::normal_distribution<T>`, `std::uniform_distribution<T>`, etc.	`<random>`	Nombres aléatoires et lois usuelles
Pointeurs intelligents	`std::unique_ptr<T>`, `std::shared_ptr<T>`, etc.	`<memory>`	Gestion avancée de la mémoire.
Calcul parallèle	`std::thread`	`<thread>`	Tâches à exécuter parallèlement sur plusieurs processeurs.

FIGURE 4.1 – Récapitulatif de bibliothèque standard.

La ligne 2 est inévitable pour assurer une bonne communication entre le programme et l'utilisateur ; en effet, sans elle, on aurait l'impression que le programme ne fait rien ou tourne indéfiniment alors qu'il est simplement en attente d'une action de l'utilisateur que ce dernier ne peut en général pas deviner.

L'utilisation de `std::cin` reste dangereuse bien qu'inévitable car aucune vérification n'est faite par le programme sur la compatibilité entre la chaîne tapée et le type de valeur attendu. Le programme tente une conversion implicite selon un certain nombre de schémas classiques (un entier est attendu sous la forme d'une suite de chiffres, un double comme une suite de chiffres avec éventuellement une virgule et une puissance de 10, etc) : si cela ne correspond pas à ce qui attendu, la valeur choisie est absolument quelconque. Pour éviter tout problème, c'est au programmeur de faire toutes les vérifications nécessaires, quitte à redemander à l'utilisateur une nouvelle valeur tant que cela ne correspond à une valeur valide.

4.2.4 Lecture et écriture dans un fichier.

Cela requiert la bibliothèque `<fstream>` . Il faut alors déclarer un flux par fichier en précisant le chemin d'accès au fichier souhaité :

```
std::ofstream Fichier1("chemin/nom_du_fichier1"); // pour l'écriture
std::ifstream Fichier2("chemin/nom_du_fichier2"); // pour la lecture
```

Si le fichier est dans le même répertoire que le programme, il n'est pas nécessaire de spécifier le chemin d'accès. L'utilisation est ensuite très simple puisque tout se fait avec les opérateurs d'injection et d'extraction `<<` et `>>` . Enfin, il ne faut pas oublier de fermer les flux de fichiers après leur utilisation avec les commandes

```
Fichier1.close():
Fichier2.close();
```

C'est un oubli fréquent qui a pour conséquence de parfois tronquer les fichiers avant la fin.

L'opérateur d'extraction `>>` navigue dans un fichier d'espace en espace jusqu'à arriver à la fin du fichier. Pour décider quand s'arrêter, il faut savoir si l'on est à la fin ou non : cela se fait par la commande `Fichier2.eof()` qui renvoie `true` si et seulement la fin du fichier est déjà dépassée (et non atteinte). Une solution alternative bien commode est de placer directement la commande d'extraction de flux dans une condition de test : la conversion implicite d'un flux vers un booléen donne `true` si et seulement si l'extraction s'est bien déroulée.

Exemple : formatage de fichier. Supposons que nous ayons un fichier dont le nom est `a_trier.txt` et qui contient trois colonnes sous la forme :

```
2       3       5.16
1       2       7.18
4       5       -1.23
7       2       2.47
```

et nous souhaitons en faire deux fichiers intitulés `tri1.txt` et `tri2.txt` qui contiennent :

```
2       3
1       2
4       5
7       2
```

et

```
2       5.16
1       7.18
7       2.47
```

i.e. le premier contient les deux premières colonnes et le deuxième les première et troisième colonnes pour les lignes dont le troisième terme est positif. Tout cela peut être réalisé par :

```
std::ifstream source("a_trier.txt");
std::ofstream sortie1("tri1.txt"), sortie2("tri2.txt");
int x,y;
double v;
while (source >> x >> y >> v)
{
        sortie1 << x << "  " << y << std::endl;
        if (v>=0.) sortie2 << x << "  " << v << std::endl;
}
source.close();
sortie1.close();
sortie2.close();
```

Le code précédent est dangereux car il suppose que le programmeur et l'utilisateur sont une seule et même personne : autrement dit, les lignes pour l'extraction supposent que le fichier `a_trier.txt` existe déjà et correspond au bon format. Si le programme est destiné à d'autres utilisateurs, il faut bien évidemment ajouter de nombreuses lignes de vérifications de l'état du fichier afin de gérer les cas où l'utilisateur aurait fourni un fichier au mauvais format.

4.3 Les chaînes de caractères `string`

Documentation. Deux références complètes sont disponibles aux adresses :

http://www.cplusplus.com/reference/string/string/
et
https://en.cppreference.com/w/cpp/string/basic_string

Description. Nous avons vu en section 3.1.3 comment gérer des chaînes de caractères avec le type `char []` , déjà présent en `C` . La STL fournit une classe[1] `std::string` pour traiter les chaînes de caractère avec de nombreuses fonctionnalités supplémentaires.

1. Il faut faire attention à ne pas confondre cette bibliothèque avec la bibliothèque `cstring` qui permet d'utiliser en `C++` une bibliothèque faite en `C` qui comporte moins de fonctionnalités.

Il faut tout d'abord charger la bibliothèque dans les en-têtes de fichier avec :

```
#include <string>
```

La déclaration d'une chaîne de nom `s` se fait alors avec la ligne `std::string s;` avec éventuellement `std::string s("abcdef");` pour l'initialiser à partir d'un tableau de caractères.

Voici quelques fonctionnalités utiles :

— l'accès au caractère numéro i (à partir de 0) se fait par `s[i]` ;

— la commande `s.size()` donne la longueur de la chaîne ;

— l'ajout d'un caractère à la fin de la chaîne se fait par `s += 'a';` ;

— étant données deux chaînes `s1` et `s2` , `s1+s2` donne la concaténation des deux chaînes (attention, ce n'est pas commutatif !) ;

— étant donnés une chaîne `s1` et un entier positif `i` , `s.insert(i,s1)` insère la chaîne `s1` dans la chaîne `s` à la i-ème position ;

— étant donnés un entier positif `i` (et éventuellement un autre entier positif `l`)), la commande `s.erase(i);` (ou éventuellement `s.erase(i,p);`) efface le i-ème caractère (ou p caractères à partir du i-ème).

— la fonction `getline` lit une ligne complète (sans s'occuper des espaces) dans un flux et la stocke dans un objet de type `string` avec

```
std::getline(std::cin, s);
```

Il existe également des commandes permettant de trouver des caractères ou des sous-chaînes dans une chaîne plus grande, d'isoler une sous-chaîne dans une chaîne, etc.

4.4 Les conteneurs ordonnés `std::vector` et `std::list`

4.4.1 Les tableaux

Documentation. Deux références complètes sont disponibles aux adresses :

http://www.cplusplus.com/reference/vector/vector/
et
https://en.cppreference.com/w/cpp/container/vector

Description. Nous avons vu dans la section 3.1.3 comment gérer des tableaux statiques. Comme nous l'avions annoncé alors, ces tableaux statiques sont particulièrement adaptés à des tailles petites et fixes, or de nombreuses situations requièrent des tableaux dont la taille est grande et/ou peut fluctuer. De plus, les tableaux statiques hérités du langage `C` ont une structure syntaxique qui ne permet pas de tirer partie pleinement des autres outils de la STL présentés en section 4.6 et sont à éviter [2]. Pour décrire des tableaux dynamiques, i.e. de taille quelconque et potentiellement variable, la STL fournit, entre autres, un modèle de classe `std::vector` .

2. Pour conserver la propriété de taille fixe et avoir une implémentation équivalente mais compatible avec la STL, il suffit d'utiliser le modèle `std::array` du standard `C++` 11)

Commençons néanmoins par une remarque mathématique sur ce nom trompeur de « vecteurs ». En mathématiques, un vecteur est un élément d'un espace vectoriel, de dimension finie ou non, sur un corps ; en revanche, en `C++`, un « vector » est une suite finie d'éléments d'un ensemble T (qui n'est pas nécessairement un corps...), autrement un élément de :

$$\bigcup_{n\in\mathbb{N}} T^n$$

avec la convention $T^0 = \{\omega\}$ où ω est un élément appelé le vecteur vide. Un élément de E^n est un n-uplet $(x_0, x_1, \ldots, x_{n-1})$, i.e. une suite finie traditionnellement numérotée en `C++` à partir de 0.

Le modèle de classe `std::vector<T>` où `T` est un type quelconque (qui correspond à l'ensemble mathématique précédent) est défini dans la bibliothèque

```
#include <vector>
```

La déclaration d'un vecteur `v` d'objets de type `T` se fait par l'une des lignes :

```
std::vector<T>  v;
std::vector<T>  v(n);   //n entier
std::vector<T>  v(n,t); //n entier, t de type T
std::vector<T>  v(w); // copie de w de type std::vector<T>
```

Dans le premier cas, le vecteur est vide ; dans le deuxième cas, il est de longueur `n` et ses valeurs sont initialisées à la valeur par défaut du type `T` (si elle existe) ; dans le troisième cas, le vecteur est de taille `n` et toutes ses cases contiennent la valeur `t`.

Cet objet possède de nombreuses méthodes qui peuvent toutes être appelées avec la syntaxe

```
objet.methode(args);
```

Par exemple, il existe une méthode `size()` qui donne le nombre d'éléments de l'objet par la syntaxe :

```
unsigned int p;
p=v.size();
```

L'accès à l'élément d'indice i (toujours à partir de 0, attention !) se fait également par la syntaxe `v[i]`. Si i est supérieur ou égal à la taille du vecteur, le comportement du programme est indéfini [3]. Les accès aux premier et dernier éléments se font aussi directement par

```
v.front();
v.back();
```

3. Autrement dit, il peut s'interrompre brutalement ou tout aussi bien continuer en renvoyant une valeur quelconque

Il est facile d'ajouter ou d'enlever un élément *à la fin du vecteur*, en faisant ainsi augmenter ou diminuer sa taille d'une unité avec les commandes `push_back(x)` et `pop_back()` [4]. Une valeur x de même type que les éléments du `vector` peut être ajoutée en fin du tableau par la commande :

```
v.push_back(x);
```

Cet élément peut être supprimé du tableau par la commande

```
v.pop_back();
```

Pour des raisons algorithmiques de bonne gestion de la mémoire, il n'est pas possible d'ajouter un élément en tête de vecteur (si on souhaite le faire, il est possible d'utiliser le conteneur `std::deque`) et il est faisable mais coûteux en temps d'ajouter un élément au milieu du vecteur par la méthode `insert` : au prix d'une perte de performances sur d'autres méthodes, ces opérations deviennent faciles et rapides sur les `std::list` présentées plus loin.

Un exemple récapitulatif. Voici un code que nous vous encourageons à exécuter sur votre ordinateur, voire à modifier pour tester les modifications de la classe.

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
int main() {
        vector<double> V(13,1); //Vecteur de taille 2000 initialisé à 1
        for (unsigned int i = 0; i < V.size(); i++) {
                V[i]=i*i;
        }
        for (unsigned int i = 0; i<V.size(); i++) {
                cout << i << ": " << V[i] << "\n";
        }

        V.pop_back();
        cout << "Taille après pop_back: " << V.size()  << "\n";
        V.push_back(M_PI);
        cout << "Dernier élément: "<< V.back()
                << " (taille après push_back: " << V.size()  << ")\n";

        int k=0;
        for (unsigned int i = 0; i < V.size(); i++)
        {
                if( cos(V[i])<0. ) k++;
        }
        cout << "Il y a " << k << " éléments de cosinus négatifs.\n";
```

4. Cette dernière ne peut bien sûr par être appelée sur un vecteur vide.

```
26          return 0;
}
```

Ce code, typique du standard C++ 03, est parfaitement valide et respecte toutes les règles du C++. Néanmoins, il est source de nombreuses erreurs d'étourderie : mauvais indices (à partir de 0 ou 1), lecture parfois difficiles des boucles `for` (surtout si les blocs sont longs), identification parfois difficile des opérations réalisées, etc. Nous verrons en section 4.6 (page 63) comment réécrire ce code en C++ 11 pour le rendre à la fois plus moderne, plus facile à lire et à comprendre et moins sujets aux étourderies.

Structure en mémoire (ne lire qu'après avoir lu le chapitre 7 !). Il nous reste à comprendre quelle astuce est utilisée en mémoire pour réaliser tout cela. Derrière ce modèle de classe, ce sont bien des tableaux statiques mais qui ont une taille réelle en mémoire, appelée *capacité*, différente de leur taille apparente. On accède à la capacité par la syntaxe `v.capacity()` .

Augmenter la taille d'un tableau statique est long car cela demande de trouver une place plus grande en mémoire et de copier toutes les cases vers le nouvel emplacement. La notion de capacité permet de raréfier ces réallocations par la dynamique suivante. Si on utilise `push_back()` juste après la déclaration du tableau, la taille augmente de 1 mais la capacité est multipliée par 2 ; si, au contraire, on utilise `pop_back()` , la taille décroît de un mais la capacité ne bouge pas (et la dernière case reste réservée mais non utilisée). Cela permet d'éviter des réallocations lors des prochains usages de `push_back()` .

Attention, une conséquence de cette gestion de la mémoire est qu'il ne faut pas remplir un vecteur vide à travers n usages de `push_back` , sans avoir au préalable réserver la bonne capacité à travers `v.reserve(n);` qui permet d'augmenter en une seule fois la capacité sans faire varier la taille : cela évite ainsi toutes les copies partielles à chaque doublement de capacité.

Remarques. Il existe de nombreuses autres méthodes dont la documentation est disponible en lignes mais, attention, ce n'est pas parce qu'elles existent qu'il faut les utiliser sans réfléchir. En effet, il en existe plusieurs qui sont pratiques mais inefficaces et qu'il vaut mieux utiliser le moins possible. Toutes les méthodes présentées dans cette section correspondent à des algorithmes efficaces.

Il est important de remarquer également que rien n'est prévu pour l'opérateur `<<` car l'affichage dépend beaucoup de l'usage souhaité par l'utilisateur. Une simple boucle `for` pour parcourir un vecteur fait le plus souvent l'affaire.

4.4.2 Les listes

Documentation. Deux références complètes sont disponibles aux adresses :

http://www.cplusplus.com/reference/list/list/
et
https://en.cppreference.com/w/cpp/container/list

Description. Le conteneur `std::list<T>` , utilisable après l'inclusion dans les en-têtes

Méthode	Fonctionnalité	Exemple	Complexité
`push_back(T x)`	Ajoute un élément à la fin du vecteur	`v.push_back(x);`	$O(1)$ (*)
`pop_back()`	Supprime le dernier élément	`v.pop_back();`	$O(1)$
`operator[](int i)`	Accesseur et mutateur de la case `i`	`v[i]`	$O(1)$
`size()`	Renvoie la taille du vecteur	`v.size()`	$O(1)$
`resize(int n)`	Change la taille à `n`	`v.resize()`	variable (*)
`clear()`	Réinitialise le vecteur à une taille nulle	`v.clear()`	$O(1)$
`front()` , `back()`	Accède au premier et dernier élément du vecteur	`v.front()` , `v.back()`	$O(1)$
`begin()` , `end()`	Renvoie un itérateur vers le premier et le après-le-dernier élément	`v.begin()` et `v.end()`	$O(1)$
`insert(It, x)`	insère la valeur `x` à l'endroit pointé par l'itérateur `It` (**inefficace**)	`v.insert(v.begin(),x)` est équivalent à `push_front()` pour des listes	$O(N)$
`erase(It)`	efface la case pointée par l'itérateur `It`	`v.erase(v.begin())` efface la première case	$O(N)$
`swap(vector<T> u)`	échange les contenus de l'objet courant et `u`	`v.swap(u)`	variable

TABLE 4.1 – Autres fonctionnalités du modèle de classe `std::vector<>` . Pour alléger le tableau, l'espace de nom `std::` a été supprimé et `T` désigne un type quelconque. Les complexités étiquetées (*) ne sont vraies que lorsque la capacité n'a pas à varier.

```
#include <list>
```

fournit un conteneur alternatif qui décrit le même objet mathématique $\bigcup_{n\in\mathbb{N}} T^n$ mais qui utilise une gestion de la mémoire complètement différent et propose ainsi des fonctionnalités différentes.

Commençons par les points communs. Comme pour les vecteurs, les déclarations de variables se font par l'une des quatre instructions

```
std::list<T>  L;
std::list<T>  L(n);   //n entier
std::list<T>  L(n,t); //n entier, t de type T
std::list<T>  L(w); // copie de w de type std::list<T>
```

avec les mêmes effets. Il y a également des méthodes

```
L.size();
L.front();
L.back();
```

qui permettent d'accéder à la taille, au premier élément et au dernier élément.

Contrairement à `std::vector` , le stockage en mémoire ne se fait plus de manière contiguë mais par un jeu astucieux de pointeurs (cf. chapitre 7). L'avantage est d'avoir cette fois-ci les *six* opérations d'ajout et de retrait de valeurs en début comme en fin de listes, qui ont toutes une complexité $O(1)$:

```
L.pop_back(); //retrait du dernier élément
L.push_back(x); //ajout d'un élément x à la fin
L.pop_front(); //retrait du premier élément
L.push_front(x); //ajout d'un élément x en tête.
L.insert(It,x); //ajout d'un élément juste avant le lieu It
L.erase(It);//retrait de élément au lien It
```

où les objets `It` sont des itérateurs décrits en section 4.6.

Le prix à payer pour ces nouvelles fonctionnalités à la complexité la plus faible possible est qu'il n'est plus possible d'accéder directement à l'élément d'indice i (autre que le premier et le dernier). Heureusement, dans de nombreuses applications, cela reste inutile et l'on souhaite seulement pouvoir parcourir la liste dans l'ordre : cela sera fait en détail là encore en section 4.6. Le code de test de la classe `std::vector` présent page 4.4.1 ne peut pas être écrit sans itérateur pour les listes ; en revanche, sa réécriture en page 63 est directement adaptable aux listes comme nous le verrons.

Remarque. Une étude détaillée du modèle de mémoire derrière les vecteurs et les listes est présentée au chapitre 7 et sera approfondie en amphithéâtre.

4.5 Les conteneurs non ordonnés std::map et std::set

4.5.1 Les ensembles, std::set et std::unordered_set

Documentation. Deux références complètes sont disponibles aux adresses :

http://www.cplusplus.com/reference/set/set/,
http://www.cplusplus.com/reference/unordered_set/unordered_set/
et
https://en.cppreference.com/w/cpp/container/set,
https://en.cppreference.com/w/cpp/container/unordered_set

Description. Les deux conteneurs précédents `std::vector` et `std::list` correspondent à des n-uplets mathématiques, i.e. n éléments numérotés et éventuellement égaux. Les conteneurs `std::set<E>` et `std::unordered_set<E>` permettent de considérer des ensembles finis à n éléments (nécessairement différents) de même type `E`. En supposant que le type `E` est une implémentation de l'ensemble mathématique E, cela revient ainsi à décrire l'ensemble des parties finies de E.

L'utilisation de ces conteneurs requiert les inclusions

```
#include <set>
#include <unordered_set>
```

La déclaration d'un ensemble de type `E` se fait par l'une des instructions suivantes :

```
std::set<E> A; //ensemble vide
std::set<E> B(A); //copie de A
std::set<E> B(v.begin(),v.end()); // v std::vector<E> ou std::list<E>
```

La dernière construction correspond mathématiquement à la transformation $(x_1, \dots, x_n) \mapsto \{x_1, \dots, x_n\}$ où l'ensemble à droite peut être de cardinal strictement inférieur à n si certains éléments sont redondants.

Les opérations essentielles disponibles sont les suivantes :

```
A.clear(); // remet A à l'ensemble vide
A.insert(x); // ajout de x (de type E) s'il est absent, rien sinon
A.erase(x); // retrait de x (de type E) s'il est présent, rien sinon
A.erase(It); // retrait de l'élément présent au "lieu" It
A.size(); // taille/cardinal de A
```

Afficher le contenu complet d'un `std::set<E>` se fera là encore grâce aux itérateurs en section 4.6.

Remarque technique. On ne peut utiliser dans `std::set<E>` que les types `E` muni d'un ordre $<$; en revanche, cet ordre peut être quelconque. Il existe en général déjà pour la plupart des types existants et, en son absence, on peut aussi en définir un soi-même. Ce conteneur existe depuis longtemps.

Pour pallier cette question d'ordre, le standard C++11 a ajouté le nouveau conteneur `std::unordered_set<E>` qui, comme le nom l'indique, ne requiert plus d'ordre $<$ (mais

seulement des fonctions de hachage sur lesquelles nous ne dirons rien). Il est fréquent que ce conteneur soit plus rapide que `std::set<E>`, c'est pourquoi nous le mentionnons.

4.5.2 Les fonctions à domaine de définition fini et le conteneur map

Documentation. Deux références complètes sont disponibles aux adresses :

http://www.cplusplus.com/reference/map/map/,
http://www.cplusplus.com/reference/unordered_map/unordered_map/
et
https://en.cppreference.com/w/cpp/container/map,
https://en.cppreference.com/w/cpp/container/unordered_map

et les paires sont documentées sur cette page :

http://www.cplusplus.com/reference/utility/pair/

Description. Se donner une fonction $f : D \subset E \to F$ de domaine de définition D fini revient à se donner l'ensemble fini des paires $\{(x, f(x)); x \in D\}$. De telles fonctions sont ainsi encodées dans la STL par les deux conteneurs `std::map<E,F>` et `std::unordered_map<E,F>`, qui sont essentiellement basées sur la classe `std::pair<E,F>` pour décrire les éléments $(x, f(x))$ et sur les conteneurs `std::set<T>` et `std::unordered_set<T>`, où `T` est remplacé par `std::pair<E,F>` pour gérer les ensembles de paires. Nous nous contenterons ici du seul conteneur `std::map`.

Les paires. La classe `std::pair<E,F>` fonctionne de la manière suivante :

```
std::pair<E,F> U; //E et F deux types quelconques
U.first; //le premier élément de U
U.second; // le deuxième élément
U=std::make_pair(x,y);//crée une paire à partir de x et y de types E et F
```

et nous soulignons le fait que `first` et `second` *ne* sont *pas* des méthodes mais directement des champs publics de la classe : il n'y a donc pas de double parenthèse `()` à la fin.

Le conteneur `std::map`. L'usage de `std::map` requiert l'inclusion en en-tête de

```
#include <map>
```

La définition d'une variable de ce type se fait par :

```
std::map<E,F> f; //fonction vide
std::map<E,F> g(f); // copie de g
std::map<E,F> g(v.begin(),v.end());
        // v de type std::vector< std::pair<E,F> > (ou liste)
```

Les fonctionnalités basiques sont les suivantes :

```
y=f[x]; // si x est présent, renvoie son image f(x),
        // si x est absent, renvoie la valeur par défaut vF du type F
        //        et ajoute (x, vF) à f (dangereux !!!)
y=f.at(x); // si x est présent, renvoie son image f(x)
        // si x est absent, le programme s'interrompt brutalement
f.size(); // taille du domaine de déf.
f.clear(); // remet f à l'état vide.
```

Nous verrons là encore en section 4.6 comment afficher l'intégralité du contenu d'un objet de type `std::map` .

Un exemple d'application. Ce type de conteneur est pratique par exemple pour établir, entre autres, des dictionnaires ou des correspondances. On pourrait par exemple associer à trois lieux *Paris*, *Toulouse* et *Clermont-Ferrand* leur superficie en kilomètres carrés. Cela peut se faire de la manière suivante :

```
std::map< std::string, double> m;
m["Paris"]= 105.4;
m["Toulouse"]= 118.3;
m["Clermont-Ferrand"]= 42.67;
std::cout << "Nombre de villes: " << m.size() << "\n";
std::cout << "Superficie de Paris: " << m["Paris"] << " km2\n";
std::cout << "Nombre de villes: " << m.size() << "\n";
std::cout << "Superficie de Marseille" << m["Marseille"] << "km2\n";
std::cout << "Nombre de villes: " << m.size() << "\n"; //ATTENTION !!!!!!
```

4.6 Itérateurs et algorithmes sur les conteneurs : *toute la puissance de la STL !*

4.6.1 Un exemple simple et basique : dénombrer des éléments selon un critère.

Considérons le code suivant :

```
std::vector<double> v(200);
double a=2.4;
// ... remplissage de v ... //
int k=0;
for(unsigned int i=0; i< 200 ; ++i ){
        if( v[i] >= a ) k=k+1;
}
std::cout << k;
```

Derrière cette apparente simplicité se cachent de nombreux défauts à plus au moins long terme :

— il n'est pas évident de savoir que comprendre ce que fait exactement ce code [5] (bien sûr, on pourrait ajouter une ligne de commentaire pour expliquer).
— source d'erreurs d'étourderie dans l'écriture de la boucle (en particulier la taille du vecteur) ;
— modifier le programme *a minima* pour une liste ou un ensemble est impossible ;
— cela n'est pas du tout dans l'esprit du chapitre 6 dans lequel on cherche à écrire des morceaux de code qui s'appliquent à un maximum de structures possibles.

Remarquons tout de suite que ce type de code reflète typiquement ce que l'on écrirait en `C` ou en Fortran et possède des équivalents directs dans à peu près tous les langages de programmation (sauf les fonctionnels). Une meilleure manière de l'écrire, *typique du `C++` et sa bibliothèque standard* consiste à écrire :

```
std::vector<double> v(200);
double a=2.4;
// ... remplissage de v ... /
auto larger_than_a = [=](double x) { return x>=a; };
std::cout << std::count_if(v.begin(),v.end(),larger_than_a);
```

Dans ce morceau de code, il y a plusieurs avantages :
— le nom de la λ-fonction `larger_than_a` ainsi que le nom `count_if` de l'algorithme rend immédiatement clair que l'on est en train de compter des éléments selon un critère de taille ;
— chaque ligne correspond à une brique élémentaire de raisonnement mathématique ;
— il n'y ni la taille du vecteur (seulement les mots-clefs explicites que sont `begin` et `end`), ni d'indice pour énumérer les cases ;
— les deux dernières lignes seraient *exactement les mêmes* pour des `std::list` ou des `std::set` de nombres réels.

Le but de la présente section est de comprendre quelle est la nature des objets (les itérateurs) derrière `begin()` et `end()` qui rendent cela possible, comment manipuler ces objets, où trouver et comment utiliser les modèles de fonctions comme `std::count_if` .

4.6.2 Les itérateurs

Définition. Ici, un conteneur est n'importe quel objet de type `vector` , `list` , `set` , `unordered_set` , `map` , etc, de la STL.

De manière simplifiée, un *itérateur* sur un *conteneur* est un objet qui permet de parcourir toutes les cases d'un conteneur sans en oublier aucune (en respectant autant que possible une notion d'ordre). Ainsi, toutes les opérations comme
— afficher toutes les valeurs
— compter les valeurs selon un critère
— appliquer une transformation à toutes les valeurs
— trouver si une valeur est présente
— sommer des valeurs sur un conteneur
reviennent à parcourir toutes les cases d'un conteneur en effectuant une certaine opération sur chaque case. Cela signifie qu'on peut donc utiliser à profit utiliser les itérateurs.

De manière plus technique, un *itérateur* sur un conteneur `C` de la STL satisfait les propriétés suivantes :

5. Ici, l'accusation est légère car le programme fait 3 lignes avec un `std::cout` mais il faut imaginer un programme un peu plus long et un peu moins trivial.

1. `C` possède deux méthodes `C.begin()` et `C.end()` qui renvoient des opérateurs sur le début du tableau et la fin du tableau (attention, la fin n'est pas la dernière case mais un indicateur que la dernière case est déjà franchie) ;
2. un itérateur `It` désigne l'emplacement mémoire d'une case du tableau (avec parfois quelques informations supplémentaires) ;
3. un itérateur `It` possède une opération `*It` qui renvoie la valeur stockée à l'emplacement désigné par `It` ;
4. un itérateur `It` possède une opération `++It` qui permet d'avancer d'une case à la suivante ;
5. on a la garantie qu'en commençant à `C.begin()` et terminant à `C.end()` et en utilisant seulement `++It` , alors on aura parcouru *toutes* les cases du conteneur ;
6. pour les conteneurs ordonnés, le parcours de toutes les cases se fait selon l'ordre du conteneur ;

Syntaxe des itérateurs au sein de la STL.

Pour déclarer une variable de type itérateur, il faut utiliser :

