
LA PROGRAMMATION POUR...

☒ les élèves ingénieurs

☒ ... *ou les collégiens*

☒ débutants

☒ ... *ou confirmés*

Cours de l'École des Ponts ParisTech - 2017/2018
Renaud Keriven et Pascal Monasse
IMAGINE - École des Ponts ParisTech
monasse@imagine.enpc.fr

Version électronique et programmes :
<http://imagine.enpc.fr/~monasse/Info/>

*"Ne traitez pas vos ordinateurs comme des êtres vivants...
Ils n'aiment pas ça !"*

-
- *"Cet ordinateur ne fait pas du tout ce que je veux !"*
 - *"Exact... Il fait ce que tu lui demandes de faire !"*
-

Table des matières

1	Préambule	7
1.1	Pourquoi savoir programmer ?	10
1.2	Comment apprendre ?	10
1.2.1	Choix du langage	10
1.2.2	Choix de l'environnement	11
1.2.3	Principes et conseils	12
2	Bonjour, Monde !	15
2.1	L'ordinateur	17
2.1.1	Le micro-processeur	17
2.1.2	La mémoire	19
2.1.3	Autres Composants	21
2.2	Système d'exploitation	22
2.3	La Compilation	23
2.4	L'environnement de programmation	25
2.4.1	Noms de fichiers	25
2.4.2	Debuggeur	25
2.5	Le minimum indispensable	26
2.5.1	Pour comprendre le TP	26
2.5.2	Un peu plus...	27
2.5.3	Le debuggeur	28
2.5.4	TP	28
3	Premiers programmes	31
3.1	Tout dans le <code>main()</code> !	31
3.1.1	Variables	31
3.1.2	Tests	35
3.1.3	Boucles	37
3.1.4	Récréations	39
3.2	Fonctions	41
3.2.1	Retour	43
3.2.2	Paramètres	45
3.2.3	Passage par référence	46
3.2.4	Portée, Déclaration, Définition	48
3.2.5	Variables locales et globales	49
3.2.6	Surcharge	50
3.3	TP	51
3.4	Fiche de référence	52

4	Les tableaux	55
4.1	Premiers tableaux	55
4.2	Initialisation	57
4.3	Spécificités des tableaux	58
4.3.1	Tableaux et fonctions	58
4.3.2	Affectation	60
4.4	Récréations	61
4.4.1	Multi-balles	61
4.4.2	Avec des chocs !	63
4.4.3	Mélanger les lettres	66
4.5	TP	67
4.6	Fiche de référence	68
5	Les structures	71
5.1	Révisions	71
5.1.1	Erreurs classiques	71
5.1.2	Erreurs originales	72
5.1.3	Conseils	73
5.2	Les structures	73
5.2.1	Définition	73
5.2.2	Utilisation	74
5.3	Récréation : TP	76
5.4	Fiche de référence	76
6	Plusieurs fichiers !	79
6.1	Fichiers séparés	80
6.1.1	Principe	80
6.1.2	Avantages	81
6.1.3	Utilisation dans un autre projet	82
6.1.4	Fichiers d'en-têtes	82
6.1.5	A ne pas faire...	85
6.1.6	Implémentation	85
6.1.7	Inclusions mutuelles	85
6.1.8	Chemin d'inclusion	86
6.2	Opérateurs	87
6.3	Récréation : TP suite et fin	88
6.4	Fiche de référence	88
7	La mémoire	91
7.1	L'appel d'une fonction	91
7.1.1	Exemple	91
7.1.2	Pile des appels et débogueur	93
7.2	Variables Locales	95
7.2.1	Paramètres	95
7.2.2	La pile	96
7.3	Fonctions récursives	96
7.3.1	Pourquoi ça marche ?	96
7.3.2	Efficacité	97

7.4	Le tas	99
7.4.1	Limites	99
7.4.2	Tableaux de taille variable	99
7.4.3	Essai d'explication	101
7.5	L'optimiseur	101
7.6	Assertions	102
7.7	Examens sur machine	102
7.8	Fiche de référence	103
8	Allocation dynamique	105
8.1	Tableaux bidimensionnels	105
8.1.1	Principe	105
8.1.2	Limitations	106
8.1.3	Solution	107
8.2	Allocation dynamique	108
8.2.1	Pourquoi ça marche ?	108
8.2.2	Erreurs classiques	109
8.2.3	Conséquences	110
8.3	Structures et allocation dynamique	111
8.4	Boucles et <code>continue</code>	114
8.5	TP	115
8.6	Fiche de référence	115
9	Premiers objets	119
9.1	Philosophie	119
9.2	Exemple simple	120
9.3	Visibilité	122
9.4	Exemple des matrices	122
9.5	Cas des opérateurs	125
9.6	Interface	127
9.7	Protection	128
9.7.1	Principe	128
9.7.2	Structures vs Classes	129
9.7.3	Accesseurs	129
9.8	TP	130
9.9	Fiche de référence	131
10	Constructeurs et Destructeurs	135
10.1	Le problème	135
10.2	La solution	136
10.3	Cas général	136
10.3.1	Constructeur vide	136
10.3.2	Plusieurs constructeurs	138
10.3.3	Tableaux d'objets	139
10.4	Objets temporaires	140
10.5	TP	141
10.6	Références Constantes	141
10.6.1	Principe	141

10.6.2 Méthodes constantes	142
10.7 Destructeur	144
10.8 Destructeurs et tableaux	146
10.9 Constructeur de copie	146
10.10 Affectation	147
10.11 Objets avec allocation dynamique	148
10.11.1 Construction et destruction	148
10.11.2 Problèmes !	149
10.11.3 Solution !	150
10.12 Fiche de référence	152
11 Chaînes de caractères, fichiers	157
11.1 Chaînes de caractères	157
11.2 Fichiers	159
11.2.1 Principe	159
11.2.2 Chaînes et fichiers	160
11.2.3 Objets et fichiers	161
11.3 Valeurs par défaut	162
11.3.1 Principe	162
11.3.2 Utilité	162
11.3.3 Erreurs fréquentes	163
11.4 Accesseurs	163
11.4.1 Référence comme type de retour	163
11.4.2 Utilisation	164
11.4.3 <code>operator()</code>	164
11.4.4 Surcharge et méthode constante	165
11.4.5 "inline"	166
11.5 Assertions	167
11.6 Types énumérés	168
11.7 Fiche de référence	168
12 Fonctions et classes paramétrées (templates)	173
12.1 <code>template</code>	173
12.1.1 Principe	173
12.1.2 <code>template</code> et fichiers	174
12.1.3 Classes	175
12.1.4 STL	178
12.2 Opérateurs binaires	180
12.3 Valeur conditionnelle	181
12.4 Boucles et <code>break</code>	182
12.5 Variables statiques	183
12.6 <code>const</code> et tableaux	184
12.7 Fiche de référence	185
A Travaux Pratiques	191
A.1 L'environnement de programmation	191
A.1.1 Bonjour, Monde !	191
A.1.2 Premières erreurs	193

A.1.3	Debugger	195
A.1.4	S'il reste du temps	196
A.1.5	Installer Imagine++ chez soi	197
A.2	Variables, boucles, conditions, fonctions	198
A.2.1	Premier programme avec fonctions	198
A.2.2	Premier programme graphique avec Imagine++	198
A.2.3	Jeu de Tennis	200
A.3	Tableaux	202
A.3.1	Mastermind Texte	202
A.3.2	Mastermind Graphique	204
A.4	Structures	207
A.4.1	Etapas	207
A.4.2	Aide	209
A.4.3	Théorie physique	210
A.5	Fichiers séparés	212
A.5.1	Fonctions outils	212
A.5.2	Vecteurs	212
A.5.3	Balle à part	213
A.5.4	Retour à la physique	213
A.6	Images	216
A.6.1	Allocation	216
A.6.2	Tableaux statiques	216
A.6.3	Tableaux dynamiques	217
A.6.4	Charger un fichier	217
A.6.5	Fonctions	217
A.6.6	Structure	218
A.6.7	Suite et fin	219
A.7	Premiers objets et dessins de fractales	220
A.7.1	Le triangle de Sierpinski	220
A.7.2	Une classe plutôt qu'une structure	221
A.7.3	Changer d'implémentation	221
A.7.4	Le flocon de neige	222
A.8	Tron	223
A.8.1	Serpent	223
A.8.2	Tron	224
A.8.3	Graphismes	224
B	Imagine++	227
B.1	Common	227
B.2	Graphics	228
B.3	Images	229
B.4	LinAlg	230
B.5	Installation	230
C	Compilateur, CMake, Linker, Qt... Au secours !	231
C.1	Compilation	231
C.1.1	Compilateur et linker	231
C.2	En ligne de commande	233

C.3	Make et CMake	234
C.4	Utilisation d'un IDE	235
D	Fiche de référence finale	237

Chapitre 1

Préambule

Note : Ce premier chapitre maladroït correspond à l'état d'esprit dans lequel ce cours a débuté en 2003, dans une période où l'Informatique avait mauvaise presse à l'École des Ponts. Nous le maintenons ici en tant que témoin de ce qu'il fallait faire alors pour amener les élèves à ne pas négliger l'Informatique. Si l'on ignore la naïveté de cette première rédaction (et le fait que Star Wars n'est plus autant à la mode!), l'analyse et les conseils qui suivent restent d'actualité.

—

(Ce premier chapitre tente surtout de motiver les élèves ingénieurs dans leur apprentissage de la programmation. Les enfants qui se trouveraient ici pour apprendre à programmer sont sûrement déjà motivés et peuvent sauter au chapitre suivant ! Profitons-en pour tenir des propos qui ne les concernent pas...)

—

- *Le Maître Programmeur*¹ : "Rassure-toi ! Les ordinateurs sont stupides ! Programmer est donc facile."
- *L'Apprenti Programmeur*² : "Maître, les ordinateurs ne sont certes que des machines et les dominer devrait être à ma portée. Et pourtant... Leur manque d'intelligence fait justement qu'il m'est pénible d'en faire ce que je veux. Programmer exige de la précision et la moindre erreur est sanctionnée par un message incompréhensible, un *bug*³ ou même un *crash* de la machine. Pourquoi doit-on être aussi... précis ?" *Programmer rend maniaque ! D'ailleurs, les informaticiens sont tous maniaques. Et je n'ai pas envie de devenir comme ça...*

1. Permettez ce terme ouvertement Lucasien. Il semble plus approprié que l'habituel *Gourou* souvent utilisé pour décrire l'expert informaticien. Nous parlons bien ici d'un savoir-faire à transmettre de *Maître* à *Apprenti* et non d'une secte...

2. Le jeune *Padawan*, donc, pour ceux qui connaissent...

3. Je n'aurai aucun remord dans ce polycopié à utiliser les termes habituels des informaticiens... en essayant évidemment de ne pas oublier de les expliquer au passage. Anglicismes souvent incompréhensibles, ils constituent en réalité un *argot* propre au métier d'informaticien, argot que doit bien évidemment accepter et faire sien l'*Apprenti* sous peine de ne rien comprendre au discours de ses collègues d'une part, et d'employer des adaptations françaises ridicules ou peu usitées d'autre part. Naviguer sur la *toile*, envoyer un *courriel* ou avoir un *bogue* commencent peut-être à devenir des expressions compréhensibles. Mais demandez-donc à votre voisin s'il reçoit beaucoup de *pourriels* (terme proposé pour traduire "Spams") !

- M.P. : "La précision est indispensable pour communiquer avec une machine. C'est à l'Homme de s'adapter. Tu dois faire un effort. En contre-partie tu deviendras son maître. Réjouis-toi. Bientôt, tu pourras créer ces êtres obéissants que sont les programmes."
- A.P. : "Bien, Maître..." *Quel vieux fou ! Pour un peu, il se prendrait pour Dieu. La vérité, c'est qu'il parle aux machines parce qu'il ne sait pas parler aux hommes. Il comble avec ses ordinateurs son manque de contact humain. L'informaticien type... Il ne lui manque plus que des grosses lunettes et les cheveux gras*⁴. "Maître, je ne suis pas sûr d'en avoir envie. Je n'y arriverai pas. Ne le prenez pas mal, mais je crois être davantage doué pour les Mathématiques ! Et puis, à quoi savoir programmer me servira-t-il ?"
- M.P. : "Les vrais problèmes qui se poseront à toi, tu ne pourras toujours les résoudre par les Mathématiques. Savoir programmer, tu devras !"
- A.P. : "J'essaierai..." *Je me demande s'il a vraiment raison ! Je suis sûr qu'il doit être nul en Maths. Voilà la vérité !*
- ...

—

Oublions là ce dialogue aussi caricatural que maladroit. Il montre pourtant clairement la situation. Résumons :

- Pour celui qui sait, programmer :
 - est un jeu d'enfant.
 - est indispensable.
 - est une activité créatrice et épanouissante.
- Pour celui qui apprend, programmer :
 - est difficile.
 - ne sert à rien.
 - est une activité ingrate qui favorise le renfermement⁵ sur soi-même.

Dans le cas où l'élève est ingénieur, nous pouvons compléter le tableau :

- Pour le professeur, apprendre à programmer :
 - devrait être simple et rapide pour un élève ingénieur.
 - est plus utile qu'apprendre davantage de Mathématiques.
- Pour l'élève, programmer :
 - est un travail de "technicien"⁶ qu'il n'aura jamais à faire lui-même.
 - n'est pas aussi noble que les Mathématiques, bref, n'est pas digne de lui.

En fait, les torts sont partagés :

- Le professeur :

4. Toute ressemblance avec des personnages réels ou imaginaires, etc.

5. Utiliser un ordinateur pour programmer a tout aussi mauvaise presse que de jouer aux jeux vidéo. Programmer est pourtant souvent un travail d'équipe.

6. avec tout le sens péjoratif que ce terme peut avoir pour lui.

- ne réalise pas que ses élèves ont un niveau avancé en maths parce qu'ils en font depuis plus de dix ans, et qu'il leur faudra du temps pour apprendre ne serait-ce que les bases de la programmation. Du temps... et de la pratique, car, si programmer est effectivement simple en regard de ce que ses élèves savent faire en maths, il nécessite une tournure d'esprit complètement différente et beaucoup de travail personnel devant la machine.
- oublie qu'il a le plus souvent appris seul quand il était plus jeune, en programmant des choses simples et ludiques⁷. Il devrait donc faire venir ses élèves à la programmation par le côté ludique, et non avec les mêmes sempiternels exemples⁸.
- L'élève :
 - ne se rend pas compte que savoir programmer lui sera utile. Il s'agit pourtant d'une base qui se retrouve dans tous les langages et même dans la plupart des logiciels modernes⁹. Et puis, considéré comme "le jeune" donc le moins "allergique" aux ordinateurs, il se verra vraisemblablement confier à son premier poste la réalisation de quelques petits programmes en plus de ses attributions normales.
 - s'arrange un peu trop facilement d'un mépris de bon ton pour la programmation. Il lui est plus aisé d'apprendre une n-ième branche des mathématiques que de faire l'effort d'acquérir par la pratique une nouvelle tournure d'esprit.

On l'aura compris, il est à la fois facile et difficile d'apprendre à programmer. Pour l'*ingénieur*, cela demandera de la motivation et un peu d'effort : essentiellement de mettre ses maths de côté et de retrouver le goût des choses basiques. Pour un *collégien*, motivation et goût de l'effort seront au rendez-vous. Il lui restera malgré tout à acquérir quelques bases d'arithmétique et de géométrie. Comme annoncé par le titre de ce cours, collégien et ingénieur en sont au même point pour l'apprentissage de la programmation. De plus, et c'est un phénomène relativement nouveau, il en est de même pour le *débutant* et le "*geek*"¹⁰. Expliquons-nous : le passionné d'informatique a aujourd'hui tellement de choses à faire avec son ordinateur qu'il sera en général incollable sur les jeux, internet, les logiciels graphiques ou musicaux, l'installation ou la configuration de son système, l'achat du dernier gadget USB à la mode, etc. mais qu'en contrepartie il sera mauvais programmeur. Il y a quelques années, il y avait peu à faire avec son ordinateur sinon programmer. Programmer pour combler le manque de possibilités de l'ordinateur. Aujourd'hui, faire le tour de toutes les possibilités d'un ordinateur est une occupation à plein temps ! Ainsi, le "fana info" passe-t-il sa journée à se tenir au courant des nouveaux logiciels¹¹ et en oublie qu'il pourrait lui aussi

7. C'est une erreur fréquente de croire qu'il intéressera ses élèves en leur faisant faire des programmes centrés sur les mathématiques ou le calcul scientifique. De tels programmes leur seront peut-être utiles plus tard, mais ne sont pas forcément motivants. L'algèbre linéaire ou l'analyse numérique sont des domaines passionnants à étudier... mais certainement pas à programmer. Il faut admettre sans complexe que programmer un flipper, un master-mind ou un labyrinthe 3D est tout aussi formateur et plus motivant qu'inverser une matrice creuse.

8. La liste est longue, mais tellement vraie : quel cours de programmation ne rabâche pas les célèbres "factorielle", "suites de Fibonacci", "Quick Sort", etc ?

9. Savoir programmer ne sert pas seulement à faire du C++ ou du Java, ni même du Scilab, du Matlab ou du Maple : une utilisation avancée d'Excel ou du Word demande parfois de la programmation !

10. Une récompense à qui me trouve un substitut satisfaisant à cette expression consacrée.

11. Sans même d'ailleurs avoir le temps d'en creuser convenablement un seul !

en créer. En conclusion, collégiens ou ingénieurs, débutants ou passionnés, **tous les élèves sont à égalité**. C'est donc sans complexe que l'ingénieur pourra apprendre à programmer en même temps que le fils de la voisine.

1.1 Pourquoi savoir programmer ?

Nous venons partiellement de le dire. Résumons et complétons :

1. C'est la base. Apprendre un langage précis n'est pas du temps perdu car les mêmes concepts se retrouvent dans la plupart des langages. De plus, les logiciels courants eux-mêmes peuvent se programmer.
2. Il est fréquent qu'un stage ou qu'une embauche en premier poste comporte un peu de programmation, même, et peut-être surtout, dans les milieux où peu de gens programment.
3. Savoir programmer, c'est mieux connaître le matériel et les logiciels, ce qui est possible techniquement et ce qui ne l'est pas. Même à un poste non technique, c'est important pour prendre les bonnes décisions.

1.2 Comment apprendre ?

1.2.1 Choix du langage

Il faut d'abord choisir un langage de programmation. Un ingénieur pourrait évidemment être tenté d'apprendre à programmer en Maple, Matlab, Scilab ou autre. Il faut qu'il comprenne qu'il s'agit là d'outils spécialisés pour mathématicien ou ingénieur qui lui seront utiles et qui, certes, se programment, mais pas à proprement parler de langages généralistes complets. Sans argumenter sur les défauts respectifs des langages qui en font partie, il nous semble évident qu'il ne s'agit pas du bon choix pour l'apprentissage de la programmation.

En pratique, le choix actuel se fait souvent entre C++ et Java. Bien que Java ait été conçu, entre autres, dans un souci de simplification du C++¹², nous préférons C++ pour des raisons pédagogiques :

1. C++ est plus complexe dans son ensemble mais n'en connaître que les bases est déjà bien suffisant. Nous ne verrons donc dans ce cours qu'un sous ensemble du C++, suffisant en pratique.
2. Plus complet, C++ permet une programmation de haut niveau mais aussi une programmation simple, adaptée au débutant¹³. C++ permet également une programmation proche de la machine, ce qui est important pour le spécialiste mais aussi pour le débutant, car seule une bonne compréhension de la machine aboutit à une programmation convenable et efficace¹⁴.
3. C++ est souvent incontournable dans certains milieux, par exemple en finance.

12. Nous ne réduisons évidemment pas Java à un sous ensemble de C++. Il lui est supérieur sur certains aspects mais il est d'expressivité plus réduite.

13. Java force un cadre de programmation objet, déroutant pour le débutant.

14. Ne pas comprendre ce que la machine doit faire pour *exécuter* un programme, conduit à des programmes inconsidérément gourmands en temps ou mémoire.

4. Enfin, certains aspects pratiques et pourtant simples de C++ ont disparu dans Java¹⁵.

Depuis quelques années, un langage qui s'impose de plus en plus est le Python. La raison est qu'il est portable, puissant et facile d'accès. Cependant, il présente des inconvénients. Il est en constante évolution, non standardisé, et la compatibilité entre les versions n'est pas garantie¹⁶. De plus, les structures de données de Python, certes très utiles, cachent la complexité qu'il y a derrière du point de vue de la gestion mémoire, et il est important pour un ingénieur d'être conscient de ce qui se passe en coulisse. Encore une fois, répétons que le choix du langage n'est pas le plus important et que l'essentiel est d'apprendre à programmer.

1.2.2 Choix de l'environnement

Windows et Linux ont leurs partisans, souvent farouchement opposés, à tel point que certains n'admettent pas qu'il est possible d'être partisan des deux systèmes à la fois. Conscients des avantages et des inconvénients de chacun des deux systèmes, nous n'en prôtons aucun en particulier¹⁷. Ceci dit, pour des raisons pédagogiques, nous pensons qu'un *environnement de programmation intégré*, c'est à dire un logiciel unique permettant de programmer, est préférable à l'utilisation de multiples logiciels (éditeur, compilateur, débogueur, etc.). C'est vrai pour le programmeur confirmé, qui trouve en général dans cet environnement des outils puissants, mais c'est encore plus crucial pour le débutant. Un environnement de programmation, c'est :

- Toutes les étapes de la programmation regroupées en un seul outil de façon cohérente.
- Editer ses fichiers, les transformer en programme, passer en revue ses erreurs, détecter les bugs, parcourir la documentation, etc. tout cela avec un seul outil ergonomique.

Différents environnements de développement peuvent être choisis. L'environnement de référence sous Windows est celui de Microsoft, Visual Studio, qui existe en version gratuite, Express. C'est probablement l'un des meilleurs logiciels issus de la firme de Redmond¹⁸. Sous Linux, conseillons KDevelop et sous Mac XCode. L'inconvénient est qu'ils n'ont pas par défaut les mêmes raccourcis clavier que Visual. Certains fonctionnent sous toutes les plates-formes (Linux, Windows, Mac), en particulier QtCreator. L'avantage de ce dernier est qu'il utilise par défaut les mêmes raccourcis clavier que Visual Studio, mais qu'il comprend en plus le format CMake (dont nous reparlerons dans les TP). Son interface est fortement inspirée de KDevelop et XCode. Comme pour le choix du langage, le choix de l'environnement n'est pas limitant et en connaître un permet de s'adapter facilement à n'importe quel autre.

Le reste de ce poly est orienté vers QtCreator. Notons qu'il est facile d'installation sur toutes les plates-formes. Sous Windows et Mac, il vient avec les bibliothèques Qt, dont de toutes façons nous avons besoin pour utiliser *Imagine++*, que nous utiliserons pour tout ce qui est interface graphique. De plus, sous Windows nous n'avons pas de

15. Les opérateurs par exemple.

16. Ainsi l'opération $3/4$ donnera 0 en Python 2 (quotient de division euclidienne) et 0.75 en Java 3.

17. L'idéal est en fait d'avoir les deux "sous la main".

18. Le seul utilisable, diront les mauvaises langues...

compilateur C++ par défaut¹⁹, mais en installant la version de Qt pour MinGW on bénéficie justement du compilateur MinGW²⁰. Un autre compilateur pour Windows est celui de Microsoft qui vient avec VisualStudio, qui est certes gratuit mais ne l'installez que si ça ne vous dérange pas de donner à Microsoft des informations personnelles qui ne le concernent en rien²¹.

1.2.3 Principes et conseils

Au niveau auquel nous prétendons l'enseigner, la programmation ne requiert ni grande théorie, ni connaissances encyclopédiques. Les concepts utilisés sont rudimentaires mais c'est leur mise en oeuvre qui est délicate. S'il n'y avait qu'un seul conseil à donner, ce serait la *règle des trois "P"* :

1. **Programmer**
2. **Programmer**
3. **Programmer**

La pratique est effectivement essentielle. C'est ce qui fait qu'un enfant a plus de facilités, puisqu'il a plus de temps. Ajoutons quand même quelques conseils de base :

1. **S'amuser.** C'est une évidence en matière de pédagogie. Mais c'est tellement facile dans le cas de la programmation, qu'il serait dommage de passer à côté ! Au pire, si programmer n'est pas toujours une partie de plaisir pour tout le monde, il vaut mieux que le programme obtenu dans la douleur soit intéressant pour celui qui l'a fait !
2. **Bricoler.** Ce que nous voulons dire par là, c'est qu'il ne faut pas hésiter à tâtonner, tester, fouiller, faire, défaire, casser, etc. L'ordinateur est un outil expérimental. Mais sa programmation est elle aussi une activité expérimentale à la base. Même si le programmeur aguerri trouvera la bonne solution du premier jet, il est important pour le débutant d'apprendre à connaître le langage et l'outil de programmation en jouant avec eux.
3. **Faire volontairement des erreurs.** Provoquer les erreurs pendant la phase d'apprentissage pour mieux les connaître est le meilleur moyen de comprendre beaucoup de choses et aussi de repérer ces erreurs quand elles ne seront plus volontaires.
4. **Rester (le) maître**²² (de la machine et de son programme). Que programmer soit expérimental ne signifie pas pour autant qu'il faille faire n'importe quoi jusqu'à ce que ça marche plus ou moins. Il faut avancer progressivement, méthodiquement, en testant au fur et à mesure, sans laisser passer la moindre erreur ou imprécision.

19. sous Linux, le compilateur usuel est GCC (GNU Compiler Collection) et Clang sous Mac, soutenu par Apple. Le projet GNU développe de nombreux outils indispensables et libres pour compléter le système d'exploitation. Clang s'inspire fortement de GCC pour l'utilisation, mais dispose d'une licence différente, qui ne ferme pas la porte à des extensions non libres.

20. Minimal GNU for Windows, une adaptation de GCC pour Windows.

21. Cette intrusion n'est plus de mise avec VisualStudio 2015, enfin !

22. Le vocabulaire n'est pas choisi au hasard : un programme est une suite d'*ordres*, de *commandes* ou d'*instructions*. On voit bien qui est le chef !

5. **Debugger.** Souvent, la connaissance du débogueur (l'outil pour rechercher les bugs) est négligée et son apprentissage est repoussé au stade avancé. Cet outil est pourtant pratique pour comprendre ce qui se passe dans un programme, même dépourvu de bugs. Il faut donc le considérer comme essentiel et faisant partie intégrante de la conception d'un programme. Là encore, un bon environnement de programmation facilite la tâche.

—

Gardons bien présents ces quelques principes car il est maintenant temps de...

passer à notre premier programme !

Chapitre 2

Bonjour, Monde !

(Si certains collégiens sont arrivés ici, ils sont bien courageux ! Lorsque je disais tout à l'heure qu'ils pouvaient facilement apprendre à programmer, je le pensais vraiment. Par contre, c'est avec un peu d'optimisme que j'ai prétendu qu'ils pouvaient le faire en lisant un polycopié destiné à des ingénieurs. Enfin, je suis pris à mon propre piège ! Alors, à tout hasard, je vais tenter d'expliquer au passage les mathématiques qui pourraient leur poser problème.)

—

Si l'on en croit de nombreux manuels de programmation, un premier programme doit toujours ressembler à ça :

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

int main()
{
    cout << "Hello ,_World!" << endl;
    return 0;
}
```

Eh bien, allons-y ! Décortiquons-le ! Dans ce programme, qui *affiche à l'écran*¹ le texte "Hello, World!", les lignes 1 et 2 sont des instructions *magiques*² qui servent à pouvoir utiliser dans la suite `cout` et `endl`. La ligne 4 `int main()` définit une *fonction* appelée `main()`, qui *renvoie*³ un nombre entier. Cette fonction est spéciale car c'est la fonction principale d'un programme C++, celle qui est appelée automatiquement⁴ quand le

1. Cette expression, vestige de l'époque où les ordinateurs étaient dotés d'un écran capable de n'afficher que des caractères et non des *graphiques* (courbes, dessins, etc.), signifie aujourd'hui que l'affichage se fera dans une *fenêtre* simulant l'écran d'un ordinateur de cette époque. Cette fenêtre est appelée *terminal*, **console**, fenêtre de commande, fenêtre DOS, `xterm`, etc. suivant les cas. Souvenons nous avec un minimum de respect que c'était déjà un progrès par rapport à la génération précédente, dépourvue d'écran et qui utilisait une imprimante pour communiquer avec l'homme...ce qui était relativement peu interactif !

2. Entendons par là des instructions que nous n'expliquons pas pour l'instant. Il n'y a (mal ?)-heureusement rien de magique dans la programmation.

3. On dit aussi *retourne*. À qui renvoie-t-elle cet entier ? Mais à celui qui l'a *appelée*, voyons !

4. Voilà, maintenant vous savez qui appelle `main()`. Dans un programme, les fonctions s'appellent les unes les autres. Mais `main()` n'est appelée par personne puisque c'est la première de toutes. (Du moins en apparence car en réalité le programme a plein de choses à faire avant d'arriver dans `main()`)

programme *est lancé*⁵. Délimitée par les accolades ({ ligne 5 et } ligne 8), la fonction `main()` se termine ligne 7 par **return 0**; qui lui ordonne de retourner l'entier 0. Notons au passage que toutes les instructions se terminent par un point-virgule ;. Enfin, à la ligne 6, seule ligne "intéressante", `cout << "Hello, \_World!" << endl`; affiche, grâce à la *variable*⁶ `cout` qui correspond à la sortie *console*⁷, des données séparées par des <<. La première de ces données est la *chaîne de caractères*⁸ "Hello, _World!". La deuxième, `endl`, est un *retour à la ligne*⁹.

Ouf! Que de termes en italique. Que de concepts à essayer d'expliquer! Et pour un programme aussi simple! Mais là n'est pas le problème. Commencer par expliquer ce programme, c'est être encore dans le vide, dans le magique, dans l'abstrait, dans l'approximatif. Nous ne sommes pas réellement maîtres de la machine. Taper des instructions et voir ce qui se passe sans **comprendre ce qui se passe** n'est pas raisonnable. En fait, c'est même très dommageable pour la suite. On ne donne pas efficacement d'ordre à quelqu'un sans comprendre comment il fonctionne ni ce que les ordres donnés entraînent comme travail. De même,

on ne programme pas convenablement sans comprendre ce que l'ordinateur aura exactement besoin de faire pour exécuter ce programme.

C'est toute cette approche qui est négligée quand on commence comme nous venons de le faire. Donc...

Stop! Stop! Stop! Faux départ! On reprend le :

et il commence par plusieurs autres fonctions que le programmeur n'a pas à connaître et qui finissent par appeler `main()`. D'ailleurs, si personne ne l'appelait, à qui `main()` retournerait-elle un entier?)

5. Je savais bien que vouloir expliquer tous les barbarismes propres aux informaticiens m'interromprait souvent. Mais bon. Donc, un programme *démarre* ou *est lancé*. Après quoi, il *s'exécute* ou *tourne*. Enfin, il *se termine* ou *meurt*.

6. Les *données* sont *rangées* ou *stockées* dans des *variables* qui mémorisent des *valeurs*. Ces *variables* ne sont d'ailleurs pas toujours variables au sens usuel, puisque certaines sont constantes!

7. Qu'est-ce que je disais! On affiche dans une fenêtre console!

8. En clair, un texte.

9. Ce qui signifie que la suite de l'affichage sur la console se fera sur une nouvelle ligne.

Chapitre 2 (deuxième essai)

Comment ça marche ?

Le problème avec le programme précédent est qu'il est très loin de ce qu'un ordinateur sait faire naturellement. En fait, un ordinateur ne sait pas faire de C++. Il ne sait que calculer¹⁰, transformer des nombres en autres nombres. Bien que peu compréhensible pour le débutant, un programme en C++ se veut le plus proche possible de l'Homme, tout en restant évidemment accessible¹¹ à la machine. Le C++ est un langage très complet, peut-être même trop. Il peut être relativement proche de la machine si nécessaire et au contraire de "haut niveau" quand il le faut. La largeur de son spectre est une des raisons de son succès. C'est aussi ce qui fait que son apprentissage complet demande un long travail et nous ne verrons ici qu'une partie restreinte du C++ !

2.1 L'ordinateur

Pour savoir ce qu'un ordinateur sait vraiment faire, il faut commencer par son organe principal : le micro-processeur.

2.1.1 Le micro-processeur

Quel qu'il soit¹² et quelle que soit sa vitesse¹³, un micro-processeur ne sait faire que des choses relativement basiques. Sans être exhaustif, retenons juste ceci :

- Il sait exécuter une suite ordonnée d'instructions.
- Il possède un petit nombre de mémoires internes appelées registres.
- Il dialogue avec le monde extérieur via de la mémoire¹⁴ en plus grande quantité que ses registres.
- Cette mémoire contient, sous forme de nombres, les instructions à exécuter et les données sur lesquelles travailler.
- Les instructions sont typiquement :
 - Lire ou écrire un nombre dans un registre ou en mémoire.
 - Effectuer des calculs simples : addition, multiplication, etc.
 - Tester ou comparer des valeurs et décider éventuellement de sauter à une autre partie de la suite d'instructions.

10. Un *computer*, quoi !

11. Cette notion est évidemment dépendante de notre savoir faire informatique à l'instant présent. Les premiers langages étaient plus éloignés de l'Homme car plus proches de la machine qui était alors rudimentaire, et l'on peut envisager que les futurs langages seront plus proches de l'Homme.

12. Pentium ou autre

13. Plus exactement la fréquence à laquelle il exécute ses instructions. Aujourd'hui l'horloge va environ à 3GHz. (Mais attention : une instruction demande plus d'un cycle d'horloge !)

14. Aujourd'hui, typiquement 1Go (giga-octets), soit $1024 \times 1024 \times 1024$ mémoires de 8 bits (mémoires pouvant stocker des nombres entre 0 et 255).

Voici par exemple ce que doit faire le micro-processeur quand on lui demande d'exécuter "c=3*a+2*b;" en C++, où a,b,c sont trois variables entières :

```
00415A61  mov  eax,dword ptr [a] // mettre dans le registre eax
                                     // le contenu de l'adresse où
                                     // est mémorisée la variable a
00415A64  imul  eax,eax,3          // effectuer eax=eax*3
00415A67  mov  ecx,dword ptr [b] // idem mais b dans ecx
00415A6A  lea  edx,[eax+ecx*2]    // effectuer edx=eax+ecx*2
00415A6D  mov  dword ptr [c],edx // mettre le contenu du registre edx
                                     // à l'adresse où est mémorisée la
                                     // variable c
```

Ce programme est désigné comme du *Code Machine*. Le nombre au début de chaque ligne est une adresse. Nous allons en reparler. A part lui, le reste est relativement lisible pour l'Homme (attention, c'est moi qui ai ajouté les remarques sur le coté droit !). Ceci parce qu'il s'agit d'un programme en langage *assembleur*, c'est-à-dire un langage où chaque instruction est vraiment une instruction du micro-processeur, mais où le nom de ces instructions ainsi que leurs arguments sont explicites. En réalité, le micro-processeur ne comprend pas l'assembleur. Comprendre "mov eax,dword ptr [a]" lui demanderait non seulement de décoder cette suite de symboles, mais aussi de savoir où est rangée la variable a. Le vrai langage du micro-processeur est le *langage machine*, dans lequel les instructions sont des nombres. Voici ce que ça donne pour notre "c=3*a+2*b;" :

```
00415A61 8B 45 F8
00415A64 6B C0 03
00415A67 8B 4D EC
00415A6A 8D 14 48
00415A6D 89 55 E0
```

A part encore une fois la colonne de gauche, chaque suite de nombres¹⁵ correspond évidemment à une instruction précise. C'est tout de suite moins compréhensible¹⁶ ! Notons que chaque micro-processeur à son *jeu d'instructions* ce qui veut dire que la traduction de c=3*a+2*b; en la suite de nombres 8B45F86BC0038B4DEC8D14488955E0 est propre au Pentium que nous avons utilisé pour notre exemple :

Une fois traduit en langage machine pour un micro-processeur donné, un programme C++ n'a de sens que pour ce micro-processeur.

Remarquons aussi que les concepteurs du Pentium ont décidé de créer une instruction spécifique pour calculer edx=eax+ecx*2 en une seule fois car elle est très fréquente. Si on avait demandé c=3*a+3*b;, notre programme serait devenu :

```
00415A61 8B 45 F8    mov  eax,dword ptr [a]
00415A64 6B C0 03    imul  eax,eax,3
00415A67 8B 4D EC    mov  ecx,dword ptr [b]
```

15. Nombres un peu bizarres, certes, puisqu'il contiennent des lettres. Patience, jeune *Padawan* ! Nous en reparlons aussi tout de suite !

16. Et pourtant, les informaticiens programmaient comme cela il n'y a pas si longtemps. C'était déjà très bien par rapport à l'époque antérieure où il fallait programmer en base 2... et beaucoup moins bien que lorsqu'on a pu enfin programmer en assembleur !