```
std::CONTENEUR<TYPE>::iterator It;
std::CONTENEUR<TYPE>::const_iterator Itc;
```

Dans le premier cas, la syntaxe `*It` permet d'accéder aux valeurs en lecture comme en écriture ; dans le second, seul l'accès en lecture est autorisé : de tels itérateurs étiquetés sont requis dans toutes les fonctions qui manipulent un conteneur avec une étiquette `const` .

On peut ainsi réécrire le code de la page 61 de la manière suivante pour des vecteurs comme pour des ensembles :

```
std::vector<double> v(200);
std::set<double> E;
double a=2.4;
// ... remplissage de v et E ... //
int k1=0;
for(std::vector<double>::const_iterator it=v.begin(); it != v.end();++it){
        if( *it >= a) k1=k1+1;
}
std::cout << k1 << "(pour le vector) \n";
int k2=0;
for(std::set<double>::const_iterator it=E.begin(); it != E.end();++it){
        if( *it >= a) k2=k2+1;
}
std::cout << k2 << "(pour le set) \n";
```

On voit ainsi la similarité de code entre les deux conteneurs et cela ouvre la voie vers le chapitre 6. Cela résout ainsi l'un des inconvénients mentionnés précédemment mais malheureusement les autres restent inchangés et la syntaxe est lourde [6].

6. Cette lourdeur de syntaxe, avec des noms de type très longs et malheureusement absolument nécessaires, a souvent été reprochée au `C++` dans les versions antérieures à C++11.

Le standard C++11 a résolu la question de la lourdeur de syntaxe par deux introductions majeures. Tout d'abord, il y a eu l'ajout du mot-clef `auto` : puisqu'il est clair qu'au sein de la STL, tout conteneur produit via `begin()` un itérateur sur ce conteneur, il est inutile que le programmeur rappelle ce type explicitement. Ainsi, on peut remplacer les débuts de boucles du code précédent par

```
for(auto it=v.begin(); it != v.end();++it){...}
```

et laisser ainsi le compilateur récupérer le nom exact du type dans le code de la STL.

Le deuxième ajout consiste en les syntaxes appelées *range-for* :

```
for( auto x : v) {
        //manipulation de x qui vaut successivement
        // les valeurs de chaque case
}
for( auto & x : v) {
        //manipulation de x qui référence successivement
        //chaque case pour les modifier
}
for( const auto & x : v) {
        //manipulation de x qui référence successivement
        //chaque case sans les modifier
}
```

qui sont strictement équivalentes aux codes suivants :

```
for( auto p = v.begin(); p != v.end(); ++p) {
        auto x = *p;
        // manipulation de x obtenu par copie.
}
for( auto p = v.begin(); p != v.end(); ++p) {
        auto & x = *p;
        // manipulation de x
}
for( auto p = v.begin(); p != v.end(); ++p) {
        const auto & x = *p;
        // manipulation de x
}
```

et permettent un code beaucoup plus léger et lisible.

De manière générale, il est assez rare — et heureusement !— en C++11 de devoir utiliser les noms de type des itérateurs. Cela ne dispense pas néanmoins de les connaître au cas où on se verrait imposer le standard C++03.

4.6.3 Les algorithmes de `<algorithm>` et `<numeric>`

Documentation. Deux références complètes sont disponibles aux adresses :

http://www.cplusplus.com/reference/algorithm/ (<algorithm>),
http://www.cplusplus.com/reference/numeric/ (<numeric>)
et
https://en.cppreference.com/w/cpp/algorithm (<algorithm>),
https://en.cppreference.com/w/cpp/numeric (<numeric>)

Description. L'idée générale derrière cette partie de la STL est qu'en pratique on est toujours en train d'utiliser les mêmes briques élémentaires algorithmiques : compter selon un critère, trier, appliquer la même transformation à de nombreux points, trouver un élément, etc. Ainsi, plutôt que de réécrire toujours la même boucle avec les problèmes inhérents d'étourderie et de changement de noms de variables, la STL propose des implémentations d'algorithmes génériques pour de très nombreuses tâches.

Les implémentations sont garanties de la meilleure qualité possible et certaines d'entre elles sont mêmes parallélisées à partir du standard C++17.

Nous vous encourageons à vous reporter à la documentation ci-dessus pour découvrir les immenses possibilités. Outre la syntaxe de base, ce qui distingue un bon programmeur et un très bon programmeur est souvent une profonde connaissance des outils de la STL.

À partir de C++11, l'usage de <algorithm> est considérablement amélioré par l'existence des λ-fonctions décrites en section 3.3.2 et nous vous conseillons de relire cette dernière pour profiter pleinement de la présente section.

Exemple : le code de la page 61 remanié. En utilisant toutes les possibilités de la STL (itérateurs et algorithmes), nous pouvons réécrire le code précédent sous la forme concise et parfaitement explicite suivante :

```
std::vector<double> v(200);
double a=2.4;
// ... remplissage de v ... //
std::cout<< std::count_if(v.begin(),v.end(),[a](double x){return x>= a;});
```

et la même syntaxe s'appliquerait à tout autre conteneur de la STL sans la moindre modification !

Exemple : le tri de vecteur ou de liste. Si les éléments d'un vecteur ou d'une liste C sont d'un type ordonné (i.e. pour lequel l'opérateur < a un sens), la bibliothèque <algorithm> fournit un algorithme de tri rapide (complexité $O(N \log N)$ en la taille de la structure) par la fonction std::sort . Le tri des éléments d'un vecteur C complet se fait par la syntaxe :

```
std::sort(C.begin(),C.end());
```

Si on souhaite trier les éléments selon un autre critère de tri, il est possible d'ajouter en argument supplémentaire une fonction ou une λ-fonction de prototype

```
bool f(const T & x,const T & y)
```

où `T` est le type des éléments du tableau et qui renvoie `true` si et seulement si `x` est strictement plus petit que `y`.

Bien évidemment, ce genre d'algorithme n'a aucun sens sur un ensemble `std::set` qui n'est pas un conteneur ordonné. En cas d'erreur, cela se traduit par une erreur à la compilation.

Exemple : transformer selon une formule Soient deux vecteurs (ou listes) *de même taille*, le premier constitué de paires de nombres réels et le second de nombre réels. On souhaite parcourir le premier, calculer le produit des deux nombres et le stocker dans la case correspondante du second.

Cela peut se faire de la manière suivante :

```
std::vector< std::pair<double,double> > source;
// ... remplissage de source ...//
std::vector< double > cible(source.size());
auto f = [](std::pair<double,double> p) { return p.first*p.second; };
std::transform(source.begin(),source.end(),cible.begin(),f);
```

Exemple : somme/produit/maximum d'évaluations. Soit $f : E \to \mathbb{R}_+$ une fonction implémentée par une certaine fonction `double f(const E &)` en `C++`. Supposons que, étant donné un conteneur C ordonné ou non, nous souhaitions évaluer :

$$S = \sum_{x \in C} f(x) \qquad P = \prod_{x \in C} f(x) \qquad M = \max_{x \in C} f(x)$$

Le code naïf consisterait à parcourir le conteneur par un itérateur du premier jusqu'au dernier élément en calculant les sommes, produits et maxima partiels.

Mathématiquement, cela correspond à la construction suivante : étant donnée une suite $(c_k)_{0 \leq k < n}$ de valeurs de E (correspondant au parcours de C du premier élément au dernier), un nombre inital u_0 et une fonction $t : \mathbb{R} \times E \to \mathbb{R}$, on construit la suite $(U_k)_{0 \leq k \leq n}$ par

$$\begin{cases} U_0 = u_0, \\ U_{k+1} = t(U_k, c_k), \quad 1 \leq k \leq n \end{cases}$$

Les valeurs de S, P et M correspondent toutes au terme U_n mais avec des fonctions t différentes : $t(u, c) = u + f(c)$ pour la première, $t(u, c) = uf(c)$ pour la deuxième et $t(u, c) = \max(u, f(c))$ pour la dernière.

Le calcul de U_n par cette récurrence est précisément ce que fait le modèle de fonction `std::accumulate` de la bibliothèque numérique de prototype

```
std::accumulate(debut_iterator,fin_iterator,u0,t)
```

où `t` est la formule permettant de calculer la somme/produit/maximum partiel(le) à partir de le(la) précédent(e) et de la nouvelle valeur et `u0` est la valeur initiale.

Étant donné un conteneur `C` préalablement rempli de nombres complexes et la fonction module $\mathbb{C} \to \mathbb{R}_+$, $z \mapsto |x|$, nous pouvons écrire :

```
auto partial_max= [](double x, std::complex<double> u)
        { return std::max(x,std::abs(u)); };
auto partial_sum= [](double x, std::complex<double> u)
        { return x+std::abs(u); };
auto partial_prod= [](double x, std::complex<double> u)
        { return x*std::abs(u); };
double S= std::accumulate(C.begin(),C.end(),0.,partial_sum);
double P= std::accumulate(C.begin(),C.end(),1.,partial_prod);
double M= std::accumulate(C.begin(),C.end(),0.,partial_max);
```

pour calculer les sommes, produits et module maximal d'un ensemble de nombres complexes, quel que soit le conteneur utilisé.

4.6.4 Des extensions des itérateurs bien utiles en pratique

Les itérateurs d'entrée-sortie

Les données dans un fichier ou même l'affichage dans le terminal consistent souvent en une suite ordonnée d'objets de même type ; afin de pouvoir éviter les outils de `<algorithm>` , il devient souhaitable d'avoir des itérateurs sur un fichier de données ou sur le terminal. Cela est effectivement fait dans la bibliothèque `<iterator>` via les objets `std::ostream_iterator` (pour l'écriture) et `std::istream_iterator` (pour la lecture).

La documentation est accessible ici :

http://www.cplusplus.com/reference/iterator/iterator/

Cela nécessite l'inclusion suivante dans l'en-tête :

```
#include <iterator>
```

Écriture. On définit un itérateur sur le *début* d'un flux de sortie (terminal ou fichier) à remplir par des objets de type `T` par :

```
std::ostream_iterator<T> output_it1(std::cout,", ");
                // terminal et séparateur , suivi d'une espace
std::ofstream file_flux("nomdufichier.txt");
std::ostream_iterator<T> output_it2(file_flux,"\n");
                //fichier et séparateur saut de ligne
```

Cela s'utilise ensuite comme un `C.begin()` partout où cela a un sens. Par exemple, on peut souhaiter écrire le contenu de n'importe quel conteneur d'objets de type `T` dans le terminal séparé par des virgules, en adaptant l'exemple suivant.

```
std::list<double> L(10);
// remplissage de L par les entiers successifs commençant à 3
std::iota(L.begin(),L.end(),3); // dans <numeric>
// affichage:
```

```
std::ostream_iterator<double> output_it(std::cout,",");
std::copy(L.begin(),L.end(),output_it);
```

et cela affichera dans le terminal :

```
3,4,5,6,7,8,9,10,11,12,
```

Lecture. On définit un itérateur d'objets de type `T` (lisibles par `input >> t`) en lecture sur le terminal ou sur un fichier de la manière suivante :

```
std::istream_iterator<T> input_it1(std::cout);
                // terminal et séparateur , suivi d'une espace
std::ifstream file_flux("nomdufichier.txt");
std::istream_iterator<T> input_it2(file_flux);
                //fichier et séparateur saut de ligne
```

Cette fois-ci, il n'y a plus de séparateur comme deuxième argument. En revanche, il est nécessaire d'avoir l'équivalent de `C.end()` pour décider à quel moment les algorithmes vont arrêter la lecture : le constructeur par défaut joue précisément ce rôle, que ce soit sur le terminal ou sur un fichier, via

```
std::istream_iterator<T> end_of_flux;
```

qui, s'il doit être utilisé fréquemment, peut être défini une fois pour toute au début de la fonction `main()` ou comme variable globale.

Supposons alors que nous ayons un fichier de nom `complexnumbers.txt` composé des trois lignes suivantes :

```
(1,0)
(0,-2)
(4,3)
(1,1)
```

correspondant aux nombres complexes 1, $-2i$, $4+3i$ et $1+i$ avec la syntaxe qu'il faut pour lire un nombre complexe par `>>` . Il est ainsi possible ainsi de remplir un vecteur de taille 4 par la syntaxe suivante :

```
std::vector< std::complex<double> > V(4);
std::ifstream F("complexnumbers.txt");
std::istream_iterator< std::complex<double> > input(F);
std::istream_iterator< std::complex<double> > end_of_file;
std::copy(input,end_of_file,V.begin());
F.close();
```

sans besoin d'utiliser la syntaxe avec des `while` et des `>>` vue au chapitre précédent.

Cela permet aussi de faire des statistiques sur un très grand fichier sans avoir à le charger entièrement dans la RAM via un conteneur. Imaginons ainsi un fichier contenant des millions [7] de nombres complexes écrits dans un fichier `large_data.txt` et un scénario dans lequel nous souhaiterions écrire leurs modules dans un fichier `modules.txt`. Cela peut se faire facilement via :

```
// Ouverture de la source
std::ifstream F("large_data.txt");
std::istream_iterator< std::complex<double> > input(F);
std::istream_iterator< std::complex<double> > eof;
// Ouverture de la cible
std::ofstream G("modules.txt");
std::ostream_iterator< double > output(G);
// Conversion
std::transform(input,eof,output,std::abs<double>);
// Fermeture;
F.close(); G.close();
```

Les itérateurs d'insertion

Des algorithmes tels que `std::transform`, `std::copy`, etc., utilisent des itérateurs sur deux conteneurs, l'un correspondant à la source des valeurs, l'autre correspondant à la cible où l'on stocke les nouvelles valeurs. Pour des conteneurs ordonnés tels `std::vector` ou `std::list`, cela nécessite seulement que le conteneur cible soit d'une taille suffisamment grande pour recevoir les résultats.

Si l'on souhaite que la cible soit un conteneur non ordonné de type `std::set`, il n'est plus possible de prévoir une telle taille à l'avance et affecter les valeurs aux cases restées vides. En revanche, on peut toujours partir d'un conteneur cible vide et *insérer* les valeurs pour faire grandir le conteneur. Pour cela, il faut un autre type d'itérateur qui puisse accéder aux cases et ajouter des nouvelles cases à chaque fois. Cela est possible grâce aux *itérateurs d'insertion* prévus dans la bibliothèque `<iterator>`.

Illustrons cela sur un exemple. Supposons que nous souhaitions transformer un vecteur en un ensemble, i.e. construire l'ensemble $\{x_0, \ldots, x_{n-1}\}$ à partir du n-uplet $(x_0, \ldots, x_{n-1})$, en utilisant l'algorithme `std::copy`. Cela donnerait le code suivant :

```
std::vector<double> V;
//... remplissage de V... //
std::set<double> E; // l'ensemble E est initialement vide
std::copy(v.begin(),v.end(),std::inserter(E,E.end()));
```

où `std::inserter` est défini dans `<iterator>` et construit un itérateur d'insertion sur l'ensemble `E` à partir de sa fin [8].

Nous n'irons pas plus loin sur ces questions et nous vous encourageons à explorer la vaste STL à partir de sa documentation et à poser vos questions à vos enseignants.

7. Nul besoin de savoir le nombre exact !

8. Pour un `std::set`, il est équivalent de mettre `E.begin()` ou `E.end()` car le conteneur est ordonné. En revanche, pour insérer dans une liste à partir d'un endroit donné, le deuxième argument est important.

4.7 L'aléatoire en C++

Ce bref chapitre traite essentiellement de la programmation de variables aléatoires en `C++`. D'une part, les problèmes posés ne sont pas évidents du point de vue conceptuel et, d'autre part, la gestion de l'aléatoire en `C++` a beaucoup évolué avec les derniers standard du `C++`. La norme de 2011 a apporté beaucoup de nouveaux outils très performants mais cette norme n'est pas encore utilisée partout et il est nécessaire de connaître également les anciens outils.

4.7.1 L'aléatoire et la programmation

Génération de nombres aléatoires

Commençons par un constat accablant : il n'est pas possible de produire de l'aléatoire par un simple programme sans apport extérieur. En effet, toutes les instructions vues dans les chapitres précédents sont — heureusement ! — déterministes et ne laissent donc place à aucune variabilité lors de leur exécution. Il existe alors deux solutions :

— écrire un programme qui va lire des données produites de manière aléatoire par autre chose qu'un programme informatique,
— essayer d'imiter de manière déterministe des données aléatoires.

La première solution est celle que nous ne suivrons pas car elle est coûteuse en temps et la qualité obtenue n'est pas nécessairement meilleure que celle produite par la seconde solution : elle n'est utile que dans quelques applications cryptographiques. La vitesse lente vient du fait qu'il faut aller lire une grande quantité de données (à chaque échantillonnage d'une variable aléatoire) sur une source externe (autre que la RAM) : cela prend du temps et il n'y a pas beaucoup de moyens de produire des données aléatoires rapidement.

La seconde solution consiste à trouver une fonction `int aleatoire()` qui produit à chaque appel un nouvel entier et telle que la suite d'entiers produits *mime* — dans un sens à préciser — une suite d'entiers indépendants et identiquement distribués.

Cela est fait en général par un code du type :

```
unsigned int x=graine;

unsigned int aleatoire() {
        static unsigned int valeur=x;
        valeur=f(valeur);
        return valeur;
}
```

où `f` est une fonction d'itération. Ce code signifie qu'on commence par choisir une valeur d'initialisation appelée *graine* qui n'a rien d'aléatoire et on calcule ensuite les itérations de `f` , $f(x)$, $f(f(x))$,..., $f \circ f \circ \ldots \circ f(x)$, etc, qu'on renvoie successivement à chaque nouvel appel de `aleatoire()` .

Le nombre de valeurs admises par le type `unsigned int` est fini et donc toute suite $(f^{\circ n}(x))_{n\in\mathbb{N}}$ est *périodique*. Soit P sa période. Au bout de P itérations, on retrouve les mêmes valeurs que celles de départ. Or, aucune suite de variables aléatoires i.i.d. non constantes p.s. ne peut être périodique. Un générateur de nombres aléatoires ne génère donc pas une quantité *infinie* de valeurs i.i.d. On peut contourner cette impossibilité de trois manières : soit on se limite à N utilisations du générateur avec N petit devant P, soit on utilise les cycles de longueurs P pour différentes v.a. dans différents morceaux

du programme, soit on croise les doigts pour que la périodicité n'ait pas d'impact sur le résultat du calcul !

La qualité d'un générateur de nombres aléatoires se mesure selon trois paramètres. Premièrement, plus la périodicité P est grande, plus les problèmes liés à la périodicité tardent à se manifester. C'est donc un paramètre important dans le choix d'un générateur. Deuxièmement, si on prend N petit devant P, il faut que les N valeurs obtenues donnent l'impression de suivre la loi de N v.a. indépendantes : il existe de nombreux tests statistiques qui mesurent cela et que le générateur doit passer. Cela restreint grandement la fonction d'itération `f` . Le troisième aspect est la rapidité du générateur pour produire une itération. Malheureusement, il faut souvent trouver un compromis entre ce troisième paramètre et les deux premiers...

Un exemple historique. L'exemple le plus simple et utilisé pendant longtemps malgré sa faible qualité est le suivant : pour des entiers codés sur 31 bits, i.e. de valeurs comprises entre 0 et $M = 2^{31} - 1$, on peut considérer la fonction $f(x) = 48271x \mod M$. Puisque 48271 est premier avec M, la période est M. C'est extrêmement rapide du point de vue algorithmique mais les corrélations résiduelles sont nombreuses et ce générateur ne passe pas de nombreux tests statistiques.

Passer d'une loi de v.a. à une autre. Un générateur de nombres pseudo-aléatoires fournit en général des valeurs de loi uniformes sur un ensemble d'entiers $n \in \{0, 1, 2, \ldots, M-1\}$. Si M est grand, la variable réelle `n/(double)M` fournit une bonne approximation de la loi uniforme sur l'intervalle $[0, 1[$. Pour obtenir les autres lois, il faut revenir consulter un cours de probabilité pour retrouver des formules algébriques reliant des v.a. de lois différentes (Box-Mueller pour des gaussiennes, méthode du rejet, pseudo-inversion de la fonction de répartition, etc). Cela fera l'objet d'un TP spécifique. De plus, le nouveau standard `C++` 2011 offre partiellement de telles possibilités.

4.7.2 Les générateurs de nombres aléatoires en C++

[HORS PROGRAMME]Avant la norme `C++` 2011

Cette sous-section présente des outils qui sont à présent périmés et à éviter au maximum. Néanmoins, puisque certains vieux codes contiennent encore des usages cette méthode, nous la présentons néanmoins. Il est interdit de l'utiliser en TP cette année.

La bibliothèque `<cstdlib>` fournit deux fonctions `srand()` et `rand()` et un nombre `RAND_MAX` prédéfini pour la génération de nombres pseudo-aléatoires.

La fonction `srand(n)` fixe à `n` la valeur de la graine du générateur de nombres pseudo-aléatoires et chaque appel `rand()` fournit une nouvelle valeur pseudo-aléatoire entière entre 0 et `RAND_MAX` obtenue par itération du système dynamique sous-jacent (cf. l'explication de la section précédente).

Faisons une remarque importante : le générateur pseudo-aléatoire de la fonction `rand()` est de **mauvaise qualité** et peut donner lieu à des résultats erronés si on l'utilise pour des applications probabilistes fines.

En pratique, si on l'on souhaite simuler quelques jets de dé pour la programmation d'un jeu, alors `rand()` suffit amplement : l'utilisateur ne se rendra pas compte de cette mauvaise qualité. Si l'on souhaite programmer une méthode de Monte-Carlo, l'usage de `rand()` est dangereux : il donne le plus souvent un résultat correct pour des programmes

très simples mais sa qualité n'est pas suffisamment fiable pour des calculs plus compliqués. Son usage est même à proscrire complètement pour des utilisations cryptographiques. Si l'on souhaite utiliser des générateurs de nombres pseudo-aléatoires de meilleure qualité en restant sur une norme C++ d'avant 2011, il faut aller chercher des bibliothèques spécifiques héritées du langage C : utilisation de `random()` sur des systèmes d'exploitation de type POSIX (cela exclut donc Windows), utilisation de la bibliothèque scientifique `GSL` , etc.

Nous nous contenterons donc ici de présenter brièvement l'usage des fonctions `rand()` et `srand()` car l'usage d'autres bibliothèques modifie la syntaxe et non la méthode générale.

Tout programme faisant usage de `rand()` doit contenir au début de la fonction `main()` une ligne d'initialisation de la graine :

```
#include <cstdlib>
#include <ctime>
#... // autres bibliothèque à inclure

int main() {
        srand( ??? ); // ??? doit être remplacé par une valeur à choisir
        ...           // suite du programme
        return 0;
}
```

La valeur `???` initialise le générateur de nombres pseudo-aléatoires. Choisir la même valeur `???` pour deux exécutions du programme produira la *même* suite de valeurs pour `rand()` : mathématiquement, cela revient à prendre le même $\omega \in \Omega$ et on observera les mêmes trajectoires de processus. Cela peut être utile pour déboguer un programme ou pour exhiber des trajectoires atypiques mais, en pratique, c'est rarement ce que l'on souhaite.

Pour obtenir une valeur `???` qui varie d'une exécution du programme à une autre, une astuce simple consiste à récupérer la date et l'heure de lancement du programme et à utiliser cette valeur comme graine. Cela se fait par la commande `time(NULL)` qui donne le nombre de secondes écoulées depuis une date de référence. L'initialisation se fait donc en substituant à la ligne 6 précédente l'instruction :

```
        srand(time(NULL));
```

Chaque appel à `rand()` fournit ensuite une nouvelle valeur pseudo-aléatoire entière, uniformément entre 0 et `RAND_MAX` (qui dépend de la machine et du compilateur).

[A CONNAÎTRE] La norme C++ 2011 et les outils de la STL

Depuis la version 2011 du C++, la STL s'est enrichie d'une nouvelle bibliothèque `<random>` que l'on invoque par la commande d'en-tête :

```
#include <random>
```

La compilation selon le standard 2011 se fait donc en ajoutant l'option `-std=c++11` dans la commande de compilation (les points de suspensions résument les options habituelles décrites en section 2.4) :

```
g++ -std=c++11 nomdufichier.cpp ...
```

Deux documentations complètes sont disponibles dans les liens suivants :

http://www.cplusplus.com/reference/random/
et
https://en.cppreference.com/w/cpp/header/random

La bibliothèque fournit deux types d'objets :

1. des générateurs pseudo-aléatoires (des alternatives à `rand()` de meilleure qualité le plus souvent)
2. des méthodes de génération des lois usuelles à partir de lois uniformes (cf. la ligne 10 de l'exemple précédent)

La génération d'une variable d'une loi donnée doit donc toujours associer un objet de chaque nature.

Dans ce cours, nous prendrons systématiquement le générateur appelé « Mersenne twister » et de type `C++` donné par `std::mt19937_64` . Il faut déclarer le générateur, lui donner un nom (mettons `G`) et l'initialiser par une valeur entière `n` qui fixera sa graine (l'équivalent de `srand()`) :

```
std::mt19937_64 G(n);
```

Il est intéressant de noter que l'initialisation de la graine se fait au moment de la déclaration. Si on souhaite changer la graine ultérieurement par une nouvelle valeur entière `val` , il suffit d'écrire `G.seed(val);` . Là encore, on pourra utiliser :