```
00415A6A 6B C9 03    imul   ecx,ecx,3
00415A6D 03 C1          add    eax,ecx
00415A6F 89 45 E0    mov    dword ptr [c],eax
```

car "lea edx, [eax+ecx*3]" n'existe pas !

Mais revenons à nos nombres...

2.1.2 La mémoire

La mémoire interne du micro-processeur est gérée comme des registres, un peu comme les variables du C++, mais en nombre prédéfini. Pour *stocker*¹⁷ la suite d'instructions à lui fournir, on utilise de la mémoire en quantité bien plus importante, désignée en général par *la mémoire de l'ordinateur*. Il s'agit des fameuses "barrettes"¹⁸ de mémoire que l'on achète pour augmenter la capacité de sa machine et dont les prix fluctuent assez fortement par rapport au reste des composants d'un ordinateur. Cette mémoire est découpée en octets. Un octet¹⁹ correspond à un nombre binaire de 8 bits²⁰, soit à $2^8 = 256$ valeurs possibles. Pour se repérer dans la mémoire, il n'est pas question de donner des noms à chaque octet. On numérote simplement les octets et on obtient ainsi des *adresses mémoire*. Les nombres 00415A61, etc. vus plus haut sont des adresses ! Au début, ces nombres étaient écrits en binaire, ce qui était exactement ce que comprenait le micro-processeur. C'est devenu déraisonnable quand la taille de la mémoire a dépassé les quelques centaines d'octets. Le contenu d'un octet de mémoire étant lui aussi donné sous la forme d'un nombre, on a opté pour un système adapté au fait que ces nombres sont sur 8 bits : plutôt que d'écrire les nombre en binaire, le choix de la base 16 permettait de représenter le contenu d'un octet sur deux chiffres (0,1,...,9,A,B,C,D,E,F). Le système *hexadécimal*²¹ était adopté... Les conversions de binaire à hexadécimal sont très simples, chaque chiffre hexadécimal valant pour un paquet de 4 bits, alors qu'entre binaire et décimal, c'est moins immédiat. Il est aujourd'hui encore utilisé quand on désigne le contenu d'un octet ou une adresse²². Ainsi, notre fameux `c=3*a+2*b`; devient en mémoire :

17. Encore un anglicisme...

18. Aujourd'hui, typiquement une ou plusieurs barrettes pour un total de 1 ou 2Go, on l'a déjà dit. Souvenons nous avec une larme à l'oeil des premiers PC qui avaient 640Ko (kilo-octet soit 1024 octets), voire pour les plus âgés d'entre nous des premiers ordinateurs personnels avec 4Ko, ou même des premières cartes programmables avec 256 octets !

19. *byte* en anglais. Attention donc à ne pas confondre byte et bit, surtout dans des abréviations comme 512kb/s données pour le débit d'un accès internet... b=bit, B=byte=8 bits

20. **Le coin des collégiens** : en binaire, ou base 2, on compte avec deux chiffres au lieu de dix d'habitude (c'est à dire en décimal ou base 10). Cela donne : 0, 1, 10, 11, 100, 101, 110, 111, ... Ainsi, 111 en binaire vaut 7. Chaque chiffre s'appelle un bit. On voit facilement qu'avec un chiffre on compte de 0 à 1 soit deux nombres possibles ; avec deux chiffres, de 0 à 3, soit $4 = 2 \times 2$ nombres ; avec 3 chiffres, de 0 à 7, soit $8 = 2 \times 2 \times 2$ nombres. Bref avec n bits, on peut coder 2^n (2 multiplié par lui-même n fois) nombres. Je me souviens avoir appris la base 2 en grande section de maternelle avec des cubes en bois ! Étrange programme scolaire. Et je ne dis pas ça pour me trouver une excuse d'être devenu informaticien. Quoique...

21. **Coin des collégiens (suite)** : en base 16, ou hexadécimal, on compte avec 16 chiffres. Il faut inventer des chiffres au delà de 9 et on prend A,B,C,D,E,F. Quand on compte, cela donne : 0, 1, 2, ..., 9, A, B, C, D, E, F, 10, 11, 12, 13, ..., 19, 1A, 1B, 1C, ... Ainsi 1F en hexadécimal vaut 31. Avec 1 chiffre, on compte de 0 à 15 soit 16 nombres possibles ; avec 2 chiffres, de 0 à 255 soit $256 = 16 \times 16$ nombres possibles, etc. Un octet peut s'écrire avec 8 bits en binaire, ou 2 nombres en hexadécimal et va de 0 à 255, ou 11111111 en binaire, ou FF en hexadécimal.

22. Dans ce cas, sur plus de 2 chiffres : 8 pour les processeurs 32 bits, 16 pour les processeurs 64 bits.

adresse mémoire	contenu	représente
00415A61	8B	mov eax, dword ptr [a]
00415A62	45	
00415A63	F8	
00415A64	6B	imul eax, eax, 3
00415A65	C0	
...	...	

La mémoire ne sert pas uniquement à stocker la suite d'instructions à exécuter mais aussi toutes les variables et données du programme, les registres du micro-processeur étant insuffisants. Ainsi nos variables a, b, c sont stockées quelque part en mémoire sur un nombre d'octets suffisant²³ pour représenter des nombres entiers (ici 4 octets) et dans un endroit décidé par le C++, de tel sorte que l'instruction 8B45F8 aille bien chercher la variable a ! C'est un travail pénible, que le C++ fait pour nous et que les programmeurs faisaient autrefois à la main²⁴. Bref, on a en plus²⁵ :

adresse mémoire	contenu	représente
...	...	
00500000	a_1	a
00500001	a_2	
00500002	a_3	
00500003	a_4	
00500004	b_1	b
00500005	b_2	
00500006	b_3	
00500007	b_4	
...	...	

où les octets $a_1, \dots, a_4$ combinés donnent l'entier a sur 32 bits. Certains processeurs (dits *big-endian*)²⁶ décident $a = a_1a_2a_3a_4$, d'autres (*little-endian*)²⁷ que $a = a_4a_3a_2a_1$. Cela signifie que :

Tout comme pour les instructions, un nombre stocké par un micro-processeur dans un fichier peut ne pas être compréhensible par un autre micro-processeur qui relit le fichier !

23. Les variables ayant plus de 256 valeurs possibles sont forcément stockées sur plusieurs octets. Ainsi, avec 4 octets on peut compter en binaire sur $4 \times 8 = 32$ bits, soit 2^{32} valeurs possibles (plus de 4 milliards).

24. Ce qui était le plus pénible n'était pas de décider où il fallait ranger les variables en mémoire, mais d'ajuster les instructions en conséquence. Si on se trompait, on risquait d'écrire au mauvais endroit de la mémoire. Au mieux, cela effaçait une autre variable — ce comportement est encore possible de nos jours — au pire, cela effaçait des instructions et le programme pouvait faire de "grosses bêtises" — ceci est aujourd'hui impossible sous Windows ou Linux, et ne concerne plus que certains systèmes.

25. Nous faisons ici un horrible mensonge à des fins simplificatrices. Dans notre cas, les variables étaient des variables locales à la fonction `main()` donc stockées dans la *pile*. Elles ne sont pas à une adresse mémoire définie à l'avance de manière absolue mais à une adresse relative à l'emplacement où la fonction rangera ses variables locales en fonction de ce que le programme aura fait avant. Cela explique la simplicité de l'instruction `mov eax, dword ptr [a]` dans notre cas. Nous verrons tout cela plus tard.

26. Comme les PowerPC des vieux Macs

27. Comme les processeurs Intel et AMD

2.1.3 Autres Composants

Micro-processeur et mémoire : nous avons vu le principal. Complétons le tableau avec quelques autres éléments importants de l'ordinateur.

Types de mémoire

La mémoire dont nous parlions jusqu'ici est de la *mémoire vive* ou RAM. Elle est rapide²⁸ mais a la mauvaise idée de s'effacer quand on éteint l'ordinateur. Il faut donc aussi de la *mémoire morte* ou ROM, c'est-à-dire de la mémoire conservant ses données quand l'ordinateur est éteint mais qui en contre-partie ne peut être modifiée²⁹. Cette mémoire contient en général le minimum pour que l'ordinateur démarre et exécute une tâche prédéfinie. Initialement, on y stockait les instructions nécessaires pour que le programmeur puisse remplir ensuite la RAM avec les instructions de son programme. Il fallait retaper le programme à chaque fois³⁰ ! On a donc rapidement eu recours à des *moyens de stockage* pour sauver programmes et données à l'extinction de l'ordinateur. Il suffisait alors de mettre en ROM le nécessaire pour gérer ces moyens de stockages.

Moyens de stockage

Certains permettent de lire des données, d'autres d'en écrire, d'autres les deux à la fois. Certains ne délivrent les données que dans l'ordre, de manière séquentielle, d'autres, dans l'ordre que l'on veut, de manière aléatoire. Ils sont en général bien plus lents que la mémoire et c'est sûrement ce qu'il faut surtout retenir ! On recopie donc en RAM la partie des moyens de stockage sur laquelle on travaille.

Faire travailler le micro-processeur avec le disque dur est BEAUCOUP plus lent qu'avec la mémoire (1000 fois plus lent en temps d'accès, 100 fois plus en débit)^a

^a. Rajoutez un facteur 50 supplémentaire entre la mémoire et la mémoire cache du processeur !

Au début, les moyens de stockages étaient mécaniques : cartes ou bandes perforées. Puis ils devinrent magnétiques : mini-cassettes³¹, disquettes³², disques durs³³ ou bandes magnétiques. Aujourd'hui, on peut rajouter les CD, DVD, les cartes mémoire, les "clés USB", etc, etc.

28. Moins que les registres, ou même que le cache mémoire du processeur, dont nous ne parlerons pas ici.

29. Il est pénible qu'une ROM ne puisse être modifiée. Alors, à une époque, on utilisait des mémoires modifiables malgré tout, mais avec du matériel spécialisé (EPROMS). Maintenant, on a souvent recours à de la mémoire pouvant se modifier de façon logicielle (mémoire "flashable") ou, pour de très petites quantités de données, à une mémoire consommant peu (CMOS) et complétée par une petite pile. Dans un PC, la mémoire qui sert à démarrer s'appelle le BIOS. Il est flashable et ses paramètres de réglage sont en CMOS. Attention à l'usure de la pile !

30. A chaque fois qu'on allumait l'ordinateur mais aussi à chaque fois que le programme plantait et s'effaçait lui-même, c'est-à-dire la plupart du temps !

31. Très lent et très peu fiable, mais le quotidien des ordinateurs personnels.

32. Le luxe. Un lecteur de 40Ko coûtait 5000F !

33. Les premiers étaient de véritables moteurs de voiture, réservés aux importants centres de calcul.

Périphériques

On appelle encore périphériques différents appareils reliés à l'ordinateur : clavier, souris, écran, imprimante, modem, scanner, etc. Ils étaient initialement là pour servir d'interface avec l'Homme, comme des entrées et des sorties entre le micro-processeur et la réalité. Maintenant, il est difficile de voir encore les choses de cette façon. Ainsi les cartes graphiques, qui pouvaient être considérées comme un périphérique allant avec l'écran, sont-elles devenues une partie essentielle de l'ordinateur, véritables puissances de calcul, à tel point que certains programmeur les utilisent pour faire des calculs sans même afficher quoi que ce soit. Plus encore, c'est l'ordinateur qui est parfois juste considéré comme maillon entre différents appareils. Qui appellerait périphérique un caméscope qu'on relie à un ordinateur pour envoyer des vidéos sur internet ou les transférer sur un DVD ? Ce serait presque l'ordinateur qui serait un périphérique du caméscope !

2.2 Système d'exploitation

Notre vision jusqu'ici est donc la suivante :

1. Le processeur démarre avec les instructions présentes en ROM.
2. Ces instructions lui permettent de lire d'autres instructions présentes sur le disque dur et qu'il recopie en RAM.
3. Il exécute les instructions en question pour il lire des données (entrées) présentes elles-aussi sur le disque dur et générer de nouvelles données (sorties). A moins que les entrées ou les sorties ne soient échangées via les périphériques.

Assez vite, ce principe a évolué :

1. Le contenu du disque dur a été organisé en fichiers. Certains fichiers représentaient des données³⁴, d'autres des programmes³⁵, d'autres encore contenaient eux-mêmes des fichiers³⁶.
2. Les processeurs devenant plus rapides et les capacités du disque dur plus importantes, on a eu envie de gérer plusieurs programmes et d'en exécuter plusieurs : l'un après l'autre, puis plusieurs en même temps (multi-tâches), puis pour plusieurs utilisateurs en même temps (multi-utilisateurs)³⁷, enfin avec plusieurs processeurs par machine.

Pour gérer tout cela, s'est dégagé le concept de *système d'exploitation*³⁸. Windows, Unix (dont linux) et MAC/OS sont les plus répandus. Le système d'exploitation est aujourd'hui responsable de gérer les fichiers, les interfaces avec les périphériques ou les utilisateurs³⁹, mais son rôle le plus délicat est de gérer les programmes (ou tâches ou

34. Les plus courantes étaient les textes, où chaque octet représentait un caractère. C'était le célèbre code ASCII (65 pour A, 66 pour B, etc.). A l'ère du multimédia, les formats sont aujourd'hui nombreux, concurrents, et plus ou moins normalisés.

35. On parle de fichier *exécutable*...

36. Les répertoires.

37. Aujourd'hui, c'est pire. Un programme est souvent lui même en plusieurs parties s'exécutant en même temps (*les threads*). Quant au processeur, il exécute en permanence plusieurs instructions en même temps (on dit qu'il est *super-scalaire*) !

38. Operating System

39. Espérons qu'un jour les utilisateurs ne seront pas eux-aussi des périphériques !

process) en train de s'exécuter. Il doit pour cela essentiellement faire face à deux problèmes⁴⁰ :

1. Faire travailler le processeur successivement par petites tranches sur les différents programmes. Il s'agit de donner la main de manière intelligente et équitable, mais aussi de replacer un process interrompu dans la situation qu'il avait quittée lors de son interruption.
2. Gérer la mémoire dédiée à chaque process. En pratique, une partie ajustable de la mémoire est réservée à chaque process. La mémoire d'un process devient *mémoire virtuelle* : si un process est déplacé à un autre endroit de la *mémoire physique* (la RAM), il ne s'en rend pas compte. On en profite même pour mettre temporairement hors RAM (donc sur disque dur) un process en veille. On peut aussi utiliser le disque dur pour qu'un process utilise plus de mémoire que la mémoire physique : mais attention, le disque étant très lent, ce process risque de devenir lui aussi très lent.

Lorsqu'un process a besoin de trop de mémoire, il utilise, sans prévenir, le disque dur à la place de la mémoire et peut devenir très lent. On dit qu'il *swappe* (ou *pagine*). Seule sa lenteur (et le bruit du disque dur !) permet en général de s'en rendre compte (on peut alors s'en assurer avec le gestionnaire de tâche du système).

Autre progrès : on gère maintenant la mémoire virtuelle de façon à séparer les process entre eux et, au sein d'un même process, la mémoire contenant les instructions de celle contenant les données. Il est rigoureusement impossible qu'un process buggé puisse modifier ses instructions ou la mémoire d'un autre process en écrivant à un mauvais endroit de la mémoire⁴¹.

Avec l'arrivée des systèmes d'exploitation, les fichiers exécutables ont du s'adapter pour de nombreuses raisons de gestion et de partage de la mémoire. En pratique, un programme exécutable linux ne tournera pas sous Windows et réciproquement, même s'ils contiennent tous les deux des instructions pour le même processeur.

Un fichier exécutable est spécifique, non seulement à un processeur donné, mais aussi à un système d'exploitation donné.

Au mieux, tout comme les versions successives d'une famille de processeur essaient de continuer à comprendre les instructions de leurs prédécesseurs, tout comme les versions successives d'un logiciel essaient de pouvoir lire les données produites avec les versions précédentes, les différentes versions d'un système d'exploitation essaient de pouvoir exécuter les programmes faits pour les versions précédentes. C'est la *compatibilité ascendante*, que l'on paye souvent au prix d'une complexité et d'une lenteur accrues.

2.3 La Compilation

Tout en essayant de comprendre ce qui se passe en dessous pour en tirer des informations utiles comme la gestion de la mémoire, nous avons entrevu que transformer

40. Les processeurs ont évidemment évolué pour aider le système d'exploitation à faire cela efficacement.

41. Il se contente de modifier anarchiquement ses données, ce qui est déjà pas mal !

un programme C++ en un fichier exécutable est un travail difficile mais utile. Certains logiciels disposant d'un langage de programmation comme Maple ou Scilab ne transforment pas leurs programmes en langage machine. Le travail de traduction est fait à l'exécution du programme qui est alors analysé au fur et à mesure⁴² : on parle alors de *langage interprété*. L'exécution alors est évidemment très lente. D'autres langages, comme Java, décident de résoudre les problèmes de *portabilité*, c'est-à-dire de dépendance au processeur et au système, en plaçant une couche intermédiaire entre le processeur et le programme : la *machine virtuelle*. Cette machine, évidemment écrite pour un processeur et un système donnés, peut exécuter des programmes dans un langage machine virtuel⁴³, le "*byte code*". Un programme Java est alors traduit en son équivalent dans ce langage machine. Le résultat peut être exécuté sur n'importe quelle machine virtuelle Java. La contrepartie de cette portabilité est évidemment une perte d'efficacité.

La traduction en *code natif* ou en *byte code* d'un programme s'appelle la **compilation**⁴⁴. Un *langage compilé* est alors à opposer à un *langage interprété*. Dans le cas du C++ et de la plupart des langages compilés (Fortran, C, etc), la compilation se fait vers du code natif. On transforme un fichier *source*, le programme C++, en un fichier *objet*, suite d'instructions en langage machine.

Cependant, le fichier objet ne se suffit pas à lui-même. Des instructions supplémentaires sont nécessaires pour former un fichier exécutable complet :

- de quoi lancer le `main()` ! Plus précisément, tout ce que le process doit faire avant et après l'exécution de `main()`.
- des fonctions ou variables faisant partie du langage et que le programmeur utilise sans les reprogrammer lui-même, comme `cout`, `cout < min()`, etc. L'ensemble de ces instructions constitue ce qu'on appelle une *bibliothèque*⁴⁵.
- des fonctions ou variables programmées par le programmeur lui-même dans d'autres fichiers source compilés par ailleurs en d'autres fichiers objet, mais qu'il veut utiliser dans son programme actuel.

La synthèse de ces fichiers en un fichier exécutable s'appelle **l'édition des liens**. Le programme qui réalise cette opération est plus souvent appelé *linker* qu'éditeur de liens...

En résumé, la production du fichier exécutable se fait de la façon suivante :

- 1. *Compilation* : fichier source → fichier objet.**
- 2. *Link* : fichier objet + autres fichiers objets + bibliothèque standard ou autres → fichier exécutable.**

42. même s'il est parfois pré-traité pour accélérer l'exécution.

43. Par opposition, le "vrai" langage machine du processeur est alors appelé *code natif*.

44. Les principes de la compilation sont une des matières de base de l'informatique, traditionnelle et très formatrice. Quand on sait programmer un compilateur, on sait tout programmer (Évidemment, un compilateur est un programme ! On le programme avec le compilateur précédent ! Même chose pour les systèmes d'exploitation...). Elle nécessite un cours à part entière et nous n'en parlerons pas ici !

45. Une bibliothèque est en fait un ensemble de fichiers objets pré-existants regroupés en un seul fichier. Il peut s'agir de la bibliothèque des fonctions faisant partie de C++, appelée bibliothèque standard, mais aussi d'une bibliothèque supplémentaire fournie par un tiers.

2.4 L'environnement de programmation

L'environnement de programmation⁴⁶ est le logiciel permettant de programmer. Dans notre cas il s'agit de QtCreator. Dans d'autres cas, il peut simplement s'agir d'un ensemble de programmes. Un environnement contient au minimum un *éditeur* pour créer les fichiers sources, un *compilateur/linker* pour créer les exécutables, un *debugueur* pour traquer les erreurs de programmation, et un *gestionnaire de projet* pour gérer les différents fichiers sources et exécutables avec lesquels on travaille.

Nous reportons ici le lecteur au texte du premier TP et à l'appendice C. En plus de quelques notions rudimentaires de C++ que nous verrons au chapitre suivant, quelques informations supplémentaires sont utiles pour le suivre.

2.4.1 Noms de fichiers

L'*extension* (le suffixe) sert à se repérer dans les types de fichier :

- Un fichier source C++ se terminera par `.cpp`⁴⁷.
- Un fichier objet sera en `.obj` (Windows) ou `.o` (Linux/Mac)
- Un fichier exécutable en `.exe` (Windows) ou sans extension (Linux/Mac)

Nous verrons aussi plus loin dans le cours :

- Les "en-tête" C++ ou *headers* servant à être inclus dans un fichier source : fichiers `.h`
- Les bibliothèques (ensembles de fichiers objets archivés en un seul fichier) : fichiers `.lib` ou `.dll` (Windows) ou `.a` ou `.so` (Linux/Mac)

2.4.2 Debugueur

Lorsqu'un programme ne fait pas ce qu'il faut, on peut essayer de comprendre ce qui ne va pas en truffant son source d'instructions pour imprimer la valeur de certaines données ou simplement pour suivre son déroulement. Ca n'est évidemment pas très pratique. Il est mieux de pouvoir suivre son déroulement instruction par instruction et d'afficher à la demande la valeur des variables. C'est le rôle du *debugueur*⁴⁸.

Lorsqu'un langage est interprété, il est relativement simple de le faire s'exécute *pas à pas* car c'est le langage lui-même qui exécute le programme. Dans le cas d'un langage compilé, c'est le micro-processeur qui exécute le programme et on ne peut pas l'arrêter à chaque instruction ! Il faut alors mettre en place des *points d'arrêt* en modifiant temporairement le code machine du programme pour que le processeur s'arrête lorsqu'il atteint l'instruction correspondant à la ligne de source à debugger. Si c'est compliqué à mettre au point, c'est très simple à utiliser, surtout dans un environnement de programmation graphique.

Nous verrons au fur et à mesure des TP comment le debugueur peut aussi inspecter les appels de fonctions, espionner la modification d'une variable, etc.

46. souvent appelé IDE, Integrated Development Environment.

47. Un fichier en `.c` sera considéré comme du C.

48. Débogueur en français !

2.5 Le minimum indispensable

2.5.1 Pour comprendre le TP

Voici le programme C++ strictement minimal, écrit dans un fichier `main.cpp` :

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

Le mot réservé `main` indique le point d'entrée du programme. C'est une fonction (un bloc de code). Comme une fonction mathématique elle prend une ou plusieurs entrées et a une valeur de retour. Pour cette fonction, il n'y a rien entre les parenthèses, ce qui signifie qu'elle n'a pas d'argument en entrée. Par contre, elle retourne une valeur entière (type `int` comme integer). Cette fonction ne fait rien, la liste de ses instructions est vide (bloc entre accolades). Mais quelle est la valeur retournée par cette fonction ? Cette fonction retourne la valeur 0, qui par convention indique que tout est normal pour ce programme. Pour être plus explicite, il aurait mieux valu écrire

```
int main() {
    return 0;
}
```

ce qui a exactement le même effet, car la fonction `main` retourne 0 si on ne le précise pas. Mais le cas de cette fonction est spécial, il est préférable d'être plus explicite et de préciser qu'on retourne la valeur 0. Notons au passage qu'on a écrit l'ordre `return` sur une nouvelle ligne, que comme toute instruction elle se termine avec point-virgule obligatoire, et qu'on a décalé cette instruction vers la droite (on dit "indenté") pour bien montrer qu'on est à l'intérieur du bloc. Le compilateur est indifférent aux retours à la ligne et aux indentations, ils sont présents uniquement pour faciliter la lecture pour nous, le programmeur. Cependant, ne les considérez pas comme optionnels, il est primordial d'écrire un programme propre et lisible. Pour ceux qui connaissent ce langage, notez que le Python se passe des accolades et marque un bloc par l'indentation, qui devient obligatoire. Le fait que celle-ci ne soit pas significative en C++ ne veut pas dire qu'on doit faire n'importe quoi. Insistons donc déjà sur le fait qu'il faut indenter.

Ce programme ne fait rien, mais il faut quand même vérifier qu'il est correct, c'est-à-dire qu'il respecte la syntaxe et qu'il suffit à créer un programme exécutable. Pour cela, nous allons utiliser `Cmake`, qui a l'avantage d'être multi plates-formes et d'être compatible avec tous les environnements de développement courants. Créons donc un fichier `CMakeLists.txt` de contenu suivant :

```
add_executable(EssaiQtCreator main.cpp)
```

Cela indique qu'on a un fichier de code source, `main.cpp`, et que notre programme exécutable s'appellera `EssaiQtCreator`. On écrit ce fichier dans le même dossier que le fichier source `main.cpp`, dossier source). On lance `QtCreator` et l'option "Open file or project" dans le menu `File`. On lui donne le fichier `CMakeLists.txt` comme projet. Dans la fenêtre de configuration, on sélectionne un dossier où générer les fichiers (dossier de build). Pour cela, cliquer sur `Browse` et créer un nouveau dossier, puis cliquer le bouton "Configure project". On peut voir un message du type

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/pascal/TEMP/Build
```

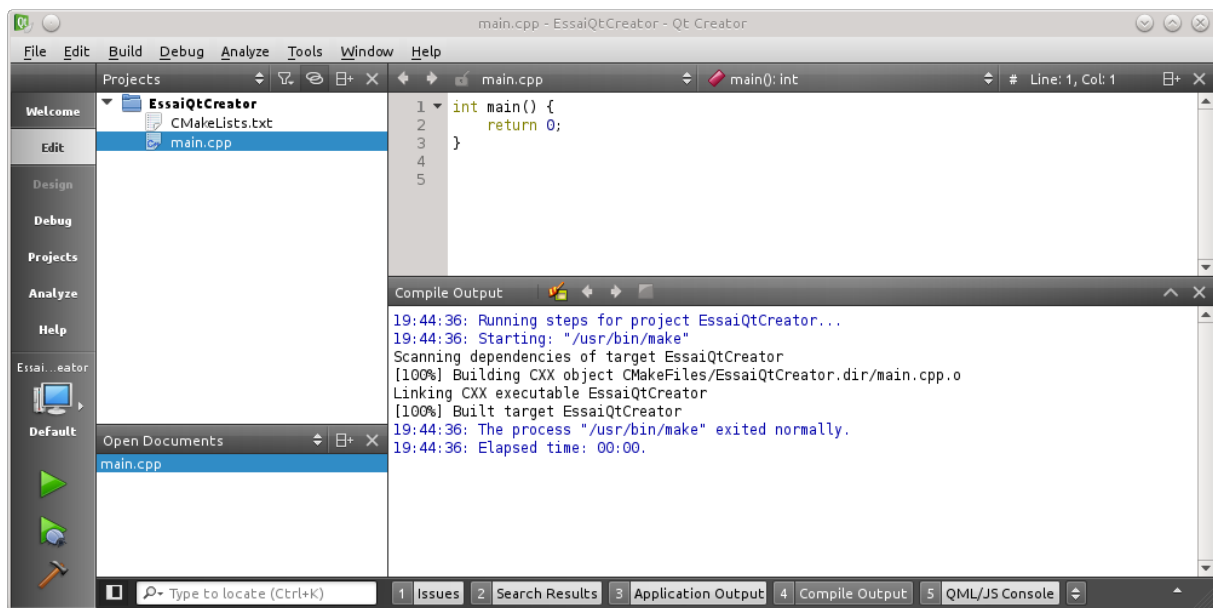


FIGURE 2.1 – QtCreator après compilation du programme de test

En coulisses, QtCreator a lancé le programme CMake et on voit les messages d'information de ce programme. Allons voir dans ce dossier `/home/pascal/TEMP/Build`. On y trouve entre autres le fichier `CMakeCache.txt`, qui a été généré. Celui-ci contient des variables que nous changerons plus tard. Dans QtCreator, cliquer sur le marteau pour lancer la phase de compilation. C'est la même effet que lancer l'option "Build all" du menu "Build". On peut constater Figure 2.1 dans la fenêtre "Compile Output" ce qu'il a fait :

- créer le fichier `main.cpp.o` en compilant `main.cpp` ;
- édition de liens "Linking..." ;
- création du programme exécutable `EssaiQtCreator`.

On peut vérifier l'existence de ce programme dans le répertoire de build. Le triangle vert permet de lancer le programme depuis QtCreator qui indique dans la fenêtre "Application Output" :

```
Starting /home/pascal/TEMP/Build/EssaiQtCreator...
/home/pascal/TEMP/Build/EssaiQtCreator exited with code 0
```

Le programme s'est bien lancé et est sorti avec la valeur 0.

2.5.2 Un peu plus...

On se sert aussi dans le TP du `cout` qui signifie "character output" et qui permet d'afficher ce qu'on veut dans le terminal (`endl` signifie end of line, retour à la ligne). On pourra aussi tester le `cin` qui permet de lire dans le terminal ce qu'entre l'utilisateur :

```
int i=2, j;
cout << "i=" << i << endl;
cout << "Entrez_un_entier:_";
cin >> j;
cout << "Le_double_de_" << j << "_est_" << 2*j << endl;
```

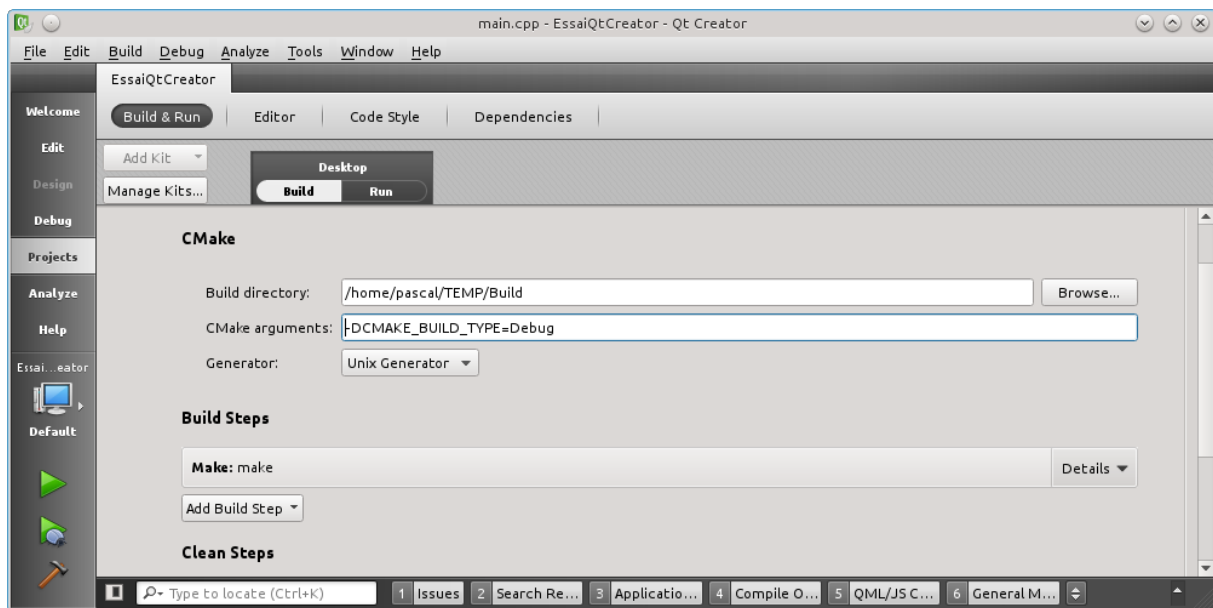


FIGURE 2.2 – QtCreator configuré pour compiler en mode Debug.

A noter que le terminal intégré de QtCreator ne supporte pas l'entrée par l'utilisateur. Il faut donc lancer avec un terminal extérieur, en allant dans l'onglet Projects, rubrique Run et cliquer le bouton "Run in terminal".

Enfin, le TP utilise la commande conditionnelle `if - else`.

2.5.3 Le debugueur

Il est important de pouvoir suivre pas à pas le programme au cours de son exécution, consulter la valeur des variables, etc. Pour cela, il faut compiler dans un mode spécial, dit Debug. Cela se fait avec une variable de CMake, qu'on peut modifier directement dans le fichier `CMakeCache.txt`, ou mieux sans quitter QtCreator : dans l'onglet Projects, passer comme argument à CMake `-DCMAKE_BUILD_TYPE=Debug`, puis recompiler (voir Figure 2.2). La touche F9 permet de mettre un point d'arrêt à la ligne courante.

2.5.4 TP

Vous devriez maintenant aller faire le TP en annexe A.1. Si la pratique est essentielle, en retenir quelque chose est indispensable ! Vous y trouverez aussi comment installer les outils sur votre ordinateur (lien <http://imagine.enpc.fr/~monasse/Imagine++> mentionné à la fin du TP). Voir en Figure 2.3 ce qu'il faut retenir du TP.

—

Nous en savons maintenant assez pour apprendre un peu de C++...

Nous commençons maintenant à nous fabriquer une "fiche de référence" qui servira d'aide mémoire lorsqu'on est devant la machine. Nous la compléterons après chaque chapitre avec ce qui est vu dans le chapitre. Les nouveautés par rapport à la fiche précédente sont en rouge. La fiche finale est en annexe D.

1. Sous Windows, toujours travailler en local et sauvegarder sur le disque partagé, clé USB, etc.
2. Utiliser CMake pour construire une solution Windows.
3. Nettoyer ("clean") quand on quitte.
4. Lancer directement une exécution sauve et génère automatiquement. Attention toutefois de ne pas confirmer l'exécution si la génération s'est mal passée.
5. Double-cliquer sur un message d'erreur positionne l'éditeur sur l'erreur.
6. Toujours bien indenter.
7. Ne pas laisser passer des warnings !
8. Savoir utiliser le debugueur.
9. Touches utiles :

F5 =  = Debug/Continue

F10 =  = Step over

F11 =  = Step inside

Ctrl+A, Ctrl+I (QtCreator) = Indent selection

FIGURE 2.3 – Ce qu'il faut retenir du TP 1

Fiche de référence (1/1)

Entrées/Sorties

```
— #include <iostream>
  using namespace std;
  ...
  cout <<"I="<<i<<endl;
  cin >> i >> j;
```

Clavier

- Debug : F5 
- Step over : F10 
- Step inside : F11 
- Indent : Ctrl+A, Ctrl+I

Conseils

- Nettoyer en quittant.
- Erreurs et warnings : cliquer.
- Indenter.
- Ne pas laisser de warning.
- Utiliser le debugueur.

Chapitre 3

Premiers programmes

Parés à expérimenter au fur et à mesure avec notre environnement de programmation, il est temps d'apprendre les premiers rudiments du C++. Nous allons commencer par programmer n'importe comment... puis nous ajouterons un minimum d'organisation en apprenant à faire des fonctions.