```
std::mt19937_64 G(time(NULL));
```

à la place de `n` pour obtenir des réalisations qui changent à chaque exécution du programme. La fonction `time()` est définie dans la bibliothèque `<ctime>` qu'il est nécessaire d'ajouter dans les en-têtes.

Vous verrez parfois, même dans notre cours et nos corrigé, l'usage du générateur `std::mt19937` : il faut préférer ce dernier sur les machines anciennes (en 32 bits) et préférer `std::mt19937_64` sur les machines récentes (64 bits) pour lesquelles il est optimisé.

Les générateurs de nombres aléatoires autres que `std::mt19937_64` sont également présentés sur la page web mentionnée ci-dessus. Cette page web indiquée ci-dessus présente également les lois usuelles déjà présentes dans la STL que nous récapitulons partiellement dans le tableau 4.2 avec leurs syntaxes.

La production d'une nouvelle valeur aléatoire de loi `Loi` avec le générateur `G` se fait simplement par appel de :

```
x=Loi(G);
```

où `x` est une variable du type associé à la loi `Loi` .

Remplissage d'un vecteur par des valeurs aléatoires. Supposons que nous voulions remplir un vecteur d'entiers par $N = 100$ valeurs aléatoires i.i.d. de loi de Poisson de paramètre 1 en exploitant au maximum la STL et afficher le vecteur. Cela peut être fait par :

```
#include <vector>
#include <iostream>
#include <random>
#include <algorithm> //pour std::generate et std::copy
#include <iterator> //pour std::ostream_iterator
#include <functional> //pour std::bind et std::ref
using namespace std;
int main() {
        const int N=100;
        mt19937_64 G(time(nullptr));
        poisson_distribution<int> P(1.);
        vector<int> values(N);
        generate(values.begin(),values.end(),bind(P,ref(G)));
        ostream_iterator<int> out(cout," / ");
        copy(values.begin(),values.end(),out);
        return 0;
}
```

Revenons en détail sur la ligne 13. L'algorithme `std::generate(debut,end,f)` remplit chaque case d'un conteneur ordonné entre les itérateurs `debut` et `end` par des appels à `f()`. Ici, on souhaite que le contenu soit `P(G)`. La fonctionnelle `std::bind()` permet de composer les fonctions `P(.)` et `G()` pour construire une fonction dont l'appel donne `P(G)`. Le passage de `G` par référence via `std::ref(G)` est nécessaire pour que `G` ne soit pas copié (et que son état interne soit bien incrémenté).

Exemple. Supposons par exemple que nous souhaitions calculer $S_1 = \sum_{i=1}^{n} \cos(U_i)$ et $S_2 = \sum_{i=1}^{n} U_i^2$ pour des variables $(U_i)_{i\geq 1}$ indépendantes et de loi exponentielle et $n = 10000$. Il faut d'abord se référer à un cours de probabilité pour savoir que, si $(Z_i)_{i\geq 1}$ est une suite de v.a. i.i.d. uniformes sur $]0, 1[$ alors chaque $U_i = -\log(Z_i)$ suit une loi exponentielle. Nous avons donc le code suivant (si nous ne souhaitons pas stocker l'ensemble des 10000 nombres, ce qui est souvent le cas en pratique) :

```
#include <cmath> // pour pouvoir utiliser cos
#include <iostream> // pour pouvoir afficher le résultat
#include <random>
#include <ctime>
int main(void) {
        std::mt19937_64 G(time(NULL));
        std::exponential_distribution<double> U(1.);
        unsigned int N=10000;          //nombre d'itérations
        double S1=0., S2=0.; // initialisation des sommes
        double u;
        for(unsigned int i=0; i<N ; i++) {
```

Loi	Syntaxe
Unif. sur $\{a,\dots,b\} \subset \mathbb{Z}$	`uniform_int_distribution<int> Loi(a,b);`
Uniforme sur $[a,b[\subset \mathbb{R}$	`uniform_real_distribution<double> Loi(a,b);`
Bernoulli(p) (`bool`)	`bernoulli_distribution Loi(p);`
Binomiale(n,p)	`binomial_distribution<int> Loi(n,p);`
Géométrique(p) sur $\mathbb{N}$	`geometric_distribution<int> Loi(p);`
Poisson(a)	`poisson_distribution<int> Loi(a);`
Exponentielle(a)	`exponential_distribution<double> Loi(a);`
Gaussienne $\mathcal{N}(m,\sigma)$	`normal_distribution<double> Loi(m,sigma);`

TABLE 4.2 – Lois usuelles présentes dans `random` et leur syntaxe de déclaration (sans le préfixe `std::`) pour une variable `Loi` . Si l'on souhaite changer la précision des valeurs générées, il suffit de changer `int` et `double` par d'autres types comme `long int` ou `float` .

```
12                  u= U(G);
                    S1 += cos(u);
14                  S2 += u*u;
            }
16          std::cout << "S1 et S2: " << S1 << " , " << S2 << std::endl;
            return 0;
18  }
```

Dans ce code, la ligne 12 fournit, à chaque parcours de la boucle, une nouvelle valeur réelle indépendante des précédentes de loi exponentielle de paramètre 1. Cette valeur, stockée dans `u` , est ensuite utilisée plusieurs fois dans différentes formules utilisant un même U_i.

Une autre loi utile. Il existe un dernier type de loi très utile. Soit la loi ν sur $\{0,1,\dots,N\}$ définie par

$$\nu(\{k\}) = \frac{a_k}{\sum_{j=0}^{N} a_j}$$

où les nombres a_j sont positifs ou nuls et de somme non nulle. Supposons qu'en `C++`, les nombres a_j soient stockés dans une structure ordonnée `A` de type tableau, vecteur ou liste (`array` , `vector` ou `list`) de la STL. Alors il est possible de créer une loi `nu` à partir de `A` avec la syntaxe :

```
std::discrete_distribution<int> nu(A.begin(),A.end());
```

Pièges à éviter

Nous résumons ici quelques pièges classiques à éviter à tout prix.

Piège 1. Il ne faut pas oublier d'initialiser le générateur pseudo-aléatoire avec l'heure de lancement du programme. Si on oublie cela, les séquences de valeurs sont identiques d'une exécution du programme à une autre et on observe toujours la même chose.

Piège 2. Dès la génération d'une valeur aléatoire par `rand()` ou par `Loi(G)` selon le standard, il faut absolument stocker cette valeur dans une variable auxiliaire pour pouvoir utiliser cette même valeur à différents endroits du programme car sinon elle est perdue ; en effet, un nouvel appel `rand()` fournit une *nouvelle* valeur indépendante de la précédente et il n'est pas possible de revenir en arrière...

Piège 3. Il ne faut jamais se fier aveuglément à des simulations basées sur des générateurs pseudo-aléatoires. Il suffit d'excéder la période du système dynamique sous-jacent pour casser toutes les propriétés d'indépendance des valeurs générées. Si des résultats semblent bizarres lorsque la taille des simulations augmente et si on a déjà vérifié maintes fois le code, alors il peut s'agir d'un problème lié au générateur pseudo-aléatoire ; il faut alors essayer de changer de générateur pour voir si le problème persiste.

Exercices

Exercice 4.1. (chaînes de caractères) Soit le programme suivant :

```
_____
int ____() {
        std::string Debut("Les sanglots longs des violons de l'automne");
        std::string Fin("blessent mon coeur d'une langueur monotone.");
        std::string Phrase = Debut + " " + Fin;
        _____cout << Phrase  << _____endl;
        int pos= Phrase.find("automne");
        _____cout << pos << _____endl;
        Phrase[pos] = 'A';
        std::cout << Phrase << _____endl;
        int counter=0;
        for (int i = 0; i < Phrase.size(); i++)
        {
                ____;
        }
        _____cout << "Nombre de 'o' : " << counter << _____endl;
}
```

1. Ajouter les bibliothèques nécessaires à la compilation du fichier.
2. Comprendre le programme et le compléter pour que la dernière instruction affichent le nombre d'occurrences de la lettre 'o' dans la phrase.
3. Qui est l'auteur de cette phrase et dans quel contexte historique a-t-elle été réutilisée ?

Exercice 4.2. (vecteurs de la STL) Voici un programme qui génère un vecteur d'un nombre aléatoire K (borné par 1000) de réels aléatoires de $[0, 1]$:

```
#include <iostream>
#include <vector>
#include <random>
using ______ std;
int main() {
        mt19937_64 G(666);
        uniform_real_distribution<double> U(0,1);
        uniform_int_distribution<int> K(0,1000);
        vector<double> V(K(G));
        for(int i=0; i<V.size(); i++) V[i]=U(G);
        cout << "Le nombre d'éléments est : " << /* bloc 0 */ << endl;
        int N=0;
        /* bloc 1 */
        cout << "Le nombre d'éléments <= 0.5 est: " << N << endl;
        double S=0;
        /* bloc 2 */
```

```
        cout << "La somme des éléments est: " << S << endl;
        cout << "Le vecteur complet est: " << endl;
        /* bloc 3 */
        return 0;
}
```

1. Remplacer les blocs et les morceaux manquants pour que le programme donne le bon résultat.
2. Ajouter des lignes au programme pour remplacer tous les nombres plus grands que 3/4 par 2/3. *Faites attention...*
3. Refaire les deux questions précédentes en utilisant des fonctions de `<algorithm>` .

Exercice 4.3. (bibliothèque `<algorithm>`) Voici plusieurs codes qui contiennent plusieurs boucles `for` pour manipuler deux vecteurs de `double` nommés `v` et `w` . À chaque fois, remplacer la boucle `for` par un modèle de fonction de `<algorithm>` et la λ-fonction adéquate si besoin.

1. Remplissage :

```
for(unsigned int i=0; i<v.size() ; i++) {
        v[i]=1.2345;
}
```

à remplacer par :

```
std::____(v._____(),v.___(),______);
```

2. Transformation par une fonction :

```
for(unsigned int i=0; i<v.size() ; i++) {
        w[i]= 1./(1.+exp(-v[i]));
}
```

à remplacer par :

```
____ f = [](______ x) { return ______ ; }
std::transform(_________,________,w._______, ___ );
```

3. Comptage de différence (on cherchera la solution autour de la fonction `inner_prod` de `<numeric>`) :

```
unsigned c=0;
for(unsigned int i=0; i<v.size() ; i++) {
        if( w[i] != v[i] ) c++;
}
```

4. Copie conditionnelle :

```
int j=0;
for(unsigned int i=0; i<v.size() ; i++) {
        if( v[i] > 0) {
                w[j]=v[i];
                j++;
        }
}
```

Exercice 4.4. (problème de graine) Soit le code suivant :

```
#include <iostream>
#include <random>//compiler avec -std=c++11
#include <ctime>
int f() {
        std::mt19937_64 G(time(nullptr));
        std::uniform_int_distribution<int> U(0,4);
        return U(G);
}
int main() {
        for(int i=0; i<100 ; ++i) {
                std::cout << f() << " " ;
        }
        return 0;
}
```

Compiler et exécuter plusieurs fois le programme. Qu'observez-vous ? Pourquoi le résultat de `f()` n'est-il pas aléatoire ? Où aurait-il fallu positionner l'initialisation du générateur pseudo-aléatoire ?

Exercice 4.5. (problème de perte de valeur non-sauvegardées) Exécuter le programme suivant plusieurs fois et expliquer le variabilité des valeurs des deux sommes. Expliquer également la présence de `&` dans le type de l'argument de la première fonction (et essayer sans !).

```cpp
#include <iostream>
#include <random>//compiler avec -std=c++11
#include <ctime>
int signe_aleatoire(std::mt19937_64 & G) { return 2*(G()%2)-1;}
int main() {
        std::mt19937_64 G(time(nullptr));
        int somme1=0;
        int somme2=0; int sauvegarde;
        for (int i = 0; i < 100; i++)
        {
                somme1 += signe_aleatoire(G)*signe_aleatoire(G);
                sauvegarde=signe_aleatoire(G);
                somme2 += sauvegarde*sauvegarde;
        }
        std::cout << somme1 << " vs " << somme2 << std::endl;
        return 0;
}
```

Exercice 4.6. (pierre-papier-ciseaux) Soit le programme suivant :

```cpp
____
using namespace std;
string chifoumi(mt19937_64 & G) {
        _____
}

int main() {
        mt19937_64 G(____);
        cout << chifoumi(G) << endl;
        return 0;
}
```

Compléter-le pour qu'à chaque exécution, il écrive `Pierre` , `Papier` ou `Ciseau` aléatoirement avec probabilité 1/3 chacun dans le terminal.

Exercice 4.7. (estimation de π) Une estimation de $\pi/4$ par la méthode de Monte-Carlo consiste à tirer des v.a. indépendantes uniformément sur $[-1,1]^2$ un grand nombre de fois et à compter le nombre de fois où elles tombent à l'intérieur du disque unité. Estimer π par cette méthode en générant 1000000 de points indépendants.

Chapitre 5

Programmation orientée objet et classes

Attention : un chapitre sur les classes est nécessairement incomplet tant que la notion de pointeur n'a pas été encore vue. Cette notion n'apparaîtra qu'au chapitre 7. Nous avons donc fait le choix d'éviter cette notion au moins tout le début du chapitre et d'ajouter une section sur les classes dans le chapitre 7 pour compléter l'information. Néanmoins, vers la fin du chapitre, vous verrez apparaître progressivement quelques pointeurs : lors d'une première lecture, vous pourrez tout simplement ne pas y faire attention et vous focaliser sur la notion très proche de référence. Une fois le chapitre 7 digéré, nous vous encourageons alors à relire le présent chapitre pour maîtriser les classes plus en profondeur.

Les chapitres précédents présentent essentiellement le `C++` pour des programmeurs qui souhaitent *utiliser* des types déjà existants (par nature ou dans la STL). Par exemple, nous ne nous sommes pas souciés de savoir comment étaient programmées les `int`, `double`, `std::vector`, `std::string`, `std::cout`, etc, et nous avons fait confiance aux développeurs du `C++` et de la STL pour avoir des algorithmes de bonne qualité. Dans la pratique, il se peut qu'il faille améliorer un type existant, l'adapter à des besoins propres, l'optimiser pour des données précises ou même encore créer des nouveaux types pour décrire de nouvelles structures mathématiques ou algorithmiques.

Par exemple, la STL ne gère pas les matrices. Pour cela, divers programmeurs ont bâti des nouveaux types au sein de bibliothèques comme `boost`, `armadillo` ou `Eigen` (nous avons déjà rencontrées cette dernière au début du chapitre 3 et dans les TP). La STL ne contient pas non plus d'outils mathématiques classiques : la conception de telles classes sera faite en TP.

Le langage `C++` permet de créer facilement de nouveaux objets avec de nouvelles fonctionnalités tout en conservant une syntaxe limpide. C'est l'avantage de la programmation orientée objet (par rapport à un langage comme le `C`). Néanmoins, la construction d'une classe contient de nombreux ingrédients — tous indispensables — car il faut revenir à des notions fondamentales.

La première section de ce chapitre décortique comment on utilise une classe pour établir la liste des choses à faire pour créer une nouvelle classe. Les sections suivantes décrivent les ingrédients d'une classe l'un après l'autre. La dernière section présente comment faire interagir différentes classes les unes avec les autres.

5.1 Généralités sur les classes

5.1.1 Étude d'un exemple préliminaire

Supposons que nous voulions effectuer des opérations matricielles. Nous souhaitons faire fonctionner le code suivant :

```
unsigned int N=4;
std::vector<double> V(N,2.); // construit le vecteur (2,2,2,2)
Matrice A(N,1.); // construit une matrice de taille N par N
        //tous ces coefficients sont 1
Matrice B(V); // construit une matrice carrée de diagonale donnée par V
Matrice C(N); // construit une matrice carrée de taille N par N
        // ses coefficients sont initialisés à 0
C= A+B;
C(3,2) = 7.;
double x = C(1,2);
std::cout << "La matrice C est: " << std::endl << C << std::endl;
std::cout << "Sa taille est: " << C.taille() << std::endl;
std::cout << "Son carre est: " << std::endl;
std::cout << C*C << std::endl;
double t=C.trace();
std::cout << "Sa trace est: "<<  t << std::endl;
```

La syntaxe est assez minimaliste et relativement simple à comprendre. Nous souhaitons que l'affichage donne :

```
La matrice C est:
3. 1. 1. 1.
1. 3. 1. 1.
1. 7. 3. 1.
1. 1. 1. 3.

Sa taille est: 4
Son carre est:
12. 14. 8. 8.
8. 18. 8. 8.
14. 44. 18. 14.
8. 14. 8. 12.

Sa trace est: 12.
```

Malheureusement, un tel type `Matrice` n'existe pas pour l'instant ! Le code précédent ne pourra être compilé que lorsque nous aurons créé une classe `Matrice` avec les bonnes fonctionnalités.

Commentons le code précédent et faisons la liste des ingrédients minimaux nécessaires à sa compilation :

— Les lignes 3 à 6 déclarent des matrices et les initialisent (ou non) de trois manières différentes : la première avec une taille et une seule valeur pour tous les coefficients,

la deuxième avec la donnée de sa diagonale sous la forme d'un vecteur et la troisième avec seulement la taille (et une valeur par défaut cachée). Il faudra donc deux *constructeurs* et une valeur par défaut.
- La ligne 8 nécessite de définir l'opérateur `+` d'addition de deux matrices ainsi que l'opérateur `=` qui permet d'affecter un résultat.
- La ligne 9 permet d'accéder directement à un coefficient donné en écriture et doit pouvoir changer sa valeur.
- La ligne 10 permet d'accéder directement à un coefficient donné en lecture et doit pouvoir lire sa valeur.
- La ligne 11 permet d'afficher directement une matrice : il faut donc étendre l'opérateur `<<` au type `Matrice`.
- les lignes 12 et 15 doivent permettre de calculer la taille et la trace avec une syntaxe du type `Variable.Methode(arguments)`.
- la ligne 14 nécessite de donner un sens au produit `*` de deux éléments de type `Matrice`.
- à la fin du code, il faudra effacer proprement toutes les variables pour ne pas perdre de place. Il faudra donc un *destructeur*.

Rien dans le code précédent ni dans le résultat de l'exécution ne fait apparaître la structure de la matrice en mémoire, ni la manière dont elle a été codée. La syntaxe ressemble ainsi beaucoup au code de la page 12 bien que nous allons choisir une structure très différente de celle choisie par les programmeurs qui ont conçu la bibliothèque `eigen`.

Afin de compléter l'exemple, nous donnons dans la figure 5.1 le prototype de la classe `Matrice` que nous allons expliciter dans les sections suivantes.

Commentons cette définition de classe en lien avec le code précédent. Cette définition de classe ne contient que des prototypes, sauf pour la méthode `taille()`. Le code des méthodes et des opérateurs sera écrit dans un fichier `.cpp` alors que la définition précédente sera dans un en-tête `.hpp`. Les constructeurs des lignes 6 et 7 donneront un sens aux lignes 3, 5 et 6 du code précédent. Les méthodes des lignes 10 à 13 permettent de donner un sens à toute fonction attachée à un objet donnée (la taille, la trace, l'affectation, l'accès à un coefficient). Les lignes 15 à 17 définissent des opérateurs qui manipulent plusieurs objets issus d'une ou plusieurs classes et *surchargent* les définitions d'opérateurs qui existent déjà pour d'autres types.

Il est intéressant de remarquer dès maintenant les usages de `const` et de références. Les méthodes des lignes 11 et 12 contiennent un `const` car aucune valeur de matrice ne varie : c'est un simple calcul à partir de valeurs existantes. En revanche, les lignes 10 et 13 ne contient pas de `const` après le prototype car elles servent à modifier un coefficient existant. Enfin, chaque fois qu'une matrice est passée en argument d'une fonction pour un calcul, on préfère éviter des copies inutiles qui prennent du temps et encombrent la mémoire et on utilise des références comme dans les chapitres précédents.

Les mots-clefs `public` et `private` permettent ici de séparer ce qui est utilisable dans un code qui utilise des objets de type `Matrice` (la partie publique) et ce qui est interne à la classe (la partie privée), manipulé par les méthodes de la classe mais qu'un utilisateur de la classe n'a ni à voir ni à modifier.

En guise d'exemple, la figure 5.2 montre à quoi ressemble le code d'un constructeur, du destructeur et de quelques méthodes. On remarquera la syntaxe `Matrice::methode` qui indique que l'on décrit la fonction `methode` qui appartient à la classe `Matrice` : en effet, plusieurs classes peuvent posséder des méthodes de même nom. On remarquera également que l'allocation de mémoire a lieu dans les constructeurs et la désallocation dans le destructeur. Enfin, le dernier opérateur n'est pas une méthode de la classe mais

```
class Matrice {
private:
        unsigned int n;
        std::vector<double> coeffs;
public:
        Matrice(unsigned N=1, double x=0.); // constructeur 1
        Matrice(std::vector<double> v); // constructeur 2
        //Matrice(const Matrice &);//construc. par copie inutile ici (*)
        //~Matrice(); // destructeur inutile ici (*)
// les méthodes:
        double & operator() (unsigned int,unsigned int); //mutateur
        double operator() (unsigned int,unsigned int) const ;//accesseur
        double taille() const { return n;} //accesseur
        double trace() const;
        //Matrice & operator= (const Matrice & U);//inutile ici (*)
// les opérateurs externes:
        friend Matrice operator+(const Matrice &, const Matrice &);
        friend Matrice operator*(const Matrice &, const Matrice &);
        friend std::ostream & operator<<(std::ostream &,const Matrice &);
};
```

FIGURE 5.1 – Définition de la classe `Matrice` avec les champs privés et l'ensemble des méthodes disponibles. Les méthodes marquées d'une étoile sont inutiles ici car la mémoire est directement gérée par la classe `std::vector` : elles deviendraient nécessaire avec l'usage de pointeurs tels que définis dans le chapitre 7.

```
Matrice::Matrice(unsigned N, double x): n(N), coeffs(N*N,x) {}

double & Matrice::operator() (unsigned int i,unsigned int j) {
        return coeffs[n*(i-1)+(j-1)];
}        // la numérotation des éléments matriciels débute à 1 !

double Matrice::operator() const (unsigned int i,unsigned int j) {
        return coeffs[n*(i-1)+(j-1)];
}

double Matrice::trace() const {
        double t=0.;
        for(unsigned int i=0; i<n ; i++)
                t+= coeffs[i*n+i];
        return t;
}

std::ostream & operator<<(std::ostream & o,const Matrice & A) {
        for(unsigned int i=0; i<A.n ; i++) {
                for( unsigned int j=0; j<A.n ; j++ ) {
                        o << A.coeffs[A.n*i+j] << " ";
                }
                o << std::endl;
        }
        return o;
}
```

FIGURE 5.2 – Codes d'un constructeur et de quelques méthodes de la classe `Matrice`. L'écriture des autres fonctions et méthodes est laissée en exercice.

une fonction externe qui utilise des éléments de la classe : *a priori* elle n'a pas accès aux champs étiquetés `private`, c'est le mot-clef `friend` qui lui octroie cet accès.

5.1.2 Vocabulaire et définition d'une classe : champs et méthodes

Une *classe* est la définition d'un nouveau type en `C++` : c'est une généralisation des structures `struct` héritées du langage `C` et présentées dans la section 3.1.4. Les structures présentées précédemment ne contiennent que des champs qui sont des emplacements dans la mémoire ; une *classe* contient de tels champs et ajoute à ceux-ci toutes les opérations naturelles que l'on peut effectuer dessus, les *méthodes*. C'est le principe de la *programmation orientée objet* : un *objet* (ici une classe) est la donnée jointe de variables et des fonctions qui agissent dessus.

Si nous osions un parallèle mathématique, nous dirions que c'est la même chose qu'une structure `struct` correspond à $\mathbb{Z}$ vu comme un ensemble d'éléments alors qu'une classe `class` avec ses méthodes correspond à $\mathbb{Z}$ vu comme un anneau, i.e. comme un ensemble muni des opérations d'addition, de multiplication et d'élements neutres. La deuxième

description de $\mathbb{Z}$ est beaucoup plus utile que la première !

Une *classe* est une liste de champs et de méthodes déclarée par :

```
#ifndef REF_DE_LA_CLASSE
#define REF_DE_LA_CLASSE
class NOM_DE_LA_CLASSE {
private:// ou protected selon les besoins
        type1 champ1;
        type2 champ2;
        ...
        typeN champN;
        //prototype de la methode 1:
        t1 nom_de_la_methode_1 (arguments_1) ;
        // prototype de la methode 2: ...
        t2 nom_de_la_methode_2 (arguments_2) ;
        ...
public:
        typeA champA;
        ...
        tA nom_de_la_methode_A (arguments_A) ;
        ...        // autres champs et autres methodes
//Constructeurs:
        NOM_DE_LA_CLASSE(arguments1); //constructeur 1
        ...
        NOM_DE_LA_CLASSE(argumentsM); //constructeur M
//Destructeurs:
        ~NOM_DE_LA_CLASSE() ; //destructeur
}; //ATTENTION, le point-virgule final est absolument nécessaire !!!
#endif
```

Cette définition doit être écrite dans un en-tête `.hpp` qui doit être inclus dans tous les fichiers qui utilisent des objets de cette classe. **Attention**, la déclaration de la classe doit se terminer absolument par un point-virgule !

La partie `public`. La définition d'un objet de type `NOM_DE_LA_CLASSE` se fait comme toutes les déclarations de variables par :

```
NOM_DE_LA_CLASSE nom(argumentsK);
```

où la liste des arguments `argumentsK` doit correspondre à l'un des constructeurs de la classe. Comme pour les types usuels, à l'issue de cette déclaration, une zone de la mémoire est allouée à la variable `nom` : cette zone correspond à tous les champs publics et privés `champ1`, etc., `champA`, etc. Si le constructeur numéro `K` contient des `new`, de la mémoire supplémentaire est allouée. L'initialisation de toutes ces zones mémoires se fait selon les instructions écrites dans les constructeurs ; si rien n'est écrit à ce sujet, les valeurs d'initialisation sont imprévisibles.

Une fois cette variable `nom` déclarée, le programmeur a le droit d'accéder à tous les champs étiquetés `public` par la syntaxe :

```
nom.champ
```

comme pour les `struct`. Le programmeur a également le droit d'appeler toutes les méthodes étiquetées `public` par la syntaxe :

```
nom.nom_de_la_methode_K(arguments_K)
```

où les arguments respectent les types donnés dans le prototype de la méthode. Cela produit alors une valeur de type `tK`.

Seule la partie `public` intéresse les *utilisateurs* de la classe.

La partie `private`. Les champs et méthodes étiquetés `private` ne sont utilisables qu'*à l'intérieur* du code des méthodes publiques ou privées (ainsi que par les fonctions externes déclarées `friend`, cf. plus bas) et nulle par ailleurs. Cela permet d'éviter qu'un utilisateur de la classe qui ne connaît pas son fonctionnement interne n'écrive des instructions dangereuses. La partie `private` n'intéresse donc que les *concepteurs* de la classe et non les utilisateurs.

Description des méthodes de la classe. Seuls les prototypes des méthodes figurent dans la classe. Le code des méthodes est écrit dans un fichier d'extension `.cpp` qui doit inclure le `.hpp` de définition de la classe et qui doit être compilé séparément.

Dans ce fichier `.cpp`, il n'y a plus de distinctions entre `public` et `private` et l'ordre d'écriture des codes n'importe pas. Normalement, un utilisateur de la classe n'a jamais à ouvrir ce fichier ; d'ailleurs, il arrive fréquemment que seule la version compilée soit fournie à l'utilisateur.

La définition d'une méthode se fait comme la définition d'une fonction, à ceci près qu'il faut spécifier la classe à laquelle elle appartient avec l'*opérateur de résolution* `::` comme ceci :

```
t1 NOM_DE_CLASSE::nom_de_la_methode_1(arguments1) {
        ...
        //instructions de la methode;
        return ... ; // un objet de type t1
}
```

Lors de l'écriture du code d'une méthode, il faut toujours garder à l'esprit qu'elle contient un argument de plus que ceux écrits dans son prototype. En effet, elle est toujours utilisée associée à un objet par `var.nom_de_la_methode(args)` et il faut considérer l'objet d'appel comme un argument supplémentaire fantôme. On accède à son adresse par le pointeur `this` ou en faisant mention du champ ou de la méthode sans préfixe.

Par exemple, si nous reprenons le code de la figure 5.2, la méthode `trace()` accède au champ `coeffs` directement : par convention, à chaque appel de `trace()` sous la forme `var.trace()`, c'est le champ `coeffs` de l'objet `var` qui est utilisé. Pour lever le doute, nous aurions pu écrire de manière complètement équivalente à la ligne 17 :

```
        t+= this->coeffs[i*n+i];
```

Qualificatif `const` . Lorsqu'une méthode d'une classe ne modifie pas les champs de l'objet sur lequel elle est appelée, il faut l'étiqueter `const` . Ce mot-clef doit être placé à la fin du prototype à l'intérieur de la classe ainsi que lors de la définition de la méthode. Il n'est appuyé à aucun argument de la liste d'arguments et par convention s'applique à l'objet d'appel pointé par `this` .

Son usage est *obligatoire*. En effet, une méthode étiquetée `const` (ou une fonction extérieure à la classe dont un argument est un objet de la classe et est marqué `const`) ne peut faire intervenir bien évidemment que des méthodes étiquetées `const` . Oublier de qualifier une méthode `const` restreint donc son utilisation. Or, de nombreux opérateurs déjà présents dans des bibliothèques comportent des `const` , comme par exemple `+` et `<<` . Oublier tous les `const` dans une classe empêche donc d'utiliser ces opérateurs !

Qualificatif `static` devant un camp privé. Il est possible d'ajouter le mot-clef `static` devant un champ privé dans la définition d'une classe. Alors, ce champ sera *commun* à tous les objets de la classe. Cela peut être utile pour éviter de stocker une même information, parfois volumineuse, dans toutes les instances d'une classe ou bien pour étudier des caractéristiques globables d'une classe, par exemple connaître en temps réel le nombre d'objets existants de la classe. Il faut alors faire attention aux constructeurs et destructeurs comme expliqué ci-dessous.

L'initialisation d'une telle variable doit se faire *une seule fois* dans le code à travers la syntaxe

```
TYPE NOM_CLASSE::NOM_VARIABLE = VALEUR;
```

De plus, cette variable n'étant plus liée nécessairement à un objet de la classe, elle peut être utilisée librement avec la syntaxe `NOM_CLASSE::NOM_VARIABLE` , indépendamment de toute déclaration d'objets de la classe.

5.2 Présentation des différentes méthodes

5.2.1 Accesseurs et mutateurs

Plusieurs méthodes sont très simples et consistent à donner l'accès à certains champs privés, en lecture ou en écriture : ces méthodes sont appelées *accesseurs* (lecture seule) et *mutateurs* (écriture).

Il pourrait sembler *a priori* stupide de restreindre l'accès à des champs par le mot-clef `private` pour ensuite le rétablir avec des accesseurs et mutateurs. Néanmoins, cela ne l'est pas pour plusieurs raisons. Tout d'abord le mot-clef `private` ne distingue pas lecture et écriture. D'autre part, cela simplifie grandement la structure du code à écrire lorsqu'il y a de l'héritage de classe (cf. la section 5.5). Enfin, nous verrons dans le chapitre 6 qu'il est pratique d'avoir des méthodes qui portent les mêmes noms dans différentes classes, même si, en arrière-plan, la structure interne des champs privés des objets est très différente.

Sauf exception, la règle d'or à adopter est de mettre tous les champs dans la partie `private` et de rétablir les accès en lecture et écriture par des accesseurs et des mutateurs au cas par cas.

Un accesseur et un mutateur doivent avoir en général le même nom (puisqu'ils concernent le même champ !) et ne diffèrent que par leur prototype. Un *accesseur* donne l'accès en lecture donc doit être étiqueté `const` et produit une valeur qui sera stockée ou utilisée ailleurs. Un *mutateur* donne l'accès en écriture donc ne doit pas être étiqueté `const` et produit une référence vers l'objet à modifier. Dans les deux cas, leur code tient en une seule ligne et consiste à renvoyer le champ souhaité : pour que le compilateur puisse optimiser le programme, il est préférable d'écrire ce code directement dans la définition de la classe.

Un exemple fondamental. Prenons un exemple abstrait mais minimal :

```
class A {
        private:
           double x;
        public:
           A(double y): x(y) {} //cf. constructeur section suivante
           double   X() const {return x;}  //accesseur
           double & X() {return x;}  //mutateur
           void clone(const A & r) { x=r.X(); }
};
```

Considérons les lignes de code suivantes, trions celles qui sont acceptables ou non et essayons de comprendre si c'est le mutateur ou l'accesseur qui est utilisé :

```
double u,v;
A a(4), b(5);        //creation de deux objets a et b
        // champs a.x et b.x initialisés à 4 et 5
a.x= u;         // impossible car x est privé
v= b.x;         // impossible car x est privé
v= a.X();         // OK. accesseur, v est modifié mais pas a.x
a.X()= u;         // OK. mutateur, a.x est modifié.
b.X() = a.X();         // OK. mutateur pour b, accesseur pour a
b.clone(a); // accesseur pour a
std::cout << a.X(); // accesseur pour a
```

Dans les deux dernières lignes, il faut se référer aux prototypes de `reset` et `<<` qui contiennent un `const` sur l'argument et donc seule une méthode `const` peut être utilisé sur `a` : cela exclut donc le mutateur.

Dans l'exemple de la figure 5.2 sur les matrices, l'opérateur `()` qui renvoie un `double &` est un mutateur de la case (i, j) de la matrice, alors que le même opérateur, du même nom, mais avec un `const` et sans référence est un accesseur. De même la méthode `taille()` est un accesseur à la taille de la matrice.

Un exemple plus élaboré *(hors programme mais instructif)* qui illustre le rôle des copies et références. Le but de cet exemple est de montrer que, dans certaines circonstances, le prototype des accesseurs peut être amélioré pour maintenir une bonne efficacité de calcul. Considérons ainsi la classe suivante :

```
class B {
   private:
      std::vector<double> v;
   public:
      B(int n,double u): v(n,u) {} //constructeur, cf. plus bas
      double operator[](int i) const { return v[i];} //accesseur OK
      std::vector<double> to_v() const {return v;}//mauvais accesseur !
};
```

Pour la ligne 6, c'est un accesseur à une case du vecteur `v` .Il n'y a rien de nouveau par rapport à l'exemple de la figure 5.2 avec l'opérateur `operator()` .

En revanche, dans certaines situations, on pourrait souhaiter donner accès à tout le vecteur `v` , par exemple pour utiliser des algorithmes de `<algorithm>` sur le champ privé. C'est ce que la ligne 7 tente de faire mais c'est un échec : il conduit à un programme qui compile mais ne marche pas ! Illustrons-le par un exemple :

```
B b(10,1.);
std::for_each(b.to_v().begin(),b.to_v().end(),
                [](double x) { std::cout << x << "\n";});
```

Ce code compile et donne

```
1
1
1
1
0
5.14322e-321
2.94941e+179
2.16942e-313
0
0
0
...une centaine de lignes avec des 0
0
Segmentation fault
```

Que se passe-t-il en pratique ? Chaque appel à l'accesseur `b.to_v()` produit une nouvelle copie du vecteur `v` . En particulier, les deux itérateurs `b.to_v().begin()` et `b.to_v().end()` pointent sur les extrêmités de *deux* vecteurs différents : ainsi, une fois passée la fin du premier vecteur copié lors de `b.to_v().begin()` , le `std::for_each` peut faire absolument n'importe quoi.

Une fois compris ce mécanisme de copie de `v` à chaque appel, on peut tout à fait imaginer également qu'il est absolument catastrophique de devoir copier le vecteur `v` dès qu'on veut accéder à une information de `v` (un itérateur de début, sa taille, une case, etc.).

La solution est alors de changer le type de retour de l'accesseur avec une référence pour donner accès à l'emplacement de `v` (comme pour le mutateur). Le simple ajout d'une

`&` ne suffit pas et donne cette fois-ci une erreur de compilation : en effet, le `const` de `to_v` garantit l'absence de modification de `v` alors que la méthode produit une référence sur `v` et donc un moyen de le modifier. Il faut alors ajouter un `const` supplémentaire dans le type de retour ! Le prototype correct de l'accesseur qui doit remplacer la ligne 7 est maintenant :

```
const std::vector<double> & to_v() const { return v; }
```

Cela résout alors tous les problèmes sus-mentionnés !

5.2.2 Constructeurs et destructeurs

Un *constructeur* d'une classe — nommons-la `A` pour l'exemple — est une méthode utilisée lors de la déclaration d'un objet de cette classe. Son nom est systématiquement le nom de la classe.

Usage.

Une classe peut posséder autant de constructeurs que l'on souhaite. Néanmoins, deux d'entre eux sont toujours nécessaires et jouent des rôles particuliers : le constructeur par défaut et le constructeur par copie. Le premier donne un sens à :

```
A a;
```

sans précision de valeurs d'initialisation lors de l'allocation de mémoire. Le second donne un sens aux trois commandes suivantes :

```
A b(a); // où a est déjà déclaré comme de type A
A b=a; // où a est déjà déclaré comme de type A
f(a) ; // où f a pour prototype T f(A a)
```

Dans les deux premiers cas, le nouvel objet `b` est construit à partir de l'objet `a` en dupliquant toutes les valeurs de champs de `a`. Nous avons vu précédemment que l'appel d'une fonction sur un objet donné en argument commence par cloner ce dernier pour pouvoir ensuite manipuler le clone en interne : ce clonage est exécuté par le constructeur par copie. Bien que le constructeur par copie n'apparaisse pas explicitement dans la ligne 2, il est utilisé lors de l'appel de la fonction `f`.

Les autres constructeurs sont utilisés lors de déclarations du style de :

```
A a(u1,u2,...,uN);
```

où les `ui` sont des objets préexistants. Pour trouver le bon constructeur, le compilateur regarde les types des objets `ui` et cherchent parmi les constructeurs un prototype qui correspond à la bonne succession de type. S'il n'en trouve pas, il y a une erreur de compilation. Un corollaire de ce mode de fonctionnement est que deux constructeurs différents ne peuvent pas avoir le même prototype.

À la fin d'une fonction ou d'un bloc d'instructions, nous avons vu au chapitre 3 que toutes les variables déclarées dans cette fonction ou ce bloc sont effacées de la mémoire.

Lors de la création d'une classe, il faut donc spécifier comment un objet de cette classe doit être effacé de la mémoire : c'est le rôle du *destructeur*. Remarquons immédiatement qu'on ne fait jamais appel directement au destructeur d'une classe : c'est le programme lui-même qui l'utilise au moment d'effacer une variable. Même s'il est souvent trivial, il ne faut cependant pas l'oublier !

Définitions.

Dans la définition de la classe, les constructeurs apparaissent toujours sous la forme :

```
        NOM_DE_LA_CLASSE(arguments);
```

Dans le fichier `.cpp` qui contient le code des méthodes, sa définition est toujours de l'un des deux types suivants, soit avec des affectations du type :

```
NOM_DE_LA_CLASSE::NOM_DE_LA_CLASSE(arguments) {
        champ1 = ... //affectation
        champ2 = ... //affectation
        ...
        return;
}
```

ou bien avec des listes d'initialisation :

```
NOM_DE_LA_CLASSE::NOM_DE_LA_CLASSE(arguments): //ne pas oublier les ":"
        champ1(valeur1), champ2(valeur2),...
{
        ... // autres initialisations
        return;
}
```

La différence entre les deux syntaxes est que la première construit d'abord les champs avec les constructeurs par défaut des types correspondants *puis* on change leur valeur par des affectations dans le code, alors que la deuxième utilise directement d'autres constructeurs pour créer les champs. La deuxième solution permet souvent de gagner du temps et se retrouve parfois nécessaire en cas d'absence d'opérateurs d'affectations `=`, de mutateurs ou tout simplement de constructeur par défaut.

Le constructeur par défaut est défini soit par le prototype `NOM_DE_LA_CLASSE()`, soit en attribuant des valeurs par défauts à tous les arguments d'un autre constructeur.

Le constructeur par copie est *toujours* défini par la prototype

```
NOM_DE_LA_CLASSE(const NOM_DE_LA_CLASSE &);
```

sans exception possible. En l'absence de redéfinition du constructeur par copie, par défaut, le compilateur tente d'utiliser pour chaque champ le constructeur par copie du type du

champ ; s'il n'en trouve pas pour au moins un champ, alors une erreur de compilation se produit.

Le destructeur est *toujours* défini par le prototype

```
~NOM_DE_LA_CLASSE() ;
```

Là encore, par défaut, le compilateur utilise pour chaque champ le destructeur du type du champ.

Les notions de constructeurs par copie et destructeurs sont relativement transparentes tant que les champs ne sont pas des pointeurs : la situation se corse lorsque l'on souhaite gérer soi-même la mémoire comme cela est décrit dans le chapitre 7.

Gestion des champs globaux. Une autre situation où l'on doit utiliser un constructeur par copie non trivial ou un destructeur est l'interaction avec des champs étiquetés `static`. Imaginons, comme expliqué précédemment que l'on veuille connaître en temps réel le nombre d'objets existants de la classe. On peut alors considérer la situation suivante :

```
class A {
        protected:
                double u;
                static unsigned int nb;
        public:
                A(double u0): u(u0) { nb++; }
                A(const A & a): u(a.u) { nb++; }
                ~A() { nb--; }
                double operator() const { return u;}
                double & operator() { return u;}
};

unsigned int  A::nb=0;//initialisation de la variable statique
```

dans laquelle le champ `A::nb` vaut constamment le nombre d'objets de type `A` actuellement déclaré.

Affichage d'information. Il faut également réécrire les définitions des constructeurs par copie et destructeurs si l'on souhaite qu'ils affichent des messages dans le terminal à l'exécution du programme comme illustré dans l'un des exercices en fin de chapitre. C'est très utile dans la période de débuggage du programme pour vérifier que ce que l'on a écrit est correct.

5.2.3 Incrémentation et décrémentation

Parmi les méthodes, il est possible de donner un sens au opérateur `++`, `+=`, `--`, `-=`, `!` avec des déclarations à l'intérieur de la classe du style :

```
        A& operator++(); //idem pour -- et !
        A& operator+=(argument); //idem pour -=
```

et les codes correspondants dans le fichier `.cpp` :

```
A& A::operator++() {
...//instructions
} //idem pour -- et !

A& A::operator+=(argument) {
...//instructions
} //idem pour -=
```

Les opérateurs `++` et `--` s'utilisent alors nécessairement sous forme préfixée sur un objet avec `++a;` ou `--a;` .

Le cas de l'opérateur `[]` est un peu différent. Pour respecter une compatibilité avec l'usage de `[]` pour les tableaux et les vecteurs, le seul prototype possible pour cet opérateur défini comme méthode d'une classe `A` est de prendre un entier en argument :

```
Type A::operator[](int i) const {
...//instructions
}        //version accesseur

Type & A::operator[](int i) {
...//instructions
}        //version mutateur
```

5.2.4 Méthodes de conversion

Dans la présentation des types usuels `int` et `double` , nous avons vu que l'inclusion naturelle des entiers dans les réels pouvait être faite par la notation `(double)i` ou `double(i)` où `i` est une variable de type `int` .

De la même manière, si nous disposons de deux classes `B` et `A` avec une méthode de conversion naturelle des objets de `A` en objets de `B` , il est possible d'écrire :

```
A a;
B b,c;
b=(B)a;
c=B(c);
```

à condition d'ajouter comme méthode dans la partie publique de la classe `A` (attention !) un opérateur de prototype :

```
        operator B(); // SANS TYPE DE RETOUR !!!
```

défini ensuite par :

```
A::operator B() {
        ...//instructions
        return ...; // un objet de type B doit être produit
}
```

5.3 Surcharge d'opérateurs, de fonctions et amitié

Les sections précédentes de ce chapitre ont permis de créer de nouveaux objets avec des méthodes associées. Dans de nombreux cas, on souhaite étendre également des fonctions préexistantes. Par exemple, tout comme les entiers `int` et les flottants `double`, l'ensemble des matrices peut être muni d'une addition, d'une multiplication et d'une exponentielle ; il faut alors donner un sens à `A+B`, `A*B` et `exp(A)` pour des objets `A` et `B` appartenant à un nouveau type.

Pour cela, il suffit d'ajouter des fonctions avec les prototypes idoines après la définition de la nouvelle classe dans le fichier `.hpp` et d'ajouter leur code dans le fichier `.cpp`. Attention néanmoins, ces fonctions additionnelles ne sont plus des méthodes donc :

- il n'y a plus de syntaxe d'appel du style `objet.fonction()`,
- le pointeur `this` n'est plus défini dans ces fonctions
- ces fonctions sont externes à la classe donc, *a priori*, elles n'ont le droit d'utiliser que les champs et méthodes étiquetées `public`.

Cette dernière règle est salutaire car elle évite à un utilisateur qui connaîtrait mal l'organisation interne de la classe de nuire à son bon fonctionnement. Néanmoins, cela requiert souvent des contorsions syntaxiques ; il est alors possible de contourner la dernière règle grâce au mot-clef `friend`. L'ajout d'une ligne

```
        friend  PROTOTYPE DE LA FONCTION ;
```

dans la partie publique de la définition d'une classe octroie au programmeur d'utiliser les méthodes et champs privés de la classe à l'intérieur du code de la fonction.

Dans l'exemple des figures 5.1 et 5.2, c'est effectivement le cas des additions, multiplications et `<<` et on remarque en effet que les lignes 22 à 24 de la figure 5.2 utilise les champs privés de la matrice `A` passée donnée en argument.

Toute fonction peut être surchargée, ainsi que la quasi-totalité des opérateurs arithmétiques, les tests logiques, les opérateurs de comparaisons, les opérateurs d'entrée-sortie `<<` et `>>`, etc. Avoir une seule et même notation pour des fonctions définies sur différentes classes prendra toute son importance au chapitre suivant.

Il est également possible de déclarer des liens d'amitié entre classes. L'écriture

```
class A {
        ...//parties privées et protégées
public:
        ...//méthodes publiques
        friend class B;
};
```

permet à toutes les méthodes de la classe `B` d'accéder aux champs et méthodes privés de la classe `A`.

5.4 Méthodes et opérateurs par défaut : liste, déclarations et désactivation

5.4.1 Méthodes nécessaires pour un fonctionnement standard

Dans tout ce qui précède, nous avons supposé que le compilateur générait automatiquement certaines opérations : c'est le cas par exemple de toutes les méthodes étiquetées `(*)` en commentaire du code de la figure 5.1. Dans la majorité des cas, c'est effectivement le cas mais :

- il est possible que le résultat ne soit pas du tout celui que l'on souhaite : c'est par exemple le cas important traité dans le chapitre 7 ;
- il est possible que le compilateur ne puisse pas faire cette génération automatique ;
- il est possible que la solution générée par le compilateur ne soit pas optimale ou ne soit pas souhaitable du tout.

Il est à noter que, pour que cette génération automatique puisse avoir lieu, ces méthodes très particulières doivent avoir toujours *le même prototype* et les mêmes règles d'utilisation spécifiées par la norme du langage `C++`.

Nous allons commencer par faire la liste des méthodes par défaut nécessaires à un bon fonctionnement d'un programme qu'en général un compilateur peut générer automatiquement. Nous illustrerons chaque cas par une situation d'usage et par le code généré par le compilateur. Selon le standard, il s'agit de la *règle des trois* ou *règle des cinq*. Afin que cela apparaisse une fois clairement, voici un exemple de classe avec les prototypes et les nos de toutes ces méthodes par défaut expliquées ensuite :

```
class A {
        public:
                A(); // constructeur par défaut

                // Règle des trois (< standard C++11)
                A(const & A); // constructeur par copie
                A & operator=(const & A &);// affectation par copie
                ~A(); // destructeur

                // Extension à la règle des cinq (>= C++11)
                A(A &&); // constructeur par déplacement
                A & operator=(A &&);//affectation par déplacement

                //plein de choses
        private:
                //plein de choses
};
```

Le constructeur par défaut. Il est appelé à de très nombreux endroits, parfois sous forme cachée. Voici une liste de situations où il apparaît pour une classe `A` donnée :

```
A a;
A a {}; //une autre syntaxe peu présente dans ce cours
A* p = new A;//voir chapitre (7) sur la mémoire
std::vector<A> v(10);
v.resize(20);
```

Cela correspond dans les trois cas à la déclaration d'une ou plusieurs variables de type `A` sans spécification de la valeur de `A` : dans ces cas-là, le compilateur doit allouer de la mémoire pour un objet de type `A` mais il reste à savoir comment la remplir.

Le prototype d'un constructeur par défaut dans une classe `A` quelconque est le suivant :

```
class A {
        private:
        TYPE1 champ1;        //champs privés
        TYPE2 champ2;
        ...
        public:
                A(); // prototype du constructeur par défaut.
};
```

La règle par défaut est la suivante : si le constructeur n'est ni spécifié explicitement dans la classe, ni désactivé, le compilateur y substitue la ligne

```
A::A(): champ1(), champ2(), .... {};
```

autrement dit, chaque champ est créé avec le constructeur par défaut de son type. Bien évidemment, cela marche si et seulement si tous les types qui apparaissent dans la classe ont un constructeur par défaut.

Il est possible à partir des standards récents de C++ de prescrire des valeurs de champs privés par défaut avec la syntaxe dans la classe :

```
        private:
                TYPE1 champ1 = val_par_defaut_1;
                TYPE2 champ2 = val_par_defaut_2;
                ...
```

Constructeur par copie. Comme le constructeur par défaut, il est appelé à de très nombreux endroits, parfois sous forme cachée. Voici une liste de situations où il apparaît pour une classe `A` donnée :

```
A a(b); //où b est de type A
A a=b ; //si b est de type A et le compilateur est correctement paramétré
A* p = new A(b);//voir chapitre (7) sur la mémoire
f(a); //si f est de prototype f(A x): passage par valeur
std::vector<A> v(w) ;//si v et w sont deux vecteurs, copie des cases
```

En reprenant le même exemple que précédemment, le prototype du constructeur par copie est toujours donné dans la classe par :

```
A(const A &); //constructeur par copie.
```

La règle par défaut est la suivante : si le constructeur n'est ni spécifié explicitement dans la classe ni désactivé, alors le compilateur crée automatiquement la ligne :

```
A::A(const A & x): champ1(x.champ1), champ2(x.champ2), ... {};
```

i.e. il copie chaque champ de `A` en utilisant le constructeur par copie du type correspondant.

Opérateur d'affectation par copie. Son usage est facile et correspond à la situation

```
a=b ; // avec deux objets de type A
```

à l'exception du code `A a=b;` qui correspond à une construction initialisée gérée par le constructeur par copie (sauf option contraire donnée au compilateur).

Le prototype dans la classe est figé et doit correspondre à :

```
// dans la classe:
A & operator=(const A &);
```

Il décrit en fait une méthode et la ligne `a=b=c;` est en fait interprétée comme

```
a.operator=(b.operator=(c));
```

En l'absence de définition explicite dans la classe, le compilateur génère un opérateur d'affectation canonique de code quasi-équivalent (nous passons sur certains détails) à :

```
A & A::operator=(const A & x) {
        champ1 = x.champ1;
        champ2 = x.champ2;
        ...
        return *this;
}
```

i.e. chaque champ est affecté aux champs de l'objet courant par l'opérateur d'affectation associé à son propre type.

Destructeur. Chaque fois qu'une variable arrive en fin de vie, soit à la fin d'un bloc soit à l'effacement d'une mémoire allouée (voir chapitre 7), le compilateur fait appel au *destructeur* de la classe de prototype imposé par `A::~A()` . Sauf situation explicite où l'on alloue la mémoire soi-même ou bien où l'on ajoute des compteurs statiques, on laisse le compilateur générer lui-même le destructeur explicitement (il utilise alors le destructeur associé à chaque champ).

Opérateurs de déplacement par défaut à partir du standard C++11. Le C++11 ajoute la possibilité de « déplacer » une variable qui va disparaître juste après. Cela correspond à une situation équivalente à :

```
A a;
{// un bloc quelconque ou l'exécution d'une fonction
        A b;
        .../calcul de b
        a = b;
}
```

L'avant-dernière ligne copie la valeur de `b` dans `a` et il y a donc temporairement deux fois le même contenu de valeurs dans la mémoire et, immédiatement après, `b` est effacée car elle arrive en fin de vie. Il y a manifestement une copie inutile : il vaudrait mieux que la variable `a` récupère l'emplacement mémoire de `b` qui était de toute façon destiné à disparaître. Ce phénomène devient crucial lorsque les objets sont gros.

Le standard C++11 définit des *rvalue references* pour décrire des objets voués à disparaître (c'est dit de manière trop simpliste mais l'idée est là) avec le type `T &&` où `T` est un type quelconque. Le compilateur génère alors automatiquement les méthodes de prototypes :

```
A::A(A &&);
A & A::operator=(A &&);
```

qui utilisent les mêmes opérateurs sur chaque champ avec le type associé.

Opérateur de comparaison par défaut en C++20. Les conteneurs de la STL comme `std::set` (ou les algorithmes de tri comme `std::sort`) requièrent un opérateur de comparaison `<` sur les objets d'une classe. Plusieurs situations se présentent alors :

1. soit le type des objets correspond à un objet mathématique avec une relation d'ordre qu'on définit soi-même en surchargeant `operator<` comme une fonction amie ou une méthode.
2. soit le type des objets correspond à un objet mathématique avec une relation d'ordre héritée de l'ordre sur chaque champ et de l'ordre lexicographique sur chaque champ
3. soit le type des objets n'a pas de relation d'ordre clairement identifiée mais toute relation d'ordre ferait algorithmiquement l'affaire (pour `std::set` par exemple).

Dans le premier cas, tout est fait par le développeur de la classe. Dans les deuxième et troisième cas, on souhaiterait plutôt laisser le soin au compilateur d'écrire l'opérateur de comparaison correspondant. C'est effectivement possible depuis le standard C++20 où le compilateur fournit une définition par défaut lorsque la ligne suivante apparaît dans la classe `A` (comme une méthode) :