—

On organise souvent un manuel de programmation de façon logique par rapport au langage, en différents points successifs : les expressions, les fonctions, les variables, les instructions, etc. Le résultat est indigeste car il faut alors être exhaustif sur chaque point. Nous allons plutôt ici essayer de voir les choses telles qu'elles se présentent quand on apprend : progressivement et sur un peu tous les sujets à la fois¹ ! Ainsi, ce n'est que dans un autre chapitre que nous verrons la façon dont les fonctions mémorisent leurs variables dans la "pile".

3.1 Tout dans le `main()` !

Rien dans les mains, rien dans les poches... mais tout dans le `main()`. Voici comment un débutant programme².

C'est déjà une étape importante que de programmer *au kilomètre*, en plaçant l'intégralité du programme dans la fonction `main()`. L'essentiel est avant tout de faire un programme qui marche !

3.1.1 Variables

Types

Les **variables** sont des *mémoires* dans lesquelles sont stockées des valeurs (ou données). Une donnée ne pouvant être stockée n'importe comment, il faut à chaque fois décider de la *place prise en mémoire* (nombre d'octets) et du *format*, c'est-à-dire de la façon dont les octets utilisés vont représenter les valeurs prises par la variable. Nous avons

1. La contre-partie de cette présentation est que ce polycopié, s'il est fait pour être lu dans l'ordre, est peut-être moins adapté à servir de manuel de référence. .

2. Et bien des élèves, dès que le professeur n'est plus derrière !

déjà rencontré les **int** qui sont le plus souvent aujourd'hui stockés sur quatre octets, soit 32 bits, et pouvant prendre $2^{32} = 4294967296$ valeurs possibles³. Par convention, les **int** stockent les nombres entiers relatifs⁴, avec autant de nombres négatifs que de nombres positifs⁵, soit, dans le cas de 32 bits⁶, de -2147483648 à 2147483647 suivant une certaine correspondance avec le binaire⁷.

Dire qu'une variable est un **int**, c'est préciser son **type**. Certains langages n'ont pas la notion de type ou essaient de deviner les types des variables. En C++, c'est initialement pour préciser la mémoire et le format des variables qu'elles sont typées. Nous verrons que le compilateur se livre à un certain nombre de vérifications de cohérence de type entre les différentes parties d'un programme. Ces vérifications, pourtant bien pratiques, n'étaient pas faites dans les premières versions du C, petit frère du C++, car avant tout, répétons-le :

Préciser un type, c'est préciser la place mémoire et le format d'une variable. Le compilateur, s'il pourra mettre cette information à profit pour détecter des erreurs de programmation, en a avant tout besoin pour traduire le source C++ en langage machine.

Définition, Affectation, Initialisation, Constantes

Avant de voir d'autres types de variables, regardons sur un exemple la syntaxe à utiliser :

```

1      int i; // Définition
2      i=2;   // Affectation
3      cout << i << " ";
4      int j;
5      j=i;
6      i=1;   // Ne modifie que i, pas j!
7      cout << i << " " << j << " ";
8      int k,l,m; // Définition multiple
9      k=l=3;   // Affectation multiple
10     m=4;
11     cout << k << " " << l << " " << m << " ";
12     int n=5,o=n,p=INT_MAX; // Initialisations
13     cout << n << " " << o << " " << p << endl;
14     int q=r=4; // Erreur!
15     const int s=12;
16     s=13; // Erreur!
```

3. Nous avons aussi vu que cette simple idée donne déjà lieu à deux façons d'utiliser les 4 octets : *big-endian* ou *little-endian*.

4. **Coin des collégiens** : c'est à dire 0, 1, 2, ... mais aussi -1, -2, -3, ...

5. à un près!

6. En fait, les **int** s'adaptent au processeur et un programme compilé sur un processeur 64 bits aura des **int** sur 64 bits ! Si l'on a besoin de savoir dans quel cas on est, le C++ fournit les constantes `INT_MIN` et `INT_MAX` qui sont les valeurs minimales et maximales prises par les **int**.

7. Là, tout le monde fait pareil ! On compte en binaire à partir de 0, et arrivé à 2147483647, le suivant est -2147483648, puis -2147483647 et ainsi de suite jusqu'à -1. On a par exemple : $0 = 000 \dots 000$, $1 = 000 \dots 001$, $2147483647 = 011 \dots 111$, $-2147483648 = 100 \dots 000$, $-2147483647 = 100 \dots 001$, $-2 = 111 \dots 110$, $-1 = 111 \dots 111$

Dans ce programme :

- Les lignes 1 et 2 **définissent** une variable nommée `i`⁸ de type `int` puis **affecte** 2 à cette variable. La représentation binaire de 2 est donc stockée en mémoire là où le compilateur décide de placer `i`. Ce qui suit le "*double slash*" (`//`) est une **remarque** : le compilateur ignore toute la fin de la ligne, ce qui permet de mettre des commentaires aidant à la compréhension du programme.
- La ligne 3 affiche la valeur de `i` puis un espace (sans aller à la ligne)
- Les lignes 4, 5 et 6 définissent un `int` nommé `j`, recopie la valeur de `i`, soit 2, dans `j`, puis mémorise 1 dans `i`. Notez bien que `i` et `j` sont bien deux variables différentes : `i` passe à 1 mais `j` reste à 2 !
- La ligne 8 nous montre comment définir simultanément plusieurs variables du même type.
- La ligne 9 nous apprend que l'on peut affecter des variables simultanément à une même valeur.
- A la ligne 12, des variables sont définies et affectées en même temps. En fait, on parle plutôt de variables **initialisées** : elles prennent une valeur initiale en même temps qu'elles sont définies. Notez que, pour des raisons d'efficacité, **les variables ne sont pas initialisées par défaut** : tant qu'on ne leur a pas affecté une valeur et si elles n'ont pas été initialisées, **elles valent n'importe quoi**⁹ !
- Attention toutefois, il est inutile de tenter une initialisation simultanée. C'est interdit. La ligne 14 provoque une erreur.
- Enfin, on peut rajouter **const** devant le type d'une variable : celle-ci devient alors constante et on ne peut modifier son contenu. La ligne 15 définit une telle variable et la ligne 16 est une erreur.

En résumé, une fois les lignes 14 et 16 supprimées, ce (passionnant !) programme affiche¹⁰ :

```
2 1 2 3 3 4 5 5 2147483647
```

Les noms de variable sont composés uniquement des caractères `a` à `z` (et majuscules), chiffres et underscore `_` (évitez celui-ci, il n'est pas très esthétique), mais ne peuvent pas commencer par un chiffre. N'utilisez pas de caractères accentués, car cela pose des problèmes de portabilité.

Portée

Dans l'exemple précédent, les variables ont été définies au fur et à mesure des besoins. Ce n'est pas une évidence. Par exemple, le C ne permettait de définir les variables que toutes d'un coup au début du `main()`. En C++, on peut définir les variables en cours de route, ce qui permet davantage de clarté. Mais attention :

8. Le *nom* d'une variable est aussi appelé *identificateur*. Les messages d'erreur du compilateur utiliseront plutôt ce vocabulaire !

9. Ainsi, un entier ne vaut pas 0 lorsqu'il est créé et les octets où il est mémorisé gardent la valeur qu'il avaient avant d'être réquisitionnés pour stocker l'entier en question. C'est une mauvaise idée d'utiliser la valeur d'une variable qui vaut n'importe quoi et un compilateur émettra généralement un warning si on utilise une variable avant de lui fournir une valeur !

10. du moins sur une machine 32 bits, cf. remarque précédente sur `INT_MAX`

les variables "n'existent" (et ne sont donc utilisables) qu'à partir de la ligne où elles sont définies. Elles ont une durée de vie limitée et meurent dès que l'on sort du *bloc* limité par des accolades auquel elles appartiennent^a. C'est ce qu'on appelle la *portée* d'une variable.

a. C'est un peu plus compliqué pour les *variables globales*. Nous verrons ça aussi...

Ainsi, en prenant un peu d'avance sur la syntaxe des tests, que nous allons voir tout de suite, le programme suivant provoque des erreurs de portée aux lignes 2 et 8 :

```
int i;
i=j; // Erreur: j n'existe pas encore!
int j=2;
if (j>1) {
    int k=3;
    j=k;
}
i=k; // Erreur: k n'existe plus.
```

Autres types

Nous verrons les différents types au fur et à mesure. Voici malgré tout les plus courants :

```
int i=3;           // Entier relatif
double x=12.3;     // Nombre réel (double précision)
char c='A';        // Caractère
string s="hop";    // Chaîne de caractères
bool t=true;       // Booléen (vrai ou faux)
```

Les nombres réels sont en général approchés par des variables de type **double** ("double précision", ici sur 8 octets). Les caractères sont représentés par un entier sur un octet (sur certaines machines de -128 à 127, sur d'autres de 0 à 255), la correspondance caractère/entier étant celle du code ASCII (65 pour A, 66 pour B, etc.), qu'il n'est heureusement pas besoin de connaître puisque la syntaxe 'A' entre simples guillemets est traduite en 65 par le compilateur, etc. Les doubles guillemets sont eux réservés aux "chaînes" de caractères¹¹. Enfin, les booléens sont des variables qui valent vrai (**true**) ou faux (**false**).

Voici, pour information, quelques types supplémentaires :

```
float y=1.2f;      // Nombre réel simple précision
unsigned int j=4;   // Entier naturel
signed char d=-128; // Entier relatif un octet
unsigned char d=254; // Entier naturel un octet
complex<double> z(2,3); // Nombre complexe
```

où l'on trouve :

- les **float**, nombres réels moins précis mais plus courts que les **double**, ici sur 4 octets (Les curieux pourront explorer la documentation de Visual et voir que les

11. Attention, l'utilisation des string nécessite un `#include<string>` au début du programme.

float valent au plus FLT_MAX (ici, environ 3.4×10^{38} ¹²) et que leur valeur la plus petite strictement positive est FLT_MIN (ici, environ 1.2×10^{-38}), de même que pour les **double** les constantes DBL_MAX et DBL_MIN valent ici environ 1.8×10^{308} et 2.2×10^{-308} ,

- les **unsigned int**, entiers positifs utilisés pour aller plus loin que les **int** dans les positifs (de 0 à UINT_MAX, soit 4294967295 dans notre cas),
- les **unsigned char**, qui vont de 0 à 255,
- les **signed char**, qui vont de -128 à 127,
- et enfin les nombres complexes¹³.

3.1.2 Tests

Tests simples

Les tests servent à exécuter telle ou telle instruction en fonction de la valeur d'une ou de plusieurs variables. Ils sont toujours entre parenthèses. Le "et" s'écrit `&&`, le "ou" `||`, la négation `!`, l'égalité `==`, la non-égalité `!=`, et les inégalités `>`, `>=`, `<` et `<=`. Si plusieurs instructions doivent être exécutées quand un test est vrai (**if**) ou faux (**else**), on utilise des accolades pour les regrouper. Tout cela se comprend facilement sur l'exemple suivant :

```
if (i==0) // i est-il nul?
    cout << "i_est_nul" << endl;
...
if (i>2) // i est-il plus grand que 2?
    j=3;
else
    j=5; // Si on est ici, c'est que i<=2
...
// Cas plus compliqué!
if (i!=3 || (j==2 && k!=3) || !(i>j && i>k)) {
    // Ici, i est différent de 3 ou alors
    // j vaut 2 et k est différent de 3 ou alors
    // on n'a pas i plus grand a la fois de j et de k
    cout << "Une_première_instruction" << endl;
    cout << "Une_deuxième_instruction" << endl;
}
```

Les variables de type booléen servent à mémoriser le résultat d'un test :

```
bool t= ((i==3) || (j==4));
if (t)
    k=5;
```

12. **Coin des collégiens** : 10^{38} ou 1×10^{38} vaut 1 suivi de 38 zéros, 10^{-38} ou 1×10^{-38} vaut 0.000...01 avec 37 zéros avant le 1. En compliquant : 3.4×10^{38} vaut 34 suivis de 37 zéros (38 chiffres après le 3) et 1.2×10^{-38} vaut 0.00...012 toujours avec 37 zéros entre la virgule et le 1 (le 1 est à la place 38).

13. Il est trop tôt pour comprendre la syntaxe "objet" de cette définition mais il nous paraît important de mentionner dès maintenant que les complexes existent en C++.

Coin des collégiens : pas de panique ! Vous apprendrez ce que sont les nombres complexes plus tard. Ils ne seront pas utilisés dans ce livre.

Enfin, une dernière chose très importante : penser à utiliser `==` et non `=` sous peine d'avoir des surprises.¹⁴ C'est peut-être l'erreur la plus fréquente chez les débutants. Elle est heureusement signalée aujourd'hui par un warning...

Attention : utiliser `if (i==3)` et non `if (i=3)` !

Le "switch"

On a parfois besoin de faire telle ou telle chose en fonction des valeurs possibles d'une variable. On utilise alors souvent l'instruction **switch** pour des raisons de clarté de présentation. Chaque cas possible pour les valeurs de la variable est précisé avec **case** et **doit se terminer par break**¹⁵. Plusieurs **case** peuvent être utilisés pour préciser un cas multiple. Enfin, le *mot clé* **default**, à placer en dernier, correspond aux cas non précisés. Le programme suivant¹⁶ réagit aux touches tapées au clavier et utilise un **switch** pour afficher des commentaires passionnants !

```

1 #include <iostream>
2 using namespace std;
3 #include <conio.h> // Non standard !
4
5 int main()
6 {
7     bool fini=false;
8     char c;
9     do {
10         c=_getch(); // Non standard !
11         switch (c) {
12             case 'a':
13                 cout << "Vous_avez_tapé_'a' !" << endl;
14                 break;
15             case 'f':
16                 cout << "Vous_avez_tapé_'f'._Au_revoir !" << endl;
17                 fini=true;
18                 break;
19             case 'e':
20             case 'i':
21             case 'o':
22             case 'u':
23             case 'y':
24                 cout << "Vous_avez_tapé_une_autre_voyelle !" << endl;

```

14. Faire `if (i=3)` affecte 3 à `i` puis renvoie 3 comme résultat du test, ce qui est considéré comme vrai car la convention est qu'un booléen est en fait un entier, faux s'il est nul et vrai s'il est non nul !

15. C'est une erreur grave et fréquente d'oublier le **break**. Sans lui, le programme exécute aussi les instructions du cas suivant !

16. Attention, un `cin >> c`, instruction que nous verrons plus loin, lit bien un caractère au clavier mais ne réagit pas à chaque touche : il attend qu'on appuie sur la touche *Entrée* pour lire d'un coup toutes les touches frappées ! Récupérer juste une touche à la console n'est malheureusement pas standard et n'est plus très utilisé dans notre monde d'interfaces graphiques. Sous Windows, il faudra utiliser `_getch()` après avoir fait un `#include <conio.h>` (cf. lignes 3 et 10) et sous Unix `getch()` après avoir fait un `#include <curses.h>`.


```

25         break;
26     default:
27         cout << "Vous_avez_tapé_autre_chose!" << endl;
28         break;
29     }
30 } while (!fini);
31 return 0;
32 }

```

Si vous avez tout compris, le **switch** précédant ceci est équivalent à¹⁷ :

```

if (c=='a')
    cout << "Vous_avez_tapé_'a' !" << endl;
else if (c=='f') {
    cout << "Vous_avez_tapé_'f'._Au_revoir!" << endl;
    fini=true;
} else if (c=='e' || c=='i' || c=='o' || c=='u' || c=='y')
    cout << "Vous_avez_tapé_une_autre_voyelle!" << endl;
else
    cout << "Vous_avez_tapé_autre_chose!" << endl;

```

Avant tout, rappelons la principale source d'erreur du **switch** :

Dans un switch, ne pas oublier les break !

Vous avez pu remarquer cette ligne 2 un peu cryptique. Un **namespace** est un préfixe pour certains objets. Le préfixe des objets standard du langage est `std`. Ainsi `cout` et `endl` ont pour nom complet `std::cout` et `std::endl`. La ligne 2 permet d'omettre ce préfixe.

3.1.3 Boucles

Il est difficile de faire un programme qui fait quelque chose sans avoir la possibilité d'exécuter plusieurs fois la même instruction. C'est le rôle des boucles. La plus utilisée est le **for()**, mais ça n'est pas la plus simple à comprendre. Commençons par le **do ... while**, qui "tourne en rond" tant qu'un test est vrai. Le programme suivant attend que l'utilisateur tape au clavier un entier entre 1 et 10, et lui réitère sa question jusqu'à obtenir un nombre correct :

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int i;
7     do { // Début de la boucle
8         cout << "Un_nombre_entre_1_et_10,_SVP:_ ";

```

17. On voit bien que le **switch** n'est pas toujours plus clair ni plus court. C'est comme tout, il faut l'utiliser à bon escient... Et plus nous connaissons de C++, plus nous devons nous rappeler cette règle et éviter de faire des fonctions pour tout, des structures de données pour tout, des objets pour tout, des fichiers séparés pour tout, etc.

```

9      cin >> i;
10     } while (i<1 || i>10); // Retourne au début de la boucle si
11                               // ce test est vrai
12     cout << "Merci!_Vous_avez_tapé_" << i << endl;
13     return 0;
14 }

```

Notez la ligne 9 qui met dans `i` un nombre tapé au clavier. La variable `cin` est le pendant en entrée ("console in") de la sortie `cout`.

Vient ensuite le **while** qui vérifie le test au début de la boucle. Le programme suivant affiche les entiers de 1 à 100 :

```

int i=1;
while (i<=100) {
    cout << i << endl;
    i=i+1;
}

```

Enfin, on a créé une boucle spéciale tant elle est fréquente : le **for()** qui exécute une instruction avant de démarrer, effectue un test au début de chaque tour, comme le **while**, et exécute une instruction à la fin de chaque boucle. Instruction initiale, test et instruction finale sont séparées par un `;`, ce qui donne le programme suivant, absolument équivalent au précédent :

```

int i;
for (i=1;i<=100;i=i+1) {
    cout << i << endl;
}

```

En général, le **for()** est utilisé comme dans l'exemple précédent pour effectuer une boucle avec une variable (un *indice*) qui prend une série de valeurs dans un certain intervalle. On trouvera en fait plutôt :

```

for (int i=1;i<=100;i++)
    cout << i << endl;

```

quand on sait que :

- On peut définir la variable dans la première partie du **for()**. Attention, cette variable admet le **for()** pour portée : elle n'est plus utilisable en dehors du **for()** ¹⁸.
- `i++` est une abbréviation de `i=i+1`
- Puisqu'il n'y a ici qu'une seule instruction dans la boucle, les accolades étaient inutiles.

On utilise aussi la virgule `,` pour mettre plusieurs instructions ¹⁹ dans l'instruction finale du **for**. Ainsi, le programme suivant part de `i=1` et `j=100`, et augmente `i` de 2 et diminue `j` de 3 à chaque tour jusqu'à ce que leurs valeurs se croisent ²⁰ :

18. Les vieux C++ ne permettaient pas de définir la variable dans la première partie du **for()**. Des C++ un peu moins anciens permettaient de le faire mais la variable survivait au **for()** !

19. Pour les curieux : ça n'a en fait rien d'extraordinaire, car plusieurs instructions séparées par une virgule deviennent en C++ une seule instruction qui consiste à exécuter l'une après l'autre les différentes instructions ainsi rassemblées.

20. Toujours pour les curieux, il s'arrête pour `i=39` et `j=43`.

```

    for (int i=1, j=100; j>i; i=i+2, j=j-3)
        cout << i << " " << j << endl;

```

Notez aussi qu'on peut abréger `i=i+2` en `i+=2` et `j=j-3` en `j-=3`.

3.1.4 Récréations

Nous pouvons déjà faire de nombreux programmes. Par exemple, jouer au juste prix. Le programme choisit le prix, et l'utilisateur devine :

```

1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 int main()
6 {
7     int n=rand()%100; // nombre à deviner entre 0 et 99
8     int i;
9     do {
10         cout << "Votre_prix:_";
11         cin >> i;
12         if (i>n)
13             cout << "C'est_moins" << endl;
14         else if (i<n)
15             cout << "C'est_plus" << endl;
16         else
17             cout << "Gagne!" << endl;
18     } while (i!=n);
19     return 0;
20 }

```

Seule la ligne 7 a besoin d'explications :

- la fonction `rand()` fournit un nombre entier au hasard entre 0 et `RAND_MAX`. On a besoin de rajouter `#include <cstdlib>` pour l'utiliser
- `%` est la fonction modulo²¹.

C'est évidemment plus intéressant quand c'est le programme qui devine. Pour cela, il va procéder par *dichotomie*, afin de trouver au plus vite :

```

1 #include<iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Choisissez_un_nombre_entre_1_et_100" << endl;
7     cout << "Repondez_par_+,_-_ou_" << endl;
8     int a=1,b=100; // Valeurs extrêmes

```

21. **Coin des collégiens** : compter "modulo N", c'est retomber à 0 quand on atteint N. Modulo 4, cela donne : 0,1,2,3,0,1,2,3,0,... Par exemple `12%10` vaut 2 et `11%3` aussi ! Ici, le modulo 100 sert à retomber entre 0 et 99.

```

9      bool trouve=false;
10     do {
11         int c=(a+b)/2; // On propose le milieu
12         cout << "Serait-ce_" << c << "?:_" ;
13         char r;
14         do
15             cin >> r;
16         while (r!='=' && r!='+' && r!='-');
17         if (r=='=')
18             trouve=true;
19         else if (r=='-')
20             b=c-1; // C'est moins, on essaie entre a et c-1
21         else
22             a=c+1; // C'est plus, on essaie entre c+1 et b
23     } while (!trouve && (a<=b));
24     if (trouve)
25         cout << "Quel_boss_je_suis!" << endl;
26     else
27         cout << "Vous_avez_triché!" << endl;
28     return 0;
29 }

```

On peut aussi compléter le programme "supplémentaire" du TP de l'annexe A.1. Il s'agissait d'une balle rebondissant dans un carré. (Voir l'annexe B pour les instructions graphiques...)

```

1  #include <Imagine/Graphics.h>
2  using namespace Imagine;
3
4  int main()
5  {
6      int w=300,h=210;
7      openWindow(w,h); // Fenêtre graphique
8      int i=0,j=0;       // Position
9      int di=2,dj=3;    // Vitesse
10     while (true) {
11         fillRect(i,j,4,4,RED); // Dessin de la balle
12         milliSleep(10); // On attend un peu...
13         if (i+di>w || i+di<0) {
14             di=-di; // Rebond horizontal si on sort
15         }
16         int ni=i+di; // Nouvelle position
17         if (j+dj>h || j+dj<0) {
18             dj=-dj; // Rebond vertical si on sort
19         }
20         int nj=j+dj;
21         fillRect(i,j,4,4,WHITE); // Effacement
22         i=ni; // On change de position
23         j=nj;

```

```

24     }
25     endGraphics ();
26     return 0;
27 }

```

Notez ce `endGraphics()` dont la fonction est d'attendre un clic de l'utilisateur avant de terminer le programme, de manière à laisser la fenêtre visible le temps nécessaire. Cette fonction n'est pas standard, et elle est dans le namespace `Imagine`. La ligne 2 permet de l'appeler sans utiliser son nom complet `Imagine::endGraphics()`. Les autres fonctions appelées dans ce petit programme (`openWindow`, `fillRect` et `milliSleep`) sont aussi fournies par `Imagine`.

3.2 Fonctions

Lorsqu'on met tout dans le `main()` on réalise très vite que l'on fait souvent des *copier/coller* de bouts de programmes. Si des lignes de programmes commencent à se ressembler, c'est qu'on est vraisemblablement devant l'occasion de faire des fonctions. On le fait pour des raisons de clarté, mais aussi pour faire des économies de frappe au clavier !

Il faut regrouper les passages identiques en fonctions :

- pour obtenir un programme clair...
- et pour moins se fatiguer !

Attention à bien comprendre quand faire une fonction et à ne pas simplement découper un programme en petits morceaux sans aucune logique^a.

^a. ou juste pour faire plaisir au professeur. Mal découper un programme est la meilleure façon de ne plus avoir envie de le faire la fois suivante. Encore une fois, le bon critère est ici que la bonne solution est généralement la moins fatigante.

En fait, pouvoir réutiliser le travail déjà fait est le fil conducteur d'une bonne programmation. Pour l'instant, nous nous contentons, grâce aux fonctions, de réutiliser ce que nous venons de taper quelques lignes plus haut. Plus tard, nous aurons envie de réutiliser ce qui aura été fait dans d'autres programmes, ou longtemps auparavant, ou dans les programmes d'autres personnes, et nous verrons alors comment faire.

Prenons le programme suivant, qui dessine des traits et des cercles au hasard, et dont la figure 3.1 montre un résultat :

```

1  #include <Imagine/Graphics.h>
2  using namespace Imagine;
3  #include <cstdlib>
4  using namespace std;
5
6  int main()
7  {
8      openWindow(300,200);
9      for (int i=0;i<150;i++) {
10         int x1=rand()%300; // Point initial
11         int y1=rand()%200;

```

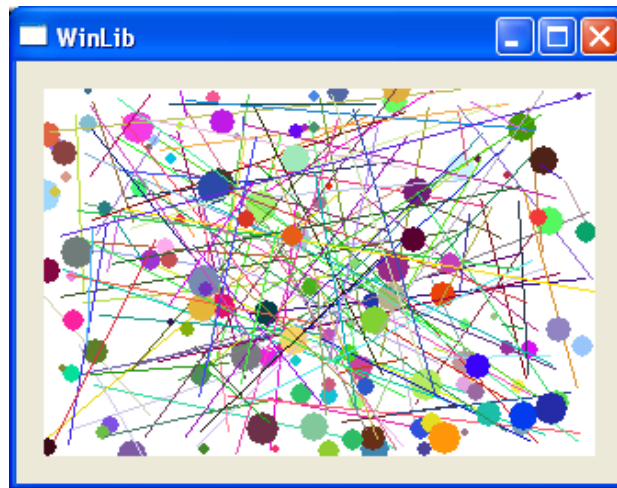


FIGURE 3.1 – Traits et cercles au hasard...

```

12     int x2=rand()%300; // Point final
13     int y2=rand()%200;
14     Color c=Color(rand()%256,rand()%256,rand()%256); // RVB
15     drawLine(x1,y1,x2,y2,c); // Tracé de segment
16     int xc=rand()%300; // Centre du cercle
17     int yc=rand()%200;
18     int rc=rand()%10; // Rayon
19     Color cc=Color(rand()%256,rand()%256,rand()%256); // RVB
20     fillCircle(xc,yc,rc,cc); // Cercle
21 }
22 endGraphics();
23 return 0;
24 }

```

La première chose qui choque²², c'est l'appel répété à `rand()` et à modulo pour tirer un nombre au hasard. On aura souvent besoin de tirer des nombres au hasard dans un certain intervalle et il est naturel de le faire avec une fonction. Au passage, nous corrigeons une deuxième chose qui choque : les entiers 300 et 200 reviennent souvent. Si nous voulons changer les dimensions de la fenêtre, il faudra remplacer dans le programme tous les 300 et tous les 200. Il vaudrait mieux mettre ces valeurs dans des variables et faire dépendre le reste du programme de ces variables. C'est un défaut constant de tous les débutants et il faut le corriger tout de suite.

Il faut dès le début d'un programme repérer les paramètres constants utilisés à plusieurs reprises et les placer dans des variables dont dépendra le programme. On gagne alors beaucoup de temps^a quand on veut les modifier par la suite.

^a. Encore la règle du moindre effort... Si on fait trop de *copier/coller* ou de *remplacer* avec l'éditeur, c'est mauvais signe !

22. à part évidemment la syntaxe "objet" des variables de type `Color` pour lesquelles on se permet un `Color(r,v,b)` bien en avance sur ce que nous sommes censés savoir faire...

Bref, notre programme devient :

```
// Nombre entre 0 et n-1
int hasard(int n)
{
    return rand()%n;
}

int main()
{
    const int w=300,h=200;
    openWindow(w,h);
    for (int i=0;i<150;i++) {
        int x1=hasard(w),y1=hasard(h); // Point initial
        int x2=hasard(w),y2=hasard(h); // Point final
        Color c=Color(hazard(256),hazard(256),hazard(256));
        drawLine(x1,y1,x2,y2,c); // Tracé de segment
        int xc=hasard(w),yc=hasard(h); // Centre du cercle
        int rc=hasard(w/20); // Rayon
        Color cc=Color(hazard(256),hazard(256),hazard(256));
        fillCircle(xc,yc,rc,cc); // Cercle
    }
    endGraphics();
    return 0;
}
```

On pourrait penser que hasard(w) est aussi long à taper que rand()%w et que notre fonction est inutile. C'est un peu vrai. Mais en pratique, nous n'avons alors plus à nous souvenir de l'existence de la fonction rand() ni de comment on fait un modulo. C'est même mieux que ça : nous devenons indépendant de ces deux fonctions, et si vous voulions tirer des nombres au hasard avec une autre fonction²³, nous n'aurions plus qu'à modifier la fonction hasard(). C'est encore une règle importante.

On doit également faire une fonction quand on veut séparer et factoriser le travail. Il est ensuite plus facile^a de modifier la fonction que toutes les lignes qu'elle a remplacées !

a. Moindre effort, toujours !

3.2.1 Retour

Nous venons de définir sans l'expliquer une fonction hasard() qui prend un paramètre n de type **int** et qui retourne un résultat, de type **int** lui aussi. Il n'y a pas grand chose à savoir de plus, si ce n'est que :

23. Pourquoi vouloir le faire ? Dans notre cas parce que la fonction rand() utilisée est suffisante pour des applications courantes mais pas assez précise pour des applications mathématiques. Par exemple, faire un modulo ne répartit pas vraiment équitablement les nombres tirés. Enfin, nous avons oublié d'initialiser le générateur aléatoire. Si vous le permettez, nous verrons une autre fois ce que cela signifie et comment le faire en modifiant juste la fonction hasard().

1. Une fonction peut ne rien renvoyer. Son type de retour est alors **void** et il n'y a pas de **return** à la fin. Par exemple :

```
void dis_bonjour_a_la_dame(string nom_de_la_dame) {
    cout << "Bonjour ,_Mme_" << nom_de_la_dame << "!" << endl;
}

...
dis_bonjour_a_la_dame("Germaine");
dis_bonjour_a_la_dame("Fitzgerald");
```

2. Une fonction peut comporter plusieurs instructions **return**²⁴. Cela permet de sortir quand on en a envie, ce qui est bien plus clair et plus proche de notre façon de penser :

```
int signe_avec_un_seul_return(double x) {
    int s;
    if (x==0)
        s=0;
    else if (x<0)
        s=-1;
    else
        s=1;
    return s;
}

int signe_plus_simple(double x) {
    if (x<0)
        return -1;
    if (x>0) // Notez l'absence de else , devenu inutile!
        return 1;
    return 0;
}
```

3. Pour une fonction **void**, on utilise **return** sans rien derrière pour un retour en cours de fonction :

```
void telephoner_avec_un_seul_return(string nom) {
    if (j_ai_le_telephone) {
        if (mon_telephone_marche) {
            if (est_dans_l_annuaire(nom)) {
                int numero=numero_telephone(nom);
                composer(numero);
                if (ca_decroche) {
                    parler();
                    raccrocher();
                }
            }
        }
    }
}
```

24. Contrairement à certains vieux langages, comme le Pascal


```

}
void telephoner_plus_simple(string nom) {
    if (!j_ai_le_telephone)
        return;
    if (!mon_telephone_marche)
        return;
    if (!est_dans_l_annuaire(nom))
        return;
    int numero=numero_telephone(nom);
    composer(numero);
    if (!ca_decroche)
        return;
    parler();
    raccrocher();
}

```

3.2.2 Paramètres

Nous n'avons vu que des fonctions à un seul paramètre. Voici comment faire pour en passer plusieurs ou n'en passer aucun :

```

// Nombre entre a et b
int hasard2(int a,int b)
{
    return a+(rand()%(b-a+1));
}

// Nombre entre 0 et 1
double hasard3()
{
    return rand()/double(RAND_MAX);
}

...
int a=hasard2(1,10);
double x=hasard3();
...

```

Attention à bien utiliser `x=hasard3()` et non simplement `x=hasard3` pour appeler cette fonction sans paramètre. Ce simple programme est aussi l'occasion de parler d'une erreur très fréquente : la division de deux nombres entiers donne un nombre entier ! Ainsi, écrire `double x=1/3;` est une erreur car le C++ commence par calculer `1/3` avec des entiers, ce qui donne 0, puis convertit 0 en `double` pour le ranger dans x. *Il ne sait pas au moment de calculer 1/3 qu'on va mettre le résultat dans un double !* Il faut alors faire en sorte que le 1 ou le 3 soit une `double` et écrire `double x=1.0/3;` ou `double x=1/3.0;`. Si, comme dans notre cas, on a affaire à deux variables de type `int`, il suffit de convertir une de ces variables en `double` avec la syntaxe `double (...)` que nous verrons plus tard.

1. **Fonction sans paramètre : `x=hop()`; et non `x=hop`;**
2. **Division entière :**
 - **`double x=1.0/3`; et non `double x=1/3`;**
 - **`double x=double(i)/j`; et non `double x=i/j`;; ni même `double x=double(i/j)`;^a**

a. Cette conversion en **double** arrive trop tard !

3.2.3 Passage par référence

Lorsqu'une fonction modifie la valeur d'un de ses paramètres, et si ce paramètre était une variable dans la fonction appelante, alors la variable en question n'est pas modifiée. Plus clairement, le programme suivant échoue :

```
void triple(int x) {
    x=x*3;
}
...
int a=2;
triple(a);
cout << a << endl;
```

Il affiche 2 et non 6. En fait, le paramètre `x` de la fonction `triple` vaut bien 2, puis 6. Mais son passage à 6 ne modifie pas `a`. Nous verrons plus loin que `x` est mémorisé à un endroit différent de `a`, ce qui explique tout ! C'est la valeur de `a` qui est passée à la fonction `triple()` et non pas la variable `a` ! On parle de **passage par valeur**. On peut toutefois faire en sorte que la fonction puisse vraiment modifier son paramètre. On s'agit alors d'un **passage par référence** (ou *par variable*). Il suffit de rajouter un `&` derrière le type du paramètre :

```
void triple(int& x) {
    x=x*3;
}
```

Généralement, on choisit l'exemple suivant pour justifier le besoin des références :

```
void echanger1(int x,int y) {
    int t=x;
    x=y;
    y=t;
}
void echanger2(int& x,int& y) {
    int t=x;
    x=y;
    y=t;
}
...
int a=2,b=3;
echanger1(a,b);
cout << a << " " << b << " ";
```

```

    echanger2(a,b);
    cout << a << " " << b << endl;
...

```

Ce programme affiche 2 3 3 2, echanger1() ne marchant pas.

Une bonne façon de comprendre le passage par référence est de considérer que les variables `x` et `y` de `echanger1` sont des variables vraiment indépendantes du `a` et du `b` de la fonction appelante, alors qu'au moment de l'appel à `echanger2`, le `x` et le `y` de `echanger2` deviennent des "liens" avec `a` et `b`. A chaque fois que l'on utilise `x` dans `echanger2`, c'est en fait `a` qui est utilisée. Pour encore mieux comprendre **allez voir le premier exercice du TP 2 (A.2.1)** et sa solution.

En pratique,

on utilise aussi les références pour faire des fonctions retournant plusieurs valeurs à la fois,

et ce, de la façon suivante :

```

void un_point(int& x, int& y) {
    x = ...;
    y = ...;
}
...
    int a,b;
    un_point(a,b);

```

Ainsi, notre programme de dessin aléatoire deviendrait :

```

1 #include <Imagine/Graphics.h>
2 using namespace Imagine;
3 #include <cstdlib>
4 using namespace std;
5
6 // Nombre entre 0 et n-1
7 int hasard(int n)
8 {
9     return rand()%n;
10 }
11
12 Color une_couleur() {
13     return Color(hazard(256),hazard(256),hazard(256));
14 }
15
16 void un_point(int w,int h,int& x,int& y){
17     x=hasard(w);
18     y=hasard(h);
19 }
20
21 int main()
22 {
23     const int w=300,h=200;

```

```

24  openWindow(w,h);
25  for (int i=0;i<150;i++) {
26      int x1,y1; // Point initial
27      un_point(w,h,x1,y1);
28      int x2,y2; // Point final
29      un_point(w,h,x2,y2);
30      Color c=une_couleur();
31      drawLine(x1,y1,x2,y2,c); // Tracé de segment
32      int xc,yc; // Centre du cercle
33      un_point(w,h,xc,yc);
34      int rc=hasard(w/20); // Rayon
35      Color cc=une_couleur();
36      fillCircle(xc,yc,rc,cc); // Cercle
37  }
38  endGraphics();
39  return 0;
40 }

```

Avec le conseil suivant :

penser à utiliser directement le résultat d'une fonction et ne pas le mémoriser dans une variable lorsque c'est inutile.

Il devient même :

```

26  int x1,y1; // Point initial
27  un_point(w,h,x1,y1);
28  int x2,y2; // Point final
29  un_point(w,h,x2,y2);
30  drawLine(x1,y1,x2,y2,une_couleur()); // Tracé de segment
31  int xc,yc; // Centre du cercle
32  un_point(w,h,xc,yc);
33  int rc=hasard(w/20); // Rayon
34  fillCircle(xc,yc,rc,une_couleur()); // Cercle

```

3.2.4 Portée, Déclaration, Définition

Depuis le début, nous créons des fonctions en les **définissant**. Il est parfois utile de ne connaître que le type de retour et les paramètres d'une fonction sans pour autant savoir comment elle est programmée, c'est-à-dire sans connaître le *corps* de la fonction. Une des raisons de ce besoin est que :

comme les variables, les fonctions ont une portée et ne sont connues que dans les lignes de source qui lui succèdent.

Ainsi, le programme suivant ne compile pas :

```

1  int main()
2  {
3      f();

```

```

4     return 0;
5 }
6 void f() {
7 }

```

car à la ligne 3, `f()` n'est pas connue. Il suffit ici de mettre les lignes 6 et 7 avant le `main()` pour que le programme compile. Par contre, il est plus difficile de faire compiler :

```

void f()
{
    g(); // Erreur: g() inconnue
}

void g() {
    f();
}

```

puisque les deux fonctions ont besoin l'une de l'autre, et qu'aucun ordre ne conviendra. Il faut alors connaître la règle suivante :

- Remplacer le corps d'une fonction par un ; s'appelle *déclarer la fonction*.
- Déclarer une fonction suffit au compilateur, qui peut "patienter"^a jusqu'à sa *définition*.

a. En réalité, le compilateur n'a besoin que de la déclaration. C'est le linker qui devra trouver quelque part la définition de la fonction, ou plus exactement le résultat de la compilation de sa définition !

Notre programme précédent peut donc se compiler avec une ligne de plus :

```

void g(); // Déclaration de g

void f()
{
    g(); // OK: fonction déclarée
}

void g() { // Définition de g
    f();
}

```

3.2.5 Variables locales et globales

Nous avons vu section 3.1.1 la portée des variables. La règle des accolades s'applique évidemment aux accolades du corps d'une fonction.

Les variables d'une fonction sont donc inconnues en dehors de la fonction

On parle alors de **variables locales** à la fonction. Ainsi, le programme suivant est interdit :

```

void f()
{
    int x;
    x=3;
}

void g() {
    int y;
    y=x; // Erreur: x inconnu
}

```

Si vraiment deux fonctions utilisent des variables communes, il faut alors les "sortir" des fonctions. Elles deviennent alors des **variables globales**, dont voici un exemple :

```

1 int z; // globale
2
3 void f()
4 {
5     int x; // locale
6     ...
7     if (x<z)
8         ...
9 }
10
11 void g()
12 {
13     int y; // locale
14     ...
15     z=y;
16     ...
17 }

```

L'utilisation de variables globales est tolérée et parfois justifiée. Mais elle constitue une solution de facilité dont les débutants abusent et il faut combattre cette tentation dès le début :

les variables globales sont à éviter au maximum car

- elles permettent parfois des communications abusives entre fonctions, sources de bugs^a.
- les fonctions qui les utilisent sont souvent peu réutilisables dans des contextes différents.

En général, elles sont le signe d'une mauvaise façon de traiter le problème.

^a. C'est pourquoi les variables globales non constantes ne sont pas tolérées chez le débutant. Voir le programme précédent où g() parle à f() au travers de z.

3.2.6 Surcharge

Il est parfois utile d'avoir une fonction qui fait des choses différentes suivant le type d'argument qu'on lui passe. Pour cela on peut utiliser la *surcharge* :

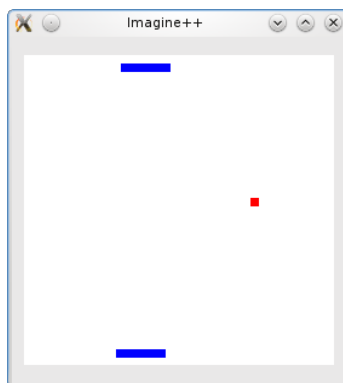


FIGURE 3.2 – Mini tennis...

Deux fonctions qui ont des listes de paramètres différentes peuvent avoir le même nom^a. Attention : deux fonctions aux types de retour différents mais aux paramètres identiques ne peuvent avoir le même nom^b.

a. Car alors la façon de les appeler permettra au compilateur de savoir laquelle des fonctions on veut utiliser

b. Car alors le compilateur ne pourra différencier leurs appels.

Ainsi, nos fonctions "hasard" de tout à l'heure peuvent très bien s'écrire :

```

1 // Nombre entre 0 et n-1
2 int hasard(int n) {
3     return rand()%n;
4 }
5 // Nombre entre a et b
6 int hasard(int a,int b) {
7     return a+(rand()%(b-a+1));
8 }
9 // Nombre entre 0 et 1
10 double hasard() {
11     return rand()/double(RAND_MAX);
12 }
13 ...
14 int i=hasard(3); // entre 0 et 2
15 int j=hasard(2,4) // entre 2 et 4
16 double k=hasard(); // entre 0 et 1

```

3.3 TP

Nous pouvons maintenant aller faire le deuxième TP donné en annexe A.2 afin de mieux comprendre les fonctions et aussi pour obtenir un mini jeu de tennis (figure 3.2).

3.4 Fiche de référence

Fiche de référence (1/2)		
Variables — Définition : <pre>int i; int k,l,m;</pre> — Affectation : <pre>i=2; j=i; k=l=3;</pre> — Initialisation : <pre>int n=5,o=n;</pre> — Constantes : <pre>const int s=12;</pre> — Portée : <pre>int i; // i=j; interdit! int j=2; i=j; // OK! if (j>1) { int k=3; j=k; // OK! } //i=k; interdit!</pre> — Types : <pre>int i=3; double x=12.3; char c='A'; string s="hop"; bool t=true; float y=1.2f; unsigned int j=4; signed char d=-128; unsigned char d=25; complex<double> z(2,3);</pre> — Variables globales : <pre>int n; const int m=12; void f() { n=10; // OK int i=m; // OK ...</pre> — Conversion : <pre>int i=int(x),j; float x=float(i)/j;</pre> Tests — Comparaison : <pre>== != < > <= >=</pre>	— Négation : ! — Combinaisons : && — if (i==0) j=1; — if (i==0) j=1; else j=2; — if (i==0) { j=1; k=2; } — bool t=(i==0); if (t) j=1; — switch (i) { case 1: ...; ...; break; case 2: case 3: ...; break; default: ...; } Boucles — do { ... } while(!ok); — int i=1; while(i<=100) { ... i=i+1; } — for(int i=1;i<=10;i++) ... — for(int i=1,j=10;j>i; i=i+2,j=j-3) ... Fonctions — Définition : <pre>int plus(int a,int b){ int c=a+b; return c; } void affiche(int a) { cout << a << endl; }</pre>	— Déclaration : <pre>int plus(int a,int b);</pre> — Retour : <pre>int signe(double x) { if (x<0) return -1; if (x>0) return 1; return 0; } void afficher(int x, int y) { if (x<0 y<0) return; if (x>=w y>=h) return; DrawPoint(x,y,RED); } — Appel : <pre>int f(int a) { ... } int g() { ... } ... int i=f(2),j=g();</pre> — Références : <pre>void swap(int& a, int& b){ int tmp=a; a=b;b=tmp; } ... int x=3,y=2; swap(x,y);</pre> — Surcharge : <pre>int hasard(int n); int hasard(int a, int b); double hasard();</pre> Divers — i++; i--; i-=2; j+=3; — j=i%n; // Modulo — #include <cstdlib> ... i=rand()%n; x=rand()/ double(RAND_MAX);</pre>

Fiche de référence (2/2)		
Entrées/Sorties — #include <iostream> using namespace std; ... cout <<"I="<<i<<endl; cin >> i >> j; Erreurs fréquentes — Pas de définition de fonction dans une fonction ! — int q=r=4; // NON! — if (i=2) // NON! if i==2 // NON! if (i==2) then // NON!	— for(int i=0,i<100,i++) // NON! — int f() {...} int i=f; // NON! — double x=1/3; // NON! int i,j; x=i/j; // NON! x=double(i/j); //NON! Imagine++ — Voir documentation... Clavier — Debug : F5	— Step over : F10 — Step inside : F11 — Indent : Ctrl+A, Ctrl+I Conseils — Nettoyer en quittant. — Erreurs et warnings : cliquer. — Indenter. — Ne pas laisser de warning. — Utiliser le debugueur. — Faire des fonctions.

Chapitre 4

Les tableaux

*Tout en continuant à utiliser les fonctions pour les assimiler, nous allons rajouter les **tableaux** qui, sinon, nous manqueraient rapidement. Nous n'irons pas trop vite et ne verrons pour l'instant que les tableaux à une dimension et de taille fixe. Nous étudierons dans un autre chapitre les tableaux de taille variable et les questions de mémoire ("pile" et "tas").*

—

4.1 Premiers tableaux

De même qu'on a tout de suite ressenti le besoin d'avoir des boucles pour faire plusieurs fois de suite la même chose, il a été rapidement utile de faire plusieurs fois la même chose mais sur des variables différentes. D'où les tableaux... Ainsi, le programme suivant :

```
int x1,y1,u1,v1; // Balle 1
int x2,y2,u2,v2; // Balle 2
int x3,y3,u3,v3; // Balle 3
int x4,y4,u4,v4; // Balle 4
...
BougeBalle(x1,y1,u1,v1);
BougeBalle(x2,y2,u2,v2);
BougeBalle(x3,y3,u3,v3);
BougeBalle(x4,y4,u4,v4);
```

pourra avantageusement être remplacé par :

```
int x[4],y[4],u[4],v[4]; // Balles
...
for (int i=0;i<4;i++)
    BougeBalle(x[i],y[i],u[i],v[i]);
```

dans lequel `int x[4]` définit un *tableau* de 4 variables de type `int` : `x[0]`, `x[1]`, `x[2]` et `x[3]`. En pratique, le compilateur réserve quelque part en mémoire de quoi stocker les 4 variables en question et gère de quoi faire en sorte que `x[i]` désigne la bonne variable.

Un autre exemple pour mieux comprendre, qui additionne des **double** deux par deux en mémorisant les résultats :

```
double x[100],y[100],z[100];
...
... // ici, les x[i] et y[i] prennent des valeurs
...
for (int i=0;i<100;i++)
    z[i]=x[i]+y[i];
...
... // ici, on utilise les z[i]
...
```

Il y a deux choses essentielles à retenir.

1. D'abord :

les indices d'un tableau t de taille n vont de 0 à $n-1$. Tout accès à $t[n]$ peut provoquer une erreur grave pendant l'exécution du programme. C'EST UNE DES ERREURS LES PLUS FRÉQUENTES EN C++. Soit on va lire ou écrire dans un endroit utilisé pour une autre variable^a, soit on accède à une zone mémoire illégale et le programme peut "planter"^b.

a. Dans l'exemple ci-dessus, si on remplaçait la boucle pour que i aille de 1 à 100, $x[100]$ irait certainement chercher $y[0]$ à la place. De même, $z[100]$ irait peut-être chercher la variable i de la boucle, ce qui risquerait de faire ensuite des choses étranges, i valant n'importe quoi !

b. Ci-dessus, $z[i]$ avec n'importe quoi pour i irait écrire en dehors de la zone réservée aux données, ce qui stopperait le programme plus ou moins délicatement !

Dans le dernier exemple, on utilise $x[0]$ à $x[99]$. L'habitude est de faire une boucle avec $i<100$ comme test, plutôt que $i\leq 99$, ce qui est plus lisible. Mais attention à ne pas mettre $i\leq 100$!

2. Ensuite :

un tableau doit avoir une taille fixe connue à la compilation. Cette taille peut être un nombre ou une variable constante, mais pas une variable.

Même si on pense que le compilateur pourrait connaître la taille, il joue au plus idiot et n'accepte que des constantes :

```
1 double x[10],y[4],z[5]; // OK
2 const int n=5;
3 int i[n],j[2*n],k[n+1]; // OK
4 int n1; // n1 n'a même pas de valeur
5 int t1[n1]; // donc ERREUR
6 int n2;
7 cin >> n2; // n2 prend une valeur, mais connue
8 // uniquement à l'exécution
9 int t2[n2]; // donc ERREUR
10 int n3;
11 n3=5; // n3 prend une valeur, connue
12 // à l'exécution, mais... non constante
13 int t3[n3]; // donc ERREUR (SI!)
```

Connaissant ces deux points, on peut très facilement utiliser des tableaux. Attention toutefois :

ne pas utiliser de tableau quand c'est inutile, notamment quand on traduit une formule mathématique.

Je m'explique. Si vous devez calculer $s = \sum_{i=1}^{100} f(i)$ pour f donnée¹, par exemple $f(i) = 3i + 4$, n'allez pas écrire, comme on le voit parfois :

```
1 double f[100];
2 for (int i=1; i<=100; i++)
3     f[i]=3*i+4;
4 double s;
5 for (int i=1; i<=100; i++)
6     s=s+f[i];
```

ni, même, ayant corrigé vos bugs :

```
5 double f[100];           // Stocke f(i) dans f[i-1]
6 for (int i=1; i<=100; i++)
7     f[i-1]=3*i+4;        // Attention aux indices!
8 double s=0;              // Ca va mieux comme ça!
9 for (int i=1; i<=100; i++)
10    s=s+f[i-1];
```

mais plutôt directement sans tableau :

```
5 double s=0;
6 for (int i=1; i<=100; i++)
7     s=s+(3*i+4); // Ou mieux: s+=3*i+4
```

ce qui épargnera, à la machine, un tableau (donc de la mémoire et des calculs), et à vous des bugs (donc vos nerfs!). Notez qu'ici on utilise la relation de récurrence

$$s_k = \sum_{i=1}^k f(i) = s_{k-1} + f(k)$$

pour calculer s_{100} . Comme pour calculer s_k on n'a besoin que de garder en mémoire s_{k-1} , on peut se contenter d'une seule variable s qu'on met à jour.

4.2 Initialisation

Tout comme une variable, un tableau peut être initialisé :

```
int t[4]={1,2,3,4};
string s[2]={"hip","hop"};
```

Attention, la syntaxe utilisée pour l'initialisation ne marche pas pour une affectation² :

```
int t[2];
t={1,2}; // Erreur!
```

1. **Coin des collégiens** : c'est-à-dire $s = f(1) + f(2) + \dots + f(100)$.

2. Nous verrons plus bas que l'affectation ne marche même pas entre deux tableaux ! Tout ceci s'arrangera avec les objets...

4.3 Spécificités des tableaux

Les tableaux sont des variables un peu spéciales. Ils ne se comportent pas toujours comme les autres variables³...

4.3.1 Tableaux et fonctions

Tout comme les variables, on a besoin de passer les tableaux en paramètres à des fonctions. La syntaxe à utiliser est simple :

```
void affiche(int s[4]) {
    for (int i=0;i<4;i++)
        cout << s[i] << endl;
}
...
int t[4]={1,2,3,4};
affiche(t);
```

mais il faut savoir deux choses :

- Un tableau est toujours passé par référence bien qu'on n'utilise pas le '&'^a.
- Une fonction ne peut pas retourner un tableau^b.

a. Un `void f(int& t[4])` ou toute autre syntaxe est une erreur.

b. On comprendra plus tard pourquoi, par souci d'efficacité, les concepteurs du C++ ont voulu qu'un tableau ne soit ni passé par valeur, ni retourné.

donc :

```
1 // Rappel: ceci ne marche pas
2 void affecte1(int x,int val) {
3     x=val;
4 }
5 // Rappel: c'est ceci qui marche!
6 void affecte2(int& x,int val) {
7     x=val;
8 }
9 // Une fonction qui marche sans '&'
10 void remplit(int s[4],int val) {
11     for (int i=0;i<4;i++)
12         s[i]=val;
13 }
14 ...
15 int a=1;
```

3. Il est du coup de plus en plus fréquent que les programmeurs utilisent directement des variables de type `vector` qui sont des objets implémentant les fonctionnalités des tableaux tout en se comportant davantage comme des variables standard. Nous préférons ne pas parler dès maintenant des `vector` car leur compréhension nécessite celle des objets et celle des "template". Nous pensons aussi que la connaissance des tableaux, même si elle demande un petit effort, est incontournable et aide à la compréhension de la gestion de la mémoire.

```

16 affecte1(a,0);           // a ne sera pas mis à 0
17 cout << a << endl;      // vérification
18 affecte2(a,0);           // a sera bien mis à 0
19 cout << a << endl;      // vérification
20 int t[4];
21 remplit(t,0);            // Met les t[i] à 0
22 affiche(t);              // Vérifie que les t[i] valent 0
    et aussi :
1  // Somme de deux tableaux qui ne compile même pas
2                                // Pour retourner un tableau
3 int sommel(int x[4],int y[4])[4] { // on peut imaginer mettre le
4                                // [4] ici ou ailleurs :
5                                // rien n'y fait !
6     int z[4];
7     for (int i=0;i<4;i++)
8         z[i]=x[i]+y[i];
9     return z;
10 }
11 // En pratique , on fera donc comme ça !
12 // Somme de deux tableaux qui marche
13 void somme2(int x[4],int y[4],int z[4])
14     for (int i=0;i<4;i++)
15         z[i]=x[i]+y[i]; // OK: 'z' est passé par référence !
16 }
17
18 int a[4],b[4];
19 ...           // remplissage de a et b
20 int c[4];
21 c=sommel(a,b); // ERREUR
22 somme2(a,b,c); // OK

```

Enfin, et c'est utilisé tout le temps,

Une fonction n'est pas tenue de travailler sur une taille de tableau unique...mais il est impossible de demander à un tableau sa taille !

On utilise la syntaxe `int t[]` dans les paramètres pour un tableau dont on ne précise pas la taille. Comme il faut bien parcourir le tableau dans la fonction et qu'on ne peut retrouver sa taille, on la passe en paramètre en plus du tableau :

```

1 // Une fonction qui ne marche pas
2 void affiche1(int t[]) {
3     for (int i=0;i<TAILLE(t);i++) // TAILLE(t) n'existe pas !
4         cout << t[i] << endl;
5 }
6 // Comment on fait en pratique
7 void affiche2(int t[],int n) {
8     for (int i=0;i<n;i++)
9         cout << t[i] << endl;

```

```

10 }
11 ...
12 int t1[2]={1,2};
13 int t2[3]={3,4,5};
14 affiche2(t1,2); // OK
15 affiche2(t2,3); // OK

```

4.3.2 Affectation

C'est simple :

Affecter un tableau ne marche pas ! Il faut traiter les éléments du tableau un par un...

Ainsi, le programme :

```

int s[4]={1,2,3,4}, t[4];
t=s; // ERREUR de compilation

```

ne marche pas et on est obligé de faire :

```

int s[4]={1,2,3,4}, t[4];
for (int i=0;i<4;i++)
    t[i]=s[i]; // OK

```

Le problème, c'est que :

Affecter un tableau ne marche jamais mais ne génère pas toujours une erreur de compilation, ni même un warning. C'est le cas entre deux paramètres de fonction. Nous comprendrons plus tard pourquoi et l'effet exact d'une telle affectation...

```

.
1 // Fonction qui ne marche pas
2 // Mais qui compile très bien!
3 void set1(int s[4],int t[4]) {
4     t=s; // Ne fait pas ce qu'il faut!
5         // mais compile sans warning!
6 }
7 // Fonction qui marche (et qui compile!-)
8 void set2(int s[4],int t[4]) {
9     for (int i=0;i<4;i++)
10         t[i]=s[i]; // OK
11 }
12 ...
13 int s[4]={1,2,3,4}, t[4];
14 set1(s,t); // Sans effet
15 set2(s,t); // OK

```

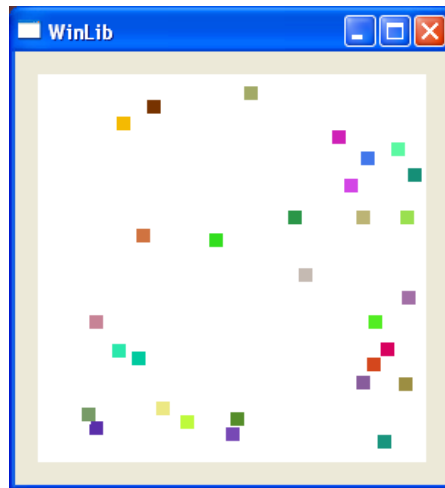



FIGURE 4.1 – Des balles qui rebondissent... (momentanément figées ! Allez sur la page du cours pour un programme animé !)

4.4 Récréations

4.4.1 Multi-balles

Nous pouvons maintenant reprendre le programme de la balle qui rebondit, donné à la section 3.1.4, puis amélioré avec des fonctions et de constantes lors du TP de l'annexe A.2. Grâce aux tableaux, il est facile de faire se déplacer plusieurs balles à la fois. Nous tirons aussi la couleur et la position et la vitesse initiales des balles au hasard. Plusieurs fonctions devraient vous être inconnues :

- L'initialisation du générateur aléatoire avec `srand((unsigned int)time(0))`, qui est expliquée dans le TP 3 (annexe A.3)
- Les fonctions `noRefreshBegin` et `noRefreshEnd` qui servent à accélérer l'affichage de toutes les balles (voir documentation de `Imagine++` annexe B).

Voici le listing du programme (exemple d'affichage (malheureusement statique !) figure 4.1) :

```

1 #include <Imagine/Graphics.h>
2 using namespace Imagine;
3 #include <cstdlib>
4 #include <ctime>
5 using namespace std;
6 //////////////////////////////////////
7 // Constantes du programme
8 const int width=256;    // Largeur de la fenetre
9 const int height=256;  // Hauteur de la fenetre
10 const int ball_size=4;  // Rayon de la balle
11 const int nb_balls=30; // Nombre de balles
12 //////////////////////////////////////
13 // Generateur aleatoire
14 // A n'appeler qu'une fois , avant Random()
```

```

15 void InitRandom()
16 {
17     srand((unsigned int)time(0));
18 }
19 // Entre a et b
20 int Random(int a,int b)
21 {
22     return a+(rand()%(b-a+1));
23 }
24 ///////////////////////////////////////////////////
25 // Position et vitesse aleatoire
26 void InitBalle(int &x,int &y,int &u,int &v,Color &c) {
27     x=Random(ball_size,width-ball_size);
28     y=Random(ball_size,height-ball_size);
29     u=Random(0,4);
30     v=Random(0,4);
31     c=Color(byte(Random(0,255)),
32             byte(Random(0,255)),
33             byte(Random(0,255)));
34 }
35 ///////////////////////////////////////////////////
36 // Affichage d'une balle
37 void DessineBalle(int x,int y,Color col) {
38     fillRect(x-ball_size,y-ball_size,2*ball_size+1,2*ball_size+1,col);
39 }
40 ///////////////////////////////////////////////////
41 // Deplacement d'une balle
42 void BougeBalle(int &x,int &y,int &u,int &v) {
43     // Rebond sur les bords gauche et droit
44     if (x+u>width-ball_size || x+u<ball_size)
45         u=-u;
46     // Rebond sur les bords haut et bas et comptage du score
47     if (y+v<ball_size || y+v>height-ball_size)
48         v=-v;
49     // Mise a jour de la position
50     x+=u;
51     y+=v;
52 }
53 ///////////////////////////////////////////////////
54 // Fonction principale
55 int main()
56 {
57     // Ouverture de la fenetre
58     openWindow(width,height);
59     // Position et vitesse des balles
60     int xb[nb_balls],yb[nb_balls],ub[nb_balls],vb[nb_balls];
61     Color cb[nb_balls]; // Couleurs des balles
62     InitRandom();

```

```

63     for (int i=0;i<nb_balls;i++) {
64         InitBalle(xb[i],yb[i],ub[i],vb[i],cb[i]);
65         DessineBalle(xb[i],yb[i],cb[i]);
66     }
67     // Boucle principale
68     while (true) {
69         milliSleep(25);
70         noRefreshBegin();
71         for (int i=0;i<nb_balls;i++) {
72             DessineBalle(xb[i],yb[i],White);
73             BougeBalle(xb[i],yb[i],ub[i],vb[i]);
74             DessineBalle(xb[i],yb[i],cb[i]);
75         }
76         noRefreshEnd();
77     }
78     endGraphics();
79     return 0;
80 }

```

4.4.2 Avec des chocs !

Il n'est ensuite pas très compliqué de modifier le programme précédent pour que les balles rebondissent entre-elles. Le listing ci-après a été construit comme suit :

1. Lorsqu'une balle se déplace, on regarde aussi si elle rencontre une autre balle. Il faut donc que BougeBalle connaisse les positions des autres balles. On modifie donc BougeBalle en passant les tableaux complets des positions et des vitesses, et en précisant juste l'indice de la balle à déplacer (lignes 71 et 110). La boucle de la ligne 78 vérifie ensuite via le test de la ligne 81 si l'une des autres balles est heurtée par la balle courante. Auquel cas, on appelle ChocBalles qui modifie les vitesses des deux balles. Notez les lignes 79 et 80 qui évitent de considérer le choc d'une balle avec elle-même (nous verrons l'instruction **continue** une autre fois).
2. Les formules du choc de deux balles peuvent se trouver facilement dans un cours de prépa... ou sur le web. La fonction ChocBalles implémente ces formules. (Notez l'inclusion du fichier <cmath> pour avoir accès à la racine carré `sqrt()`, aux sinus et cosinus `cos()` et `sin()`, et à l'arc-cosinus `acos()`).
3. On réalise ensuite que les variables entières qui stockent positions et vitesses font que les erreurs d'arrondis s'accumulent et que les vitesses deviennent nulles ! On bascule alors toutes les variables concernées en **double**, en pensant bien à les reconverter en **int** lors de l'affichage (ligne 37).

Le tout donne un programme bien plus animé. On ne peut évidemment constater la différence sur une figure dans un livre. Téléchargez donc le programme sur la page du cours !

```

23 // //////////////////////////////////////
24 // Position et vitesse aleatoire
25 void InitBalle(double &x,double &y,double &u,double &v,Color &c){
26     x=Random(ball_size,width-ball_size);

```

```

27     y=Random(ball_size,height-ball_size);
28     u=Random(0,4);
29     v=Random(0,4);
30     c=Color(byte(Random(0,255)),
31             byte(Random(0,255)),
32             byte(Random(0,255)));
33 }
34 //////////////////////////////////////////////////
35 // Affichage d'une balle
36 void DessineBalle(double x,double y,Color col) {
37     fillRect(int(x)-ball_size,int(y)-ball_size,
38             2*ball_size+1,2*ball_size+1,col);
39 }
40 //////////////////////////////////////////////////
41 // Choc elastique de deux balles spheriques
42 // cf labo.ntic.org
43 #include <cmath>
44 void ChocBalles(double&x1,double&y1,double&u1,double&v1,
45                double&x2,double&y2,double&u2,double&v2)
46 {
47     // Distance
48     double o2o1x=x1-x2,o2o1y=y1-y2;
49     double d=sqrt(o2o1x*o2o1x+o2o1y*o2o1y);
50     if (d==0) return; // Même centre?
51     // Repère (o2,x,y)
52     double Vx=u1-u2,Vy=v1-v2;
53     double V=sqrt(Vx*Vx+Vy*Vy);
54     if (V==0) return; // Même vitesse
55     // Repère suivant V (o2,i,j)
56     double ix=Vx/V,iy=Vy/V,jx=-iy,jy=ix;
57     // Hauteur d'attaque
58     double H=o2o1x*jx+o2o1y*jy;
59     // Angle
60     double th=acos(H/d),c=cos(th),s=sin(th);
61     // Vitesse après choc dans (o2,i,j)
62     double v1i=V*c*c,v1j=V*c*s,v2i=V*s*s,v2j=-v1j;
63     // Dans repère d'origine (O,x,y)
64     u1=v1i*ix+v1j*jx+u2;
65     v1=v1i*iy+v1j*jy+v2;
66     u2+=v2i*ix+v2j*jx;
67     v2+=v2i*iy+v2j*jy;
68 }
69 //////////////////////////////////////////////////
70 // Deplacement d'une balle
71 void BougeBalle(double x[],double y[],double u[],double v[],int i)
72 { // Rebond sur les bords gauche et droit
73     if (x[i]+u[i]>width-ball_size || x[i]+u[i]<ball_size)
74         u[i]=-u[i];

```

```

75 // Rebond sur les bords haut et bas et comptage du score
76 if (y[i]+v[i]<ball_size || y[i]+v[i]>height-ball_size)
77     v[i]=-v[i];
78 for (int j=0;j<nb_balls;j++) {
79     if (j==i)
80         continue;
81     if (abs(x[i]+u[i]-x[j])<2*ball_size
82         && abs(y[i]+v[i]-y[j])<2*ball_size) {
83         ChocBalles(x[i],y[i],u[i],v[i],x[j],y[j],u[j],v[j]);
84     }
85 }
86 // Mise a jour de la position
87 x[i]+=u[i];
88 y[i]+=v[i];
89 }
90 ///////////////////////////////////////////////////
91 // Fonction principale
92 int main()
93 {
94     // Ouverture de la fenetre
95     openWindow(width,height);
96     // Position et vitesse des balles
97     double xb[nb_balls],yb[nb_balls],ub[nb_balls],vb[nb_balls];
98     Color cb[nb_balls]; // Couleurs des balles
99     InitRandom();
100    for (int i=0;i<nb_balls;i++) {
101        InitBalle(xb[i],yb[i],ub[i],vb[i],cb[i]);
102        DessineBalle(xb[i],yb[i],cb[i]);
103    }
104    // Boucle principale
105    while (true) {
106        milliSleep(25);
107        noRefreshBegin();
108        for (int i=0;i<nb_balls;i++) {
109            DessineBalle(xb[i],yb[i],White);
110            BougeBalle(xb,yb,ub,vb,i);
111            DessineBalle(xb[i],yb[i],cb[i]);
112        }
113        noRefreshEnd();
114    }
115    endGraphics();
116    return 0;
117 }

```

4.4.3 Mélanger les lettres

Le programme suivant considère une phrase et permute aléatoirement les lettres intérieures de chaque mot (c'est-à-dire sans toucher aux extrémités des mots). Il utilise pour cela le type `string`, chaîne de caractère, pour lequel `s[i]` renvoie le *i*-ème caractère de la chaîne `s`, et `s.size()` le nombre de caractères de `s` (nous expliquerons plus tard la notation "objet" de cette fonction). La phrase considérée ici devient par exemple :

Ctete pteite psahre dreviat erte ecorne libslie puor vorte parvue ceeravu

L'avez-vous comprise ? Peu importe ! C'est le listing que vous devez comprendre :

```

1 #include <iostream>
2 #include <string>
3 #include <cstdlib>
4 #include <ctime>
5 using namespace std;
6
7 //////////////////////////////////////
8 // Generateur aleatoire
9 // A n'appeler qu'une fois , avant Random()
10 void InitRandom()
11 {
12     srand((unsigned int)time(0));
13 }
14 // Entre a et b
15 int Random(int a,int b)
16 {
17     return a+(rand()%(b-a+1));
18 }
19
20 //////////////////////////////////////
21 // Permuter les lettres interieures de s n fois
22 string Melanger(string s,int n)
23 {
24     int l=int(s.size());
25     if (l<=3)
26         return s;
27     string t=s;
28     for (int i=0;i<n;i++) {
29         int a=Random(1,l-2);
30         int b;
31         do
32             b=Random(1,l-2);
33         while (a==b);
34         char c=t[a];
35         t[a]=t[b]; t[b]=c;
36     }
37     return t;
38 }
```

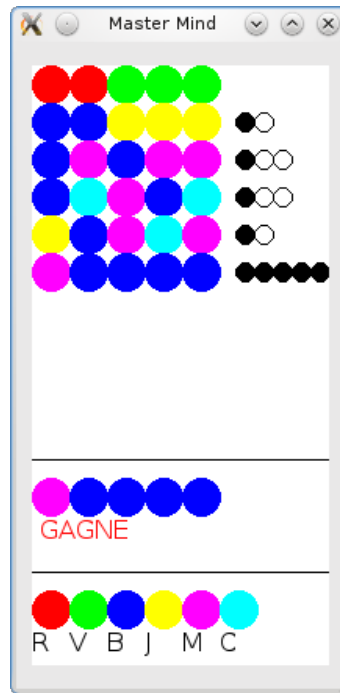


FIGURE 4.2 – Master mind...