```
auto operator<=> (const A &) const = default;
```

Le symbole `<=>` permet de définir en une seule fois tous les opérateurs d'ordre `<` , `<=` , `>` , `>=` grâce à une astuce sur le type de retour qui permet de les ordres partiels et totaux et que nous laissons le compilateur déterminer seul par `auto` . Contrairement aux autres

méthodes où il n'y a rien à écrire, cet opérateur — non-essentiel en fait — requiert l'écriture de la ligne précédente pour exister.

5.4.2 Insuffisance, programmation, désactivation et conséquences

Insuffisance des solutions par défaut. Il se peut que les codes par défaut générés précédemment ne correspondent pas à ce que l'on souhaite, voire mènent à des problèmes majeurs ; c'est le cas par exemple du chapitre 7 et nous vous encourageons à revenir sur la présente section après la lecture de ce chapitre.

On peut toujours programmer soi-même les méthodes par défaut précédentes : il suffit pour cela de respecter le prototype scrupuleusement lors des déclarations et définitions des méthodes. Le compilateur ne génère des méthodes par défaut qu'en l'absence de méthodes définies explicitement.

Désactivation. Lorsqu'on refuse qu'existe une méthode par défaut, il est donc nécessaire de le spécifier explicitement par une ligne du type par le mot-clef `=delete` à partir de C++11 de la manière suivante :

```
class A {
        public:
                A()= delete;
                A(const & A) = delete;
                A & operator=(const A &) =delete;
                //etc
};
```

Dans ce cas, le compilateur ne génère pas les méthodes associées (et peut en désactiver d'autres selon certaines règles que nous n'aborderons pas ici) et vérifie ensuite qu'aucune ligne de code n'utilise de tels méthodes, comme le montre l'exemple minimal suivant :

```
class B {
        public:
                B()=delete;
};

class A {
        B b;
};

int main(){
        A a;
        return 0;
}
```

qui ne fonctionne pas.

En général, il y a toujours de **très bonnes raisons** pour que certaines méthodes par défaut n'existent pas pour une classe : cela veut en général dire que c'est à vous de réfléchir au pourquoi et de trouver des solutions alternatives.

Outre la situation particulière du chapitre 7, on peut souhaiter par exemple désactiver le constructeur par copie lorsqu'on sait qu'on utilise des objets très gros dont l'existence multiple peut menacer la mémoire de saturation ou menacer le temps de calcul d'un ralentissement significatif. Le compilateur s'assure ensuite qu'aucune copie est faite et signale ainsi tout problème potentiel.

Attention, désactiver une méthode par défaut a des conséquences pratiques significatives : cela prive de nombreux outils de la STL ou de nombreuses fonctions qui utiliseraient ces méthodes. Il ne faut donc le faire qu'avec de sérieuses motivations.

Attention, afin de ne prendre aucun risque, nous vous conseillons de respecter strictement la *règle des cinq* : si vous touchez à l'une des cinq méthodes essentielles, alors vous touchez aux cinq (quitte à utiliser `=default` pour éviter de recoder celles auxquelles vous ne souhaitez pas toucher).

5.5 Héritage public entre classes

Ce cours est restreint à la notion d'héritage *public*. De même, il y a de nombreux détails subtils (méthodes par défaut, copies, etc) que nous omettrons non seulement par souci de simplicité mais aussi car l'utilisation de l'héritage doit être réduit au strict minimum. Souvent, ajouter un champ privé qui correspondrait à ce que l'on souhaiterait pour la classe mère est suffisant.

5.5.1 Définition et syntaxe

Nous avons vu dans les sections précédentes comment définir de nouveaux objets à partir de types préexistants en les assemblant comme champs privés d'une nouvelle classe. Il se peut aussi qu'on souhaite définir de nouveaux objets en enrichissant une classe existante de fonctionnalités étendues ou en la restreignant à une classe de sous-objets particuliers. *A priori*, le code des méthodes de la nouvelle classe ressemblerait en grande partir à celui de la classe existante : le recopier serait une perte du temps ; pour éviter cela, le `C++` introduit la notion d'*héritage*.

Un exemple de référence. Pour illustrer l'intérêt de cette notion, prenons un exemple géométrique. Supposons que nous voulions manipuler des polygones géométriques dans l'espace euclidien $\mathbb{R}^2$. Nous pourrions créer des classes pour chacune des formes usuelles les rectangles, les carrés, triangles, etc. Chacune de ces classes aurait des champs privés similaires (le nombre de points, les tableaux des abscisses et ordonnées de chaque point) et des fonctionnalités similaires (le périmètre, le centre de gravité) mais, par ailleurs, certaines différeraient des autres par l'ajout de fonctionnalités supplémentaires (l'orthocentre pour le triangle) ou par des simplifications supplémentaires (le carré par rapport au rectangle). Pour toutes les fonctionnalités similaires, il faudrait recopier presque le même code pour chacune des classes : ce serait fastidieux et source d'erreur.

L'héritage fournit une solution à ce problème : on construit une classe mère `Polygone` qui contient les caractéristiques communes et les méthodes communes à tous les polygones et on crée des classes filles avec des fonctionnalités supplémentaires ou des simplifications des fonctionnalités déjà existantes. Une ébauche de telles classes est fourni dans la figure 5.3.

Commentons brièvement le code présenté dans la figure 5.3. On note l'arrivée de deux nouveaux mots-clefs `protected` (à la place de `private`) et `virtual`. On remarque également :

Définition des classes :

```
class Polygone {
protected://au lieu de private
        unsigned int n; //nombre de points
        double * abs; //tableau dynamique des abscisses
        double * ord; //tableau dynamique des ordonnées
public:
        Polygone(unsigned n=1); //constructeur d'un polygone de n côtés
                // les sommets ne sont pas initialisés
        Polygone(const Polygone &); // copie;
        ... //autres constructeurs, destructeurs, autres méthodes
        virtual double perimetre() const;
};

class Triangle: public Polygone {
public:
        ...// des constructeurs
        double orthocentre_abscisse() const;
        double orthocentre_ordonnee() const;
};

class Carre: public Polygone {
public:
        Carre(double x,double y,double u,double v);
        double perimetre() const;
};
```

Définition des méthodes périmètres :

```
double Polygone::perimetre() const {
        double x=abs[0]-abs[n-1], y=ord[0]-ord[n-1];
        double p=sqrt(x*x+y*y);
        for(unsigned int i=0; i<n-1; i++) {
                x=abs[i]-abs[i+1], y=ord[i]-ord[i+1];
                p += sqrt(x*x+y*y);
        }
        return p;
}

double Carre::perimetre() const {
        double x=abs[0]-abs[n-1], y=ord[0]-ord[n-1];
        double p=sqrt(x*x+y*y);
        return 4*p;
}
```

FIGURE 5.3 – Définition de la classe `Polygone` et de deux classes filles `Carre` et `Triangle` .

— l'absence dans `Triangle` et `Carre` des attributs déjà présents dans `Polygone` (on évite les répétitions),
— l'ajout de deux nouvelles méthodes de calcul de l'orthocentre dans la seule classe `Triangle` (c'est normal : cette notion n'est pas définie pour les autres polygones),
— la redéfinition dans `Carre` de la méthode `perimetre()` avec une accélération du calcul puisqu'une seule racine carrée est calculée au lieu de quatre.

Les autres changements seront commentés plus tard à titre d'exemples.

La syntaxe générale de l'héritage public. Une classe `B` hérite d'une classe `A` lorsque sa définition commence par :

```
class B: public A {
        ...// champs et méthodes
};
```

Cela signifie qu'un objet de la classe `B` possède en mémoire *tous les champs et méthodes* de la classe `A`, en plus des siens propres donnés dans la définition de `B`. Ainsi, un objet de type `B` est un objet de type `A` dont on *étend* les capacités par l'ajout de champs et méthodes supplémentaires.

La classe `A` est dite la classe *mère* et `B` la classe *fille*. On dit parfois aussi que le classe `B` *hérite* de `A`, ou encore qu'elle *dérive* de `A`.

Toutes les caractéristiques `public` de `A` restent `public` pour `B`, toutes les caractéristiques `private` de `A` deviennent *inaccessibles* pour `B` bien que présentes dans la mémoire. On introduit alors un niveau intermédiaire supplémentaire par le mot-clef `protected` : de l'extérieur d'une classe, les champs et méthodes `protected` sont inaccessibles (comme pour les champs et méthodes `private`) mais ils restent `protected` et donc accessibles pour les méthodes des classes filles.

Dans l'exemple précédent des polygones, la méthode `Carre::perimetre()` a donc le droit d'accéder aux champs `abs`, `ord` et `n` de la classe mère `Polygone`, grâce au mot-clef `protected`.

De manière générale, toutes les définitions de méthodes et surcharges se font comme avant ; le seul danger est d'oublier que la classe fille contient — en plus des siens — tous les champs de la classe mère qu'il ne faut pas oublier de mettre à jour !

Constructeurs des classes filles et initialisation de la classe mère. Tout constructeur d'une classe fille doit utiliser un constructeur de la classe mère au début de sa liste d'initialisation. Autrement dit, tout constructeur de la classe `B` doit avoir un code du style

```
B::B(arguments):
        A(valeurs), ... //autres initialisations si necessaire
{
        ...//instructions
}
```

Le constructeur utilisé pour `A` peut bien évidemment être le constructeur par défaut si nécessaire.

Continuons l'exemple des polygones et écrivons le constructeur de `Carre` qui place un point en (x, y) et dont le premier côté issu de ce point en sens antihoraire a pour vecteur (u, v). Dans le sens antihoraire, les points du carré sont donc (x, y), $(x + u, x + v)$, $(x + u - v, x + v + u)$, $(x - v, x + u)$. Les constructeurs sont ainsi donnés par

```
Polygone::Polygone(unsigned N) {
        n=N;
        abs=new double[n];
        ord=new double[n];
        // attention, les coordonnees ne sont pas initialisées !
}

Carre::Carre(double x, double y, double u, double v): Polygone(4) {
        abs[0]=x;         ord[0]=y;
        abs[1]=x+u;         ord[1]=y+v;
        abs[2]=x+u-v;         ord[2]=x+v+u;
        abs[3]=x-v;         ord[3]=x+u;
}
```

On voit ainsi que le constructeur de la classe mère fait l'allocation mémoire de la bonne taille et le code du constructeur de la classe fille n'a plus qu'à fixer les valeurs.

Conversion naturelle et polymorphisme. Un objet d'une classe fille contient toute l'information de la classe mère : il peut donc être converti naturellement en un élément de la classe mère.

Par exemple, la syntaxe suivante est tout à fait valable :

```
Triangle T;
Polygone P(T); // constructeur par copie !
```

L'objet `P` est donc un polygone à trois côtés dont les sommets sont ceux de `T`. On a, en revanche, oublié sa nature de triangle, et le calcul de l'orthocentre sur `P` n'est plus possible par les méthodes de `Triangle`.

Plus généralement, **un objet d'une classe fille peut être utilisé partout où un objet de la classe mère est attendu**. C'est précisément cette propriété qui fait toute la puissance de l'héritage comme nous le verrons en TP et dans les exemples du cours.

Enfin, **les pointeurs vers des objets d'une classe fille sont compatibles avec les pointeurs d'une classe mère**. Cela signifie que le code suivant est tout à fait correct :

```
Carre C(0.,0.,1.,0.);
Polygone A;
Carre * p = &C;
Carre & q = C;
Polygone * r = &C;
Polygone & s = C;
Polygone * t = &A;
Polygone & u = A;
```

Néanmoins, les quatre dernières lignes conduisent à un polymorphisme dangereux : derrière les références/pointeurs `r,s,t,u` peuvent se cacher soit des objets de type `Polygone`, soit des objets de type `Carre` ou `Triangle`, bien que rien ne l'indique dans les types de `r,s,t,u`. En particulier, ce contenu réel derrière les pointeurs ne sera fixé qu'à l'exécution et non à la compilation.

Nous devons donc introduire une nouvelle distinction fine :

— *type statique* : le type statique d'une variable est le type déclaré dans le code source et est déterminé dès la compilation

— *type dynamique* : le type dynamique d'une variable de type pointeur ou référence est le type de l'objet en mémoire pointé et n'est déterminé qu'à l'exécution.

Les deux types peuvent ainsi ne pas être les mêmes. Dans l'exemple précédent, les types statique et dynamique de `p` et `q` sont `Carre` et `Carre`, ceux de `t` et `u` sont `Polygone` et `Polygone` mais ceux de `r` et `s` sont `Polygone` et `Carre`. Cette distinction devient primordiale lorsque des classes filles redéfinissent des méthodes de la classe mère.

5.5.2 Méthodes virtuelles et polymorphisme

Lors de la création de classe fille par héritage, l'ajout d'information permet souvent d'améliorer les algorithmes utilisés dans les méthodes de la classe mère. Dans l'exemple précédent, tous les côtés d'un carré ont même longueur et il est inutile de calculer quatre fois la même quantité pour obtenir le périmètre ; c'est ce que fait pourtant la méthode `perimetre()` de la classe mère `Polygone`.

Il est donc possible de redéfinir une méthode dans la classe fille en utilisant le même prototype que celui utilisé dans la classe mère (`const` compris). Le doute s'installe lors de l'utilisation de tels objets.

Regardons le code suivant

```
Carre C(0.,0.,1.,0.); //creation d'un carré
double x;
x= C.perimetre();
```

Il est important de savoir quelle méthode est utilisée entre `Polygone::perimetre()` et `Carre::perimetre()` puisque le temps de calcul est différent entre les deux. Ici c'est celle de `Carre`.

Règles d'appel des méthodes en cas d'héritage. Les règles de choix des différentes méthodes en cas de surcharge dans une classe fille sont les suivantes :

1. il est toujours possible de choisir soi-même à l'écriture du code en complétant le nom de la méthode avec `var.classe::methode()` (ici `C.Carre::perimetre()` ou `C.Polygone::perimetre()`).
2. Par défaut, le type statique a priorité sur le type dynamique.
3. Si une méthode est qualifiée `virtual` dans la classe mère, le type dynamique a priorité sur le type statique.

Exemple. Imaginons que nous complétions les classes `Polygone` et `Carre` par une méthode `quisuisje()` avec le code suivant :

```
class Polygone {
        ...// on recopie le code précédent;
        void quisuisje() const; // SANS virtual d'abord.
};

class Carre: public Polygone {
        ...// on recopie le code précédent;
        void quisuisje() const;
};

void Polygone::quisuisje() const {
std::cout << "Je suis un Polygone." << std::endl; }

void Carre::quisuisje() const {
std::cout << "Je suis un Carre." << std::endl; }
```

La fonction `main()` donnée par

```
int main(void) {
        Carre C(0.,0.,1.,0.);
        C.quisuisje();
        Carre * r = &C;
        Polygone * q= &C;
        r->quisuisje();
        q->quisuisje();
        return 0; }
```

conduit à l'affichage :

```
Je suis un Carre.
Je suis un Carre.
Je suis un Polygone.
```

Si nous ajoutons le mot-clef `virtual` devant la déclaration de `quisuisje()` dans la classe `Polygone`, l'affichage change et devient :

```
Je suis un Carre.
Je suis un Carre.
Je suis un Carre.
```

L'avantage est à présent de pouvoir écrire des fonctions qui manipulent des objets de type `Polygone`, quelle que soit leur nature précise, du moment qu'elles n'utilisent que des méthodes de la classe-mère. Lors de l'exécution, en revanche, le choix de l'exécution de la méthode issue de la classe-mère ou de la classe-fille dépend de l'usage des `virtual`.

5.5.3 Méthodes virtuelles pures

On peut généraliser l'approche précédente en réorganisant l'écriture du code selon des classes mères dites *virtuelles pures*.

Prenons l'exemple mathématique du calcul de l'inverse dans un groupe. Toutes les classes que nous pouvons avoir envie d'écrire et qui correspondent à des groupes devraient avoir une méthode `inverse()`. Imaginons que nous voulions à présent écrire une fonction qui teste si un élément x est idempotent, i.e. si $x = x^{-1}$. Pour éviter de réécrire cette fonction pour chaque classe, on souhaite écrire une unique fonction du type :

```
bool is_idempotent(Groupe x) {
        Groupe y=x.inverse();
        return (y==x)?true:false;
}
```

Malheureusement, une telle classe ne peut exister puisque les groupes peuvent être très divers et le calcul de l'inverse dépend de chaque groupe. La seule information que nous ayons pour un groupe est l'existence d'un inverse. La solution pour cela est de construire une classe dite *virtuelle pure* `Groupe` dont hériteront toutes les classes qui décrivent des objets munis d'une structure de groupe. Nous obtenons ainsi :

```
class Groupe {
public:
        virtual Groupe & inverse() const =0 ;
};
```

La nouveauté est la présence du `=0` qui qualifie la méthode `inverse`. La valeur `0` n'a aucun sens mathématique et est une pure convention pour désigner des méthodes dites *virtuelles pures*.

Une classe est dite *virtuelle pure* si elle contient au moins une méthode *virtuelle pure*.

Une méthode *virtuelle pure* est une méthode dont on ne code pas le code et qui doit être nécessairement redéfinie dans toute classe fille.

Il est donc impossible de déclarer des objets de type `Groupe` mais on est certain que, pour tout objet `x` d'une classe fille de `Groupe`, la valeur `x.inverse()` aura un sens et la quantité pourra être calculée.

L'usage de classes virtuelles pures n'apporte rien au niveau algorithmique mais permet d'appliquer des fonctions à des classes d'objets *différents* qui partagent des caractéristiques *communes* (regroupées dans des classes mères virtuelles pures).

Exercices

Exercice 5.1. (constructeurs et destructeurs) Soit le fichier `testA.hpp` :

```
#include <iostream>
#ifndef BlocA
#define BlocA
class A {
protected:        int n;
public:
        A(): n(0) {std::cout << "Constr. par defaut.\n";}
        A(int p): n(p) {std::cout << "Constr. avec n=" <<p<< "\n";}
        A(const A & a): n(a.n) {std::cout <<"Copie avec n="<< n <<"\n";}
        ~A() { std::cout << "Destr. avec n=" << n << "\n";}
        int operator()() const { return n;}
        void add(int q) { n+=q;};
};
#endif
```

1. Compiler, exécuter et comprendre le résultat du code suivant :

```
#include "testA.hpp"
using namespace std;
void f(A a) { cout << "f !" << endl; }
void g(A & a) { cout << "g !" << endl; }
void h(A * a) { cout << "h !" << endl; }
A r() { cout << "r !" << endl;return A(17);}

int main() {
        cout << "\n---Initialisation---\n" ;
        A a;
        A b(a);
        b.add(5);
        A c(3);
        cout << "\n--- Quelques usages de fonctions: ---\n";
        f(b);
        g(a);
        h(&a);
        c=r();
        cout << "\n--- Le plus étrange ---\n";
        f(r()); // Optimisation du compilateur...
        cout << "\n--- La fin de la fonction main() ---\n";
        return 0;
}
```

2. Compiler également avec l'option de compilation `-fno-elide-constructors` et observer le changement de comportement.
3. (durée de vie) Comprendre le résultat du code suivant :

```
#include "testA.hpp"
void f(const A & a,int i) { std::cout << "f: "<< a()+i <<"\n";}
int main() {
        std::cout << "\n---Première boucle---\n";
        for(int i=0; i<4 ; i++) {
                A a(50);
                f(a,i);
        }
        std::cout << "\n---Seconde boucle---\n";
        A b(70);
        for(int i=0; i<4 ; i++) {
                f(b,i);
        }
        return 0;
}
```

Qu'en concluez-vous sur l'intérêt de déclarer certaines variables dans ou avant une boucle ?

4. Comprendre le code suivant :

```
#include "testA.hpp"
#include <array>
#include <vector>
using namespace std;
int main() {
        A a(63);
        cout << "\n---Construction de U---\n" ;
        vector<A> U(3,42);
        cout << "\n---Construction de V---\n" ;
        array<A,2> V;
        cout << "\n---Retrait puis Ajout d'un élément à U---\n" ;
        U.pop_back();
        U.push_back(a);
        cout << "\n---Ajout d'un nouvel élément à U---\n" ;
        U.push_back(59); // changement de capacité du vector !
        cout << "\n";
        U.push_back(94); // rien de spécial
        cout << "\n---Terminaison---\n" ;
        return 0;
}
```

Exercice 5.2. (héritage, types statique et dynamique) Soient les classes suivantes définies dans un fichier `test-herit.hpp` :

```
#include <iostream>
#ifndef CLASSES
```

```
#define CLASSES
class F {
public:
void methodeA() const { std::cout << "Methode A de F\n";}
virtual void methodeB() const { std::cout << "Methode B de F\n";}
};

class G: public F {
public:
void methodeA() const { std::cout << "Methode A de G\n";}
void methodeB() const { std::cout << "Methode B de G\n";}
};

class H: public F {
public:
virtual void methodeA() const { std::cout << "Methode A de H\n";}
void methodeB() const { std::cout << "Methode B de H\n";}
};
#endif
```

(a) Devinez le résultat du code suivant puis vérifiez en l'exécutant :

```
#include "test-herit.hpp"
int main() {
        F u;        G v; H w;
        F * pu = new F; F * qu = new F; F & ru(u);
        F * pv = new G; G * qv = new G; F & rv(v);
        F * pw = new H; H * qw = new H; F & rw(w);
        std::cout << "\n---Premier cas---\n";
        u.methodeA(); u.methodeB();
        pu->methodeA(); pu->methodeB();
        qu->methodeA(); qu->methodeB();
        ru.methodeA(); ru.methodeB();
        std::cout << "\n---Deuxieme cas---\n";
        v.methodeA(); v.methodeB();
        pv->methodeA(); pv->methodeB();
        qv->methodeA(); qv->methodeB();
        rv.methodeA(); rv.methodeB();
        std::cout << "\n---Troisieme cas---\n";
        w.methodeA(); w.methodeB();
        pw->methodeA(); pw->methodeB();
        qw->methodeA(); qw->methodeB();
        rw.methodeA(); rw.methodeB();
        return 0;
}
```

(b) Devinez le résultat du code suivant puis vérifiez en l'exécutant :

```
#include "test-herit.hpp"
#include <vector>
int main() {
        std::vector<F> V;
        V.push_back(F());
        V.push_back(G());
        for (unsigned int i = 0; i < V.size(); i++)
        {
                std::cout << "\n---1/ Iteration "<< i << "\n";
                V[i].methodeA();
                V[i].methodeB();
        }
        std::vector<F *> W;
        W.push_back(new F);
        W.push_back(new G);
        for (unsigned int i = 0; i < W.size(); i++)
        {
                std::cout << "\n---2/ Iteration "<< i << "\n";
                W[i]->methodeA();
                W[i]->methodeB();
        }
        for (unsigned int i = 0; i < V.size(); i++) delete W[i];
        return 0;
}
```

Exercice 5.3. (accesseurs et mutateurs) Soit la classe définie dans le fichier `test-accessmut.hpp` par :

```
#include <iostream>
#ifndef CLASSE
#define CLASSE
class Bavard {
protected:        char private_var;
public:
Bavard(char c): private_var(c) {}
char val() const { std::cout << "Accesseur!\n"; return private_var;}
char & val() { std::cout << "Mutateur!\n"; return private_var;}
char methode1() const { char u=val(); return u+1;}
char methode2() { char u=val(); return u+1;}
friend std::ostream & operator<<(std::ostream &,const Bavard &);
};

std::ostream & operator<<(std::ostream & o,const Bavard & c)
{ return o << c.val();}
#endif
```

Deviner et comprendre l'affichage du code suivant :

```
#include "test-accessmut.hpp"
void f(const Bavard & h) { h.val(); }
void g(Bavard & h) { h.val(); }
int main() {
        Bavard K('z');
        std::cout <<"\n---Basique---\n";
        K.val() = 'x';
        f(K); g(K);
        K.methode1(); K.methode2();
        std::cout <<"\n---Moins basique---\n";
        std::cout << K << "\n" << K.val();
        std::cout <<"\n---Avec des pointeurs---\n";
        Bavard * p=&K;
        const Bavard * q=&K;
        Bavard * const r=&K;
        p->val(); q->val(); r->val();
        return 0;
}
```

Chapitre 6

Programmation générique et *templates*

6.1 Introduction à la programmation générique

6.1.1 Présentation de quelques exemples

Un premier exemple : le minimum. Supposons que nous souhaitons calculer le minimum de deux entiers, de deux réels en faible précision `float`, de deux réels en `double`, etc. Il faut *a priori* écrire autant de fonctions qu'il y a de types mais leurs codes ne diffèrent que par leur prototype :

```
int min(int x, int y) { return ( x<y )?x:y; }
float min(float x, float y) { return ( x<y )?x:y; }
double min(double x, double y) { return ( x<y )?x:y; }
```

Il faudrait également que toute personne qui concevrait une nouvelle classe pour laquelle le symbole `<` aurait un sens écrive également sa propre fonction `min` en recopiant une fois de plus ces instructions. C'est long, fastidieux et source d'erreurs de copie s'il faut le faire pour de nombreuses fonctions et que le code de chacune ne tient pas en une ligne...

Pour simplifier, cela le C++ offre la syntaxe suivante :

```
template <typename T>
T min( T x, T y ) { return (x<y)?x:y; }
```

Cela s'appelle un modèle de fonction (ou « *template* » en anglais) et cela décrit un algorithme de calcul du minimum sans se soucier de la nature exacte des objets. Nous voyons également que le type d'objet devient lui-même un paramètre, appelée ici `T`, qui doit donner le nom d'un type et donc qualifié par `typename`.

L'utilisation d'un tel modèle se fait alors simplement par des appels du style :

```
min<int>(3,7)        // donne 3
min<double>(-78.0,1.3e10)        // donne -78.
min<char>('d','s')        // donne 'd'
```

De la géométrie dans $\mathbb{R}^n$. Imaginons que nous souhaitions manipuler des points dans un espace de dimension n fixée sur $\mathbb{R}$ ou sur $\mathbb{C}$, par exemple pour faire de la géométrie dans le plan ou dans l'espace. La dimension de l'espace est fixée *a priori* ainsi que le corps de base. Pour un décrire un point dans $\mathbb{K}^n$, on peut ainsi construire un *modèle de classe* tel que celui décrit dans la figure 6.1.

Cette fois-ci, les deux paramètres du modèle décrivent d'une part le corps de base $\mathbb{K}$ (`K` devra être remplacé par exemple par `double` si on veut des réels en double précision) et d'autre part la dimension de l'espace. Toutes les déclarations se font comme au chapitre précédent à ceci près qu'il faut spécifier les paramètres de modèle à chaque appel au modèle de classe.

On déclare alors un point de $\mathbb{R}^3$ par `Point<double,3> P;` et un point de $\mathbb{C}^2$ par `Point<Complexe,2> Q;` à condition d'avoir préalablement défini (ou chargé à partir d'une bibliothèque) une classe qui décrive les nombres complexes avec leurs opérations usuelles.

Les modèles `std::vector` et `std::list` de la STL. Nous avons déjà manipulé des modèles de classe en utilisant les bibliothèques `std::vector` et `std::list` . En effet, la majorité des algorithmes (`push_back()` ou l'opérateur de `[]`) ne dépendent pas de la nature des objets placés dans un tableau ou une liste. Ce sont donc des modèles de classes que la STL définit et cela explique les notations `std::vector<double>` avec des chevrons.

6.1.2 Vocabulaire et syntaxe

La déclaration d'un modèle (« template ») de fonction ou de classe est toujours défini par la syntaxe :

```
template < TYPE1 A1, TYPE2 A2, ... >
PROTOTYPE DE LA FONCTION {
        ... // code de la fonction
}

template < TYPE1 A1, TYPE2 A2, ... >
class NOM_DE_CLASSE {
        ... // definition de la classe
};
```

où `A1` , `A2` , etc sont appelés *paramètres* du modèle et sont de type[1] soit `class` ou `typename` (i.e. ce sont des noms de types), soit `int` . Le prototype, le code et les définitions peuvent alors utiliser `A1` , `A2` , etc., partout où sont attendus soit des noms de type, soit des valeurs d'entiers.

L'appel d'un modèle de fonctions sur des arguments se fait par

```
NOM_DE_LA_FONCTION < VALEUR_DE_A1, VALEUR_DE_A2, ... > ( ARGUMENTS )
```

et la déclaration d'un élément d'une classe par

1. Ce n'est rigoureusement exact en fait. La vérité est que tous les types ne sont pas permis (`double` par exemple est interdit). Les entiers et les noms de type sont en revanche les plus utilisés.