```

39
40 int main()
41 {
42     const int n=11;
43     string phrase[n]={"Cette","petite","phrase","devrait","etre",
44                     "encore","lisible","pour","votre","pauvre",
45                     "cerveau"};
46
47     InitRandom();
48     for (int i=0;i<n;i++)
49         cout << Melanger(phrase[i],3) << " ";
50     cout << endl;
51
52     return 0;
53 }
```

4.5 TP

Nous pouvons maintenant aller faire le troisième TP donné en annexe A.3 afin de mieux comprendre les tableaux et aussi pour obtenir un master mind (voir figure 4.2 le résultat d'une partie intéressante!).

4.6 Fiche de référence

Comme promis, nous complétons, en rouge, la "fiche de référence" avec ce qui a été vu pendant ce chapitre et son TP.

Fiche de référence (1/2)		
Variables — Définition : <pre>int i; int k,l,m;</pre> — Affectation : <pre>i=2; j=i; k=l=3;</pre> — Initialisation : <pre>int n=5,o=n;</pre> — Constantes : <pre>const int s=12;</pre> — Portée : <pre>int i; // i=j; interdit! int j=2; i=j; // OK! if (j>1) { int k=3; j=k; // OK! } //i=k; interdit!</pre> — Types : <pre>int i=3; double x=12.3; char c='A'; string s="hop"; bool t=true; float y=1.2f; unsigned int j=4; signed char d=-128; unsigned char d=25; complex<double> z(2,3);</pre> — Variables globales : <pre>int n; const int m=12; void f() {</pre>	<pre>n=10; // OK int i=m; // OK ...</pre> — Conversion : <pre>int i=int(x),j; float x=float(i)/j;</pre>	<pre>— double x[5],y[5]; for(int i=0;i<5;i++) y[i]=2*x[i]; — const int n=5; int i[n],j[2*n];</pre>
	Tests — Comparaison : <pre>== != < > <= >=</pre> — Négation : ! — Combinaisons : && — if (i==0) j=1; — if (i==0) j=1; else j=2; — if (i==0) { j=1; k=2; } — bool t=(i==0); if (t) j=1; — switch (i) { case 1: ...; ...; break; case 2: case 3: ...; break; default: ...; }	— Initialisation : <pre>int t[4]={1,2,3,4}; string s[2]={"ab","c"};</pre> — Affectation : <pre>int s[3]={1,2,3},t[3]; for (int i=0;i<3;i++) t[i]=s[i];</pre> — En paramètre : <pre>— void init(int t[4]) { for(int i=0;i<4;i++) t[i]=0; } — void init(int t[], int n) { for(int i=0;i<n;i++) t[i]=0; }</pre>
	Tableaux — Définition : 	Boucles — do { ... } while(!ok); — int i=1; while(i<=100) { ... i=i+1; } — for(int i=1;i<=10;i++) ... — for(int i=1,j=10;j>i; i=i+2,j=j-3) ...

Fiche de référence (2/2)		
Fonctions — Définition : <pre>int plus(int a,int b){ int c=a+b; return c; } void affiche(int a) { cout << a << endl; }</pre> — Déclaration : <pre>int plus(int a,int b);</pre> — Retour : <pre>int signe(double x) { if (x<0) return -1; if (x>0) return 1; return 0; } void afficher(int x, int y) { if (x<0 y<0) return; if (x>w y>=h) return; DrawPoint(x,y,RED); }</pre> — Appel : <pre>int f(int a) { ... } int g() { ... } ... int i=f(2),j=g();</pre> — Références : <pre>void swap(int& a, int& b){ int tmp=a; a=b;b=tmp; } ... int x=3,y=2; swap(x,y);</pre> — Surcharge : <pre>int hasard(int n); int hasard(int a,</pre>	<pre>int b); double hasard();</pre> Divers — <code>i++;</code> — <code>i--;</code> — <code>i-=2;</code> — <code>j+=3;</code> — <code>j=i%n;</code> // Modulo — <code>#include <cstdlib></code> — <code>...</code> — <code>i=rand()%n;</code> — <code>x=rand()/</code> — <code>double (RAND_MAX);</code> — <code>#include <ctime></code> — <code>// Un seul appel</code> — <code>srand((unsigned int)</code> — <code>time(0));</code> — <code>#include <cmath></code> — <code>double sqrt(double x);</code> — <code>double cos(double x);</code> — <code>double sin(double x);</code> — <code>double acos(double x);</code> — <code>#include <string></code> — <code>using namespace std;</code> — <code>string s="hop";</code> — <code>char c=s[0];</code> — <code>int l=s.size();</code> Entrées/Sorties — <code>#include <iostream></code> — <code>using namespace std;</code> — <code>...</code> — <code>cout <<"I="<<i<<endl;</code> — <code>cin >> i >> j;</code> Erreurs fréquentes — Pas de définition de fonction dans une fonction ! — <code>int q=r=4;</code> // NON! — <code>if (i=2)</code> // NON! — <code>if i==2</code> // NON! — <code>if (i==2) then</code> // NON! — <code>for(int i=0,i<100,i++)</code> — <code>// NON!</code>	— <code>int f() {...}</code> — <code>int i=f;</code> // NON! — <code>double x=1/3;</code> // NON! — <code>int i,j;</code> — <code>x=i/j;</code> // NON! — <code>x=double(i/j);</code> //NON! — <code>double x[10],y[10];</code> — <code>for(int i=1;i<=10;i++)</code> — <code>y[i]=2*x[i];</code> //NON — <code>int n=5;</code> — <code>int t[n];</code> // NON — <code>int f()[4] { // NON!</code> — <code>int t[4];</code> — <code>...</code> — <code>return t;</code> // NON! — <code>}</code> — <code>int t[4]; t=f();</code> — <code>int s[3]={1,2,3},t[3];</code> — <code>t=s;</code> // NON! — <code>int t[2];</code> — <code>t={1,2};</code> // NON! Imagine++ — Voir documentation... Clavier — Debug : F5  — Step over : F10  — Step inside : F11  — Indent : Ctrl+A, Ctrl+I Conseils — Nettoyer en quittant. — Erreurs et warnings : cliquer. — Indenter. — Ne pas laisser de warning. — Utiliser le debuggeur. — Faire des fonctions. — Tableaux : pas pour transcrire une formule mathématique !

Chapitre 5

Les structures

Les fonctions et les boucles nous ont permis de regrouper des instructions identiques. Les tableaux permettent de grouper des variables de même type, mais pour manipuler plusieurs variables simultanément, il est tout aussi indispensable des fabriquer des structures de données...

—

5.1 Révisions

Avant cela, il est utile de nous livrer à une petite révision, qui prendra la forme d'un inventaire des erreurs classiques commises par de nombreux débutants...et même de celles, plus rares mais plus originales, constatées chez certains ! Enfin, nous répéterons, encore et toujours, les mêmes conseils.

5.1.1 Erreurs classiques

En vrac :

- Mettre un seul = dans les tests : `if (i=2)`
- Oublier les parenthèses : `if i==2`
- Utiliser then : `if (i==2) then`
- Mettre des virgules dans un for : `for (int i=0,i<100,i++)`
- Oublier les parenthèses quand on appelle une fonction sans paramètre :

```
int f() { ... }
```

```
...
```

```
int i=f;
```

- Vouloir affecter un tableau à un autre :

```
int s[4]={1,2,3,4}, t[4];
```

```
t=s;
```

5.1.2 Erreurs originales

Là, le débutant ne se trompe plus : il invente carrément avec sans doute le fol espoir que ça existe peut-être. Souvent, non seulement ça n'existe pas, mais en plus ça ne colle ni aux grands principes de la syntaxe du C++, ni même à ce qu'un compilateur peut comprendre ! Deux exemples :

- Mélanger la syntaxe (si peu !) :

```
void set(int t[5]) {
    ...
}
...
int s[5];           //Jusque là , tout va bien !
set(int s[5]);      // Là , c'est quand même n'importe quoi !
```

alors qu'il suffit d'un :

```
set(s);
```

- Vouloir faire plusieurs choses à la fois, ou ne pas comprendre qu'un **programme est une suite d'instructions à exécuter l'une après l'autre et non pas une formule**¹. Par exemple, croire que le **for** est un symbole mathématique comme $\sum_1^n$ ou $\bigcup_1^n$. Ainsi, pour exécuter une instruction quand tous les `ok(i)` sont vrais, on a déjà vu tenter un :

```
if (for (int i=0;i<n;i++) ok(i))    // Du grand art ...
...
```

alors qu'il faut faire :

```
bool allok=true;
for (int i=0;i<n;i++)
    allok=(allok && ok(i));
if (allok)
    ...
```

ou même mieux (voyez-vous la différence ?) :

```
bool allok=true;
for (int i=0;i<n && allok;i++)
    allok=(allok && ok(i));
if (allok)
    ...
```

qu'on peut finalement simplifier en :

```
bool allok=true;
for (int i=0;i<n && allok;i++)
    allok=ok(i);
if (allok)
    ...
```

1. Ne me faites pas dire ce que je n'ai pas dit ! Les informaticiens théoriques considèrent parfois les programmes comme des formules, mais ça n'a rien à voir !

Il est compréhensible que le débutant puisse être victime de son manque de savoir, d'une mauvaise assimilation des leçons précédentes, de la confusion avec un autre langage, ou de son imagination débordante ! Toutefois, il faut bien comprendre qu'un langage est finalement lui aussi un programme, limité et conçu pour faire des choses bien précises. En conséquence, il est plus raisonnable d'adopter la conduite suivante :

Tout ce qui n'a pas été annoncé comme possible est impossible !

5.1.3 Conseils

- Indenter. Indenter. **Indenter !**
- Cliquer sur les messages d'erreurs et de warnings pour aller directement à la bonne ligne !
- Ne pas laisser de warning.
- Utiliser le debugueur.

5.2 Les structures

5.2.1 Définition

Si les tableaux permettent de manipuler plusieurs variables d'un même type, les structures sont utilisées pour regrouper plusieurs variables afin de les manipuler comme une seule. On crée un *nouveau type*, dont les variables en question deviennent des "sous-variables" appelées *champs* de la structure. Voici par exemple un type Point possédant deux champs de type **double** nommés x et y :

```
struct Point {
    double x,y;
};
```

Les champs se définissent avec la syntaxe des variables locales d'une fonction. Attention par contre à

Ne pas oublier le point virgule après l'accolade qui ferme la définition de la structure !

L'utilisation est alors simple. La structure est un nouveau type qui se manipule exactement comme les autres, avec la particularité supplémentaire qu'on **accède aux champs avec un point** :

```
Point a;
a.x=2.3;
a.y=3.4;
```

On peut évidemment définir des champs de différents types, et même des structures dans des structures :

```
struct Cercle {
    Point centre;
    double rayon;
```

```

    Color couleur;
};
Cercle C;
C.centre.x=12.;
C.centre.y=13.;
C.rayon=10.4;
C.couleur=Red;

```

L'intérêt des structures est évident et il faut

Regrouper dans des structures des variables dès qu'on repère qu'elles sont logiquement liées. Si un programme devient pénible parce qu'on passe systématiquement plusieurs paramètres identiques à de nombreuses fonctions, alors il est vraisemblable que les paramètres en question puissent être avantageusement regroupés en une structure. Ce sera plus simple et plus clair.

5.2.2 Utilisation

Les structures se manipulent comme les autres types². La définition, l'affectation, l'initialisation, le passage en paramètre, le retour d'une fonction : tout est semblable au comportement des types de base. Seule nouveauté : **on utilise des accolades pour préciser les valeurs des champs en cas d'initialisation**³. On peut évidemment faire des tableaux de structures...et même définir un champ de type tableau! Ainsi, les lignes suivantes se comprennent facilement :

```

Point a={2.3,3.4},b=a,c;          // Initialisations
c=a;                             // Affectations
Cercle C={{12,13},10.4,Red};     // Initialisation
...
double distance(Point a,Point b) { // Passage par valeur
    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}
void agrandir(Cercle& C,double echelle) { // Par référence
    C.rayon=C.rayon*echelle; // Modifie le rayon
}
Point milieu(Point a,Point b) {   // retour
    Point M;
    M.x=(a.x+b.x)/2;
    M.y=(a.y+b.y)/2;
    return M;
}
...
Point P[10]; // Tableau de structures
for (int i=0;i<10;i++) {
    P[i].x=i;
}

```

2. D'ailleurs, nous avons bien promis que seuls les tableaux avaient des particularités (passage par référence, pas de retour possible et pas d'affectation).

3. Comme pour un tableau!

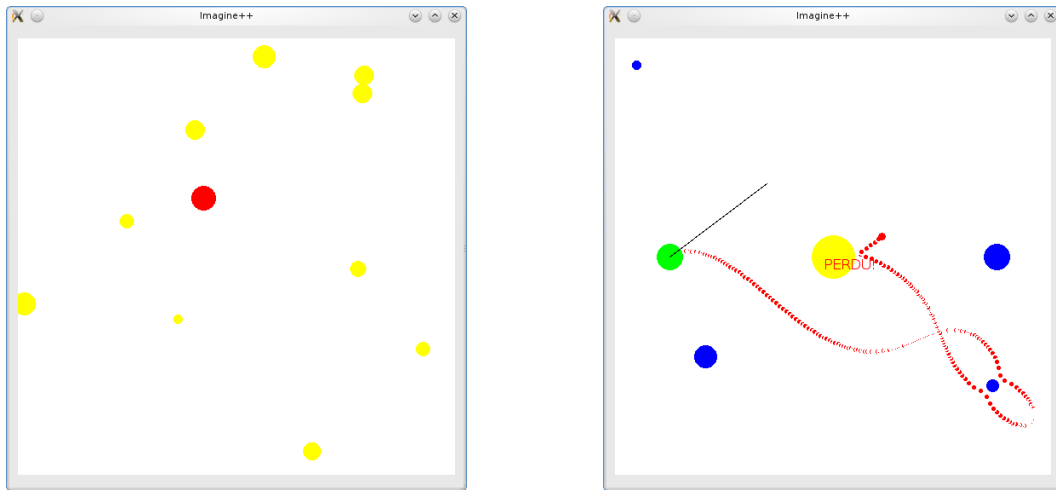


FIGURE 5.1 – Corps célestes et duel...

```

    P[i].y=f(i);
}
...
// Un début de jeu de Yam's
struct Tirage { //
    int de[5]; // champ de type tableau
};
Tirage lancer() {
    Tirage t;
    for (int i=0;i<5;i++)
        t.de[i]=1+rand()%6; // Un dé de 1 à 6
    return t;
}
...
Tirage t;
t=lancer();

```

Attention, tout comme pour les tableaux, la syntaxe utilisée pour l'initialisation ne marche pas pour une affectation⁴ :

```

Point P;
P={1,2}; // Erreur !

```

D'ailleurs, répétons-le :

Tout ce qui n'a pas été annoncé comme possible est impossible !

4. La situation s'améliorera avec les objets.

5.3 Récréation : TP

Nous pouvons maintenant aller faire le TP de l'annexe A.4 afin de mieux comprendre les structures. Nous ferons même des tableaux de structures⁵ ! Nous obtiendrons un projectile naviguant au milieu des étoiles puis un duel dans l'espace (figure 5.1) !

5.4 Fiche de référence

Encore une fois, nous complétons, en rouge, la "fiche de référence" avec ce qui a été vu pendant ce chapitre et son TP.

Fiche de référence (1/3)		
Variables — Définition : <pre>int i; int k,l,m;</pre> — Affectation : <pre>i=2; j=i; k=l=3;</pre> — Initialisation : <pre>int n=5,o=n;</pre> — Constantes : <pre>const int s=12;</pre> — Portée : <pre>int i; // i=j; interdit! int j=2; i=j; // OK! if (j>1) { int k=3; j=k; // OK! } //i=k; interdit!</pre> — Types : <pre>int i=3; double x=12.3; char c='A'; string s="hop"; bool t=true; float y=1.2f;</pre>	<pre>unsigned int j=4; signed char d=-128; unsigned char d=25; complex<double> z(2,3);</pre> — Variables globales : <pre>int n; const int m=12; void f() { n=10; // OK int i=m; // OK ... }</pre> — Conversion : <pre>int i=int(x),j; float x=float(i)/j;</pre>	<pre>— bool t=(i==0); if (t) j=1; — switch (i) { case 1: ...; break; case 2: case 3: ...; break; default: ...; }</pre>
	Tests — Comparaison : <pre>== != < > <= >=</pre> — Négation : ! — Combinaisons : && <pre>— if (i==0) j=1; — if (i==0) j=1; else j=2;</pre> <pre>— if (i==0) { j=1; k=2; }</pre>	Boucles <pre>— do { ... } while(!ok); — int i=1; while(i<=100) { ... i=i+1; } — for(int i=1;i<=10;i++) ... — for(int i=1,j=10;j>i; i=i+2,j=j-3) ...</pre>

5. **Coin des collégiens** : il y a dans ce TP des mathématiques et de la physique pour étudiant de l'enseignement supérieur... mais on peut très bien faire les programmes en ignorant tout ça !

Fiche de référence (2/3)		
Fonctions — Définition : <pre>int plus(int a,int b){ int c=a+b; return c; } void affiche(int a) { cout << a << endl; }</pre> — Déclaration : <pre>int plus(int a,int b);</pre> — Retour : <pre>int signe(double x) { if (x<0) return -1; if (x>0) return 1; return 0; } void afficher(int x, int y) { if (x<0 y<0) return; if (x>=w y>=h) return; DrawPoint(x,y,RED); } — Appel : <pre>int f(int a) { ... } int g() { ... } ... int i=f(2),j=g();</pre> — Références : <pre>void swap(int& a, int& b){ int tmp=a; a=b;b=tmp; } ... int x=3,y=2; swap(x,y);</pre> — Surcharge : <pre>int hasard(int n); int hasard(int a, int b); double hasard();</pre> </pre>	<pre>— const int n=5; int i[n],j[2*n];</pre> — Initialisation : <pre>int t[4]={1,2,3,4}; string s[2]={"ab","c"};</pre> — Affectation : <pre>int s[3]={1,2,3},t[3]; for (int i=0;i<3;i++) t[i]=s[i];</pre> — En paramètre : <pre>— void init(int t[4]) { for(int i=0;i<4;i++) t[i]=0; } — void init(int t[], int n) { for(int i=0;i<n;i++) t[i]=0; }</pre>	<pre>— #include <string> using namespace std; string s="hop"; char c=s[0]; int l=s.size();</pre>
Tableaux — Définition : <pre>— double x[5],y[5]; for(int i=0;i<5;i++) y[i]=2*x[i];</pre>	Structures <pre>— struct Point { double x,y; Color c; }; ... Point a; a.x=2.3; a.y=3.4; a.c=Red; Point b={1,2.5,Blue};</pre> Divers <pre>— i++; i--; i-=2; j+=3; — j=i%n; // Modulo — #include <cstdlib> ... i=rand()%n; x=rand()/ double(RAND_MAX); — #include <ctime> // Un seul appel srand((unsigned int) time(0)); — #include <cmath> double sqrt(double x); double cos(double x); double sin(double x); double acos(double x);</pre>	Entrées/Sorties <pre>— #include <iostream> using namespace std; ... cout <<"I="<<i<<endl; cin >> i >> j;</pre>
		Erreurs fréquentes — Pas de définition de fonction dans une fonction ! <pre>— int q=r=4; // NON!</pre> <pre>— if (i=2) // NON! if i==2 // NON! if (i==2) then // NON!</pre> <pre>— for(int i=0,i<100,i++) // NON!</pre> <pre>— int f() {...} int i=f; // NON!</pre> <pre>— double x=1/3; // NON! int i,j; x=i/j; // NON! x=double(i/j); //NON!</pre> <pre>— double x[10],y[10]; for(int i=1;i<=10;i++) y[i]=2*x[i];//NON</pre> <pre>— int n=5; int t[n]; // NON</pre> <pre>— int f()[4] { // NON! int t[4]; ... return t; // NON!</pre> <pre>— int t[4]; t=f();</pre> <pre>— int s[3]={1,2,3},t[3]; t=s; // NON!</pre> <pre>— int t[2]; t={1,2}; // NON!</pre> <pre>— struct Point { double x,y; } // NON!</pre> <pre>— Point a; a={1,2}; // NON!</pre>

Fiche de référence (3/3)		
Imagine++ — Voir documentation...	— Step inside : F11  — Indent : Ctrl+A, Ctrl+I	— Ne pas laisser de warning. — Utiliser le debugueur. — Faire des fonctions.
Clavier — Debug : F5  — Step over : F10 	Conseils — Nettoyer en quittant. — Erreurs et warnings : cliquer. — Indenter.	— Tableaux : pas pour transcrire une formule mathématique ! — Faire des structures.

Chapitre 6

Plusieurs fichiers !

Lors du dernier TP, nous avons réalisé deux projets quasiment similaires dont seuls les `main()` étaient différents. Modifier après coup une des fonctions de la partie commune aux deux projets nécessiterait d'aller la modifier dans les deux projets. Nous allons voir maintenant comment factoriser cette partie commune dans un seul fichier, de façon à en simplifier les éventuelles futures modifications. Au passage¹ nous verrons comment définir un opérateur sur de nouveaux types.

—

Résumons notre progression dans le savoir-faire du programmeur :

1. Tout programmer dans le `main()` : c'est un début et c'est déjà bien !
2. Faire des fonctions : pour être plus lisible et ne pas se répéter ! (Axe des instructions)
3. Faire des tableaux et des structures : pour manipuler plusieurs variables à la fois. (Axe des données)

Nous rajoutons maintenant :

4. Faire plusieurs fichiers : pour utiliser des parties communes dans différents projets ou solutions. (A nouveau, axe des instructions)

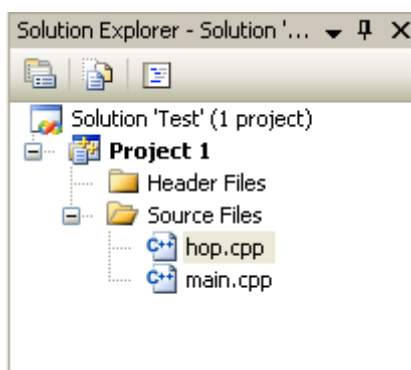


FIGURE 6.1 – Plusieurs fichiers sources (dans Visual).

1. Toujours cette idée que nous explorons les différentes composantes du langage quand le besoin s'en fait sentir.

6.1 Fichiers séparés

Nous allons répartir notre code source dans plusieurs fichiers. Mais avant toute chose :

Pour un maximum de portabilité du code, choisir des noms de fichiers avec seulement des caractères standard (pas de lettres accentuées ni d'espace)

D'ailleurs il est aussi préférable d'éviter les accents pour les noms de variables et de fonctions, tant pis pour la correction du français...

6.1.1 Principe

Jusqu'à présent un seul fichier source contenait notre programme C++. Ce fichier source était transformé en fichier objet par le compilateur puis le linker complétait le fichier objet avec les bibliothèques du C++ pour en faire un fichier exécutable. En fait, un projet peut contenir **plusieurs fichiers sources**. Il suffit pour cela de rajouter un fichier `.cpp` à la liste des sources du projet :

- Dans QtCreator, ouvrir le menu `File/New File or Project` ou faire `Ctrl+N`, choisir comme modèle `C++ Source File`, lui donner un nom et bien s'assurer qu'on le met dans le dossier des sources (et non dans le dossier de build).
- Rajouter ce fichier dans le `CMakeLists.txt` :

```
add_executable(Hop main.cpp hop.cpp)
```

Ainsi, en rajoutant un fichier C++ `hop` à un projet contenant déjà `main.cpp`, on se retrouve avec une structure de projet identique à celle de la figure 6.1.

Après cela, chaque génération du projet consistera en :

1. **Compilation** : chaque fichier source est transformé en un fichier objet (de même nom mais de suffixe `.obj`). Les fichiers sources sont donc compilés indépendamment les uns des autres.
2. **Link** : les différents fichiers objets sont réunis (et complétés avec les bibliothèques du C++) en un seul fichier exécutable (de même nom que le projet).

Une partie des instructions du fichier principal (celui qui contient `main()`) peut donc être déportée dans un autre fichier. Cette partie sera compilée séparément et réintégrée pendant l'édition des liens. Se pose alors le problème suivant : **comment utiliser dans le fichier principal ce qui se trouve dans les autres fichiers ?** En effet, nous savions (cf section 3.2.4) qu'une fonction n'était "connue" que dans les lignes qui suivaient sa définition ou son éventuelle déclaration. Par "connue", il faut comprendre que le compilateur sait qu'il existe ailleurs une fonction de tel nom avec tel type de retour et tels paramètres. Malheureusement² :

une fonction n'est pas "connue" en dehors de son fichier. Pour l'utiliser dans un autre fichier, il faut donc l'y déclarer !

En clair, nous allons devoir procéder ainsi :

2. Heureusement, en fait, car lorsque l'on réunit des fichiers de provenances multiples, il est préférable que ce qui se trouve dans les différents fichiers ne se mélange pas de façon anarchique...

— Fichier `hop.cpp` :

```
// Définitions
void f(int x) {
    ...
}
int g() {
    ...
}
// Autres fonctions
...
```

— Fichier `main.cpp` :

```
// Déclarations
void f(int x);
int g();
...
int main() {
    ...
    // Utilisation
    int a=g();
    f(a);
    ...
}
```

Nous pourrions aussi évidemment déclarer dans `hop.cpp` certaines fonctions de `main.cpp` pour pouvoir les utiliser. Attention toutefois : si des fichiers s'utilisent de façon croisée, c'est peut-être que nous sommes en train de ne pas découper les sources convenablement.

6.1.2 Avantages

Notre motivation initiale était de mettre une partie du code dans un fichier séparé pour **l'utiliser dans un autre projet**. En fait, découper son code en plusieurs fichiers a d'autres intérêts :

- Rendre le code **plus lisible** et évitant les fichiers trop longs et en regroupant les fonctions de façon structurée.
- **Accélérer la compilation**. Lorsqu'un programme devient long et complexe, le temps de compilation n'est plus négligeable. Or, lorsque l'on régénère un projet, l'environnement de programmation ne recompile que les fichiers sources qui ont été modifiés depuis la génération précédente. Il serait en effet inutile de recompiler un fichier source non modifié pour ainsi obtenir le même fichier objet³ ! Donc changer quelques lignes dans un fichier n'entraînera pas la compilation de tout le programme mais seulement du fichier concerné.

Attention toutefois à ne pas séparer en de trop nombreux fichiers ! Il devient alors plus compliqué de s'y retrouver et de naviguer parmi ces fichiers.

3. C'est en réalité un peu plus compliqué : un source peu dépendre, via des inclusions (cf section 6.1.4), d'autres fichiers, qui, eux, peuvent avoir été modifiés ! Il faut alors recompiler un fichier dont une dépendance a été modifiée. Ces dépendances sont gérées automatiquement par Cmake.

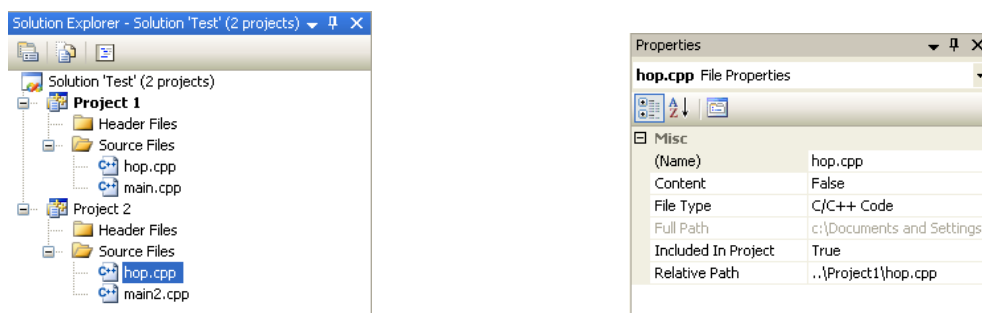


FIGURE 6.2 – Même source dans deux projets

6.1.3 Utilisation dans un autre projet

6.1.4 Fichiers d'en-têtes

Le fichier séparé nous permet de factoriser une partie du source. Toutefois, il faut taper les déclarations de toutes les fonctions utilisées dans chaque fichier les utilisant. Nous pouvons mieux faire⁴. Pour cela, il est temps d'expliquer ce que fait l'instruction **#include** que nous rencontrons depuis nos débuts :

La ligne `#include "nom"` est automatiquement remplacée par le contenu du fichier `nom` avant de procéder à la compilation.

Il s'agit bien de remplacer par le texte complet du fichier `nom` comme avec un simple copier/coller. Cette opération est faite avant la compilation par un programme dont nous n'avions pas parlé : le *pré-processeur*. Les lignes commençant par un `#` lui seront destinées. Nous en verrons d'autres. Attention : jusqu'ici nous utilisons une forme légèrement différente : **#include** <nom>, qui va chercher le fichier `nom` dans les dossiers des bibliothèques C++⁵.

Grâce à cette possibilité du pré-processeur, il nous suffit de mettre les déclarations se rapportant au fichier séparé dans un troisième fichier et de l'*inclure* dans les fichiers principaux. Il est d'usage de prendre pour ce fichier supplémentaire le même nom que le fichier séparé, mais avec l'extension `.h` : on appelle ce fichier un fichier d'en-tête⁶. Pour créer ce fichier, faire comme pour le source, mais en choisissant "C++ Header File" au lieu de "C++ Source File". Voila ce que cela donne :

— Fichier `hop.cpp` :

```
// Définitions
void f(int x) {
    ...
}
int g() {
    ...
}
```

4. Toujours le moindre effort...

5. Les fichiers d'en-tête `iostream`, etc. sont parfois appelés en-têtes *système*. Leur nom ne se termine pas toujours par `.h` (voir après), mais rien ne les distingue fondamentalement d'un fichier d'en-tête habituel. Certains noms de headers système commencent par la lettre `c`, comme `cmath`, ce sont ceux hérités du C.

6. `.h` comme header, on voit aussi parfois `.hpp` pour les distinguer des headers du C.

```

}
// Autres fonctions
...

```

— Fichier `hop.h` :

```

// Déclarations
void f(int x);
int g();

```

— Fichier `main.cpp` du premier projet :

```

#include "hop.h"
...
int main() {
    ...
    // Utilisation
    int a=g();
    f(a);
    ...
}

```

— Fichier `main2.cpp` du deuxième projet (il faut préciser l'emplacement complet de l'en-tête, qui se trouve dans le dossier du premier projet⁷) :

```

#include "../Project1/hop.h"
...
int main() {
    ...
    // Utilisation
    f(12);
    int b=g();
    ...
}

```

En fait, pour être sûr que les fonctions définies dans `hop.cpp` sont cohérentes avec leur déclaration dans `hop.h`, et bien que ça ne soit pas obligatoire, on inclut aussi l'en-tête dans le source, ce qui donne :

— Fichier `hop.cpp` :

```

#include "hop.h"
...
// Définitions
void f(int x) {
    ...
}
int g() {
    ...
}
// Autres fonctions
...

```

7. On peut aussi préciser au compilateur une liste de dossiers où il peut aller chercher les fichiers d'en-tête, voir la section 6.1.8.

En pratique, le fichier d'en-tête ne contient pas seulement les déclarations des fonctions mais aussi les définitions des nouveaux types (comme les structures) utilisés par le fichier séparé. En effet, ces nouveaux types doivent être connus du fichier séparé, mais aussi du fichier principal. Il faut donc vraiment :

1. Mettre dans l'en-tête les déclarations des fonctions et les définitions des nouveaux types.
2. Inclure l'en-tête dans le fichier principal mais aussi dans le fichier séparé.

Cela donne par exemple :

— Fichier `vect.h` :

```
// Types
struct Vecteur {
    double x,y;
};
// Déclarations
double norme(Vecteur V);
Vecteur plus(Vecteur A,Vecteur B);
```

— Fichier `vect.cpp` :

```
#include "vect.h" // Fonctions et types
// Définitions
double norme(Vecteur V) {
    ...
}
Vecteur plus(Vecteur A,Vecteur B) {
    ...
}
// Autres fonctions
...
```

— Fichier `main.cpp` du premier :

```
#include "vect.h"
...
int main() {
    ...
    // Utilisation
    Vecteur C=plus(A,B);
    double n=norme(C);
    ...
}
```

Votre environnement de développement vous permet de naviguer facilement entre les différents fichiers de votre projet. Par exemple, avec QtCreator, vous pouvez passer du header au source ou vice-versa par l'option "Switch Header/Source" (F4) du menu contextuel obtenu par un clic droit dans l'éditeur. Pour suivre un include, vous pouvez aussi vous mettre sur le nom du fichier et choisir l'option "Follow Symbol Under Cursor" (F2) du même menu.

6.1.5 A ne pas faire...

Il est "fortement" conseillé de :

1. ne pas déclarer dans l'en-tête toutes les fonctions du fichier séparé mais seulement celles qui seront utilisées par le fichier principal. Les fonctions secondaires n'ont pas à apparaître⁸.
2. ne jamais inclure un fichier séparé lui-même ! C'est généralement une "grosse bêtise"⁹. Donc

pas de `#include "vect.cpp"` !

6.1.6 Implémentation

Finalement, la philosophie de ce système est que

- **Le fichier séparé et son en-tête forment un tout cohérent, implémentant un certain nombre de fonctionnalités.**
- **Celui qui les utilise, qui n'est pas nécessairement celui qui les a programmés, se contente de rajouter ces fichiers à son projet, d'inclure l'en-tête dans ses sources et de profiter de ce que l'en-tête déclare.**
- **Le fichier d'en-tête doit être suffisamment clair et informatif pour que l'utilisateur n'ait pas à regarder le fichier séparé lui-même^a.**

a. D'ailleurs, si l'utilisateur le regarde, il peut être tenté de tirer profit de ce qui s'y trouve et d'utiliser plus que ce que l'en-tête déclare. Or, le créateur du fichier séparé et de l'en-tête peut par la suite être amené à changer dans son source la façon dont il a programmé les fonctions sans pour autant changer leurs fonctionnalités. L'utilisateur qui a "triché" en allant regarder dans le fichier séparé peut alors voir ses programmes ne plus marcher. Il n'a pas respecté la règle du jeu qui était de n'utiliser que les fonctions de l'en-tête sans savoir comment elles sont implémentées. Nous reparlerons de tout ça avec les objets. Nous pourrions alors faire en sorte que l'utilisateur ne triche pas... De toute façon, à notre niveau actuel, le créateur et l'utilisateur sont une seule et même personne. À elle de ne pas tricher !

6.1.7 Inclusions mutuelles

En passant à l'action, le débutant découvre souvent des problèmes non prévus lors du cours. Il est même en général imbattable pour cela ! Le problème le plus fréquent qui survient avec les fichiers d'en-tête est celui de l'*inclusion mutuelle*. Il arrive que les fichiers d'en-tête aient besoin d'en inclure d'autres eux-mêmes. Or, si le fichier `A.h` inclut `B.h` et si `B.h` inclut `A.h` alors toute inclusion de `A.h` ou de `B.h` se solde par un phénomène d'inclusions sans fin qui provoque une erreur¹⁰. Pour éviter cela, on utilise une instruction du pré-processeur signalant qu'un fichier déjà inclus ne doit plus l'être à nouveau : on ajoute

8. On devrait même tout faire pour bien les cacher et pour interdire au fichier principal de les utiliser. Il serait possible de le faire dès maintenant, mais nous en reparlerons plutôt quand nous aborderons les objets...

9. Une même fonction peut alors se retrouver définie plusieurs fois : dans le fichier séparé et dans le fichier principal qui l'inclut. Or, s'il est possible de déclarer autant de fois que nécessaire une fonction, il est interdit de la définir plusieurs fois (ne pas confondre avec la surcharge qui rend possible l'existence de fonctions différentes sous le même nom—cf. section 3.2.6)

10. Les pré-processeurs savent heureusement détecter ce cas de figure.

#pragma once au début de chaque fichier d'en-tête.

Certains compilateurs peuvent ne pas connaître **#pragma once**. On utilise alors une astuce que nous donnons sans explication :

- Choisir un nom unique propre au fichier d'en-tête. Par exemple `VECT_H` pour le fichier `vect.h`.
- Placer **#ifndef** `VECT_H` et **#define** `VECT_H` au début du fichier `vect.h` et **#endif** à la fin.

Cela utilise la commande `if` du préprocesseur. Notons un autre usage parfois utile en cours de développement pour que le compilateur ne regarde pas tout un bloc de code :

```
#if 0
N'importe_quoi_ici ,_ce_sera_ignoré_par_le_compilateur .
#endif
```

6.1.8 Chemin d'inclusion

Puisque le **#include** prend un fichier, la question se pose de savoir où le préprocesseur doit chercher. Pour les fichiers du système, par exemple **#include** `<iostream>`, leur chemin est connu du compilateur et on n'a pas à s'en occuper. Pour les autres, il y a deux règles de recherche :

- dans le dossier courant, celui qui contient le `cpp` ;
- dans une liste de dossiers indiqués par l'utilisateur.

Pour le deuxième cas, on dispose de l'instruction `include_directories` de Cmake, à utiliser dans le `CMakeLists.txt`. Par exemple, quand on fait¹¹

```
#include "Imagine/Graphics.h"
```

il va chercher un dossier `Imagine` contenant un fichier `Graphics.h`. (Toujours utiliser le slash direct `/`, qui fonctionne sur toutes les plates-formes, et non le backslash `\` qui ne fonctionne que sous Windows. De même, respecter les majuscules.) Pour savoir où le trouver, on a dans le `CMakeLists.txt` :

```
find_package(Imagine)
ImagineUseModules(Mastermind Graphics)
```

La commande `ImagineUseModules` est spécifique à `Imagine++` (d'où le `find_package`), mais appelle `include_directories` en lui donnant comme chemin

```
<Imagine_DIR>/include
```

avec `<Imagine_DIR>` le chemin d'installation d'`Imagine++`.

¹¹. Notez que les commandes du préprocesseur ne se terminent pas par un point-virgule, mais avec la fin de ligne.

6.2 Opérateurs

Le C++ permet de définir les opérateurs +, -, etc. quand les opérandes sont de nouveaux types. Voici très succinctement comment faire. Nous laissons au lecteur le soin de découvrir seul quels sont les opérateurs qu'il est possible de définir.

Considérons l'exemple suivant qui définit un vecteur¹² 2D et en implémente l'addition :

```
struct vect {
    double x,y;
};
vect plus(vect m,vect n) {
    vect p={m.x+n.x,m.y+n.y};
    return p;
}
int main() {
    vect a={1,2},b={3,4};
    vect c=plus(a,b);
    return 0;
}
```

Voici comment définir le + entre deux vect et ainsi remplacer la fonction plus() :

```
struct vect {
    double x,y;
};
vect operator+(vect m,vect n) {
    vect p={m.x+n.x,m.y+n.y};
    return p;
}
int main() {
    vect a={1,2},b={3,4};
    vect c=a+b;
    return 0;
}
```

Nous pouvons aussi définir un produit par un scalaire, un produit scalaire¹³, etc¹⁴.

```
// Produit par un scalaire
vect operator*(double s,vect m) {
    vect p={s*m.x,s*m.y};
    return p;
}
// Produit scalaire
double operator*(vect m,vect n) {
```

12. **Coin des collégiens** : vous ne savez pas ce qu'est un vecteur... mais vous êtes plus forts en programmation que les "vieux". Alors regardez les sources qui suivent et vous saurez ce qu'est un vecteur 2D!

13. Dans ce cas, on utilise $a*b$ et non $a.b$, le point n'étant pas définissable car réservé à l'accès aux champs de la structure

14. On peut en fait définir ce qui existe déjà sur les types de base. Attention, il est impossible de redéfinir les opérations des types de base ! Pas question de donner un sens différent à $1+1$.

```

    return m.x*n.x+m.y*n.y;
}
int main() {
    vect a={1,2},b={3,4};
    vect c=2*a;
    double s=a*b;
    return 0;
}

```

Remarquez que les deux fonctions ainsi définies sont différentes bien que de même nom (**operator***) car elles prennent des paramètres différents (cf surcharge section 3.2.6).

6.3 Récréation : TP suite et fin

Le programme du TP précédent étant un exemple parfait de besoin de fichiers séparés (structures bien identifiées, partagées par deux projets), nous vous proposons, dans le TP A.5 de convertir (et terminer ?) notre programme de simulation de gravitation et de duel dans l'espace !

6.4 Fiche de référence

La fiche habituelle...

Fiche de référence (1/3)		
Variables — Définition : <pre>int i; int k,l,m;</pre> — Affectation : <pre>i=2; j=i; k=l=3;</pre> — Initialisation : <pre>int n=5,o=n;</pre> — Constantes : <pre>const int s=12;</pre> — Portée : <pre>int i; // i=j; interdit! int j=2; i=j; // OK! if (j>1) { int k=3; j=k; // OK! } //i=k; interdit!</pre> — Types : <pre>int i=3;</pre>	<pre>double x=12.3; char c='A'; string s="hop"; bool t=true; float y=1.2f; unsigned int j=4; signed char d=-128; unsigned char d=25; complex<double> z(2,3);</pre> — Variables globales : <pre>int n; const int m=12; void f() { n=10; // OK int i=m; // OK ... }</pre> — Conversion : <pre>int i=int(x),j; float x=float(i)/j;</pre> Tests — Comparaison : <pre>== != < > <= >=</pre> — Négation : !	— Combinaisons : && <pre>— if (i==0) j=1; — if (i==0) j=1; else j=2; — if (i==0) { j=1; k=2; }</pre> — <pre>bool t=(i==0); if (t) j=1;</pre> — <pre>switch (i) { case 1: ...; ...; break; case 2: case 3: ...; break; default: ...; }</pre>

Fiche de référence (2/3)		
Boucles — do { ... } while(!ok); — int i=1; while(i<=100) { ... i=i+1; } — for(int i=1;i<=10;i++) ... — for(int i=1,j=10;j>i; i=i+2,j=j-3) ...	a=b;b=tmp; } ... int x=3,y=2; swap(x,y); — Surcharge : int hasard(int n); int hasard(int a, int b); double hasard(); — Opérateurs : vect operator+(vect A,vect B) { ... } ... vect C=A+B;	a.x=2.3; a.y=3.4; a.c=Red; Point b={1,2.5,Blue}; Compilation séparée — #include "vect.h", aussi dans vect.cpp — Fonctions : déclarations dans le .h, définitions dans le .cpp — Types : définitions dans le .h — Ne déclarer dans le .h que les fonctions utiles. — #pragma once au début du fichier. — Ne pas trop découper...
Fonctions — Définition : int plus(int a,int b){ int c=a+b; return c; } void affiche(int a) { cout << a << endl; } — Déclaration : int plus(int a,int b); — Retour : int signe(double x) { if (x<0) return -1; if (x>0) return 1; return 0; } void afficher(int x, int y) { if (x<0 y<0) return; if (x>=w y>=h) return; DrawPoint(x,y,RED); } — Appel : int f(int a) { ... } int g() { ... } ... int i=f(2),j=g(); — Références : void swap(int& a, int& b){ int tmp=a;	Tableaux — Définition : double x[5],y[5]; for(int i=0;i<5;i++) y[i]=2*x[i]; — const int n=5; int i[n],j[2*n]; — Initialisation : int t[4]={1,2,3,4}; string s[2]={"ab","c"}; — Affectation : int s[3]={1,2,3},t[3]; for (int i=0;i<3;i++) t[i]=s[i]; — En paramètre : void init(int t[4]) { for(int i=0;i<4;i++) t[i]=0; } void init(int t[], int n) { for(int i=0;i<n;i++) t[i]=0; }	Divers — i++; i--; i-=2; j+=3; — j=i%n; // Modulo — #include <cstdlib> ... i=rand()%n; x=rand()/ double(RAND_MAX); — #include <ctime> // Un seul appel srand((unsigned int) time(0)); — #include <cmath> double sqrt(double x); double cos(double x); double sin(double x); double acos(double x); — #include <string> using namespace std; string s="hop"; char c=s[0]; int l=s.size();
	Structures — struct Point { double x,y; Color c; }; ... Point a;	Entrées/Sorties — #include <iostream> using namespace std; ... cout <<"I="<<i<<endl; cin >> i >> j;

Fiche de référence (3/3)		
Erreurs fréquentes — Pas de définition de fonction dans une fonction ! — <code>int q=r=4; // NON!</code> — <code>if (i=2) // NON!</code> <code>if i==2 // NON!</code> <code>if (i==2) then // NON!</code> — <code>for(int i=0,i<100,i++)</code> <code>// NON!</code> — <code>int f() {...}</code> <code>int i=f; // NON!</code> — <code>double x=1/3; // NON!</code> <code>int i,j;</code> <code>x=i/j; // NON!</code> <code>x=double(i/j); //NON!</code> — <code>double x[10],y[10];</code> <code>for(int i=1;i<=10;i++)</code> <code>y[i]=2*x[i];//NON</code> — <code>int n=5;</code> <code>int t[n]; // NON</code>	— <code>int f()[4] { // NON!</code> <code>int t[4];</code> <code>...</code> <code>return t; // NON!</code> <code>}</code> <code>int t[4]; t=f();</code> — <code>int s[3]={1,2,3},t[3];</code> <code>t=s; // NON!</code> — <code>int t[2];</code> <code>t={1,2}; // NON!</code> — <code>struct Point {</code> <code>double x,y;</code> <code>} // NON!</code> — <code>Point a;</code> <code>a={1,2}; // NON!</code> — <code>#include "tp.cpp"//NON</code> Imagine++ — Voir documentation... Clavier	— Debug : F5  — Step over : F10  — Step inside : F11  — Indent : Ctrl+A, Ctrl+I Conseils — Nettoyer en quittant. — Erreurs et warnings : cliquer. — Indenter. — Ne pas laisser de warning. — Utiliser le debugueur. — Faire des fonctions. — Tableaux : pas pour transcrire une formule mathématique ! — Faire des structures. — Faire des fichiers séparés. — Le .h doit suffire à l'utilisateur (qui ne doit pas regarder le .cpp)

Chapitre 7

La mémoire

Il est grand temps de revenir sur la mémoire et son utilisation. Nous pourrions alors mieux comprendre les variables locales, comment marche exactement l'appel d'une fonction, les fonctions récursives, etc. Après cela, nous pourrions enfin utiliser des tableaux de taille variable (sans pour autant rentrer vraiment dans la notion délicate de pointeur).

—

7.1 L'appel d'une fonction

Il s'agit là d'une nouvelle occasion pour vous de comprendre enfin ce qui se passe dans un programme...

7.1.1 Exemple

Considérons le programme suivant :

```
1 #include <iostream>
2 using namespace std;
3
4 void verifie(int p, int q,int quo,int res) {
5     if (res<0 || res>=q || q*quo+res!=p)
6         cout << "Tiens ,_c'est_bizarre!" << endl;
7 }
8
9 int divise(int a,int b,int& r) {
10     int q;
11     q=a/b;
12     r=a-q*b;
13     verifie(a,b,q,r);
14     return q;
15 }
16 int main()
17 {
18     int num,denom;
19     do {
```

```

20      cout << "Entrez_deux_entiers_positifs:_";
21      cin >> num >> denom;
22  } while (num<=0 || denom<=0);
23  int quotient, reste;
24  quotient=divise(num,denom,reste);
25  cout << num << "/" << denom << "_=" << quotient
26      << "_(Il_reste_" << reste << ")" << endl;
27  return 0;
28  }

```

Calculant le quotient et le reste d'une division entière, et vérifiant qu'ils sont corrects, il n'est pas passionnant et surtout inutilement long (en fait, ce sont juste les lignes 11 et 12 qui font tout!). Il s'agit par contre d'un bon exemple pour illustrer notre propos. Une bonne façon d'expliquer exhaustivement son déroulement est de remplir le tableau suivant, déjà rencontré au TP A.2. En ne mettant que les lignes où les variables changent, en supposant que l'utilisateur rentre 23 et 3 au clavier, et **en indiquant avec des lettres les différentes étapes d'une même ligne**¹, cela donne :

Ligne	num	denom	quotient	reste	a	b	r	q _d	ret _d	p _v	q _v	quo	res
18	?	?											
21	23	3											
23	23	3	?	?									
24a	23	3	?	?									
9	23	3	?	?	23	3	[reste]						
10	23	3	?	?	23	3	[reste]	?					
11	23	3	?	?	23	3	[reste]	7					
12	23	3	?	2	23	3	[reste]	7					
13a	23	3	?	2	23	3	[reste]	7					
4	23	3	?	2	23	3	[reste]	7		23	3	7	2
5	23	3	?	2	23	3	[reste]	7		23	3	7	2
7	23	3	?	2	23	3	[reste]	7					
13b	23	3	?	2	23	3	[reste]	7					
14	23	3	?	2	23	3	[reste]	7	7				
15	23	3	?	2					7				
24b	23	3	7	2					7				
25	23	3	7	2									
28													

A posteriori, on constate qu'on a implicitement supposé que lorsque le programme est en train d'exécuter `divise()`, la fonction `main()` et ses variables existent encore et qu'elles attendent simplement la fin de `divise()`. Autrement dit :

Un appel de fonction est un mécanisme qui permet de partir exécuter momentanément cette fonction puis de retrouver la suite des instructions et les variables qu'on avait provisoirement quittées.

Les fonctions s'appelant les unes les autres, on se retrouve avec des appels de fonctions imbriqués les uns dans les autres : `main()` appelle `divise()` qui lui-même appelle `verifie()`². Plus précisément, cette imbrication est un *empilement* et on parle de **pile**

1. par exemple 24a et 24b

2. Et d'ailleurs `main()` a lui-même été appelé par une fonction à laquelle il renvoie un `int`.

des appels. Pour mieux comprendre cette pile, nous allons utiliser le debugueur. Avant cela, précisons ce qu'un informaticien entend par *pile*.

Pile/File

- Une **pile** est une structure permettant de mémoriser des données dans laquelle celles-ci s'empilent de telle sorte que celui qui est rangé en dernier dans la pile en est extrait en premier. En anglais, une pile (*stack*) est aussi appelée LIFO (last in first out³). On y empile (*push*) et on y dépile (*pop*) les données. Par exemple, après un `push(1)`, un `push(2)` et un `push(3)`, le premier `pop()` donnera 3, le deuxième `pop()` donnera 2 et un dernier `pop()` donnera 1.
- Pour une **file** (en anglais *queue*), c'est la même chose mais le premier arrivé est le premier sorti (FIFO). Par exemple, après un `push(1)`, un `push(2)` et un `push(3)`, le premier `pop()` donnera 1, le deuxième `pop()` donnera 2 et un dernier `pop()` donnera 3.

7.1.2 Pile des appels et debugueur

Observons donc la figure 7.1 obtenue en lançant notre programme d'exemple sous debugueur. En regardant la partie gauche de chaque étape, nous pouvons voir la pile des appels. La partie droite affiche le contenu des variables, paramètres et valeurs de retour dont nous pouvons constater la cohérence avec le tableau précédent.

- (a) Comme l'indique la pile des appels, nous sommes ligne 24 de la fonction `main()`, qui se trouve être le seul élément de la pile (`main` est la fonction d'entrée du programme). Vérifiez les variables et le fait qu'elles valent n'importe quoi (ici 0 pour `quotient` et 4197024 pour `reste`) tant qu'elles ne sont pas initialisées ou affectées.
- (b) Avançons en pas-à-pas détaillé (touche F11) jusqu'à la ligne 12. Nous sommes dans la fonction `divise()`, `q` vient de valoir 7, et la ligne 24 de `main()` est descendue d'un cran dans la pile des appels.
- (c) Nous sommes maintenant à la ligne 5 dans `verifie()`. La pile des appels à un niveau de plus, `divise()` est en attente à la ligne 13 et `main()` toujours en 24. Vérifiez au passage que la variable `q` affichée est bien celle de `verifie()`, qui vaut 3 et non pas celle de `divise()`.
- (d) Ici, **l'exécution du programme n'a pas progressé** et nous en sommes toujours à la ligne 5. Simplement, le debugueur offre la possibilité en cliquant sur la pile d'appel de regarder ce qui se passe à un des niveaux inférieurs, notamment pour afficher les instructions et les variables de ce niveau. Ici, en cliquant sur la ligne de `divise()` dans la fenêtre de la pile d'appel, nous voyons apparaître **la ligne 13 et ses variables dans leur état alors que le programme est en 5**. Entre autres, le `q` affiché est celui de `divise()` et vaut 7.
- (e) Toujours sans avancer, voici l'état du `main()` et de ses variables (entre autres, `reste` est bien passé à 2 depuis la ligne 12 de `divise()`).

3. Dernier rentré, premier sorti.

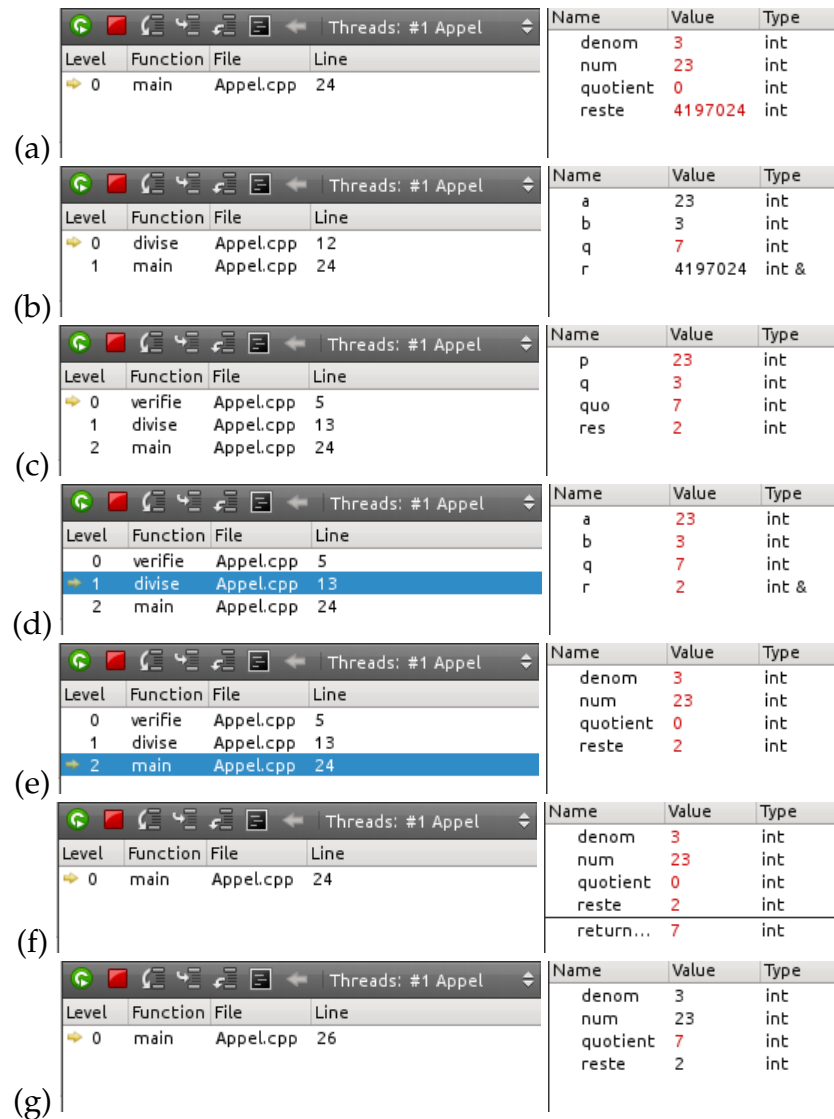


FIGURE 7.1 – Appels de fontions

pile	variable	valeur	pile	variable	valeur	pile	variable	valeur
				place libre				
			top→	p _v	23			
				q _v	3			
				quo	7			
				res	2			
top→	a	23		a	23			
	b	3		b	3			
	q _d	7		q _d	7			
	r	[reste]		r	[reste]			
	denom	3		denom	3	top→	denom	3
	num	23		num	23		num	23
	quotient	?		quotient	?		quotient	7
	reste	?		reste	2		reste	2
	pris par les fonctions	...		pris par les fonctions	...		pris par les fonctions	...
	avant main	...		avant main	...		avant main	...

FIGURE 7.2 – Pile et variables locales. De gauche à droite : étape (b) (ligne 12), étape (c) (ligne 5) et étape (g) (ligne 25/26).

- (f) Nous exécutons maintenant la suite jusqu'à nous retrouver en ligne 24 au retour de `divise()`. Pour cela, on peut faire du pas-à-pas détaillé, ou simplement deux fois de suite un pas-à-pas sortant⁴ (Maj-F11) pour relancer jusqu'à sortir de `verifie()`, puis jusqu'à sortir de `divise()`. On voit bien `quotient`, qui est encore non défini, et aussi la valeur de retour de `divise()`, non encore affectée à `quotient`.
- (g) Un pas-à-pas de plus et nous sommes en 25/26. La variable `quotient` vaut enfin 7.

7.2 Variables Locales

Il va être important pour la suite de savoir comment les paramètres et les variables locales sont stockés en mémoire.

7.2.1 Paramètres

Pour les paramètres, c'est simple :

Les paramètres sont en fait des variables locales ! Leur seule spécificité est d'être initialisés dès le début de la fonction avec les valeurs passées à l'appel de la fonction.

4. Step Out ou Maj-F11 ou . Notez aussi la possibilité de continuer le programme jusqu'à une certaine ligne sans avoir besoin de mettre un point d'arrêt temporaire sur cette ligne mais simplement en cliquant sur la ligne avec le bouton de droite et en choisissant "Run to line...", ()

7.2.2 La pile

Les variables locales (et donc les paramètres) ne sont pas mémorisées à des adresses fixes en mémoire⁵, décidées à la compilation. Si on faisait ça, les adresses mémoire en question devraient être réservées pendant toute l'exécution du programme : on ne pourrait y ranger les variables locales d'autres fonctions. La solution retenue est beaucoup plus économe en mémoire⁶ :

Les variables locales sont mémorisées dans un pile :

- **Quand une variables locale est créée, elle est rajoutée en haut de cette pile.**
- **Quand elle meurt (en général quand on quitte sa fonction) elle est sortie de la pile.**

Ainsi, au fur et à mesure des appels, les variables locales s'empilent : la mémoire est utilisée juste pendant le temps nécessaire. La figure 7.2 montre trois étapes de la pile pendant l'exécution de notre exemple.

7.3 Fonctions récursives

Un fonction récursive est une fonction qui s'appelle elle-même. La fonction la plus classique pour illustrer la récursivité est la factorielle⁷. Voici une façon simple et récursive de la programmer :

```

5 int fact1(int n)
6 {
7     if (n==1)
8         return 1;
9     return n*fact1(n-1);
10 }
```

On remarque évidemment que les fonctions récursives contiennent (en général au début, et en tout cas avant l'appel récursif !) une *condition d'arrêt* : ici si n vaut 1, la fonction retourne directement 1 sans s'appeler elle-même⁸.

7.3.1 Pourquoi ça marche ?

Si les fonctions avaient mémorisé leurs variables locales à des adresses fixes, la récursivité n'aurait pas pu marcher : l'appel récursif aurait écrasé les valeurs des variables. Par exemple, `fact1(3)` aurait écrasé la valeur 3 mémorisée dans n par un 2 en appelant `fact1(2)` ! C'est justement grâce à la pile que le n de `fact1(2)` n'est pas le même que celui de `fact1(3)`. Ainsi, l'appel à `fact1(3)` donne-t'il le tableau suivant :

5. Souvenons-nous du chapitre 2.

6. Et permettra de faire des fonctions récursives, cf section suivante !

7. **Coin des collégiens** : La factorielle d'un nombre entier n s'écrit $n!$ et vaut $n! = 1 \times 2 \times \dots \times n$.

8. Le fait de pouvoir mettre des `return` au milieu des fonctions est ici bien commode !

Ligne	$n_{fact1(3)}$	$ret_{fact1(3)}$	$n_{fact1(2)}$	$ret_{fact1(2)}$	$n_{fact1(1)}$	$ret_{fact1(1)}$
5 _{fact1(3)}	3					
9a _{fact1(3)}	3					
5 _{fact1(2)}	3		2			
9a _{fact1(2)}	3		2			
5 _{fact1(1)}	3		2		1	
8 _{fact1(1)}	3		2		1	1
10 _{fact1(1)}	3		2			1
9b _{fact1(2)}	3		2	2		1
10 _{fact1(2)}	3			2		
9b _{fact1(3)}	3	6		2		
10 _{fact1(3)}		6				

Ce tableau devient difficile à écrire maintenant qu'on sait que les variables locales ne dépendent pas que de la fonction mais changent à chaque appel ! On est aussi obligé de préciser, pour chaque numéro de ligne, quel appel de fonction est concerné. Si on visualise la pile, on comprend mieux pourquoi ça marche. Ainsi, arrivés en ligne 8 de fact1 (1) pour un appel initial à fact1 (3), la pile ressemble à :

pile	variable	valeur
	place libre	
top→	$n_{fact1(1)}$	1
	$n_{fact1(2)}$	2
	$n_{fact1(3)}$	3

ce que l'on peut aisément vérifier avec le débogueur. Finalement :

Les fonctions récursives ne sont pas différentes des autres. C'est le système d'appel des fonctions en général qui rend la récursivité possible.

7.3.2 Efficacité

Une fonction récursive est simple et élégante à écrire quand le problème s'y prête⁹. Nous venons de voir qu'elle n'est toujours pas facile à suivre ou à debugger. Il faut aussi savoir que

la pile des appels n'est pas infinie et même relativement limitée.

Ainsi, le programme suivant

```

22 // Fait déborder la pile
23 int fact3(int n)
24 {
25     if (n==1)
26         return 1;
27     return n*fact3(n+1); // erreur!
28 }
```

dans lequel une erreur s'est glissée va s'appeler théoriquement à l'infini et en pratique s'arrêtera avec une erreur de dépassement de la pile des appels¹⁰. Mais la vraie raison qui fait qu'on évite parfois le récursif est qu'

9. C'est une erreur classique de débutant que de vouloir abuser du récursif.

10. Sous Visual, il s'arrête pour $n = 5000$ environ.

appeler une fonction est un mécanisme coûteux !

Lorsque le corps d'une fonction est suffisamment petit pour que le fait d'appeler cette fonction ne soit pas négligeable devant le temps passé à exécuter la fonction elle-même, il est préférable d'éviter ce mécanisme d'appel¹¹. Dans le cas d'une fonction récursive, on essaie donc s'il est nécessaire d'écrire une version *dérécursivée* (ou *itérative*) de la fonction. Pour notre factorielle, cela donne :

```
// Version itérative
int fact2(int n)
{
    int f=1;
    for (int i=2; i<=n; i++)
        f*=i;
    return f;
}
```

ce qui après tout n'est pas si terrible.

Enfin, il arrive qu'écrire une fonction sous forme récursive ne soit pas utilisable pour des raisons de complexité. Une exemple classique est la suite de Fibonacci définie par :

$$\begin{cases} f_0 = f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \end{cases}$$

et qui donne : 1, 1, 2, 3, 5, 8, ... En version récursive :

```
32 // Très lent !
33 int fib1(int n) {
34     if (n<2)
35         return 1;
36     return fib1(n-2)+fib1(n-1);
37 }
```

cette fonction a la mauvaise idée de s'appeler très souvent : $n = 10$ appelle $n = 9$ et $n = 8$, mais $n = 9$ appelle lui aussi $n = 8$ de son côté en plus de $n = 7$, $n = 7$ qui lui-même est appelé par tous les $n = 8$ lancés, etc. Bref, cette fonction devient rapidement très lente. Ainsi, pour $n = 40$, elle s'appelle déjà 300.000.000 de fois elle-même, ce qui prend un certain temps ! Il est donc raisonnable d'en programmer une version dérécurivée :

```
39 // Dérécursivée
40 int fib2(int n) {
41     int fnm2=1, fnm1=1;
42     for (int i=2; i<=n; i++) {
43         int fn=fnm2+fnm1;
44         fnm2=fnm1;
45         fnm1=fn;
46     }
47     return fnm1;
48 }
```

11. Nous verrons dans un autre chapitre les fonctions `inline` qui répondent à ce problème.

Mentionnons aussi qu'il existe des fonctions suffisamment tordues pour que leur version récursive ne se contente pas de s'appeler un grand nombre de fois en tout, mais un grand nombre de fois *en même temps*, ce qui fait qu'indépendamment des questions d'efficacité, leur version récursive fait déborder la pile d'appels !

7.4 Le tas

La pile n'est pas la seule zone de mémoire utilisée par les programmes. Il y a aussi le *tas* (*heap* en anglais).

7.4.1 Limites

La pile est limitée en taille. La pile d'appel n'étant pas infinie et les variables locales n'étant pas en nombre illimité, il est raisonnable de réserver une pile de relativement petite taille. Essayez donc le programme :

```
32 int main ()
33 {
34     const int n=500000;
35     int t[n];
36     ...
37 }
```

Il s'exécute avec une erreur : "stack overflow". La variable locale *t* n'est pas trop grande pour l'ordinateur¹² : elle est trop grande pour tenir dans la pile. Jusqu'à présent, on savait qu'on était limité aux tableaux de taille constante. En réalité, on est aussi limité aux **petits tableaux**. Il est donc grand temps d'apprendre à utiliser le tas !

7.4.2 Tableaux de taille variable

Nous fournissons ici une règle à appliquer en aveugle. Sa compréhension viendra plus tard si nécessaire.

Lorsqu'on veut utiliser un tableau de taille variable, il n'y a que deux choses à faire, mais elle sont essentielles **toutes les deux**¹³ :

1. Remplacer `int t[n]` par `int* t=new int[n]` (ou l'équivalent pour un autre type que `int`)
2. Lorsque le tableau doit mourir (en général en fin de fonction), ajouter la ligne `delete[] t;`

Le non respect de la règle 2 fait que le tableau reste en mémoire jusqu'à la fin du programme, ce qui entraîne en général une croissance anarchique de la mémoire utilisée (on parle de *fuite de mémoire*). Pour le reste, on ne change rien. Programmer un tableau de cette façon fait qu'il est mémorisé **dans le tas et non plus dans la pile**. On fait donc ainsi :

12. 500000x4 soit 2Mo seulement !

13. Et le débutant oublie toujours la deuxième, ce qui a pour conséquence des programmes qui grossissent en quantité de mémoire occupée...

1. Pour les tableaux de taille variable.
2. Pour les tableaux de grande taille.

Voici ce que cela donne sur un petit programme :

```

1  #include <iostream>
2  using namespace std;
3
4  void rempli(int t[], int n)
5  {
6      for (int i=0;i<n;i++)
7          t[i]=i+1;
8  }
9
10 int somme(int t[], int n)
11 {
12     int s=0;
13     for (int i=0;i<n;i++)
14         s+=t[i];
15     return s;
16 }
17
18 void fixe()
19 {
20     const int n=5000;
21     int t[n];
22     rempli(t,n);
23     int s=somme(t,n);
24     cout << s << " devrait valoir " << n*(n+1)/2 << endl;
25 }
26
27 void variable()
28 {
29     int n;
30     cout << "Un entier SVP: ";
31     cin >> n;
32     int* t=new int[n]; // Allocation
33     rempli(t,n);
34     int s=somme(t,n);
35     cout << s << " devrait valoir " << n*(n+1)/2 << endl;
36     delete[] t; // Desallocation: ne pas oublier!
37 }
38
39 int main()
40 {
41     fixe();
42     variable();
43     return 0;
44 }

```


7.4.3 Essai d'explication

Ce qui suit n'est pas essentiel pour un débutant mais peut éventuellement répondre à ses interrogations. S'il comprend, tant mieux, sinon, qu'il oublie et se contente pour l'instant de la règle précédente !

Pour avoir accès à toute la mémoire de l'ordinateur¹⁴, on utilise le tas. Le tas est une zone mémoire que le programme possède et qui peut croître s'il en fait la demande au système d'exploitation (et s'il reste de la mémoire de libre évidemment). Pour utiliser le *tas*, on appelle une fonction d'*allocation* à laquelle on demande de réserver en mémoire de la place pour un certain nombre de variables. C'est ce que fait **new int[n]**.

Cette fonction retourne l'adresse de l'emplacement mémoire qu'elle a réservé. Nous n'avons jamais rencontré de type de variable capable de mémoriser une adresse. Il s'agit des pointeurs dont nous reparlerons plus tard. Un pointeur vers de la mémoire stockant des **int** est de type **int***. D'où le **int*** t pour mémoriser le retour du **new**.

Ensuite, un pointeur peut s'utiliser comme un tableau, y compris comme paramètre d'une fonction.

Enfin, il ne faut pas oublier de libérer la mémoire au moment où le tableau de taille constante aurait disparu : c'est ce que fait la fonction **delete[]** t qui libère la mémoire pointée par t.

7.5 L'optimiseur

Mentionnons ici un point important qui était négligé jusqu'ici, mais que nous allons utiliser en TP.

Il y a plusieurs façons de traduire en langage machine un source C++. Le résultat de la compilation peut donc être différent d'un compilateur à l'autre. Au moment de compiler, on peut aussi rechercher à produire un exécutable le plus rapide possible : on dit que le compilateur *optimise* le code. En général, l'optimisation nécessite un plus grand travail mais aussi des transformations qui font que le code produit n'est plus facilement debuggable. On choisit donc en pratique entre un code debuggable et un code optimisé.

Jusqu'ici, nous utilisons toujours le compilateur en mode "Debug". Lorsqu'un programme est au point (et seulement lorsqu'il l'est), on peut basculer le compilateur en mode "Release" pour avoir un programme plus performant. Dans certains cas, les gains peuvent être considérables. Un programmeur expérimenté fait même en sorte que l'optimiseur puisse efficacement faire son travail. Ceci dit, il faut respecter certaines règles :

- Ne pas debugger quand on est en mode Release (!)
- Rester en mode Debug le plus longtemps possible pour bien mettre au point le programme.

14. Plus exactement à ce que le système d'exploitation veut bien attribuer au maximum à chaque programme, ce qui est en général réglable mais en tout cas moins que la mémoire totale, bien que beaucoup plus que la taille de la pile.

7.6 Assertions

Voici une fonction très utile pour faire des programmes moins buggés ! La fonction `assert()` prévient quand un test est faux. Elle précise le fichier et le numéro de ligne où elle se trouve, offre la possibilité de debugger le programme, etc. Elle ne ralentit pas les programmes car elle disparaît à la compilation en mode Release. C'est une fonction peu connue des débutants, et c'est bien dommage ! Par exemple :

```
#include <cassert>
...
int n;
cin >> n;
assert(n>0);
int* t=new int[n]; // Allocation
```

Si l'utilisateur entre une valeur négative, les conséquences pourraient être fâcheuses. En particulier une valeur négative de `n` serait interprétée comme un grand entier (car le `[]` attend un entier non signé, ainsi -1 serait compris comme le plus grand `int` possible) et le `new` serait probablement un échec. A noter que si `n==0`, un tableau nul, l'allocation marche. Mais dans ce cas `t[0]` n'existe même pas ! La seule chose qu'on peut donc faire avec un tableau nul c'est le désallouer avec `delete[] t;`. Il est toujours utile de se prémunir contre une telle exception en vérifiant que la valeur est raisonnable.

7.7 Examens sur machine

Nous vous conseillons aussi de vous confronter aux examens proposés sur le site du cours. Vous avez toutes les connaissances nécessaires.

7.8 Fiche de référence

Fiche de référence (1/2)		
Variables — Définition : <pre>int i; int k,l,m;</pre> — Affectation : <pre>i=2; j=i; k=l=3;</pre> — Initialisation : <pre>int n=5,o=n;</pre> — Constantes : <pre>const int s=12;</pre> — Portée : <pre>int i; // i=j; interdit! int j=2; i=j; // OK! if (j>1) { int k=3; j=k; // OK! } //i=k; interdit!</pre> — Types : <pre>int i=3; double x=12.3; char c='A'; string s="hop"; bool t=true; float y=1.2f; unsigned int j=4; signed char d=-128; unsigned char d=25; complex<double> z(2,3);</pre> — Variables globales : <pre>int n; const int m=12; void f() { n=10; // OK int i=m; // OK ... }</pre> — Conversion : <pre>int i=int(x),j; float x=float(i)/j;</pre> — Pile/Tas	— Comparaison : <pre>== != < > <= >=</pre> — Négation : ! — Combinaisons : && — if (i==0) j=1; — if (i==0) j=1; else j=2; — if (i==0) { j=1; k=2; } — bool t=(i==0); if (t) j=1; — switch (i) { case 1: ...; ...; break; case 2: case 3: ...; break; default: ...; }	<pre>void affiche(int a) { cout << a << endl; }</pre> — Déclaration : <pre>int plus(int a,int b);</pre> — Retour : <pre>int signe(double x) { if (x<0) return -1; if (x>0) return 1; return 0; }</pre> <pre>void afficher(int x, int y) { if (x<0 y<0) return; if (x>=w y>=h) return; DrawPoint(x,y,RED); }</pre> — Appel : <pre>int f(int a) { ... } int g() { ... } ... int i=f(2),j=g();</pre> — Références : <pre>void swap(int& a, int& b){ int tmp=a; a=b;b=tmp; }</pre> <pre>... int x=3,y=2; swap(x,y);</pre> — Surcharge : <pre>int hasard(int n); int hasard(int a, int b); double hasard();</pre> — Opérateurs : <pre>vect operator+(vect A,vect B) { ... }</pre> <pre>... vect C=A+B;</pre> — Pile des appels — Itératif/Récuratif
Tests	Boucles — do { ... } while(!ok); — int i=1; while(i<=100) { ... i=i+1; } — for(int i=1;i<=10;i++) ... — for(int i=1,j=10;j>i; i=i+2,j=j-3) ...	
	Fonctions — Définition : <pre>int plus(int a,int b){ int c=a+b; return c; }</pre>	

Fiche de référence (2/2)**Tableaux**

— Définition :

```
— double x[5], y[5];
   for(int i=0; i<5; i++)
       y[i]=2*x[i];
```

— Initialisation :

```
— const int n=5;
   int i[n], j[2*n];
```

— Affectation :

```
— int s[3]={1,2,3}, t[3];
   for (int i=0; i<3; i++)
       t[i]=s[i];
```

— En paramètre :

```
— void init(int t[4]) {
   for(int i=0; i<4; i++)
       t[i]=0;
}

— void init(int t[],
           int n) {
   for(int i=0; i<n; i++)
       t[i]=0;
}
```

— Taille variable :

```
— int* t=new int[n];
   ...
   delete[] t;
```

Structures

```
— struct Point {
   double x,y;
   Color c;
};
...
Point a;
a.x=2.3; a.y=3.4;
a.c=Red;
Point b={1,2.5,Blue};
```

Compilation séparée

```
— #include "vect.h", aussi
   dans vect.cpp
— Fonctions : déclarations dans
   le .h, définitions dans le
   .cpp
— Types : définitions dans le .h
— Ne déclarer dans le .h que les
   fonctions utiles.
— #pragma once au début du
   fichier.
— Ne pas trop découper...
```

Divers

```
— i++;
   i--;
   i-=2;
   j+=3;
— j=i%n; // Modulo
— #include <cstdlib>
   ...
   i=rand()%n;
   x=rand()/
       double(RAND_MAX);
— #include <ctime>
   // Un seul appel
   srand((unsigned int)
       time(0));
— #include <cmath>
   double sqrt(double x);
   double cos(double x);
   double sin(double x);
   double acos(double x);
— #include <string>
   using namespace std;
   string s="hop";
   char c=s[0];
   int l=s.size();
— #include <ctime>
   s=double(clock())
       /CLOCKS_PER_SEC;
```

Entrées/Sorties

```
— #include <iostream>
   using namespace std;
   ...
   cout <<"I="<<i<<endl;
   cin >> i >> j;
```

Clavier

```
— Debug : F5 
— Step over : F10 
— Step inside : F11 
— Indent : Ctrl+A, Ctrl+I
— Step out : Maj+F11 
```

Erreurs fréquentes

```
— Pas de définition de fonction
   dans une fonction !
— int q=r=4; // NON!
— if (i=2) // NON!
   if i==2 // NON!
   if (i==2) then // NON!
— for(int i=0, i<100, i++)
   // NON!
— int f() {...}
   int i=f; // NON!
```

```
— double x=1/3; // NON!
   int i,j;
   x=i/j; // NON!
   x=double(i/j); //NON!
— double x[10], y[10];
   for(int i=1; i<=10; i++)
       y[i]=2*x[i]; //NON
— int n=5;
   int t[n]; // NON
— int f()[4] { // NON!
   int t[4];
   ...
   return t; // NON!
}
int t[4]; t=f();
— int s[3]={1,2,3}, t[3];
   t=s; // NON!
— int t[2];
   t={1,2}; // NON!
— struct Point {
   double x,y;
} // NON!
— Point a;
   a={1,2}; // NON!
— #include "tp.cpp" //NON
```

Imagine++

```
— Voir documentation...
```

Conseils

```
— Nettoyer en quittant.
— Erreurs et warnings : cliquer.
— Indenter.
— Ne pas laisser de warning.
— Utiliser le débogueur.
— Faire des fonctions.
— Tableaux : pas pour transcrire
   une formule mathématique !
— Faire des structures.
— Faire des fichiers séparés.
— Le .h doit suffire à l'utilisa-
   teur (qui ne doit pas regarder
   le .cpp)
— Ne pas abuser du récursif.
— Ne pas oublier delete.
— Compiler régulièrement.
— #include <cassert>
   ...
   assert(x!=0);
   y=1/x;
```

Chapitre 8

Allocation dynamique

Nous revenons une fois de plus sur l'utilisation du tas pour gérer des tableaux de taille variable. Après avoir mentionné l'existence de tableaux bidimensionnels de taille fixe, nous détaillons l'allocation dynamique¹ déjà vue en 7.4.2 et expliquons enfin les pointeurs, du moins partiellement. A travers l'exemple des matrices (et des images en TP) nous mélangeons structures et allocation dynamique. Il s'agira là de notre structure de donnée la plus complexe avant l'arrivée tant attendue - et maintenant justifiée - des objets...

—

8.1 Tableaux bidimensionnels

8.1.1 Principe

Il existe en C++ des tableaux à deux dimensions. Leur utilisation est similaire à celle des tableaux standards :

- Il faut utiliser des crochets (lignes 1 et 4 du programme ci-dessous). Attention : `[i][j]` et non `[i,j]`.
- L'initialisation est possible avec des accolades (ligne 5). Attention : accolades imbriquées.
- Leurs dimensions doivent être constantes (lignes 6 et 7).