```
//Annonce des templates
template <class K, int n> class Point;

template <class K, int n>
ostream & operator<<(ostream &,const Point<K,n> &);

//Declaration des templates
template <class K, int n>
class Point {
private:
        K coord[n];
public:
        Point();
        Point(const Point &);
        ~Point();
        K operator()(int i) const ; //accesseur
        void translate(const Point &);
        void oppose(void);
        friend ostream & operator<< <>(ostream &,const Point<K,n> &);
};

// Code des methodes et fonctions
template <class K, int n>
K Point<K,n>::operator() (int i) const { return coord[i]; }

template <class K, int n>
void Point<K,n>::oppose () { for(int i=0;i<n;i++) coord[i]=-coord[i]; }

template <class K, int n>
void Point<K,n>::translate (const Point<K,n> & U) {
        for(int i=0;i<n;i++) coord[i] = coord[i]+U.coord[i];
}

template <class K, int n>
ostream & operator<<(ostream & o,const Point<K,n> & P) {
        for(int i=0;i<n;i++) o << P.coord[i] << " ";
        return o;
}
```

FIGURE 6.1 – Exemple de modèle de classe pour décrire des points dans $\mathbb{K}^n$ où $\mathbb{K}$ est un corps quelconque et n un entier plus grand que 1.

```
NOM_DE_LA_CLASSE < VALEUR_DE_A1, VALEUR_DE_A2, ... > VARIABLE;
```

Un choix de valeurs de paramètres `A1` , `A2` , etc pour un modèle s'appelle une *instanciation* du modèle.

6.1.3 Structure du code et compilation

Le traitement des modèles par le compilateur est très spécifique et nécessite d'être bien maîtrisé pour comprendre l'utilisation de `template` .

Virtuellement, toutes valeurs des paramètres de modèle `A1` , `A2` , ... peuvent être choisies mais toutes les instanciations possibles ne sont pas compilées bien évidemment. Le compilateur lit l'intégralité du code qui utilise des `template` et établit la liste des instanciations nécessaires pour l'exécution du code. Ensuite, il reprend le modèle de `template` en substituant les valeurs de paramètres nécessaires et compile successivement chaque instanciation.

Cela a plusieurs conséquences majeures. La première est que les valeurs des paramètres doivent être connues des programmeurs avant la compilation et ne peuvent pas être choisies par un utilisateur du programme. Dans l'exemple de géométrie de la section précédente, il faut choisir avant la compilation du programme la dimension de l'espace et on ne peut pas demander à l'utilisateur s'il préfère $\mathbb{R}^2$ ou $\mathbb{R}^3$... Le programmeur lui impose ce choix.

La deuxième conséquence est que le code de modèles de classes ou de fonctions ne peut pas être compilé séparément à l'avance, comme c'est le cas pour des fonctions ou des classes, car on ne connaît pas les valeurs de paramètres qui seront choisies pour les instanciations : en effet, c'est le type précis des objets qui dicte la gestion de la mémoire et donc les instructions machine. **L'intégralité du code d'un `template` doit figurer dans le `.hpp`** et ne peut plus être scindé entre déclarations dans le `.hpp` et le `.cpp` . C'est une erreur très classique à laquelle il faut faire extrêmement attention.

La troisième conséquence est que la compilation en présence d'un `template` peut être longue puisque le compilateur doit parfois compiler de très nombreuses instanciations différentes d'un même `template` .

Enfin, il y a un avantage majeur à utiliser des `template` avec des paramètres entiers pour l'optimisation du code par le compilateur. Puisque la valeur d'un paramètre entier d'un `template` est connu dès la compilation, le compilateur peut faire des choix particuliers qui utilise cette valeur pour simplifier des morceaux de code et supprimer des tests inutiles. L'accélération est notable dans de nombreux cas et a fait le succès de bibliothèques comme `boost` par exemple.

Revenons sur le cas particulier de l'exemple de géométrie précédent. Sans la notion de `template` , nous aurions écrit (pour des coordonnées réelles) une classe :

```
class Point {
private:
        int n;
        double * coord;
public:
        Point();
        Point(const Point &);
        ~Point();
```

```
        double operator()(int i) const ; //accesseur
10      void translate(const Point &);
        void oppose(void);
12      friend ostream & operator<<(ostream &,const Point &);
};
```

L'entier `n` devient un simple champ et peut être choisi par l'utilisateur. Le champ `coord` ne peut plus être un tableau statique et devient un tableau dynamique qu'il faut gérer avec des `new` et `delete`. Cela laisse une plus grande liberté à l'utilisateur mais le code sera moins optimisé et le programme plus lent. Le type d'optimisation que peut appliquer le compilateur est le « déroulement de boucles » : si le compilateur sait qu'il y aura trois itérations d'une boucle du style

```
for( int i=0; i < n; i++) p[i]+=u[i];
```

parce qu'il sait que `n` vaut 3 (parce que c'est par exemple un paramètre de modèle), il prend si nécessaire la liberté de transformer le code en

```
p[0]+=u[0];
2 p[1]+=u[1];
p[2]+=u[2];
```

Cela permet de ne pas avoir à calculer l'incrément `i++` ni à faire le test `i<n` à chaque itération. Ce sont des opérations certes rapides mais malheureusement très nombreuses en calcul intensif par exemple.

Pour décider si un paramètre entier doit être passé en `template`, il faut trouver un compromis entre la liberté de fixer une valeur à l'utilisation (sans `template` donc) et le gain de temps de calcul (avec `template`). Seul l'usage que l'on destine au programme permet de juger.

6.1.4 Particularités pour les modèles de classe

La définition des méthodes d'un modèle de classe ne se fait plus dans le `.cpp` mais dans le `.hpp` après la définition de la classe. Il peut donc y avoir une compilation de moins.

La définition d'une méthode d'un modèle de classe se fait en écrivant à nouveau `template` mais la classe doit être instanciée avant l'opérateur de résolution `::` comme l'indique l'exemple de la figure 6.1 aux lignes 24, 27 et 30.

Comme pour les fonctions et les classes, il est possible de déclarer des liens d'amitié entre modèles de fonction et modèles de classes avec le mot-clef `friend`. Considérons l'exemple minimal suivant :

```
template <typename U> class A;
2 template <typename U> void f(A<U>);
```

Il faut interpréter ces modèles comme une collection de classes `A<K>` pour toutes les valeurs

de `K` et, pour chacune d'elle, on a une fonction qui prend en argument un objet de type `A<K>`. Deux liens d'amitié sont alors possibles : soit toutes les fonctions `void f(A<U>)` sont amies avec toutes les classes `A<V>` pour toutes valeurs de `U` et `V`, soit chaque fonction `void f(A<U>)` est amie avec la seule classe `A<U>`. Les deux sont possibles avec la syntaxe

```
template <typename U> class A {
        ... // definition de la classe
        friend template<typename V> void f(A<V>);
};
```

pour le premier cas (amitiés entre toutes les instanciations de chaque) et

```
template <typename U> class A {
        ... // definition de la classe
        friend void f<U>(A);
};
```

pour le deuxième cas (amitiés seulement entre des instanciations associées).

6.1.5 Redéfinition d'une instanciation particulière

Supposons que nous ayons un *template* donné par

```
template <int N, class A >
class C {
... // une certaine classe.
}
```

et que nous sachions que, si `A` est `double`, nous connaissions une version simplifiée de la structure. Alors, nous pouvons ajouter après le code précédent la nouvelle définition

```
template <int N>
class C<N,double> {
... // la structure simplifiée
};
```

Il est aussi possible de spécifier tous les arguments du *template* avec la syntaxe

```
template <>
class C<1,int> {
... // une autre version simplifiée
};
```

Pour des *templates* de fonctions, cela fonctionne de la même manière mais il n'est pas possible d'avoir des spécialisations partielles mais seulement des redéfinitions avec tous les arguments de *template* fixés. Par exemple, nous pouvons écrire

```
template <int K, int N, class A>
A f(int u, A x) {
... //un certain algorithme
}

template <>
double f<3,4,double> (int u, double x) {
... // un algorithme simplifié
}
```

Ces spécialisations sont particulièrement utiles lors de définitions récursives de *templates* sur les entiers pour fixer la valeur initiale de la récurrence.

6.2 Quelques exemples issus de différentes bibliothèques

6.2.1 La bibliothèque standard et la STL

Conteneurs et itérateurs

Comme nous l'avons déjà au chapitre 4 — dont l'acronyme signifie « Standard Template Library » !—, la bibliothèque standard contient de nombreux modèles de classes. De manière générale, ils permettent de stocker des collections, ordonnées ou non, d'objets. Le type des objets est un paramètre de chaque `template`.

Ce qui diffère d'un `template` à l'autre est la présence d'un ordre entre les éléments et la navigation dans la mémoire d'un objet de la collection à l'autre.

Outre la nature indifférenciée des éléments des conteneurs, on peut également une couche supplémentaire de *templates* en écrivant des algorithmes qui ne dépendent pas (ou presque pas) de la nature exacte du conteneur : c'est précisément ce qui fait dans la bibliothèque `<algorithm>`.

6.2.2 Objets mathématiques standards

Différents objets mathématiques classiques donnent lieu à des modèles de classe à cause de la présence de différentes classes pour décrire les mêmes objets mathématiques avec des précisions variées.

Un premier exemple est la bibliothèque `<complex>`. Les nombres réels sont remplacés par des flottants avec des précisions variées selon que l'on choisit des `float`, `double` et `long double`. Cela se répercute sur les complexes que l'on peut décrire comme des couples de réels : la bibliothèque `<complex>` permet de définir des nombres complexes avec une précision voulue avec un modèle de classe `std::complex<T>` où `T` est l'un des trois types de flottants rappelés ci-dessus.

Un deuxième exemple est la bibliothèque `<random>` en `C++` 11 déjà introduite au chapitre 4 : les variables aléatoires classiques sont à valeurs dans $\mathbb{Z}$ ou $\mathbb{R}$. En `C++`, il faut donc décider avec quelle précision les variables aléatoires doivent être générées : les lois à support dans $\mathbb{Z}$ sont donc des modèles de classes dont le paramètre du modèle est l'un des types `int`, `long int` et les lois des variables à support dans $\mathbb{R}$ sont ainsi décrites par des modèles de classe de paramètre `float`, `double` ou `long double`.

6.2.3 Un exemple d'algèbre linéaire : la bibliothèque `Eigen`

Nous avons déjà utilisé la bibliothèque `Eigen` dans divers exemples de ce cours et des TP. Cette bibliothèque permet de faire des calculs matriciels de manière très efficace. Ce n'est qu'un exemple parmi d'autres : la bibliothèque `armadillo` est d'une efficacité comparable et il en existe beaucoup d'autres.

Les matrices de la bibliothèques `Eigen` sont présentes sous deux formes différentes :

```
Eigen::Matrix<typename T, int N, int M> A; //statique
Eigen::Matrix<typename T, Eigen::Dynamic, Eigen::Dynamic> B; //dynamique
```

L'avantage du premier est la possibilité pour le compilateur d'optimiser le code s'il l'estime nécessaire (par exemple en déroulant des boucles courtes si `N` et `M` sont petits). L'avantage du second est de laisser à l'utilisateur le soin de fixer la taille des matrices `B` ainsi que de pouvoir modifier la taille des matrices en court d'exécution.

Pour les matrices qui contiennent beaucoup de zéros, `Eigen` contient également un format appelé `sparse` qui permet de ne stocker en mémoire que les coefficients non nuls des matrices.

Pour tirer un profit véritable de cette bibliothèque, il convient de compiler avec les options d'optimisation `-O2` et `-O3` : nous vous invitons à effectuer les comparaisons de temps d'exécution chez vous selon l'activation ou non de ces options ainsi que selon l'usage de taille `N` et `M` ou `Eigen::Dynamic` .

6.2.4 Les nouveautés des standards récents de C++ et la notion de std : :tuple

L'un des domaines où le standard `C++` évolue le plus et acquiert de nombreux enrichissements est le domaine de la programmation générique. Ainsi, `C++` a vu l'ajout dans le standard de 2011 des *templates variadiques* qui permettent d'avoir un nombre d'arguments du *template* indéterminé à la définition et fixé à l'instanciation.

En particulier, l'usage des `std::tuple` introduit dans la bibliothèque `<tuple>` depuis `C++` 11 permet de définir des fonctions avec plusieurs types de retours via un prototype du type

```
std::tuple<int, double, std::string> f(int a);
```

qui renvoie un 3-uplet de trois objets. La déclaration, la construction et l'accès à un élément d'un `std::tuple` se fait selon :

```
std::tuple<int, char, std::string> A(4,'k',"abc");
std::get<1>(A) = 'w';
std::get<2>(A) += "def";
std::cout << std::get<2>(A) << std::endl;

int x;
char c;
std::string S;
std::tie(x,c,S) = A;
std::cout << x << " " << c << " " << S << std::endl;
```

La fonction `std::tie` permet de répartir les éléments d'un `std::tuple` dans différentes références de variables préexistantes.

De manière générale, un *template variadique* se définit selon :

```
template < class T, class ... Types>
T f(const class Types & ... arguments) {
...
}
```

Nous vous conseillons néanmoins de bien maîtriser les *templates* présentés dans ce chapitre avant de vous intéresser aux nouvelles fonctionnalités.

Exercices

Exercice 6.1. Soit le modèle de fonction suivant :

```
template < class Conteneur >
void affichage(std::ostream & o,const Conteneur &  C) {
        for(auto iterateur= ___ ;  ____ ; ____ ){
                o << ___;
        }
        return;
}
```

1. Compléter le *template* suivant pour qu'il puisse afficher le contenu de n'importe quel conteneur possédant des itérateurs et qui possèdent des `begin()`, `end()` et `*` que la STL. Les éléments doivent être séparés par des espaces.
2. Tester le programme sur le conteneur `std::vector<double>` ainsi que sur le conteneur `std::list< std::string >`.
3. Quelle condition d'utilisation y a-t-il sur le type des objets du conteneur ?

Exercice 6.2. (important !) Écrivez des implémentations personnelles des algorithmes de la STL tels `std::any_of`, `std::count_if`, `std::copy`, `std::transform`, etc., et testez-les sur des exemples. Comparez-les ensuite au code de la STL pour le compilateur `g++`.

Exercice 6.3. (héritage, modèles récursifs ... et vertige) Comprendre la structure de la classe `Boite<10>`, en particulier combien de champs `a` elle contient réellement. Quel est le résultat du programme ?

```
#include <iostream>
template <int p> class Boite: public Boite<p-1> {
protected:
        int a;
public:
        Boite(): Boite<p-1>(), a(p) {}
        void write() const ;
};

template <int p> void Boite<p>::affichage() const {
        Boite<p-1>::write();
        std::cout << "niveau " << p<< ": " << a << std::endl; }

template <> class Boite<0> {
protected:
        int a;
public:
        Boite(): a(0) {}
        void write() const { std::cout<< "niveau 0: " << a << std::endl;}
```

```
20 };

22 int main() {
       Boite<10> u;
24     std::cout << "Contenu de u:\n";
       u.write();
26 }
```

Chapitre 7

Gestion de la mémoire

7.1 Notions sur les pointeurs

7.1.1 L'un des atouts du C et du C++

Hormis les références, la plupart des types précédemment correspondent à des objets mathématiques relativement bien définis. Toutefois, nous avons vu au début de cette section que, en informatique, une *variable* est un objet plus évolué qui comportent trois autres caractéristiques au delà de sa simple valeur (son nom, son type et son adresse) qui permettent de gérer la mémoire. Il faut donc définir des types spéciaux de variables dont les valeurs permettent de manipuler efficacement la mémoire : ce sont les *pointeurs*.

Les *pointeurs* forment un aspect incontournable des langages `C` et `C++`. D'autres langages, tel Python, ne les utilisent pas et ils ne sont donc pas incontournables *a priori*. Leur intérêt vient dès lors qu'on souhaite faire du calcul intensif car les pointeurs permettent de gérer la répartition de la mémoire et les accès à cette dernière de manière particulièrement efficace. C'est cela qui explique l'intérêt du `C` et du `C++` pour des mathématiciens et c'est aussi pourquoi ce chapitre est essentiel dans ce cours.

7.1.2 Les adresses en mémoire

La mémoire peut être décrite comme une longue suite unidimensionnelle de bits (ou plutôt d'octets, i.e. une suite de 8 bits) juxtaposés et numérotés de 1 jusqu'au dernier. Une variable de type élémentaire (cf. les exemples du chapitre 3) occupe en général une suite contiguë de bits en mémoire et son *adresse* est tout simplement les coordonnées en mémoire de son premier octet. Pour héberger ces coordonnées, il faut donc de nouvelles variables dont la valeur est l'adresse en mémoire d'un octet donné : ce sont les *pointeurs*.

Un type de pointeur générique est le type `void *`. Une variable `p` déclarée par l'instruction `void * p` est une variable dont la valeur pourra être l'adresse d'une autre variable. La variable `p` a sa propre adresse qui n'a rien à voir avec sa valeur et une valeur qui n'a rien à voir avec la valeur de la variable pointée [1].

Il faut retenir qu'un pointeur est utilisé pour naviguer efficacement entre les variables dans la mémoire sans avoir à se soucier de leur valeurs (cela est d'autant plus utile que les valeurs stockées sont complexes à manipuler). Pour cet aspect, il ressemble partiellement aux références.

1. Il est important de passer du temps à comprendre cette phrase ; pour cela n'hésitez pas à faire un schéma de ce qui se passe dans la mémoire.

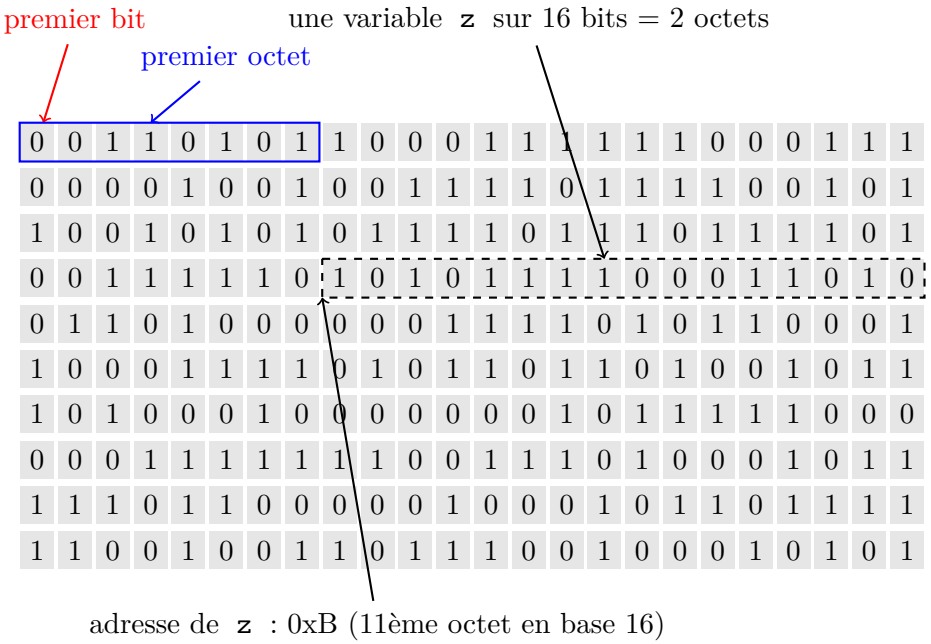

FIGURE 7.1 – Dessin schématique de la mémoire. Nous avons décidé 240 bits=30 octets consécutivement (il faut imaginer que chaque ligne suit immédiatement la précédente). La variable `z` peut être aussi bien du type `short int` ou `char[2]` car elle occupe deux octets.

On remarque également que l'adresse donnée par le pointeur ne donne aucune information sur la taille occupée en mémoire par la variable. Cet inconvénient est partiellement contournée par l'introduction de différents types de pointeurs, correspondant à tous les types déjà introduits. Un pointeur vers un entier a un type `int *`, un pointeur vers un flottant a un type `float *` ou `double *`. Ces types seront utilisés d'ici quelques paragraphes pour translater des adresses selon un mécanisme que nous préciserons mais, formellement, ils sont tous du même type et peuvent être convertis les uns dans les autres indifféremment par conversion implicite.

7.1.3 Récupérer l'adresse d'une variable et y accéder.

L'adresse d'une variable préexistante est récupérée par l'opérateur `&`, dont le symbole est appelé en français *esperluette*. L'accès à la valeur écrite à une adresse donnée se fait par l'opérateur `*`. Voici un exemple :

```
int n=147, m=78;
int * p1;
double x=3.1415;
double * p2;

p1 = &n; // récupération de l'adresse de n
std::cout << p1 << std::endl;        // affichage de l'adresse de n
        // la valeur change à chaque exécution
        // le format n'est pas simple à interpréter
m = *p1 ; // stockage dans m de la valeur pointée par p1
```

```
std::cout << m << std::endl; // affichage de m qui vaut alors 147;

p2 = &x; // récupération de l'adresse de x
std::cout << p2 << std::endl; // affichage de l'adresse de x
std::cout << *p2 << std::endl; // affichage de la valeur pointée par p2
```

Dans l'exemple précédent, l'accès à la *valeur* de `n` peut se faire par deux méthodes différentes : soit par `n=5;` , soit par `*p1=5;` puisque `p1` pointe vers `n` . Ainsi, nous avons :

```
int n=147;
int * p1= &n; // récupération de l'adresse de n
*p1 = 5;
std::cout << "n vaut maintenant : " << n << std::endl;
        // c'est bien la valeur 5 qui s'affiche !
```

Remarque. Il est clair que, pour toute variable `x` , la commande `*&x` donne la valeur de `x` : on récupère l'adresse puis on lit la valeur pointée par l'adresse. En revanche, `&*p` sur un pointeur `p` n'a pas de sens ! L'étoile permet de récupérer la valeur pointée par `p` mais une valeur n'a pas d'adresse puisqu'elle peut être écrite n'importe où. Il suffit pour cela de se convaincre[2] que `int *p = &3;` n'a pas de sens puisque 3 est une valeur, n'a pas d'endroit réservé dans la mémoire et peut être écrit dans toute variable entière.

L'usage du `const`

Comme les références, les pointeurs peuvent être utilisés pour modifier des valeurs de variables en donnant leurs adresses en argument à des fonctions ou pour éviter d'avoir à recopier leurs valeurs dans la mémoire. Là encore, pour ce dernier usage, le mot-clef `const` permet d'indiquer que la fonction ne modifie pas les valeurs derrière les adresses. C'est donc une alternative aux références de ce point de vue. Par exemple, pour échanger les valeurs de deux variables de type `TYPE` , on peut écrire la fonction `echange` suivante :

```
void echange( TYPE * p, TYPE * q) {
        TYPE a= *p;
        *p = *q;
        *q = a;
        return;
}
```

Il faut faire attention au placement du `const` dans une déclaration de pointeur. L'écriture `const int * p;` désigne un pointeur vers un entier dont la valeur ne peut changer mais la valeur du pointeur peut évoluer et pointer vers un autre entier non modifiable ; en revanche, la syntaxe `int * const p;` désigne une adresse non modifiable et définitivement figée mais l'entier pointé peut évoluer.

2. Intéressez-vous au message d'erreur que vous obtenez lors de la compilation.

Erreur de segmentation.

À chaque déclaration de variable, c'est le système d'exploitation de l'ordinateur qui attribue les adresses des variables. Il ne faut donc jamais fixer explicitement soi-même la valeur d'un pointeur : avec une probabilité très proche de 1, cette adresse ne correspond pas à une variable déclarée. Ce problème est indétectable à la compilation ; le programme s'arrête brutalement à la compilation dès qu'on tente d'accéder à une adresse inexistante et affiche simplement `Erreur de segmentation` (ou `Segmentation fault` en anglais). C'est pourquoi l'opérateur `&` est inévitable : il est le seul moyen de récupérer l'adresse d'une variable préexistante.

Pointeurs de structure et classe et syntaxe `->`

Supposons que nous ayons préalablement défini une structure (resp. classe) `S` avec des champs `champ1` , `champ2` , etc. Un pointeur `p` vers un objet de type `struct S` (resp. `S`) est donc défini par `struct S * p` (resp. `S * p;`). Pour accéder aux champs `champ1` , `champ2` (resp. aux champs ou aux méthodes) de l'objet pointé, il faut *a priori* écrire :

```
(*p).champ1
(*p).champ2
```

Les parenthèses sont inévitables et l'écriture peut être parfois lourde. Pour éviter cela, il existe une syntaxe complètement équivalente et plus légère utilisant `->` . Les deux lignes précédentes se réécrivent alors :

```
p->champ1
p->champ2
```

Cette syntaxe marchera également pour l'appel de méthodes de classe définies au chapitre 5.

7.1.4 Les références : un cas particulier de pointeur

Nous avons introduit brièvement en section 3.1.5 la notion de référence. Nous avons à présent toutes les notions sur la mémoire pour définir plus rigoureusement la notion de *référence* en `C++`. Cette notion est absente dans le langage `C`.

Une *référence* `r` est un genre particulier de pointeur avec les propriétés suivantes :

1. on ne peut pas le réaffecter (l'objet qu'il pointe est choisi lors de la déclaration de la référence),
2. on ne peut pas récupérer son adresse (`&r` renvoie l'adresse de l'objet pointé et non l'adresse de la référence `r`),
3. il se déréférence automatiquement (pas besoin d'utiliser `*` pour accéder à l'objet pointé).

Le type d'une référence est `T &` où `T` est un type déjà défini.

Selon le premier point mentionné ci-dessus, une tentative de réaffectation `r=a;` ne redéfinit pas `r` comme une référence vers `a` mais remplace directement le contenu pointé par `r` par la valeur de `a` (pour un pointeur `p` , cela correspondrait à `*p=a;`).

Comme tout pointeur, une référence peut être ou non affublée d'un `const`. Dans un prototype de fonction, l'usage d'une référence peut ainsi se substituer à celle d'un pointeur dans un argument de fonction pour éviter que l'objet pointé ne soit recopié dans la mémoire. L'usage d'une référence rend même le code de la fonction beaucoup plus lisible car cela permet de supprimer tous les `*`, `&` et `->` nécessaires lors de la manipulation de valeurs désignées par des pointeurs.

7.2 Gestion dynamique de la mémoire

7.2.1 Arithmétique des pointeurs.

Se déplacer dans la mémoire.

Comme nous l'avons écrit, un pointeur est une adresse mémoire, i.e. un numéro d'octet dans la mémoire, transcrit usuellement en base hexadécimale. Il y a tout d'abord la valeur nulle, donnée par le mot-clef `NULL` qui pointe vers un soi-disant 0-ème octet de la mémoire qui n'existe donc pas ! Tout accès par `*` à un pointeur de valeur `NULL` donne une erreur de segmentation.

Les pointeurs peuvent être translatés et incrémentés puisqu'ils sont stockés sous forme d'entiers. Un incrément de 1 par la commande `p++` sur un pointeur de type `double *` augmente la valeur de `p` de la taille nécessaire pour stocker un `double`. De manière générale, une opération `p=p+n` où `p` est un pointeur de type `TYPE *` et `n` un entier augmente `p` de `n` fois la taille occupée en mémoire par une variable de type `TYPE`.

Considérons l'exemple suivant :

```
short int *p=NULL;
int *q=NULL;
long double *r=NULL;
cout << p << "  " << q << "  " << r << endl;
        //cela affiche: 0  0  0
p++;
q++;
r++;
cout << p << "  " << q << "  " << r << endl;
        //cela affiche: 0x2  0x4  0x10
cout << q+512 ; //cela affiche:  0x804
```

Essayons de comprendre l'affichage. Comme vu précédemment, le préfixe `0x` indique qu'il faut la valeur qui suit en hexadécimal. Un entier de type `short int` est codé sur deux octets : décaler `p` de 1 consiste donc à passer du zéro-ème octet au deuxième. Un `long double` est codé sur 16 octets, or 16 s'écrit $\overline{10}$ en base 16 : incrémenter `r` par `r` décale l'adresse de 16 octets. Après la ligne 7, `q` vaut 4. On ajoute alors 512 fois la taille en octets d'un `int` codé sur 4 octets ; `q` vaut alors 2048+4=2052 qui s'écrit 804_{16} en base 16.

Pour connaître le nombre d'octets en mémoire occupé par une variable d'un type `TYPE`, il suffit d'appeler la fonction `sizeof(TYPE)`. C'est ainsi qu'on connaît l'effet de `++` sur un pointeur `TYPE *`.

Lien avec les tableaux.

L'usage de l'arithmétique des pointeurs nous permet de mieux comprendre la structure des tableaux. Un tableau `double t[50]` correspond dans la mémoire à 50 `double` stockés *consécutivement*. Techniquement, il suffit donc de connaître l'adresse de la première case et la taille d'une case ; la taille sert pour le créer en allouant la bonne quantité de mémoire et pour éviter ensuite les erreurs de segmentation.

Dans la déclaration `double t[50];` , l'objet `t` est en fait un pointeur vers le premier élément, i.e. l'élément numéroté 0. Pour le vérifier, il suffit d'afficher la valeur de `*t` et constater que cela correspond bien à la valeur du premier élément. De la même manière, l'accès à l'élément de la case numéro `k` (i.e. la `k+1` -ème case du tableau) peut se faire par la commande `*(t+k)` ; c'est bien sûr moins sympathique à écrire et c'est pourquoi existe la syntaxe `t[k]` déjà rencontrée.

Ce mécanisme de gestion des tableaux explique également pourquoi on obtient une erreur de segmentation si l'on écrit `t[i]` avec un entier `i` qui n'est pas dans l'ensemble $\{0, 1, 2, \ldots, 49\}$ puisqu'une telle case n'est pas attribuée au programme dans la mémoire.

7.2.2 Allocation et désallocation de mémoire

Jusqu'à maintenant, nous avons expliqué comment les types simples, les tableaux statiques et les modèles de classe de la STL sont codés en mémoire mais nous n'avons pas géré la mémoire nous-même : cela a été fait automatiquement par le programme et le système d'exploitation lors de la déclaration de la variable. Cela a un inconvénient majeur : il a fallu attribuer un nom à chaque emplacement lors de l'écriture du programme et réserver des plages de mémoire de taille immuable pour les tableaux statiques [3]. En pratique, cela n'est en général pas suffisant car les besoins évoluent selon l'utilisateur et la taille des données à traiter.

Les pointeurs permettent de réserver et de libérer de la mémoire de manière très flexible avec les mots-clefs respectifs `new` et `delete` .

Soit `p` un pointeur de type `TYPE *` . La commande `p=new TYPE;` demande au système d'exploitation de réserver en mémoire un emplacement de la taille nécessaire pour stocker un objet de type `TYPE` , récupère l'adresse de cet emplacement et la stocke dans la variable `p` . La valeur stockée à cet emplacement est alors accessible uniquement par `*p` .

La commande `delete p;` ordonne au système d'exploitation de libérer l'emplacement réservé précédemment. La variable `p` existe toujours bien évidemment mais ne pointe plus vers une zone accessible (la commande `*p` donne alors un résultat indéterminé et le plus souvent une erreur de segmentation).

Il existe également une version étendue de `new` et `delete` pour réserver des plages mémoires plus grandes. La syntaxe `p=new TYPE[N];` où `N` est un entier positif demande au système d'exploitation de réserver dans la mémoire une zone *contigüe* de mémoire pour stocker consécutivement `N` objets de type `TYPE` . L'accès au premier élément se fait là encore par `*p` et l'accès au k-ème par `*(p+k)` ou `p[k]` . La désallocation de la mémoire se fait alors `delete [] p;` .

Il faut faire attention à ne pas mélanger les versions simple et étendue. Si un emplacement étendue est réservé avec `new []` , il faut impérativement le libérer avec `delete []` sinon

3. Le modèle de classe `std::vector` , derrière lequel se cache un usage astucieux de `new []` et `delete []` , permettait de s'affranchir partiellement de cette contrainte avec la notion de capacité.

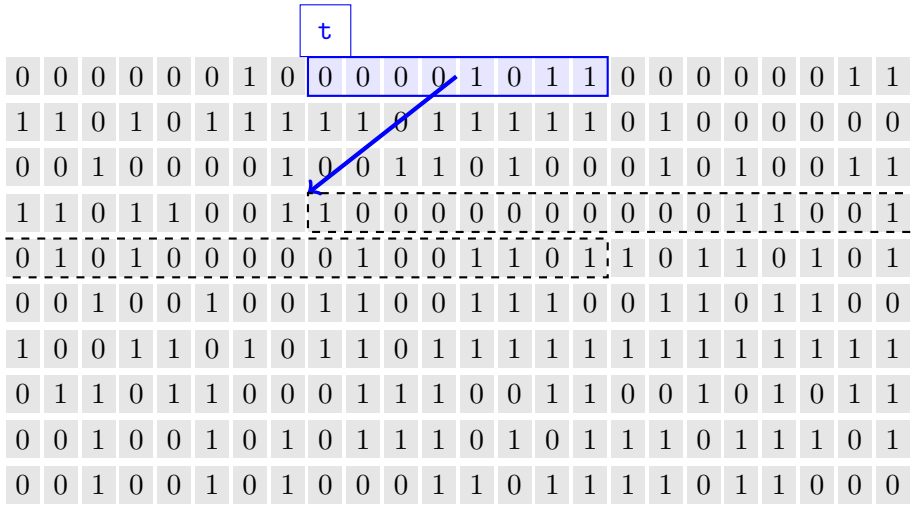

FIGURE 7.2 – Structure en mémoire d'un tableau dynamique déclaré par `short int *t = new short int [2];`. Le pointeur `t` a pour valeur l'adresse `0xB` (11 en hexadécimal, 1011 en base 2) de l'espace mémoire alloué à travers la commande `new` (ici délimité par des pointillés). L'adresse de `t` est en revanche `0x2` puisque la valeur de `t` est stockée sur le deuxième octet. L'espace alloué occupe 4 octets, i.e. deux fois la taille d'un `short int`.

le comportement du programme est *indéfini*.

Jusqu'ici, il peut sembler que rien ne distingue en apparence l'usage de `new` / `delete` sur des pointeurs de la déclaration d'une variable. Il existe une différence majeure néanmoins qui réside dans la possibilité de réutiliser un pointeur pour un usage différent dans un même programme alors qu'une déclaration de variable fige la nature de cette dernière.

Prenons l'exemple du programme suivant :

```
int main() {
        double * p;
// Premiere utilisation
        p=new double;
        *p=3.34;
        std::cout << *p << std::endl;
        delete  p;
//Seconde utilisation
        p=new double[57];
        for( int i=0; i<57; i++)        p[i]=0.5*i*i;
        delete [] p;

        return 0;
}
```

Dans ce code, une seule variable est déclarée : le pointeur `p`. Entre les lignes 4 et 7, il sert à stocker et afficher un seul nombre réel. La mémoire dont l'adresse est dans `p` est ensuite libérée et réattribuée pour stocker cette fois-ci 57 nombres réels entre les lignes 9 et 11.

Avec un seul pointeur et un usage astucieux de `new` et `delete`, nous avons pu manipuler la mémoire de manière très différente en utilisant une unique variable `p` tout au long du programme.

C'est cette liberté liée aux pointeurs qui est utilisée dans le modèle de classe `vector` (les changements de capacité se font à travers des usages de `new` et `delete`) ainsi que dans les exemples de la section 7.4. Les constructeurs et destructeurs du chapitre 5 seront également basés sur cette gestion de la mémoire par pointeur.

Valeur en cas d'échec. Les mots-clefs `new` et `delete` se révèlent également utiles lorsque d'énormes volumes de données sont en jeu. En effet, si la mémoire vive (RAM) de l'ordinateur est déjà très remplie ou tout simplement insuffisante, il se peut qu'une instruction du type `T * p=new T[N];` échoue si un objet de type `T` occupe beaucoup de place et que `N` est trop grand. Dans ce cas-là, l'espace mémoire nécessaire n'est pas réservé et, au prochain appel de type `p[i]`, une erreur de segmentation viendra interrompre subitement le programme. Ce n'est pas trop grave pour un programme de petite ampleur sur un ordinateur personnel ; c'est très grave pour un programme qui aurait tourné plusieurs semaines ou qui gère des activités critiques (cf. la ligne 14 du métro par exemple !). Il est alors absolument nécessaire de se prémunir de ce type d'erreur.

Pour cela, il existe la convention suivante : si l'allocation de mémoire échoue, alors `new` renvoie un pointeur de valeur `NULL`. Après une instruction du type `T * p=new T[N];`, il suffit de tester si `p !=NULL` pour décider de la continuation du programme si le résultat est `true`) ou d'une procédure d'arrêt (sauvegarde des données importantes, solutions alternatives, etc) si le résultat est `false`.

Le nouveau standard C++ 2011 introduit également le mot-clef `nullptr` qui vise à se substituer à `NULL`. Les différences entre les deux sont néanmoins au-delà de la perspective de cours.

7.3 Gestion de la mémoire dans les classes

7.3.1 Méthodes supplémentaires à définir

En général, une bonne partie de la gestion de la mémoire est effectuée au niveau des constructeurs : c'est lors de la déclaration d'objets qu'on réserve la mémoire nécessaire. Pour une gestion dynamique de la mémoire, si certains champs sont des pointeurs, les constructeurs contiennent toujours des `new` pour allouer la mémoire et stocker l'adresse dans les champs pointeurs.

En particulier, il faut ajouter un *constructeur par copie*. En effet, sans lui, seul l'adresse du pointeur serait copiée et deux objets de la classe pointeraient en interne vers un même tableau : l'effacement de l'un d'entre eux rendrait alors l'autre inutilisable. Ce comportement est parfois souhaitable mais c'est rarement le cas : on souhaite alors créer un nouveau tableau dans le deuxième objet et y copier les valeurs des cases du premier.

Lorsqu'un objet d'une classe est supprimé de la mémoire, les *destructeurs* correspondants aux types des champs sont appliqués sur chaque champ mais rien n'est prévu pour de la mémoire allouée par `new`. Le code du destructeur doit donc contenir un `delete` (ou `delete []` pour chaque pointeur soumis à un `new` dans un constructeur.

Nous nous retrouvons donc dans une situation opposée à celle du chapitre 5 et nous allons devoir écrire nous-même les constructeurs par copie, destructeurs, etc et respecter la *règle des trois* (avant C++11) et la *règle des cinq* (après C++11).

```
class Matrice {
private:
        unsigned int n;
        double * coeffs;
public:
        Matrice(unsigned N=1, double x=0.); // constructeur 1
        Matrice(std::vector<double> v); // constructeur 2
// Règle des trois:
        Matrice(const Matrice &); //cteur par copie nécessaire
        ~Matrice() { delete [] coeffs;} //destructeur  nécessaire
        Matrice & operator= (const Matrice & U);//nécessaire ici
//        Règle des cinq:
        Matrice(Matrice &&);//constructeur par déplacement
        Matrice & operator= (Matrice &&);//affectation par déplacement
// les méthodes:
        double & operator() (unsigned int,unsigned int);
        double taille() const { return n;}
        double trace() const;

// les opérateurs externes:
        friend Matrice operator+(const Matrice &, const Matrice &);
        friend Matrice operator*(const Matrice &, const Matrice &);
        friend std::ostream & operator<<(std::ostream &,const Matrice &);
};
```

FIGURE 7.3 – Définition de la classe `Matrice` avec les champs privés et l'ensemble des méthodes disponibles. Les méthodes marquées d'une étoile sont inutiles ici car la mémoire est directement gérée par la classe `std::vector` : elles deviendraient nécessaire avec l'usage de pointeurs tels que définis dans le chapitre 7.

Remarque. Il arrive parfois qu'on souhaite ajouter constructeur par copie et destructeur même lorsque la mémoire est gérée automatiquement par une autre bibliothèque. Cela arrive en présence de variables globables de classes ou de programme qu'on souhaite modifier à chaque création ou destruction d'un objet de la classe. Nous ne nous en préoccuperons pas dans ce cours.

Un exemple avec des matrices. Reprenons l'exemple des figures 5.1 et 5.2. Le nouveau code est à présent décrit dans les figures 7.3.

Le constructeur par défaut est bien présent ligne 6 et construit une matrice nulle de taille 1. Son code est donné par :

```
Matrice::Matrice(unsigned N, double x): n(N), coeffs(nullptr) {
        //Allocation puis initialisation
        coeffs= new double[N*N];
        for(unsigned int i=0; i<N*N; i++) coeffs[i]=x;
```

```
}
```

Il contient un `new` qui alloue la mémoire suffisante pour n^2 nombres réels. Les autres constructeurs doivent donc en faire de même et le destructeur doit effacer cette mémoire. Son code est donc donné ligne 9 de la figure 7.3.

Le constructeur par copie doit dupliquer tout le contenu d'une matrice préexistante dans l'espace alloué pour une nouvelle matrice. Son code est donc donné par :

```
Matrice::Matrice(const Matrice & A) {
2       // bien entendu, le this-> est ici optionnel.
        this->n=A.n;
4       this->coeffs=new double[n*n]; //allocation de la mémoire
        for(unsigned int i=0; i<n*n;i++) this->coeffs[i]=A.coeffs[i];
6               //on copie toutes les données contenues dans A
}
```

7.3.2 Destructeur

Étant donné que de la mémoire a été allouée dynamiquement dans les constructeurs, il faut bien libérer la mémoire une fois l'objet arrivé en fin de vie. Cela se fait par le destructeur (voir ligne 10 de la figure 7.3) qui contient les `delete` réciproques des `new` des constructeurs.

7.3.3 Opérateur d'affectation par copie

L'opérateur d'affectation permet de copier un objet déjà existant `a` dans un autre objet déjà existant `b` par la syntaxe `b=a;` . Il ressemble donc partiellement au constructeur par copie mais ce n'est pas un constructeur car les deux objets existent déjà. Il faut d'abord nettoyer l'ancien contenu de `b` (avec les `delete` nécessaires) puis créer la place nécessaire (avec des `new`) pour copier le contenu de `a` . Mais ce « nettoyage » peut être désastreux en cas de collision d'adresses mémoire du type `a=a;` . En effet, sans précaution supplémentaire, le contenu de `a` est effacé (pour nettoyer l'objet de gauche) or c'était le contenu qu'on voulait copier ; il aurait mieux valu ne rien faire.

Il semble évidemment stupide d'écrire `a=a;` mais cet usage est parfois caché dans des situations difficiles à examiner. Prenons l'exemple suivant :

```
A a;
2 A & r=a;
...//beaucoup de lignes de code
4 r=a;
```

Quand on lit la dernière ligne, on a l'impression que `r` et `a` sont des objets différents alors que `r` est construit comme référence à `a` . Il aurait fallu lire l'intégralité du code pour se rendre compte de la collision, ce qui souvent fastidieux en pratique.

Une solution à ce problème consiste à systématiquement tester si les objets à gauche et à droit du signe `=` désignent ou non les mêmes données en mémoire. Le code du l'opérateur `=` sera donc **toujours** du style :

```
A &  A::operator=(const A & c) {
        if( this == &c) return *this; // NE JAMAIS L'OUBLIER
        ... // effaçage de this avec un code similaire au destructeur
        ... /* initialisation des champs de this
                avec un code similaire au constructeur par copie */
        return *this;
}
```

Le type renvoyé par `=` permet des instructions du style `a1=a2=a3=c;` où un objet `c` est copié dans toutes les variables `a1`, `a2` et `a3`.

Exemple. Dans l'exemple des figures 5.1 et 5.2, le code de l'opérateur d'affectation est tout simplement :

```
Matrice & Matrice::operator=(const Matrice & B) {
        if( this == &B) return *this; // NE JAMAIS L'OUBLIER
        if( B.n != n) {
                delete [] coeffs;
                n=B.n;
                coeffs=new double[n*n];
        }
        for(unsigned int i=0; i<n*n ; i++) coeffs[i]=B.coeffs[i];
        return *this;
}
```

En général, le code de l'opérateur d'affectation ressemble à une combinaison – avec souvent des conditions supplémentaires – du code du destructeur et du code du constructeur par copie.

7.3.4 Opérateurs par déplacement

Tout ce qui est décrit dans cette section a été introduit dans le standard C++11 : il faut donc compiler les programmes avec l'option `-std=c++11`.

Les lignes 13 et 14 correspondent à des opérateurs *par déplacement*. Le type de l'argument est une *rvalue reference* : nous n'entrerons pas dans le détail de son sens et de son emploi et nous nous contenterons d'exhiber deux situations où cela peut être utile.

Commençons par le code suivant :

```
Matrice A;
        // des calculs sur A...
{ // début d'un bloc
        Matrice B(N);//avec N grand
        //calculs sur B;
        A = B;
}
```

Le bloc peut s'interpréter comme un bloc conditionnel d'un `if`, un appel de fonction, la fin d'une boucle, etc : ce qui est important est que la matrice `B` est détruite à la fin.

La dernière affectation sauve le résultat de `B` dans `A`. Sans opérateur d'affectation par déplacement, l'opérateur d'affectation par copie défini précédemment est appelé : l'objet `A` est préparé à la bonne taille puis les données de `B` sont copiées dans `A`. C'est dommage car les données de `B` sont effacées ensuite et on préférerait donc les récupérer directement.

C'est ce que peut faire l'opérateur d'affectation par déplacement défini par un code comme le suivant :

```
Matrice & Matrice::operator(Matrice && U) {
        if( this != & U) {
                delete [] coeffs;
                n = U.n;
                coeffs = U.coeffs;
                        //récupération de l'adresse allouée dans U
                U.n = 0;
                U.coeffs = nullptr;
        }
        return *this;
}
```

On voit ici qu'aucune copie n'a eu lieu : on a simplement « volé » la mémoire allouée dans `U`, qui était destinée de toute façon à être effacée peu après. Le `delete` efface la mémoire allouée lors de la création de l'objet courant, le `delete` effacera alors la mémoire allouée lors de la création de `U` ; le destructeur de `U` n'aura, quant à lui, plus rien à faire.

Le constructeur par déplacement fonctionne exactement de la même manière et est encore plus simple :

```
Matrice::Matrice(Matrice && U): n(U.n), coeffs(U.coeffs) {
        U.n=0;
        U.coeffs=nullptr;
}
```

Ces opérateurs par déplacement permettent ainsi de recycler des zones mémoires de toute façon effacée et sont très importants lorsque les objets sont gros.

7.3.5 Quelques règles de bonnes conduites

Manipuler la mémoire soi-même **n'est jamais anodin** et, grâce à la bibliothèque standard STL, est rarement nécessaire. De plus, comme les risques d'écrire un programme erroné avec des erreurs de segmentation difficiles à débugguer et des fuites de mémoire est relativement grand, il vaut mieux toujours se réfugier dans la STL et chercher si, dans celle-ci, il existe un conteneur qui fait ce que l'on souhaite.

En particulier, l'exemple précédent des matrices montrer que le code du chapitre 5 est nettement plus simple à manipuler que celui de la figure 7.3. Plus généralement, chaque fois que l'on a un couple fait d'un entier et d'un pointeur vers un tableau dont l'entier est la taille, on peut utiliser `std::vector`.

Bien évidemment, il y a des situations où ce n'est pas possible, comme, par exemple, la création de nouveaux conteneurs avec de nouveaux algorithmes. Nous en verrons un exemple au dernier TP. Il faut faire très attention à tous les mouvements de mémoire (allocation, accès, désallocation, déplacements, etc.).

7.4 Les structures de données usuelles

7.4.1 La problématique de la complexité

Jusque là, dès que nous avons voulu mettre plusieurs objets du même type dans une seule variable, nous avons utilisé des tableaux statiques ou dynamiques ou des vecteurs de la STL. Ce sont des structures très pratiques lorsque l'on veut accéder immédiatement à n'importe quelle case grâce à l'opérateur `[]` : cette rapidité est basée sur le fait que toutes les cases sont adjacentes dans la mémoire et l'adresse du k-ème terme se déduit immédiatement de l'adresse du 0-ème par translation. Comme nous l'avons déjà écrit, `t[i]` correspond au calcul d'adresse `*(t+i)` . Imaginons maintenant que nous souhaitions insérer un objet supplémentaire juste avant un élément donné : il faut pour cela ajouter une case à la fin du tableau puis décaler tous les éléments à partir de l'élément visé d'une place et enfin mettre l'objet supplémentaire à la place libérée. Insérer un seul élément demande de bouger une grande partie du tableau, c'est donc assez lent.

Reprenons le vocabulaire de la section 1.4. Considérons une suite de N objets avec N grand. En choisissant un tableau, accéder à un élément quelconque prend toujours le même nombre fini d'opérations par `*(t+i)` : la complexité de l'accès est donc $O(1)$. Insérer ou supprimer un élément demande de bouger en moyenne la moitié du tableau : la complexité est donc en $O(N)$. Cela signifie que le doublement de la taille du tableau ne modifie pas le temps d'accès mais multiplie en moyenne par deux le temps d'insertion/suppression d'un élément.

Une première remarque fondamentale est qu'il n'est pas possible de trouver une structure dans laquelle les deux opérations (accès et insertion) soient rapides. Une deuxième remarque est qu'en revanche il existe deux structures aux efficacités inverses : les tableaux et les listes. Le choix entre les deux lors de la programmation demande d'estimer si ce sont les accès ou les insertions/suppressions qui seront les plus utilisés.

7.4.2 Listes et itérateurs

Une liste est une suite d'éléments qui ne sont pas nécessairement contigus en mémoire mais dans laquelle chaque élément contient les adresses du précédent et du suivant. Il suffit d'alors de connaître l'adresse du premier pour pouvoir accéder à tous les autres « en suivant les liens ».

C'est une *alternative* à l'usage de `std::vector()` : beaucoup de fonctions sont identiques mais ont une complexité différente, il y a également certaines fonctions supplémentaires et d'autres sont manquantes. Le choix réside essentiellement sur une réflexion sur la complexité des algorithmes que l'on souhaite implémenter.

Nous présentons ici le cas des listes doublement chaînées sans l'utilisation de classes, introduites au chapitre suivant, afin de bien comprendre l'usage des pointeurs. Bien évidemment, l'usage de classes permet d'avoir une syntaxe plus légère : nous vous conseillons chez vous, après avoir lu et compris le chapitre sur les classes de transformer cette section en classe, de migrer les fonctions vers des méthodes, en précisant bien les `const` nécessaires.

Chaque élément d'une liste d'objets de type `TYPE` est appelé *maillon* et peut être décrit par :

```
typedef struct maillon{
        TYPE valeur;
        struct maillon * precedent;
```

```
        struct maillon * suivant;
} Maillon;
```

Le champ `precedent` (resp. `suivant`) contient l'adresse du maillon précédent (resp. suivant) dans la liste. Pour le premier (resp. dernier) maillon, la valeur de `precedent` (resp. `suivant`) est fixée par convention à `NULL`. Une liste est alors une structure :

```
typedef struct {
        unsigned int taille;
        Maillon * premier;
        Maillon * dernier;
} Liste;
```

qui indique les premiers et derniers maillons, et seulement ces deux.

Imaginons qu'on veuille ajouter une valeur `x` de type `TYPE` dans une liste `L` dans un nouveau maillon juste avant un maillon d'adresse `a`. La fonction `ajout` peut par exemple s'écrire :