```
1  int A[2][3];
2  for (int i=0;i<2;i++)
3      for (int j=0;j<3;j++)
4      A[i][j]=i+j;
5  int B[2][3]={ {1,2,3},{4,5,6}};
6  const int M=2,N=3;
7  int C[M][N];
```

B[0] →	1	2	3
B[1] →	4	5	6

Tableau 2D. Notez que B[0] est le tableau 1D {1,2,3} et B[1] le tableau 1D {4,5,6}.

La figure ci-dessus montre le tableau B. À noter que B[0] et B[1] sont des tableaux 1D représentant les lignes de B.

1. c'est-à-dire l'allocation de mémoire dans le tas avec `new` et `delete`.

8.1.2 Limitations

Vis-à-vis des fonctions, les particularités sont les mêmes qu'en 1D :

- Impossible de retourner un tableau 2D.
- Passage uniquement par variable.

mais avec une restriction supplémentaire :

On est obligé de préciser les dimensions d'un tableau 2D paramètre de fonction.

Impossible donc de programmer des fonctions qui peuvent travailler sur des tableaux de différentes tailles comme dans le cas 1D (cf 4.3.1). C'est très restrictif et explique que les tableaux 2D ne sont pas toujours utilisés. On peut donc avoir le programme suivant :

```

1 // Passage de paramètre
2 double trace(double A[2][2]) {
3     double t=0;
4     for (int i=0;i<2;i++)
5         t+=A[i][i];
6     return t;
7 }
8
9 // Le passage est toujours par référence ...
10 void set(double A[2][3]) {
11     for (int i=0;i<2;i++)
12         for (int j=0;j<3;j++)
13             A[i][j]=i+j;
14 }
15
16 ...
17 double D[2][2]={ {1,2},{3,4}};
18 double t=trace(D);
19 double E[2][3];
20 set(E);
21 ...
```

mais il est impossible de programmer une fonction trace() ou set() qui marche pour différentes tailles de tableaux 2D comme on l'aurait fait en 1D :

```

1 // OK
2 void set(double A[],int n,double x) {
3     for (int i=0;i<n;i++)
4         A[i]=x;
5 }
6 // NON!!!!!!!!!!!!!!!!!!!!
7 // double A[][] est refusé
8 void set(double A[][],double m,double n,double x) {
9     for (int i=0;i<m;i++)
10         for (int j=0;j<n;j++)
```

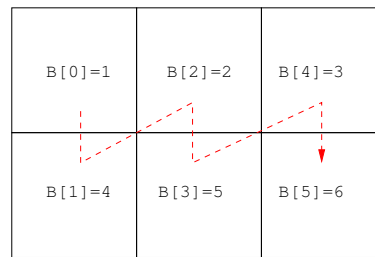


FIGURE 8.1 – La matrice B de l'exemple précédent stockée en 1D.

```

11         A[i][j]=x;
12     }

```

8.1.3 Solution

En pratique, dès que l'on doit manipuler des tableaux de dimension 2 (ou plus !) de différentes tailles, on les mémorise dans des tableaux 1D en stockant par exemple les colonnes les unes après les autres pour profiter des avantages des tableaux 1D. Ainsi, on stockera une matrice A de m lignes de n colonnes dans un tableau T de taille mn en plaçant l'élément $A(i, j)$ en $T(i + mj)$. La Fig. 8.1 montre le tableau B de l'exemple précédent stocké comme tableau 1D. On peut alors écrire :

```

1 void set(double A[], int m, int n) {
2     for (int i=0; i<m; i++)
3         for (int j=0; j<n; j++)
4             A[i+m*j]=i+j;
5 }
6 ...
7 double F[2*3];
8 set(F, 2, 3);
9 double G[3*5];
10 set(G, 3, 5);

```

ou par exemple, ce produit matrice vecteur dans lequel les vecteurs et les matrices sont stockés dans des tableaux 1D :

```

1 // y=Ax
2 void produit(double A[], int m, int n, double x[], double y[])
3 {
4     for (int i=0; i<m; i++) {
5         y[i]=0;
6         for (int j=0; j<n; j++)
7             y[i]+=A[i+m*j]*x[j];
8     }
9 }
10 ...
11 ...
12 double P[2*3], x[3], y[2];
13 ...

```

```

14      // P = ... x = ...
15      produit(P,2,3,x,y); // y=Px

```

8.2 Allocation dynamique

Il n'y a pas d'allocation dynamique possible pour les tableaux 2D. Il faut donc vraiment les mémoriser dans des tableaux 1D comme expliqué ci-dessus pour pouvoir les allouer dynamiquement dans le tas. L'exemple suivant montre comment faire. Il utilise la fonction `produit()` donnée ci-dessus sans qu'il soit besoin de la redéfinir :

```

1      int m,n;
2      ...
3      double* A=new double[m*n];
4      double* x=new double[n];
5      double* y=new double[m];
6      ...
7      // A = ... x = ...
8      produit(A,m,n,x,y); // y=Ax
9      ...
10     delete[] A;
11     delete[] x;
12     delete[] y;

```

8.2.1 Pourquoi ça marche ?

Il est maintenant temps d'expliquer pourquoi, une fois alloués, nous pouvons utiliser des tableaux dynamiques exactement comme des tableaux de taille fixe. Il suffit de comprendre les étapes suivantes :

1. **int** `t[n]` définit une variable locale, donc de la mémoire dans la pile, capable de stocker `n` variables **int**.
2. **int*** `t` définit une variable de type "pointeur" d'**int**, c'est-à-dire que `t` peut mémoriser l'adresse d'une zone mémoire contenant des **int**.
3. **new int[n]** alloue dans le tas une zone mémoire pouvant stocker `n` **int** et renvoie l'adresse de cette zone. D'où le **int*** `t=new int[n]`
4. **delete[]** `t` libère dans le tas l'adresse mémorisée dans `t`.
5. Lorsque `t` est un tableau de taille fixe `t[i]` désigne son i^{me} élément. Lorsque `t` est un pointeur d'**int**, `t[i]` désigne la variable **int** stockée i places² plus loin en mémoire que celle située à l'adresse `t`. Ainsi, après un **int** `t[n]` comme après un **int*** `t=new int[n]`, la syntaxe `t[i]` désigne bien ce qu'on veut.
6. Lorsque `t` est un tableau de taille fixe, la syntaxe `t` tout court désigne l'adresse (dans la pile) à laquelle le tableau est mémorisé. De plus, lorsqu'une fonction prend un tableau comme paramètre, la syntaxe **int** `s[]` signifie en réalité que `s` est l'adresse du tableau. Ce qui fait qu'en fin de compte :

2. Ici, une place est évidemment le nombre d'octets nécessaires au stockage d'un **int**.

- une fonction `f(int s[])` est conçue pour qu'on lui passe une adresse `s`
- elle marche évidemment avec les tableaux alloués dynamiquement qui ne sont finalement que des adresses
- c'est plutôt l'appel `f(t)`, avec `t` tableau de taille fixe, qui s'adapte en passant à `f` l'adresse où se trouve le tableau.
- logiquement, on devrait même déclarer `f` par `f(int* s)` au lieu de `f(int s[])`. Les deux sont en fait possibles et synonymes.

Vous pouvez donc maintenant programmer, *en comprenant*, ce genre de choses :

```

1 double somme(double* t, int n) { // Syntaxe "pointeur"
2     double s=0;
3     for (int i=0; i<n; i++)
4         s+=t[i];
5     return s;
6 }
7     ...
8     int t1[4];
9     ...
10    double s1=somme(t1, 4);
11    ...
12    int* t2=new int[n];
13    ...
14    double s2=somme(t2, n);
15    ...
16    delete[] t2;
```

8.2.2 Erreurs classiques

Vous comprenez maintenant aussi les erreurs classiques suivantes (que vous n'éviterez pas pour autant!).

1. Oublier d'allouer :

```

int *t;
for (int i=0; i<n; i++)
    t[i]=... // Horreur: t vaut n'importe
              // quoi comme adresse
```

2. Oublier de désallouer :

```

void f(int n) {
    int *t=new int[n];
    ...
} // On oublie delete[] t;
  // Chaque appel à f() va perdre n int dans le tas!
```

3. Ne pas désallouer ce qu'il faut :

```

int* t=new int[n];
int* s=new int[n];
```

```

...
s=t; // Aie! Du coup, s contient la même adresse que t
      // (On n'a pas recopié la zone pointée par t dans celle
      // pointée par s!)
...
delete[] t; // OK
delete[] s; // Cata: Non seulement on ne libère pas la mémoire
            // initialement mémorisée dans s, mais en plus on
            // désalloue à nouveau celle qui vient d'être libérée!

```

8.2.3 Conséquences

Quand libérer?

Maintenant que vous avez compris **new** et **delete**, vous imaginez bien qu'on n'attend pas toujours la fin de l'existence du tableau pour libérer la mémoire. Le plus tôt est le mieux et on libère la mémoire dès que le tableau n'est plus utilisé :

```

1 void f() {
2     int t[10];
3     int* s=new int[n];
4     ...
5     delete[] s; // si s ne sert plus dans la suite...
6                 // Autant libérer maintenant...
7     ...
8 } // Par contre, t attend cette ligne pour mourir.

```

En fait, le tableau dont l'adresse est mémorisée dans *s* est alloué ligne 3 et libéré ligne 5. La variable *s* qui mémorise son adresse, elle, est créée ligne 3 et meurt ligne 8!

Pointeurs et fonctions

Il est fréquent que le **new** et le **delete** ne se fassent pas dans la même fonction (attention, du coup, aux oublis!). Ils sont souvent intégrés dans des fonctions. A ce propos, lorsque des fonctions manipulent des variables de type pointeur, un certain nombre de questions peuvent se poser. Il suffit de respecter la logique :

- Une fonction qui retourne un pointeur se déclare **int* f()**;

```

1 int* alloue(int n) {
2     return new int[n];
3 }
4 ....
5 int* t=alloue(10);
6 ...

```

- Un pointeur passé en paramètre à une fonction l'est par valeur. Ne pas mélanger avec le fait qu'un tableau est passé par référence! Considérez le programme suivant :

```

1 void f(int* t, int n) {

```

```

2      ....
3      t[i]=...; // On modifie t[i] mais pas t!
4      t=... // Une telle ligne ne changerait pas 's'
5              // dans la fonction appelante
6  }
7  ...
8      int* s=new int[m];
9      f(s,m);

```

En fait, c'est parce qu'on passe l'adresse d'un tableau qu'on peut modifier ses éléments. Par ignorance, nous disions que les tableaux étaient passés par référence en annonçant cela comme une exception. Nous pouvons maintenant rectifier :

Un tableau est en fait passé via son adresse. Cette adresse est passée par valeur. Mais ce mécanisme permet à la fonction appelée de modifier le tableau. Dire qu'un tableau est passé par référence était un abus de langage simplificateur.

- Si on veut vraiment passer le pointeur par référence, la syntaxe est logique : `int*& t`. Un cas typique de besoin est :

```

1 // t et n seront modifiés (et plus seulement t[i])
2 void alloue(int*& t,int& n) {
3     cin >> n; // n est choisi au clavier
4     t=new int[n];
5 }
6     ...
7     int* t;
8     int n;
9     alloue(t,n); // t et n sont affectés par alloue()
10    ...
11    delete[] t; // Ne pas oublier pour autant!

```

Bizzarerie ? Les lignes 7 et 8 ci-dessus auraient pu s'écrire `int* t,n;`. En fait, il faut remettre une étoile devant chaque variable lorsqu'on définit plusieurs pointeurs en même-temps. Ainsi, `int *t,s,*u;` définit deux pointeurs d'`int` (les variables `t` et `u`) et un `int` (la variable `s`).

8.3 Structures et allocation dynamique

Passer systématiquement un tableau et sa taille à toutes les fonctions est évidemment pénible. Il faut les réunir dans une structure. Je vous laisse méditer l'exemple suivant qui pourrait être un passage d'un programme implémentant des matrices³ et leur produit :

3. **Coin des enfants** : les matrices et les vecteurs vous sont inconnus. Ca n'est pas grave. Comprenez le source quand même et rattrapez vous avec le TP qui, lui, joue avec des images.

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  //=====
6  // fonctions sur les matrices
7  // pourraient etre dans un matrice.h et un matrice.cpp
8
9  struct Matrice {
10     int m,n;
11     double* t;
12 };
13
14 Matrice cree(int m,int n) {
15     Matrice M;
16     M.m=m;
17     M.n=n;
18     M.t=new double[m*n];
19     return M;
20 }
21
22 void detruit(Matrice M) {
23     delete[] M.t;
24 }
25
26 Matrice produit(Matrice A,Matrice B) {
27     if (A.n!=B.m) {
28         cout << "Erreur!" << endl;
29         exit(1);
30     }
31     Matrice C=cree(A.m,B.n);
32     for (int i=0;i<A.m;i++)
33         for (int j=0;j<B.n;j++) {
34             // Cij=Ai0*B0j+Ai1*B1j+...
35             C.t[i+C.m*j]=0;
36             for (int k=0;k<A.n;k++)
37                 C.t[i+C.m*j]+=A.t[i+A.m*k]*B.t[k+B.m*j];
38
39         }
40     return C;
41 }
42
43 void affiche(string s,Matrice M) {
44     cout << s << "␣=" << endl;
45     for (int i=0;i<M.m;i++) {
46         for (int j=0;j<M.n;j++)
47             cout << M.t[i+M.m*j] << "␣";
48     cout << endl;

```

```

49     }
50 }
51
52 //=====
53 // Utilisateur
54
55 int main()
56 {
57     Matrice A=cree(2,3);
58     for (int i=0;i<2;i++)
59         for (int j=0;j<3;j++)
60             A.t[i+2*j]=i+j;
61     affiche("A",A);
62     Matrice B=cree(3,5);
63     for (int i=0;i<3;i++)
64         for (int j=0;j<5;j++)
65             B.t[i+3*j]=i+j;
66     affiche("B",B);
67     Matrice C=produit(A,B);
68     affiche("C",C);
69     detruit(C);
70     detruit(B);
71     detruit(A);
72     return 0;
73 }

```

L'utilisateur n'a maintenant plus qu'à savoir qu'il faut allouer et libérer les matrices en appelant des fonctions mais il n'a pas à savoir ce que font ces fonctions. Dans cette logique, on pourra rajouter des fonctions pour qu'il n'ait pas non plus besoin de savoir comment les éléments de la matrice sont mémorisés. Il n'a alors même plus besoin de savoir que les matrices sont des structures qui ont un champ t ! (Nous nous rapprochons vraiment de la programmation objet...) Bref, on rajoutera en général :

```

10 double get(Matrice M,int i,int j) {
11     return M.t[i+M.m*j];
12 }
13
14 void set(Matrice M,int i,int j,double x) {
15     M.t[i+M.m*j]=x;
16 }

```

que l'utilisateur pourra appeler ainsi :

```

51     for (int i=0;i<2;i++)
52         for (int j=0;j<3;j++)
53             set(A,i,j,i+j);

```

et que celui qui programme les matrices pourra aussi utiliser pour lui :

```

39 void affiche(string s,Matrice M) {
40     cout << s << "␣=" << endl;
41     for (int i=0;i<M.m;i++) {

```

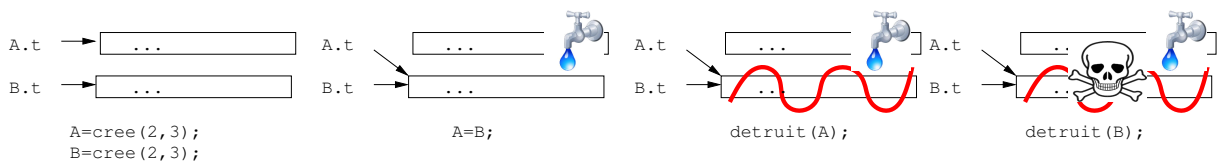


FIGURE 8.2 – Attention au double delete : le code `A=B` fait pointer deux fois sur la même zone mémoire alors qu'il n'y a plus de pointeur sur le tableau du haut (donc une fuite mémoire puisqu'il n'est plus possible de la libérer). Le `detrui(B)` libère une zone mémoire qui l'avait déjà été, avec des conséquences fâcheuses...

```

42         for (int j=0;j<M.n;j++)
43             cout << get(M,i,j) << " ";
44         cout << endl;
45     }
46 }

```

Attention, il reste facile dans ce contexte :

- D'oublier d'allouer.
- D'oublier de désallouer.
- De ne pas désallouer ce qu'il faut si on fait `A=B` entre deux matrices. (C'est alors deux fois la zone allouée initialement pour B qui est désallouée lorsqu'on libère A et B tandis que la mémoire initiale de A ne le sera jamais, comme on peut le voir sur la Fig. 8.2).

La programmation objet essaiera de faire en sorte qu'on ne puisse plus faire ces erreurs. Elle essaiera aussi de faire en sorte que l'utilisateur ne puisse plus savoir ce qu'il n'a pas besoin de savoir, de façon à rendre vraiment indépendantes la conception des matrices et leur utilisation.

8.4 Boucles et continue

Nous utiliserons dans le TP l'instruction **continue** qui est bien pratique. Voici ce qu'elle fait : lorsqu'on la rencontre dans une boucle, toute la fin de la boucle est sautée et on passe au tour suivant. Ainsi :

```

for (...) {
    ...
    if (A)
        continue;
    ...
    if (B)
        continue;
    ...
}

```

est équivalent à (et remplace avantageusement au niveau clarté et mise en page) :

```

for (...) {
    ...

```

```

if (!A) {
    ...
    if (!B) {
        ...
    }
}

```

Ceci est à rapprocher de l'utilisation du **return** en milieu de fonction pour évacuer les cas particuliers (section 7.3).



FIGURE 8.3 – Deux images et différents traitements de la deuxième (négatif, flou, relief, déformation, contraste et contours).

8.5 TP

Le TP que nous proposons en A.6 est une illustration de cette façon de manipuler des tableaux bidimensionnels dynamiques à travers des structures de données. Pour changer de nos passionnantes matrices, nous travaillerons avec des images (figure 8.3).

8.6 Fiche de référence

Fiche de référence (1/4)		
Boucles — do { ... } while(!ok); — int i=1; while(i<=100) { ...	i=i+1; } — for(int i=1;i<=10;i++) ... — for(int i=1,j=10;j>i; i=i+2,j=j-3) ...	— for (int i=...) for (int j=...) { //saute cas i==j if (i==j) continue; ... }

Fiche de référence (2/4)

Variables

- Définition :

```
int i;
int k,l,m;
```
- Affectation :

```
i=2;
j=i;
k=l=3;
```
- Initialisation :

```
int n=5,o=n;
```
- Constantes :

```
const int s=12;
```
- Portée :

```
int i;
// i=j; interdit!
int j=2;
i=j; // OK!
if (j>1) {
    int k=3;
    j=k; // OK!
}
//i=k; interdit!
```
- Types :

```
int i=3;
double x=12.3;
char c='A';
string s="hop";
bool t=true;
float y=1.2f;
unsigned int j=4;
signed char d=-128;
unsigned char d=25;
complex<double>
    z(2,3);
```
- Variables globales :

```
int n;
const int m=12;
void f() {
    n=10;    // OK
    int i=m; // OK
    ...
}
```
- Conversion :

```
int i=int(x),j;
float x=float(i)/j;
```
- Pile/Tas

Clavier

- Debug : F5 
- Step over : F10 
- Step inside : F11 
- Indent : Ctrl+A, Ctrl+I
- Step out : Maj+F11 
- Gest. tâches : Ctrl+Maj+Ech

Fonctions

- Définition :

```
int plus(int a,int b){
    int c=a+b;
    return c;
}
void affiche(int a) {
    cout << a << endl;
}
```
- Déclaration :

```
int plus(int a,int b);
```
- Retour :

```
int signe(double x) {
    if (x<0)
        return -1;
    if (x>0)
        return 1;
    return 0;
}
void afficher(int x,
              int y) {
    if (x<0 || y<0)
        return;
    if (x>=w || y>=h)
        return;
    DrawPoint(x,y,RED);
}
```
- Appel :

```
int f(int a) { ... }
int g() { ... }
...
int i=f(2),j=g();
```
- Références :

```
void swap(int& a,
          int& b){
    int tmp=a;
```

```
a=b;b=tmp;
}
```

```
...
int x=3,y=2;
swap(x,y);
```

- Surcharge :

```
int hasard(int n);
int hasard(int a,
           int b);
double hasard();
```
- Opérateurs :

```
vect operator+(
    vect A,vect B) {
    ...
}
...
vect C=A+B;
```
- Pile des appels
- Itératif/Récuratif

Tableaux

- Définition :

```
— double x[5],y[5];
  for(int i=0;i<5;i++)
    y[i]=2*x[i];
— const int n=5;
  int i[n],j[2*n];
```
- Initialisation :

```
int t[4]={1,2,3,4};
string s[2]={"ab","c"};
```
- Affectation :

```
int s[3]={1,2,3},t[3];
for (int i=0;i<3;i++)
    t[i]=s[i];
```
- En paramètre :

```
— void init(int t[4]) {
    for(int i=0;i<4;i++)
        t[i]=0;
}
— void init(int t[],
           int n) {
    for(int i=0;i<n;i++)
        t[i]=0;
}
```


Fiche de référence (3/4)		
Taille variable : <pre>int* t=new int[n]; ... delete[] t;</pre> En paramètre (suite) : <pre>void f(int* t,int n){ t[i]=... } void alloue(int*& t){ t=new int[n]; }</pre> 2D : <pre>int A[2][3]; A[i][j]=...; int A[2][3]= {{1,2,3},{4,5,6}}; void f(int A[2][2]);</pre> 2D dans 1D : <pre>int A[2*3]; A[i+2*j]=...;</pre> Taille variable (suite) : <pre>int *t,*s,n;</pre> Structures <pre>struct Point { double x,y; Color c; }; ... Point a; a.x=2.3; a.y=3.4; a.c=Red; Point b={1,2.5,Blue};</pre> Compilation séparée <pre>#include "vect.h", aussi dans vect.cpp Fonctions : déclarations dans le .h, définitions dans le .cpp Types : définitions dans le .h Ne déclarer dans le .h que les fonctions utiles. #pragma once au début du fichier. Ne pas trop découper...</pre> Tests <pre>Comparaison : == != < > <= >= Négation : !</pre>	— Combinaisons : && <pre>if (i==0) j=1; if (i==0) j=1; else j=2;</pre> <pre>if (i==0) { j=1; k=2; }</pre> <pre>bool t=(i==0); if (t) j=1;</pre> <pre>switch (i) { case 1: ...; ...; break; case 2: case 3: ...; break; default: ...; }</pre> Entrées/Sorties <pre>#include <iostream> using namespace std; ... cout <<"I="<<i<<endl; cin >> i >> j;</pre> Divers <pre>i++; i--; i-=2; j+=3; j=i%n; // Modulo #include <cstdlib> ... i=rand() %n; x=rand() / double(RAND_MAX); #include <ctime> // Un seul appel srand((unsigned int) time(0)); #include <cmath> double sqrt(double x); double cos(double x); double sin(double x); double acos(double x);</pre>	<pre>#include <string> using namespace std; string s="hop"; char c=s[0]; int l=s.size(); #include <ctime> s=double(clock()) /CLOCKS_PER_SEC; #include <cmath> double pi=M_PI;</pre> Erreurs fréquentes — Pas de définition de fonction dans une fonction ! <pre>int q=r=4; // NON!</pre> <pre>if (i=2) // NON! if i==2 // NON! if (i==2) then // NON!</pre> <pre>for(int i=0,i<100,i++) // NON!</pre> <pre>int f() {...} int i=f; // NON!</pre> <pre>double x=1/3; // NON! int i,j; x=i/j; // NON! x=double(i/j); //NON!</pre> <pre>double x[10],y[10]; for(int i=1;i<=10;i++) y[i]=2*x[i];//NON</pre> <pre>int n=5; int t[n]; // NON</pre> <pre>int f()[4] { // NON! int t[4]; ... return t; // NON! } int t[4]; t=f();</pre> <pre>int s[3]={1,2,3},t[3]; t=s; // NON!</pre> <pre>int t[2]; t={1,2}; // NON!</pre> <pre>struct Point { double x,y; } // NON!</pre> <pre>Point a; a={1,2}; // NON!</pre> <pre>#include "tp.cpp"//NON</pre>

Fiche de référence (4/4)		
<pre>int f(int t[][]); //NON int t[2,3]; // NON! t[i,j]=...; // NON! int* t; t[1]=...; // NON! int* t=new int[2]; int* s=new int[2]; s=t; // On perd s! delete[] t; delete[] s; //Déjà fait int *t,s; // s est int // non int* t=new int[n];</pre>	<pre>s=new int[n]; // NON!</pre> Imagine++ — Voir documentation... Conseils — Nettoyer en quittant. — Erreurs et warnings : cliquer. — Indenter. — Ne pas laisser de warning. — Utiliser le debugueur. — Faire des fonctions. — Tableaux : pas pour transcrire une formule mathématique !	— Faire des structures. — Faire des fichiers séparés. — Le .h doit suffire à l'utilisateur (qui ne doit pas regarder le .cpp) — Ne pas abuser du récursif. — Ne pas oublier delete. — Compiler régulièrement. — #include <cassert> ... <pre>assert(x!=0); y=1/x;</pre>

Chapitre 9

Premiers objets

Nous abordons maintenant notre dernière étape dans la direction d'une meilleure organisation des programmes. Tantôt nous structurions davantage les instructions (fonctions, fichiers), tantôt nous nous intéressions aux données (structures, tableaux). Nous allons maintenant penser données et instructions simultanément : c'est là l'idée première des objets, même s'ils possèdent de nombreux autres aspects¹. Enfin, nous justifierons l'emploi des objets par la notion d'"interface"².

—

9.1 Philosophie

Réunir les instructions en fonctions ou fichiers est une bonne chose. Réunir les données en tableaux ou structures aussi. Il arrive que les deux soient liées. C'est d'ailleurs ce que nous avons constaté naturellement dans les exemples des chapitres précédents, dans lesquels un fichier regroupait souvent une structure et un certain nombre de fonctions s'y rapportant. C'est dans ce cas qu'il faut faire des **objets**.

L'idée est simple : un objet est un type de donnée possédant un certain nombre de fonctionnalités propres³. Ainsi :

Ce ne sont plus les fonctions qui travaillent sur des données. Ce sont les données qui possèdent des fonctionnalités.

Ces "fonctionnalités" sont souvent appelées les **méthodes** de l'objet. En pratique, l'utilisation d'un objet remplacera ce genre d'instructions :

```
obj a;  
int i=f(a); // fonction f() appliquée à a  
par :
```

1. Le plus important étant l'héritage, que nous ne verrons pas dans ce cours, préférant nous consacrer à d'autres aspects du C++ plus indispensables et négligés jusqu'ici. . .

2. Nous exposerons une façon simple de créer des interfaces. Un programmeur C++ expérimenté utilisera plutôt de l'héritage et des *fonctions virtuelles pures*, ce qui dépasse largement ce cours !

3. Il arrive même parfois qu'un objet regroupe des fonctionnalités sans pour autant stocker la moindre donnée. Nous n'utiliserons pas ici cette façon de présenter les choses, dont le débutant pourrait rapidement abuser.

```
obj a;
int i=a.f(); // appel à la méthode f() de a
```

Vous l'avez compris, il s'agit ni plus ni moins de **"ranger" les fonctions dans les objets**. Attention, crions tout de suite haut et fort qu'

il ne faut pas abuser des objets, surtout lorsqu'on est débutant. Les dangers sont en effet :

- **de voir des objets là où il n'y en n'a pas. Instructions et données ne sont pas toujours liées.**
- **de mal penser l'organisation des données ou des instructions en objets.**

Un conseil donc : quand ça devient trop compliqué pour vous, abandonnez les objets.

Ce qui ne veut pas dire qu'un débutant ne doit pas faire d'objets. Des petits objets dans des cas simples sont toujours une bonne idée. Mais seule l'expérience permet de correctement organiser son programme, avec les bons objets, les bonnes fonctions, etc. Un exemple simple : lorsqu'une fonction travaille sur deux types de données, le débutant voudra souvent s'acharner à en faire malgré tout une méthode de l'un des deux objets, et transformer :

```
obj1 a;
obj2 b;
int i=f(a,b); // f() appliquée à a et b

en :

obj1 a;
obj2 b;
int i=a.f(b); // méthode f() de a appliquée à b
               // Est-ce bien la chose à faire????
```

Seuls un peu de recul et d'expérience permettent de rester simple quand il le faut. Le premier code était le plus logique : la fonction `f()` n'a souvent rien à faire chez `a`, ni chez `b`.

9.2 Exemple simple

On l'aura compris dans les exemples précédents, les méthodes des objets sont considérées comme faisant partie du type de l'objet, au même titre que ses champs. D'ailleurs, les champs d'un objet sont parfois appelés *membres* de l'objet, et ses méthodes des *fonctions membres*. Voici ce que cela donne en C++ :

```
struct obj {
    int x;           // champ x
    int f();         // méthode f()
    int g(int y);    // méthode g()
};
...
```

```
int main() {
    obj a;
    a.x=3;
    int i=a.f();
    int j=a.g(2);
    ...
}
```

Il y a juste un détail, mais d'importance : la définition de la structure `obj` ci-dessus ne fait que *déclarer les méthodes*. Elles ne sont *définies* nulle part dans le code précédent. Pour les définir, on fait comme pour les fonctions habituelles, sauf que

pour permettre à plusieurs objets d'avoir les mêmes noms de méthodes, on préfixe leur définition par le nom de l'objet suivi de `::`^a.

^a. Ce mécanisme existe aussi pour les fonctions usuelles. Ce sont les espaces de nom, que nous avons rencontrés et contournés immédiatement avec `using namespace std` pour ne pas avoir à écrire `std::cout` ...

Voici comment cela s'écrit :

```
struct obj1 {
    int x;           // champ x
    int f();         // méthode f() (déclaration)
    int g(int y);    // méthode g() (déclaration)
};

struct obj2 {
    double x;        // champ x
    double f();       // méthode f() (déclaration)
};

...
int obj1::f() {      // méthode f() de obj1 (définition)
    ...
    return ...
}
int obj1::g(int y) { // méthode g() de obj1 (définition)
    ...
    return ...
}
double obj2::f() {   // méthode f() de obj2 (définition)
    ...
    return ...
}

...
int main() {
    obj1 a;
    obj2 b;
    a.x=3; // le champ x de a est int
    b.x=3.5; // celui de b est double
    int i=a.f(); // méthode f() de a (donc obj1::f())
    int j=a.g(2); // méthode g() de a (donc obj1::g())
    double y=b.f(); // méthode f() de b (donc obj2::f())
}
```

...

9.3 Visibilité

Il y a une règle que nous n'avons pas vue sur les espaces de nom mais que nous pouvons facilement comprendre : quand on est "dans" un espace de nom, on peut utiliser toutes les variables et fonctions de cet espace sans préciser l'espace en question. Ainsi, ceux qui ont programmé `cout` et `endl` ont défini l'espace `std` puis se sont "placés à l'intérieur" de cet espace pour programmer sans avoir à mettre `std::` partout devant `cout`, `cin`, `endl` et les autres... C'est suivant cette même logique, que

dans ses méthodes, un objet accède directement à ses champs et à ses autres méthodes, c'est-à-dire sans rien mettre devant ^a !

a. Vous verrez peut-être parfois traîner le mot clé `this` qui est utile parfois en C++ et que les programmeurs venant de Java mettent partout, en se trompant d'ailleurs sur son type. Vous ne devriez pas en avoir besoin.