```
bool ajout(Liste & L, Maillon * a, TYPE x) {
        struct Maillon * p=new Maillon;
        if( p== NULL) return false; //echec de la creation: rien ne change.
        p->valeur=x;
        L.taille++;//ne pas oublier de tout mettre à jour !
// Cas où le maillon est le premier de la liste
        if( L.premier == a )
        {
                p->precedent=NULL;
                p->suivant= L.premier;
                L.premier = p;
                a->precedent=p;
                return true;
        }
// Cas où le maillon est le dernier de la liste
        if( L.dernier == a)
        {
                a->suivant=p;
                p->precedent=a;
                p->suivant=NULL;
                L.dernier=p;
                return true;
        }
// Autres cas
        p->suivant=a;
        p->precedent=a->precedent;
        a->precedent->suivant=p;
        a->precedent=p;
        return true;
}
```

Ce code montre ainsi que l'ajout d'un élément ne nécessite de ne bouger aucun autre élément et que le nombre d'opérations ne dépend pas de la taille de la liste. La complexité d'un ajout est donc en $O(1)$ en la longueur de la liste.

Supposons à présent que nous souhaitions accéder au i-ème terme de la liste. La répartition des objets dans la mémoire étant quelconque, il est hors de question de calculer l'adresse du i-ème terme comme dans les tableaux. Il reste donc à parcourir toute la liste pour la trouver. La fonction d'accès s'écrit donc :

```
Maillon * acces(const struct Liste & L, unsigned int i) {
        if( i >= L.taille) return NULL; // element inexistant
        Maillon * p=L.premier;
        while( i>0) {
                p=p->suivant;
                i--;
        }
        return p;
}
```

Cette fois-ci, la boucle `while` parcourt la liste jusqu'au bon endroit. La complexité moyenne est donc en $O(N)$.

Nous avons ainsi atteint notre but en construisant une structure qui permet l'accès en $O(N)$ et l'insertion en $O(1)$. On vérifie également qu'il est facile d'ajouter et supprimer des éléments en début ou fin de liste puisque les adresses du premier et du dernier élément sont écrites dans la structure de liste.

Le concept d'itérateur. Il existe des opérations qui prennent autant de temps dans un tableau que dans une liste et pour lesquelles les algorithmes se retrouvent être les mêmes, par exemple, calculer la somme des éléments d'une liste ou d'un tableau d'objets de type `double` . Écrivons les trois codes aussi efficaces les uns que les autres pour les trois structures étudiées jusque là :

```
double somme( const Liste & L) {
        double s=0.;
        Maillon * p= L.premier;
        while( p != NULL) {
                s +=  (*p).valeur;
                p = p-> suivant;
        }
        return s;
}
```

```
double somme(const double * t, unsigned int taille) {
        double s=0.;
        unsigned k=0;
        while( k < taille) {
                s += t[k];
```

```
6                 k++;
        }
8       return s;
}
```

```
double somme(const  double * t, unsigned int taille) {
2       double s=0.;
        double * p= t;
4       while( p != t+taille) {
                s += *p ;
6               p++;
        }
8       return s;
}
```

Le premier code est le seul moyen possible de parcourir une liste : on entre par le premier terme et on s'arrête quand on sort du dernier. Le deuxième, pour les tableaux, utilise la possibilité de trouver un élément immédiatement mais le tableau est parcouru dans le même ordre que pour une liste. L'algorithme peut être réécrit et donne alors la troisième version qui est très similaire à celle de la liste *mutatis mutandis* et utilise les mêmes ingrédients :

- ligne 3 : initialisation sur le premier élément
- ligne 4 : détection de la sortie de la structure
- ligne 5 : accès à la valeur
- ligne 6 passage à l'élément suivant

Cette similarité n'est pas un hasard et est à la base de la notion d'*itérateur*.

Un *itérateur* joue le rôle d'un pointeur vers un élément d'une structure et doit être pourvu des opérations suivantes :

- accès à la valeur par l'opérateur `*`
- passage à l'élément suivant par `++` ;
- récupération à partir d'une liste d'un itérateur vers le premier élément et d'une condition de sortie de la structure.

Le modèle de classe `std::list` de la STL. La STL contient déjà des listes doublement chaînées avec des fonctionnalités bien plus nombreuses que notre tentative précédente et nous vous encourageons à utiliser directement cette nouvelle solution. En utilisant l'en-tête

```
#include <list>
```

on a accès au modèle de classe `std::list<TYPE>` pour gérer une liste d'objets de type `TYPE` et d'itérateurs `std::list<TYPE>::iterator` vers des éléments de liste. Les opérations naturelles sur ces objets sont résumées dans le tableau de la figure 7.4 pour des objets déclaré par `std::list<TYPE> L;` , `std::list<TYPE>::iterator i;` et `TYPE x;` . Les fonctionnalités sont similaires aux fonctions précédentes.

Ainsi, si nous souhaitons construire une liste avec les carrés des 100 premiers nombres entiers puis afficher cette liste dans l'ordre inverse, nous pouvons écrire :

Méthodes	Effet	Complexité
`L.push_front(x);`	ajoute un élément en tête de liste	$O(1)$
`L.push_back(x);`	ajoute un élément en fin de liste	$O(1)$
`L.pop_front();`	enlève un élément en tête de liste	$O(1)$
`L.pop_back();`	enlève un élément en fin de liste	$O(1)$
`it=L.begin();`	renvoie un itérateur vers le premier élément	$O(1)$
`it=L.end();`	renvoie un itérateur vers l'élément fantôme après le dernier	$O(1)$
`++it` / `--it`	déplace l'itérateur vers l'élément suivant/précédent	$O(1)$
`x=*it;`	renvoie la valeur pointée par l'itérateur	$O(1)$
`L.insert(it,x);`	ajoute `x` dans la liste à l'endroit pointé par l'itérateur	$O(1)$
	Pas d'accès direct au i-ème élément	virtuellement $O(N)$

FIGURE 7.4 – Méthodes du modèle de classe `std::list<TYPE>` décrites par un exemple et leurs effets. La complexité est également précisée. La variable `x` est de type `TYPE`. La variable `it` désigne ici un itérateur de type `std::list<TYPE>::iterator`.

```
#include <list>
#include <iostream>
using namespace std;
int main() {
        list<int> L;
        for(int i=0 ; i<100; i++) L.push_front(i*i);
        list<int>::iterator it;
        for( it=L.begin(); it != L.end() ; it++) cout << *it << endl;
        return 0;
}
```

7.4.3 Listes simplement chaînées, piles, arbres, « skip lists », arbres

Il existe de nombreuses autres structures de données qui utilisent activement le système des pointeurs. L'exemple précédent est l'une des listes les plus complètes en termes de fonctionnalités : en restreignant les mouvements possibles, on gagne de la place en mémoire mais on perd en fonctionnalités. Étudions à présent plusieurs exemples.

Listes simplement chaînées et piles. Reprenons notre définition de `Maillon` dans la section précédente. Il est possible de supprimer le champ `precedent` dans `Maillon` : cela permet de gagner de la place en mémoire (un pointeur par maillon) mais il n'est alors plus possible d'utiliser `--` sur un itérateur et tout retour sur un maillon précédent demande de recommencer à partir du premier. Le gain de place en mémoire ou en temps a un prix à

payer en termes de fonctionnalités. La STL fournit la plupart de ces structures allégées sous les noms de `stack` (pile), `queue`, etc.

Toutes ces structures ainsi que les tableaux gardent en commun le fait que les éléments sont ordonnés du premier au dernier. Cet ordre est souvent pertinent (par exemple, quand il s'agit de coordonnées de vecteurs ou d'ordre d'arrivée d'objets) mais il ne l'est pas toujours.

L'exemple des listes sera traité en détail en cours.

Structures non ordonnées : ensembles, « skip-lists » et « arbres B ». Lorsque l'on préfère utiliser des ensembles plutôt que des n-uplets, i.e. lorsque l'ordre des éléments n'importent pas dans le problème considéré, il existe d'autres structures plus efficaces. Des exemples de telles situations sont des collections de fichiers, des ensembles de cas à traiter, des familles de points sur lesquelles on souhaite faire des statistiques, etc.

Dans ce cas, il est utile d'introduire un ordre, naturel ou non, partiel ou total, sur les objets. Pour les entiers ou les réels on peut se baser sur l'ordre naturel ; pour les chaînes de caractères on peut se baser sur l'ordre lexicographique. Ce qui est important, c'est de pouvoir comparer rapidement les objets (quelle que soit leur taille). On trouve alors un objet, que ce soit pour y accéder, pour le supprimer ou insérer un objet avant ou après lui, rapidement par dichotomie avec une complexité moyenne en général de l'ordre de $O(\log N)$ où N est la taille de la collection d'objets.

L'exemple des « skip-lists » sera traité en détail dans le cadre d'un TP.

7.4.4 Les itérateurs des conteneurs de la STL

Les différents conteneurs ordonnés de la STL comme `std::list`, `std::vector`, `std::array`, `std::stack`, etc possèdent tous des itérateurs qui sont présents sous deux types différents :

```
std::CONTENEUR<TYPE>::iterator It;
std::CONTENEUR<TYPE>::const_iterator Itc;
```

Dans tous les cas, ces deux types d'itérateurs se comportent comme des pointeurs `TYPE *` ou `const TYPE *` vers les contenus des cases des structures. En particulier, ils sont tous munis d'un opérateur `++It` qui permet de passer à une case suivante ; certains d'entre eux sont aussi munis de `--` qui permet de remonter dans la structure. Ils sont tous également munis de l'opérateur `*It` et `*Itc` qui permet d'accéder au contenu des cases. En revanche, il n'y a pas d'opérateur `&`.

De plus, chaque conteneur possède des méthodes `begin()` et `end()` : la première renvoie un itérateur sur le premier élément et la seconde renvoie un itérateur sur un élément « après le dernier ». Ce dernier permet de détecter l'arrivée à la fin d'une structure après usage de `++`.

La plupart des exemples de ce cours ainsi que les exemples vus en TP ont traité d'objets de type `std::vector` pour lesquels on accède aux case par les crochets `[]`. Malheureusement, c'est impossible pour la plupart des autres conteneurs et le seul moyen de parcourir ces conteneurs dans l'ordre est de manipuler des itérateurs.

7.5 [Hors programme] Une rapide revue des pointeurs intelligents en C++11

Tout ce qui est décrit dans cette section a été introduit dans le standard C++11 : il faut donc compiler les programmes avec l'option `-std=c++11`.

Le standard C++11 offre de nouvelles possibilités de gestion de la mémoire pour éviter de nombreux pièges vus précédemment dans les cas les plus simples. Cela repose sur la notion de « pointeurs intelligents » (*smart pointers* en anglais). Les utiliser requiert l'inclusion suivante en en-tête :

```
#include <memory>
```

Il en existe de plusieurs types et choisir le bon demande une réflexion supplémentaire sur les programmes que l'on écrit.

Documentation. De nombreuses informations sont récapitulées dans les réferences suivantes :

http://www.cplusplus.com/reference/memory/
et
https://en.cppreference.com/w/cpp/memory

Les pointeurs à usage unique. C'est le premier type de pointeur intelligent qui permet de définir des pointeurs dont on veut restreindre l'accès au contenu au seul environnement qui a créé le pointeur (typiquement une classe).

L'intéret est alors qu'il n'est plus nécessaire d'écrire les `delete` correspondants : le pointeur et son contenu pointé sont détruits lorsque l'environnement qui les a créé est lui-même détruit.

En revanche, par définition, ce pointeur *ne* peut *pas* être copié : c'est ce mécanisme qui fait qu'aucun objet autre que son créateur ne connaît l'adresse stockée.

La syntaxe est la suivante :

```
std::unique_ptr<TYPE> p; // construit le pointeur nul
std::unique_ptr<TYPE> p(new TYPE); // alloue la mémoire
        // pour un objet de type TYPE par défaut et écrit l'adresse
        // dans un pointeur unique pour la protéger.
```

Ensuite, cela s'utilise comme d'habitude avec les syntaxes `*p` et `p->...` pour accéder au contenu pointé. En revanche, il n'y a plus de `delete` à écrire et tout le contenu pointé est détruit lorsque `p` arrive en fin de vie.

Un exemple typique d'utilisation est la classe `Matrice` revue dans ce chapitre en figure 7.3 : le pointeur `coeff` ne doit être utiliser que par la matrice qui le possède car toute copie de matrice passe la création d'un *autre* tableau pointé et la copie du contenu. Remplacer la ligne 4 par :

```
std::unique_ptr<double> coeffs;
```

aurait les conséquences suivantes :

— le destructeur devient inutile (plus d'erreur d'étourderie !)
— les constructeurs 1 et 2 doivent utiliser la syntaxe de construction de `std::unique_ptr` précédente (presque rien...)
— l'oubli du constructeur par copie donne une erreur de compilation car le champ `coeff` n'est plus copiable et cela oblige ainsi à se poser la question de la copie (plus d'étourderie possible !)
— l'oubli de la redéfinition de `=` donne une erreur de compilation pour la même raison et oblige à se poser la question de la copie.

Les pointeurs partagés. Il est des cas où l'on souhaite l'inverse : avoir un pointeur vers un objet unique partagé par plusieurs objets. Imaginons par exemple le cas d'un grand tableau, d'une grande image, d'un long texte auxquels se référeraient plusieurs objets. Dans ce cas, on veut pouvoir copier l'adresse de ce gros objet et que ce dernier ne soit effacé qu'après que le dernier objet qui l'utilise disparaisse. Dans ce cas-là, la bonne notion est celle de *pointeur partagé*.

La déclaration d'un tel pointeur se fait par :

```
std::shared_ptr<TYPE> sp; //pointeur nul
std::shared_ptr<TYPE> sp(new TYPE);
auto sp2= std::make_shared<TYPE>(arg1,...,argn);// où arg1,...,argn
        // sont les arguments du constructeur souhaité de TYPE.
```

qui fonctionne comme précédemment. La structure interne d'un tel objet est constituée d'un pointeur nu `TYPE *` ainsi que d'un compteur qui mémorise le nombre de fois que ce pointeur est copié dans la mémoire. Il s'utilise ensuite comme un pointeur normal avec `*sp` et `sp->...` . Là encore, il n'y a plus de `delete` à écrire : la mémoire est automatiquement libérée lorsque la variable `sp` est effacée et que plus aucun objet n'utilise le pointeur (d'où le compteur interne associé...).

Nous n'entrerons pas dans les détails mais, associé à ce type, il existe la notion de `std::weak_ptr<TYPE>` qui permettent, sans rien allouer ni effacer, de pointer vers l'élément de mémoire pointé par un `std::shared_ptr<T>` (en incrémentant d'une unité le compteur de ce dernier).

Exercices

Exercice 7.1. (navigation par pointeur) Soit le programme

```
#include <iostream>
int main() {
        double * t = new double[20];
        for(double * p=t+4; p<t+16; p++) {
                *p = 1.23456;
        }
        -----
        delete [] t;
}
```

Que fait-le programme suivant ? Compléter le programme pour afficher toutes les cases du tableau pour vérifier votre intuition.

Exercice 7.2. Si vous ne faites pas ce qu'il faut à la fin de la fonction `f()` , ce code va paralyser votre machine ou bien planter après peu de temps. Soyez vigilants !

```
#include <iostream>
#include <string>
#include <complex>
double  f() {
        double *t= new double[10000000];
        double u=t[0];
        // agissez maintenant !
        return u;
}

int main() {
        double x;
        for (long long int i = 0; i < 10000000; i++) x=f();
        return 0;
}
```

Exercice 7.3. (le jeu de bonneteau des pointeurs) Voici un petit programme sympathique :

```
#include <iostream>
int main() {
        int t[6]= {0,5,4,2,1,3};
        int *p = t+5;
        p -= 1;
        p = t+ *p;
        p = p+1;
        std::cout << *p << std::endl;
```

```
        return 0;
}
```

Chapitre 8

Compléments

A REMPLIR !

Conclusion

Ces notes de cours ne constituent qu'une introduction à la programmation en `C++` pour les mathématiciens, avec une orientation vers l'aléatoire et les structures de données.

Autres aspects non abordés. Il est temps maintenant de faire un rapide panorama de tout ce que ce cours ne contient pas :

- la gestion des exceptions : lorsqu'un `new` échoue, qu'une division par zéro s'effectue, qu'un fichier devant être présent ne l'est pas, etc, il ne sert en général à rien de poursuivre l'exécution d'un programme et il vaut mieux prévenir l'utilisateur de la raison de l'arrêt soudain du programme. Pour cela, il faudrait en toute rigueur ajouter des tests pour vérifier qu'aucun de ces scénarii ne se produit et, s'il s'en produit un, prévoir une sortie rapide du programme : c'est tout l'objet du traitement des *exceptions*, que nous n'avons pas du tout abordé.
- la définition des espaces de noms : nous avons vu fréquemment apparaître des espaces de noms tels `std::` et `Eigen::` pour préciser l'appartenance de certaines classes et fonctions à des familles plus grandes. Nous avons montré comment définir des classes et, pour être complet, il faudrait également expliquer comment définir de nouveaux espaces de noms.
- les standards C++14, C++17 et C++20 (qui vient tout juste d'être rendu officiel !) : le `C++` est un langage vivant et le standard C++11 a été une étape importante dans son évolution. Nous en avons vu ici quelques aspects : les λ-fonctions, l'ajout de la bibliothèque `<random>`, le mot-clef `auto`, les pointeurs intelligents. Un autre apport longtemps attendu a été l'apparition des types `T &&`, appelées « rvalue reference » et l'opérateur `std::move()` associé qui permet d'éviter des copies inutiles lors de réaffectation ou de permutations de valeurs (avec `std::swap` par exemple). Si le temps le permet en cours, nous aborderons cette question lors des dernières séances. Les normes C++14 et C++17 apportent seulement quelques ajouts supplémentaires et essentiellement des améliorations de C++11.
  En revanche, le nouveau standard C++20 est une révolution au moins aussi grande que celle qu'a été le standard C++11 à sa sortie. Peut-être ce cours en parlera-t-il dans quelques années !
- nous avons insisté à plusieurs reprises sur l'erreur d'arrondi lié aux types de flottants et sur le dépassement de capacité des entiers, ainsi que sur la qualité approximative des générateurs de nombres aléatoires : néanmoins nous n'avons pas proposé de méthodes de contrôle et d'estimation de ces erreurs par manque de temps. Nous vous conseillons de vous référer à un cours d'analyse numérique ou de probabilités numériques pour aborder ce sujet.
- à travers les exemples vus en séances de cours et de TP, nous avons présenté quelques structures algorithmiques parmi les plus fréquentes. Il en existe beaucoup d'autres et nous vous conseillons de développer votre culture générale en vous référant à des

livres d'introduction à l'algorithmique.
— au-delà de l'aspect mathématique, l'aspect « programmation orientée objet » du `C++` permet aussi de développer aisément des interfaces graphiques (par exemple avec les outils de développement Qt). Les problématiques sont alors complètement différentes : en général on sacrifie l'efficacité numérique (la vitesse est le plus souvent limitée par les utilisateurs) au profit de la facilité de création de nouveaux outils à intégrer aux interfaces ainsi et au profit de la capacité d'évolution de ces dernières.
— nous n'avons manipulé que des jeux de données petits en TP car les machines de nos salles sont de faible puissance. En revanche, les fichiers de données réelles peuvent être gigantesques et de nombreux calculs, d'analyse comme de probabilité, peuvent être très longs. Les processeurs (ou les cœurs des processeurs) évoluent techniquement aujourd'hui moins vite qu'avant et la tendance est maintenant à la parallélisation des calculs. Cela a deux implications importantes en termes de programmation : d'une part il faut concevoir des programmes de manière moins « linéaire » (i.e. il ne faut plus percevoir le code comme une suite ordonnée d'instructions effectuées les unes à la suite des autres mais plutôt comme des blocs pouvant être lancés séparément). D'autre part, la gestion de la mémoire devient plus subtile car il peut y avoir (ou non) une mémoire propre à chaque processeur et il y a toujours une mémoire partagée : les processeurs travaillent en parallèle et il faut éviter à tout prix les collisions s'ils tentent d'accéder à un même morceau de mémoire.

Les *core guidelines*. Par ailleurs, au-delà de la simple syntaxe du langage, il existe des codes plus ou moins faciles à développer, à corriger, à mettre à jour, à compléter. Il existe également des codes plus ou moins efficaces numériquement. En TP, nous essayons de vous apprendre les bonnes manières ; mais, si vous voulez progresser bien au-delà, nous vous encourageons en fin de semestre à lire en détail les *conseils officiels* de B. Stroustrup et H. Sutter, disponibles sur :

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Autres outils. Nous vous conseillons également de jeter un coup d'oeil vers les outils `clang-format` et `clang-tidy` (paquets Linux faciles à installer). Ils sont associés au compilateur `clang` mais peuvent être utilisés indépendamment.

Le premier `clang-format` vous permet de bien présenter votre programme à travers

```
clang-format NOM.cpp --style=llvm > NOUVEAUNOM.cpp
```

Il revoie toute l'indentation du programme selon un certain style.

Le deuxième logiciel `clang-tidy` analyse votre programme pour détecter les erreurs, les manquements de style, les manques d'optimisation, la possibilité de moderniser votre code. La commande se présente sous la forme suivante :

```
clang-tidy NOM.cpp -checks="cppcoreguidelines-*,modernize-*,performance-*"
```

et affiche les remarques de la même manière que `g++` affiche les erreurs. Vous remarquerez peut-être que tous nos corrigés ne sont pas parfaits de ce point de vue là : c'est volontaire de notre part car certains points demanderaient d'aller bien plus loin dans ce cours pour être compris.

Vous pouvez également aller voir du côté des outils `clang-tools` pour toutes les options possibles (et les autres !).

Nous vous souhaitons une bonne continuation dans la découverte du `C++` moderne et de la programmation en général.

Appendice : Makefile

Nous avons vu en section 2.4 que les commandes de compilation avec `g++` peuvent être longues et nombreuses dès lors que des bibliothèques externes sont utilisées ou que le programme est découpé en de nombreux fichiers `.cpp`. Lors de la correction des erreurs dans un programme, il peut être fastidieux de relancer manuellement toutes ces commandes de compilation : tout ce processus peut être automatisé grâce à l'usage du programme `make` et la création d'un fichier `Makefile`. De plus, l'usage de `make` permet de varier les paramètres de compilation (changement de compilateur ou d'options de compilation) ainsi de nettoyer les fichiers intermédiaires produits par les compilations partielles.

La marche à suivre est de créer un fichier dont le nom est nécessairement `Makefile` dans le répertoire de travail. Ce fichier est structuré de la manière suivante :

```
1  CC= g++
2  CFLAGS= -Wall -pedantic -O3
3
4  prog.exe: fichier1.o fichier2.o principal.o
5          $(CC) -o prog.exe fichier1.o fichier2.o principal.o
6
7  fichier1.o: fichier1.hpp fichier1.cpp
8          $(CC) -c fichier1.cpp $(CFLAGS)
9
10 fichier2.o: fichier1.hpp fichier2.hpp fichier2.cpp
11         $(CC) -c fichier2.cpp $(CFLAGS) -I/usr/include/eigen3
12
13 principal.o: fichier1.hpp fichier2.hpp principal.cpp
14         $(CC) -c principal.cpp $(CFLAGS)
15
16 clean:
17         rm *.o
```

Les trois premières lignes définissent trois variables qui contiennent respectivement le nom du compilateur, les options de compilations, les liens vers les librairies pour l'assemblage des fichiers. L'accès à la valeur d'une variable `V` se fait ensuite par l'opérateur `$(V)`. Une variable peut être laissée vide comme cela a été fait dans la ligne 3 ci-dessus.

En général, on respecte les conventions suivantes : `CC` contient le nom du compilateur utilisé (par exemple `g++` ou `clang++`), `CFLAGS` contient les options de compilation (affichage des avertissements, options d'optimisation, de débuggage, etc). En cas d'utilisation de bibliothèques dynamiques, il est toujours possible d'ajouter d'autres variables contenant les directives du type `-I /usr/include/...`.

Toutes les commandes suivantes doivent suivre la syntaxe suivante :

```
cible: dependances
<TAB>        commande1
<TAB>        commande2
<TAB>        ...
```

La cible est par exemple le nom d'un fichier à bâtir. Les dépendances sont tous les fichiers dont dépend l'obtention de la cible. Par exemple, si c'est un fichier `.o`, toute modification dans les `.cpp` et `.hpp` correspondant ainsi que dans les tous les autres `.hpp` qui apparaissent dans leurs `#include "..."` induit une modification du `.o` correspondant. Les lignes suivantes doivent nécessairement commencer par une *tabulation* (la touche à gauche de la première rangée de lettres sur la majorité des claviers) puis contiennent la ligne de commandes nécessaires à l'exécution de la cible.

Une fois le `Makefile` constitué, on utilise dans le terminal le logiciel de la manière suivante :

— `make cible` fait le nécessaire pour atteindre la cible en calculant toutes les dépendances à satisfaire

— `make` fait le nécessaire pour atteindre la *première cible*.

En général, la première cible est le fichier exécutable final que l'on souhaite construire.

Lors d'un nouvel appel à `make cible` après modification de quelques fichiers, `make` cherche l'ensemble minimal de cibles à régénérer qui font intervenir les fichiers modifiés. Dans l'exemple ci-dessus, si on modifie seulement le fichier `fichier2.cpp` et qu'on appelle à nouveau `make`, seules les recompilations (dans l'ordre) de `fichier2.o` puis `prog.exe` sont effectuées et `fichier1.o` et `principal.o` restent inchangés. Si, au contraire c'est `fichier1.hpp` alors toutes les cibles sont à nouveau traitées par `make`.

Enfin, une cible peut ne pas être un fichier mais un mot quelconque. C'est utile pour nettoyer un répertoire après de multiples compilations : dans l'exemple ci-dessus, l'appel de `make clean` traite la dernière cible, qui ne dépend d'aucune autre, et efface tous les fichiers `.o` dont il n'y a plus besoin après construction de l'exécutable final.

Il existe de nombreuses autres fonctionnalités qui sont détaillées dans le manuel d'utilisation de `make`.

Bibliographie

[1] B. Stroustrup, *Le langage C++* (3ème édition), CampusPress, 1999. *(la référence par l'auteur du langage !)*

[2] le site http://www.cplusplus.com/reference/ avec une description complète de la bibliothèque standard agrémentée d'exemples faciles à comprendre.

[3] le site https://en.cppreference.com/w/ avec une description complète de la bibliothèque standard et des derniers standards.

[4] site web http://exercism.io/ avec de nombreux exercices d'entraînement.

[5] D. Ryan Stephens, C. Diggins, J. Turkanis, J. Cogswell, *C++ Cookbook : Solutions and Examples for C++ Programmers*, O'Reilly Media, 2005.

[6] E. Scheinerman, *C++ for Mathematicians. An Introduction for Students and Professionals*. CRC Press, 2006.

[7] D. Knuth, *The art of computer programming*, volumes 1 à 4, Addison Wesley, réédité en 2011. *(une excellente introduction à l'algorithmique)*