Par exemple, la fonction `obj1::f()` ci-dessus pourrait s'écrire :

```

1  int obj1::f() {           // méthode f() de obj1 (définition)
2      int i=g(3); // méthode g() de l'objet dont la méthode f() est
3                      // en train de s'exécuter
4      int j=x+i;  // champ x de l'objet dont la méthode f() est
5                      // en train de s'exécuter
6      return j;
7  }
8  ...
9  int main() {
10     obj1 a1,a2;
11     int i1=a1.f(); // Cet appel va utiliser a1.g() ligne 2
12                      // et a1.x ligne 4
13     int i2=a2.f(); // Cet appel va utiliser ligne 2 a2.g()
14                      // et a2.x ligne 4

```

Il est d'ailleurs normal qu'un objet accède simplement à ses champs depuis ses méthodes, car

si un objet n'utilise pas ses champs dans une méthode, c'est probablement qu'on est en train de ranger dans cet objet une fonction qui n'a rien à voir avec lui (cf abus mentionné plus haut)

9.4 Exemple des matrices

En programmation, un exemple de source vaut mieux qu'un long discours. Si jusqu'ici vous naviguiez dans le vague, les choses devraient maintenant s'éclaircir ! Voilà donc ce que devient notre exemple du chapitre 8 avec des objets :

```

#include <iostream>
#include <string>
using namespace std;

//=====
// fonctions sur les matrices
// pourraient etre dans un matrice.h et matrice.cpp

// ===== declarations (dans le .h)
struct Matrice {
    int m,n;
    double* t;
    void cree(int m1,int n1);
    void detruit();
    double get(int i,int j);
    void set(int i,int j,double x);
    void affiche(string s);
};
Matrice operator*(Matrice A,Matrice B);

// ===== definitions (dans le .cpp)
void Matrice::cree(int m1,int n1) {
    // Notez que les parametres ne s'appellent plus m et n
    // pour ne pas mélanger avec les champs!
    m=m1;
    n=n1;
    t=new double[m*n];
}

void Matrice::detruit() {
    delete[] t;
}

double Matrice::get(int i,int j) {
    return t[i+m*j];
}

void Matrice::set(int i,int j,double x) {
    t[i+m*j]=x;
}

void Matrice::affiche(string s) {
    cout << s << "␣=" << endl;
    for (int i=0;i<m;i++) {
        for (int j=0;j<n;j++)
            cout << get(i,j) << "␣";
        cout << endl;
    }
}

```

```

}

Matrice operator*(Matrice A, Matrice B) {
    if (A.n!=B.m) {
        cout << "Erreur!" << endl;
        exit(1);
    }
    Matrice C;
    C.cree(A.m,B.n);
    for (int i=0;i<A.m;i++)
        for (int j=0;j<B.n;j++) {
            // Cij=Ai0*B0j+Ai1*B1j+...
            C.set(i,j,0);
            for (int k=0;k<A.n;k++)
                C.set(i,j,
                    C.get(i,j)+A.get(i,k)*B.get(k,j));
        }
    return C;
}

// ===== main =====
int main()
{
    Matrice A;
    A.cree(2,3);
    for (int i=0;i<2;i++)
        for (int j=0;j<3;j++)
            A.set(i,j,i+j);
    A.affiche("A");
    Matrice B;
    B.cree(3,5);
    for (int i=0;i<3;i++)
        for (int j=0;j<5;j++)
            B.set(i,j,i+j);
    B.affiche("B");
    Matrice C=A*B;
    C.affiche("C");
    C.detrui();
    B.detrui();
    A.detrui();
    return 0;
}

```


9.5 Cas des opérateurs

Il est un peu dommage que l'opérateur `*` ne soit pas dans l'objet `Matrice`. Pour y remédier, on adopte la convention suivante :

Soit `A` un objet. S'il possède une méthode `operatorop(objB B)`, alors `AopB` appellera cette méthode pour tout `B` de type `objB`.

En clair, le programme :

```
struct objA {
    ...
};
struct objB {
    ...
};
int operator+(objA A, objB B) {
    ...
}
...
int main() {
    objA A;
    objB B;
    int i=A+B; // appelle operator+(A,B)
    ...
}
```

peut aussi s'écrire :

```
struct objA {
    ...
    int operator+(objB B);
};
struct objB {
    ...
};
int objA::operator+(objB B) {
    ...
}
...
int main() {
    objA A;
    objB B;
    int i=A+B; // appelle maintenant A.operator+(B)
    ...
}
```

ce qui pour nos matrices donne :

```
struct Matrice {
    ...
    Matrice operator*(Matrice B);
};
...
```

```

// A*B appelle A.operator*(B) donc tous
// les champs et fonctions utilisés directement
// concernent ce qui était préfixé précédemment par A.
Matrice Matrice::operator*(Matrice B) {
    // On est dans l'objet A du A*B appelé
    if (n!=B.m) { // Le n de A
        cout << "Erreur!" << endl;
        exit(1);
    }
    Matrice C;
    C.cree(m,B.n);
    for (int i=0;i<m;i++)
        for (int j=0;j<B.n;j++) {
            // Cij=Ai0*B0j+Ai1*B1j+...
            C.set(i,j,0);
            for (int k=0;k<n;k++)
                // get(i,j) sera celui de A
                C.set(i,j,
                    C.get(i,j)+get(i,k)*B.get(k,j));
        }
    return C;
}

```

Notez aussi que l'argument de l'opérateur n'a en fait pas besoin d'être un objet. Ainsi pour écrire le produit $B=A*2$, il suffira de créer la méthode :

```

Matrice Matrice::operator*(double lambda) {
    ...
}
...
B=A*2; // Appelle A.operator*(2)

```

Par contre, pour écrire $B=2*A$, on ne pourra pas créer :

```

Matrice double::operator*(Matrice A) // IMPOSSIBLE car double
                                     // n'est pas un objet!

```

car cela reviendrait à définir une méthode pour le type **double**, qui n'est pas un objet⁴. Il faudra simplement se contenter d'un opérateur standard, qui, d'ailleurs, sera bien inspiré d'appeler la méthode `Matrice::operator*(double lambda)` si elle est déjà programmée :

```

Matrice operator*(double lambda, Matrice A) {
    return A*lambda; // défini précédemment, rien à reprogrammer!
}
...
B=2*A; // appelle operator*(2,A) qui appelle à son tour
        // A.operator*(2)

```

Nous verrons au chapitre suivant d'autres opérateurs utiles dans le cas des objets...

4. et de toute façon n'appartient pas au programmeur !

9.6 Interface

Si on regarde bien le `main()` de notre exemple de matrice, on s'aperçoit qu'il n'utilise plus les champs des `Matrice` mais seulement leurs méthodes. En fait, seule la partie

```
struct Matrice {
    void cree(int m1, int n1);
    void detruit();
    double get(int i, int j);
    void set(int i, int j, double x);
    void affiche(string s);
    Matrice operator*(Matrice B);
};
```

intéresse l'utilisateur. Que les dimensions soient dans des champs `int m` et `int n` et que les éléments soient dans un champ `double* t` ne le concerne plus : c'est le problème de celui qui programme les matrices. Si ce dernier trouve un autre moyen⁵ de stocker un tableau bidimensionnel de `double`, libre à lui de le faire. En fait

Si l'utilisateur des `Matrice` se conforme aux déclarations des méthodes ci-dessus, leur concepteur peut les programmer comme il l'entend. Il peut même les reprogrammer ensuite d'une autre façon : les programmes de l'utilisateur marcheront toujours ! C'est le concept même d'une *interface* :

- Le concepteur et l'utilisateur des objets se mettent d'accord sur les méthodes qui doivent exister.
- Le concepteur les programme : il *implémente*^a l'interface.
- L'utilisateur les utilise de son côté.
- Le concepteur peut y retoucher sans gêner l'utilisateur.

En particulier le fichier d'en-tête de l'objet est le seul qui intéresse l'utilisateur. C'est lui qui précise l'interface, sans rentrer dans les détails d'implémentation. Bref, *reliées uniquement par l'interface, utilisation et implémentation deviennent indépendantes*^b.

^a. Il se trouve en général face au difficile problème du choix de l'implémentation : certaines façons de stocker les données peuvent rendre efficaces certaines méthodes au détriment de certaines autres, ou bien consommer plus ou moins de mémoire, etc. Bref, c'est lui qui doit gérer les problèmes d'algorithmique. C'est aussi en général ce qui fait que, pour une même interface, un utilisateur préférera telle ou telle implémentation : le concepteur devra aussi faire face à la concurrence !

^b. Ce qui est sûr, c'est que les deux y gagnent : le concepteur peut améliorer son implémentation sans gêner l'utilisateur, l'utilisateur peut changer pour une implémentation concurrente sans avoir à retoucher son programme.

5. Et il en existe ! Par exemple pour stocker efficacement des matrices creuses, c'est-à-dire celles dont la plupart des éléments sont nuls. Ou bien, en utilisant des objets implémentant déjà des tableaux de façon sûre et efficace, comme il en existe déjà en C++ standard ou dans des bibliothèques complémentaires disponibles sur le WEB. Etc, etc.

9.7 Protection

9.7.1 Principe

Tout cela est bien beau, mais les détails d'implémentation ne sont pas entièrement cachés : la définition de la structure dans le fichier d'en-tête fait apparaître les champs utilisés pour l'implémentation. Du coup, l'utilisateur peut être tenté de les utiliser ! Rien ne l'empêche en effet des faire des bêtises :

```
Matrice A;
A.cree(3,2);
A.m=4; // Aie! Les accès vont être faux!
```

ou tout simplement de préférer ne pas s'embêter en remplaçant

```
for (int i=0;i<3;i++)
    for (int j=0;j<2;j++)
        A.set(i,j,0);
```

par

```
for (int i=0;i<6;i++)
    A.t[i]=0; // Horreur! Et si on implémente autrement?
```

Dans ce cas, l'utilisation n'est plus indépendante de l'implémentation et on a perdu une grande partie de l'intérêt de la programmation objet... C'est ici qu'intervient la possibilité d'empêcher l'utilisateur d'accéder à certains champs ou même à certaines méthodes. Pour cela :

1. Remplacer **struct** par **class** : tous les champs et les méthodes deviennent *privés* : seules les méthodes de l'objet lui-même ou de tout autre objet du même type^a peuvent les utiliser.
2. Placer la déclaration **public** : dans la définition de l'objet pour débiter la zone^b à partir de laquelle seront déclarés les champs et méthodes *publics*, c'est-à-dire accessibles à tous.

^a. Bref, les méthodes de la classe en question !

^b. On pourrait à nouveau déclarer des passages privés avec `private`, puis `publics`, etc. Il existe aussi des passages *protégés*, notion qui dépasse ce cours...

Voici un exemple :

```
class obj {
    int x,y;
    void a_moi();
public:
    int z;
    void pour_tous();
    void une_autre(obj A);
};
void obj::a_moi() {
    x=..;    // OK
    ..=y;    // OK
```

```

    z = ..;    // OK
}
void obj::pour_tous() {
    x = ..;    // OK
    a_moi();  // OK
}
void obj::une_autre(obj A) {
    x = A.x;   // OK
    A.a_moi(); // OK
}
...
int main() {
    obj A,B;
    A.x = ..;    // NON!
    A.z = ..;    // OK
    A.a_moi();   // NON!
    A.pour_tous(); // OK
    A.une_autre(B); // OK
}

```

Dans le cas de nos matrices, que nous avons déjà bien programmées, il suffit de les définir comme suit :

```

class Matrice {
    int m,n;
    double* t;
public:
    void cree(int m1,int n1);
    void detruit();
    double get(int i,int j);
    void set(int i,int j,double x);
    void affiche(string s);
    Matrice operator*(Matrice B);
};

```

pour empêcher une utilisation dépendante de l'implémentation.

9.7.2 Structures vs Classes

Notez que, finalement, une structure est une classe où tout est public... Les anciens programmeurs C pensent souvent à tort que les structures du C++ sont les mêmes qu'en C, c'est-à-dire qu'elles ne sont pas des objets et qu'elles n'ont pas de méthode⁶.

9.7.3 Accesseurs

Les méthodes `get()` et `set()` qui permettent d'accéder en lecture (`get`) ou en écriture (`set`) à notre *classe*, sont appelées *accesseurs*. Maintenant que nos champs sont tous privés, l'utilisateur n'a plus la possibilité de retrouver les dimensions d'une matrice. On rajoutera donc deux accesseurs en lecture vers ces dimensions :

6. sans compter qu'ils les déclarent souvent comme en C avec d'inutiles `typedef`. Mais bon, ceci ne devrait pas vous concerner !

```

int Matrice::nbLin() {
    return m;
}
int Matrice::nbCol() {
    return n;
}
int main() {
    ...
    for (int i=0;i<A.nbLin();i++)
        for (int j=0;j<A.nbCol();j++)
            A.set(i,j,0);

```

mais pas en écriture, ce qui est cohérent avec le fait que changer m en cours de route rendrait fausses les fonctions utilisant `t[i+m*j]` !

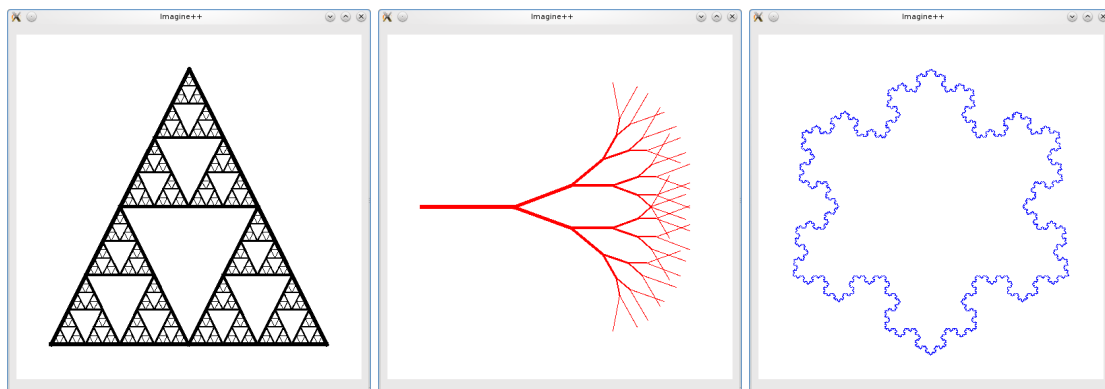


FIGURE 9.1 – Fractales

9.8 TP

Vous devriez maintenant pouvoir faire le TP en A.7 qui dessine quelques courbes fractales (figure 9.1) en illustrant le concept d’objet..

9.9 Fiche de référence

Fiche de référence (1/3)		
Boucles — do { ... } while(!ok); — int i=1; while(i<=100) { ... i=i+1; } — for(int i=1;i<=10;i++) ... — for(int i=1,j=10;j>i; i=i+2,j=j-3) ... — for (int i=...) for (int j=...) { //saute cas i==j if (i==j) continue; ... }	— Portée : int i; // i=j; interdit! int j=2; i=j; // OK! if (j>1) { int k=3; j=k; // OK! } //i=k; interdit! — Types : int i=3; double x=12.3; char c='A'; string s="hop"; bool t=true; float y=1.2f; unsigned int j=4; signed char d=-128; unsigned char d=25; complex<double> z(2,3);	— Retour : int signe(double x) { if (x<0) return -1; if (x>0) return 1; return 0; } void afficher(int x, int y) { if (x<0 y<0) return; if (x>=w y>=h) return; DrawPoint(x,y,RED); } — Appel : int f(int a) { ... } int g() { ... } ... int i=f(2),j=g(); — Références : void swap(int& a, int& b){ int tmp=a; a=b;b=tmp; } ... int x=3,y=2; swap(x,y); — Surchage : int hasard(int n); int hasard(int a, int b); double hasard(); — Opérateurs : vect operator+(vect A,vect B) { ... } ... vect C=A+B; — Pile des appels — Itératif/Récuratif
Clavier — Debug : F5  — Step over : F10  — Step inside : F11  — Indent : Ctrl+A, Ctrl+I — Step out : Maj+F11  — Gest. tâches : Ctrl+Maj+Ech	— Variables globales : int n; const int m=12; void f() { n=10; // OK int i=m; // OK ... — Conversion : int i=int(x),j; float x=float(i)/j; — Pile/Tas	
Variables — Définition : int i; int k,l,m; — Affectation : i=2; j=i; k=l=3; — Initialisation : int n=5,o=n; — Constantes : const int s=12;	Fonctions — Définition : int plus(int a,int b){ int c=a+b; return c; } void affiche(int a) { cout << a << endl; } — Déclaration : int plus(int a,int b);	

Fiche de référence (2/3)**Tableaux**

— Définition :

```
— double x[5], y[5];
  for(int i=0; i<5; i++)
    y[i]=2*x[i];
```

```
— const int n=5;
  int i[n], j[2*n];
```

— Initialisation :

```
int t[4]={1,2,3,4};
string s[2]={"ab", "c"};
```

— Affectation :

```
int s[3]={1,2,3}, t[3];
for (int i=0; i<3; i++)
  t[i]=s[i];
```

— En paramètre :

```
— void init(int t[4]) {
  for(int i=0; i<4; i++)
    t[i]=0;
}
```

```
— void init(int t[],
           int n) {
  for(int i=0; i<n; i++)
    t[i]=0;
}
```

— Taille variable :

```
int* t=new int[n];
...
delete[] t;
```

— En paramètre (suite) :

```
— void f(int* t, int n){
  t[i]=...
}
```

```
— void alloue(int*& t){
  t=new int[n];
}
```

— 2D :

```
int A[2][3];
A[i][j]=...;
int A[2][3]=
  {{1,2,3},{4,5,6}};
void f(int A[2][2]);
```

— 2D dans 1D :

```
int A[2*3];
A[i+2*j]=...;
```

— Taille variable (suite) :

```
int *t, *s, n;
```

Structures

```
— struct Point {
  double x, y;
  Color c;
};
...
Point a;
a.x=2.3; a.y=3.4;
a.c=Red;
Point b={1,2.5,Blue};
```

— Une structure est un objet entièrement public (→ cf objets!)

Objets

```
— struct obj {
  int x; // champ
  int f(); // méthode
  int g(int y);
};
int obj::f() {
  int i=g(3); // mon g
  int j=x+i; // mon x
  return j;
}
```

```
...
int main() {
  obj a;
  a.x=3;
  int i=a.f();
}
```

```
— class obj {
  int x, y;
  void a_moi();
public:
  int z;
  void pour_tous();
  void autre(obj A);
};
```

```
void obj::a_moi() {
  x=..; // OK
  ..=y; // OK
  z=..; // OK
}
```

```
void obj::pour_tous() {
  x=..; // OK
  a_moi(); // OK
}
```

```
void autre(obj A) {
  x=A.x; // OK
  A.a_moi(); // OK
}
```

```
...
int main() {
  obj A, B;
  A.x=..; //NON
  A.z=..; //OK
  A.a_moi(); //NON
  A.pour_tous(); //OK
  A.autre(B); //OK
}
```

```
— class obj {
  obj operator+(obj B);
};
...
int main() {
  obj A, B, C;
  C=A+B;
  // C=A.operator+(B)
}
```

Compilation séparée

— #include "vect.h", aussi dans vect.cpp

— Fonctions : déclarations dans le .h, définitions dans le .cpp

— Types : définitions dans le .h

— Ne déclarer dans le .h que les fonctions utiles.

— #pragma once au début du fichier.

— Ne pas trop découper...

Fiche de référence (3/3)**Tests**

```

— Comparaison :
  == != < > <= >=
— Négation : !
— Combinaisons : && ||
— if (i==0) j=1;
— if (i==0) j=1;
  else      j=2;

— if (i==0) {
    j=1;
    k=2;
  }
— bool t=(i==0);
  if (t)
    j=1;
— switch (i) {
  case 1:
    ...;
    ...;
    break;
  case 2:
  case 3:
    ...;
    break;
  default:
    ...;
}

```

Entrées/Sorties

```

— #include <iostream>
  using namespace std;
  ...
  cout <<"I="<<i<<endl;
  cin >> i >> j;

```

Erreurs fréquentes

```

— Pas de définition de fonction
  dans une fonction !
— int q=r=4; // NON!
— if (i=2) // NON!
  if i==2 // NON!
  if (i==2) then // NON!
— for(int i=0,i<100,i++)
    // NON!
— int f() {...}
  int i=f; // NON!
— double x=1/3; // NON!
  int i,j;
  x=i/j; // NON!
  x=double(i/j); //NON!

```

```

— double x[10],y[10];
  for(int i=1;i<=10;i++)
    y[i]=2*x[i]; //NON
— int n=5;
  int t[n]; // NON
— int f()[4] { // NON!
  int t[4];
  ...
  return t; // NON!
}
— int t[4]; t=f();
— int s[3]={1,2,3},t[3];
  t=s; // NON!
— int t[2];
  t={1,2}; // NON!
— struct Point {
  double x,y;
} // NON!
— Point a;
  a={1,2}; // NON!
— #include "tp.cpp"//NON
— int f(int t[][]); //NON
  int t[2,3]; // NON!
  t[i,j]=...; // NON!
— int* t;
  t[1]=...; // NON!
— int* t=new int[2];
  int* s=new int[2];
  s=t; // On perd s!
  delete[] t;
  delete[] s; //Déjà fait
— int *t,s; // s est int
    // non int*
  t=new int[n];
  s=new int[n]; // NON!

```

Divers

```

— i++;
  i--;
  i-=2;
  j+=3;
— j=i%n; // Modulo
— #include <cstdlib>
  ...
  i=rand()%n;
  x=rand()/
    double(RAND_MAX);
— #include <ctime>
  // Un seul appel
  srand((unsigned int)
    time(0));

```

```

— #include <cmath>
  double sqrt(double x);
  double cos(double x);
  double sin(double x);
  double acos(double x);

— #include <string>
  using namespace std;
  string s="hop";
  char c=s[0];
  int l=s.size();

— #include <ctime>
  s=double(clock())
    /CLOCKS_PER_SEC;

— #include <cmath>
  double pi=M_PI;

```

Imagine++

— Voir documentation...

Conseils

```

— Nettoyer en quittant.
— Erreurs et warnings : cliquer.
— Indenter.
— Ne pas laisser de warning.
— Utiliser le debugueur.
— Faire des fonctions.
— Tableaux : pas pour transcrire
  une formule mathématique !
— Faire des structures.
— Faire des fichiers séparés.
— Le .h doit suffire à l'utilisa-
  teur (qui ne doit pas regarder
  le .cpp)
— Ne pas abuser du récursif.
— Ne pas oublier delete.
— Compiler régulièrement.
— #include <cassert>
  ...
  assert(x!=0);
  y=1/x;
— Faire des objets.
— Ne pas toujours faire des ob-
  jets !
— Penser interface / implémen-
  tation / utilisation.

```


Chapitre 10

Constructeurs et Destructeurs

*Dans ce long chapitre, nous allons voir comment le C++ offre la possibilité d'intervenir sur ce qui se passe à la naissance et à la mort d'un objet. Ce mécanisme essentiel repose sur la notion de **constructeur** et de **destructeur**. Ces notions sont très utiles, même pour le débutant qui devra au moins connaître leur forme la plus simple. Nous poursuivrons par un aspect bien pratique du C++, tant pour l'efficacité des programmes que pour la découverte de bugs à la compilation : une autre utilisation du **const**. Enfin, pour les plus avancés, nous expliquerons aussi comment les problèmes de gestion du tas peuvent être ainsi automatisés.*

—

10.1 Le problème

Avec l'apparition des objets, nous avons transformé :

```
struct point {  
    int x,y;  
};  
...  
    point a;  
    a.x=2;a.y=3;  
    i=a.x;j=a.y;  
en:  
class point {  
    int x,y;  
public:  
    void get(int&X, int&Y);  
    void set(int X,int Y);  
};  
...  
    point a;  
    a.set(2,3);  
    a.get(i,j);
```

Conséquence :

```
    point a={2,3};
```

est maintenant impossible. On ne peut remplir les champs privés d'un objet, même à l'initialisation, car cela permettrait d'accéder en écriture à une partie privée¹ !

10.2 La solution

La solution est la notion de **constructeur** :

```
class point {
    int x,y;
public:
    point(int X,int Y);
};
point::point(int X,int Y) {
    x=X;
    y=Y;
}
...
point a(2,3);
```

Un constructeur est une méthode dont le nom est le nom de la classe elle-même. Il ne retourne rien mais son type de retour n'est pas `void` : il n'a pas de type de retour. Le constructeur est appelé à la création de l'objet et ses paramètres sont passés avec la syntaxe ci-dessus. Il est impossible d'appeler un constructeur sur un objet déjà créé.

Ici, c'est le constructeur `point::point(int X,int Y)` qui est défini. Notez bien qu'il est impossible d'appeler un constructeur sur un objet déjà construit :

```
point a(1,2); // OK! Valeurs initiales
// On ne fait pas comme ça pour changer les champs de a.
a.point(3,4); // ERREUR!
// Mais plutôt comme ça.
a.set(3,4);   // OK!
```

10.3 Cas général

10.3.1 Constructeur vide

Lorsqu'un objet est créé sans rien préciser, c'est le *constructeur vide* qui est appelé, c'est-à-dire celui sans paramètre. Ainsi, le programme :

```
class obj {
public:
    obj();
};
```

1. En réalité, il y a une autre raison, plus profonde et trop difficile à expliquer ici, qui fait qu'en général, dès qu'on programme des objets, cette façon d'initialiser devient impossible.

```
obj::obj() {
    cout << "hello" << endl;
}
...
obj a; // appelle le constructeur par défaut
affiche "hello".
```

Le constructeur vide `obj::obj()` est appelé à chaque fois qu'on construit un objet sans préciser de paramètre. Font exception les paramètres des fonctions et leur valeur de retour qui, eux, sont construits comme des recopies des objets passés en paramètre ou retournés^a.

a. Nous allons voir plus loin cette construction par copie.

Ainsi, le programme :

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

class obj {
public:
    obj();
};

obj::obj() {
    cout << "obj_" ;
}

void f(obj d) {
}

obj g() {
    obj e;
    cout << 6 << "_";
    return e;
}

int main()
{
    cout << 0 << "_";
    obj a;
    cout << 1 << "_";
    for (int i=2;i<=4;i++) {
        obj b;
        cout << i << "_";
    }
    f(a);
    cout << 5 << "_";
    a=g();
}
```

```

    return 0;
}

```

affiche :

```
0 obj 1 obj 2 obj 3 obj 4 5 obj 6
```

Bien repérer les deux objets non construits avec `obj :: obj()` : le paramètre `d` de `f()`, copie de `a`, et la valeur de retour de `g()`, copie de `e`.

10.3.2 Plusieurs constructeurs

Un objet peut avoir plusieurs constructeurs.

```

class point {
    int x,y;
public:
    point(int X,int Y);
    point(int V);
};
point::point(int X,int Y) {
    x=X;
    y=Y;
}
point::point(int V) {
    x=y=V;
}
...
point a(2,3); // construit avec point(X,Y)
point b(4);   // construit avec point(V)

```

Il faut cependant retenir la chose suivante :

Si on ne définit aucun constructeur, tout se passe comme s'il n'y avait qu'un constructeur vide ne faisant rien. Mais attention : dès qu'on définit soi-même un constructeur, le constructeur vide n'existe plus, sauf si on le redéfinit soi-même.

Par exemple, le programme :

```

class point {
    int x,y;
};
...
point a;
a.set(2,3);
point b;      // OK

```

devient, avec un constructeur, un programme qui ne se compile plus :

```

class point {
    int x,y;
public:

```

```

    point(int X,int Y);
};
point::point(int X,int Y) {
    x=X;
    y=Y;
}
...
    point a(2,3); // construit avec point(X,Y)
    point b;      // ERREUR! point() n'existe plus

```

et il faut alors rajouter un constructeur vide, même s'il ne fait rien :

```

class point {
    int x,y;
public:
    point();
    point(int X,int Y);
};
point::point() {
}
point::point(int X,int Y) {
    x=X;
    y=Y;
}
...
    point a(2,3); // construit avec point(X,Y)
    point b;      // OK! construit avec point()

```

10.3.3 Tableaux d'objets

Il n'est pas possible de spécifier globalement quel constructeur est appelé pour les éléments d'un tableau. C'est toujours le constructeur vide qui est appelé...

```

    point t[3]; // Construit 3 fois avec le constructeur vide
                // sur chacun des éléments du tableau
    point* s=new point[n]; // Idem, n fois
    point* u=new point(1,2)[n]; // ERREUR et HORREUR!
                                // Un essai de construire les u[i]
                                // avec point(1,2), qui n'existe pas

```

Il faudra donc écrire :

```

    point* u=new point[n];
    for (int i=0;i<n;i++)
        u[i].set(1,2);

```

ce qui n'est pas vraiment identique car on construit alors les points à vide puis on les affecte.

Par contre, il est possible d'écrire :

```

    point t[3]={point(1,2),point(2,3),point(3,4)};

```

ce qui n'est évidemment pas faisable pour un tableau de taille variable.

10.4 Objets temporaires

On peut, en appelant soi-même un constructeur^a, construire un objet sans qu'il soit rangé dans une variable. En fait il s'agit d'un objet temporaire sans nom de variable et qui meurt le plus tôt possible.

a. Attention, nous avons déjà dit qu'on ne pouvait pas appeler un constructeur d'un objet déjà construit. Ici, c'est autre chose : on appelle un constructeur sans préciser d'objet !

Ainsi, le programme :

```
void f(point p) {
    ...
}
point g() {
    point e(1,2); // pour le retourner
    return e;
}
...
point a(3,4); // uniquement pour pouvoir appeler f()
f(a);
point b;
b=g();
point c(5,6); // on pourrait avoir envie de faire
b=c;          // ça pour mettre b à (5,6)
```

peut largement s'alléger, en ne stockant pas dans des variables les points pour lesquels ce n'était pas utile :

```
1 void f(point p) {
2     ...
3 }
4 point g() {
5     return point(1,2); // retourne directement
6                       // l'objet temporaire point(1,2)
7 }
8 ...
9 f(point(3,4)); // Passe directement l'obj. temp. point(3,4)
10 point b;
11 b=g();
12 b=point(5,6); // affecte directement b à l'objet
13              // temporaire point(5,6)
```

Attention à la ligne 12 : elle est utile quand b existe déjà mais bien comprendre qu'on construit un point(5,6) temporaire qui est ensuite affecté à b. On ne remplit pas b directement avec (5,6) comme on le ferait avec un b.set(5,6).

Attention aussi à l'erreur suivante, très fréquente. Il ne faut pas écrire

```
point p=point(1,2); // NON!!!!!!!
```

mais plutôt

```
point p(1,2);      // OUI!
```


L'utilité de ces objets temporaires est visible sur un exemple réel :

```
point point::operator+(point b) {
    point c(x+b.x,y+b.y);
    return c;
}
```

...

```
point a(1,2),b(2,3);
c=a+f(b);
```

s'écrit plutôt :

```
point point::operator+(point b) {
    return point(x+b.x,y+b.y);
}
```

...

```
c=point(1,2)+f(point(2,3));
```

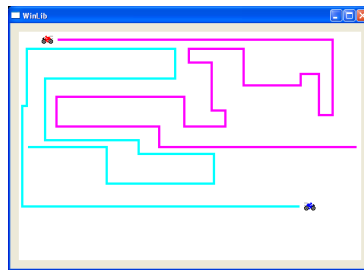


FIGURE 10.1 – Jeu de Tron.

10.5 TP

Nous pouvons faire une pause et aller faire le TP que nous proposons en A.8. Il s'agit de programmer le jeu de motos de Tron (figure 10.1).

10.6 Références Constantes

10.6.1 Principe

Lorsqu'on passe un objet en paramètre à une fonction, il est recopié. Cette copie est source d'inefficacité. Ainsi, dans le programme suivant :

```
const int N=1000;
class vecteur {
    double t[N];
    ...
};
class matrice {
    double t[N][N];
```

```

...
};
// résout AX=B
void solve(matrice A, vecteur B, vecteur& X) {
    ...
}

...
vecteur b, x;
matrice a;
...
solve(a, b, x); // résout ax=b

```

les variables A et B de la fonction solve() sont des copies des objets a et b de la fonction appelante. Notez bien que, passé par référence, le paramètre X n'est pas une copie car il s'agit juste d'un lien vers la variable x.

La recopie de a dans A n'est pas une très bonne chose. La variable a fait dans notre cas pas moins de 8 millions d'octets : les recopier dans A prend du temps ! Même pour des objets un peu moins volumineux, si une fonction est appelée souvent, cette recopie peut ralentir le programme. Lorsqu'une fonction est courte, il n'est pas rare non plus que ce temps de recopie soit supérieur à celui passé dans la fonction !

L'idée est alors, pour des objets volumineux, de les passer eux-aussi par référence, même si la fonction n'a pas à les modifier ! Il suffit donc de définir la fonction solve() ainsi :

```

void solve(matrice& A, vecteur& B, vecteur& X) {
    ...

```

pour accélérer le programme.

Cependant, cette solution n'est pas sans danger. Rien ne garantit en effet que solve ne modifie pas ses paramètres A et B. Il est donc possible, suivant la façon dont solve est programmée, qu'en sortie de solve(a,b,x), a et b eux-mêmes aient été modifiés, alors que précédemment c'étaient leurs copies A et B qui l'étaient. C'est évidemment gênant ! Le C++ offre heureusement la possibilité de *demande au compilateur de vérifier qu'une variable passée par référence n'est pas modifiée par la fonction*. Il suffit de rajouter **const** au bon endroit :

```

void solve(const matrice& A, const vecteur& B, vecteur& X) {
    ...

```

Si quelque part dans solve (ou dans les sous-fonctions appelées par solve !), la variable A ou la variable B est modifiée, alors il y aura erreur de compilation. La règle est donc :

Lorsqu'un paramètre obj o d'une fonction est de taille importante^a, c'est une bonne idée de le remplacer par const obj& o.

a. En réalité, le programme s'en trouvera accéléré pour la plupart des objets courants.

10.6.2 Méthodes constantes

Considérons le programme suivant :

```

void g(int& x) {

```

```

    cout << x << endl;
}
void f(const int& y) {
    double z=y; // OK ne modifie pas y
    g(y);       // OK?
}
...
int a=1;
f(a);

```

La fonction `f()` ne modifie pas son paramètre `y` et tout va bien. Imaginons une deuxième version de `g()` :

```

void g(int& x) {
    x++;
}

```

Alors `y` serait modifiée dans `f()` à cause de l'appel à `g()`. Le programme ne se compilerait évidemment pas... En réalité, la première version de `g()` serait refusée elle aussi car

pour savoir si une sous-fonction modifie ou non un des paramètres d'une fonction, le compilateur ne se base que sur la déclaration de cette sous-fonction et non sur sa définition^a.

^a. Le C++ n'essaie pas de deviner lui-même si une fonction modifie ses paramètres puisque la logique est que le programmeur indique lui-même avec `const` ce qu'il veut faire, et que le compilateur vérifie que le programme est bien cohérent.

Bref, notre premier programme ne se compilerait pas non plus car l'appel `g(y)` avec `const int& y` impose que `g()` soit déclarée `void g(const int& x)`. Le bon programme est donc :

```

void g(const int& x) {
    cout << x << endl;
}
void f(const int& y) {
    double z=y; // OK ne modifie pas y
    g(y);       // OK! Pas besoin d'aller regarder dans g()
}
...
int a=1;
f(a);

```

Avec les objets, nous avons besoin d'une nouvelle notion. En effet, considérons maintenant :

```

void f(const obj& o) {
    o.g(); // OK?
}

```

Il faut indiquer au compilateur si la méthode `g()` modifie ou non l'objet `o`. Cela se fait avec la syntaxe suivante :

```

class obj {
    ...
    void g() const;
    ...
};
void obj::g() const {
    ...
}
void f(const obj& o) {
    o.g(); // OK! Méthode constante
}

```

Cela n'est finalement pas compliqué :

On précise qu'une méthode est constante, c'est-à-dire qu'elle ne modifie pas son objet, en plaçant `const` derrière les parenthèses de sa déclaration et de sa définition.

On pourrait se demander si toutes ces complications sont bien nécessaires, notre point de départ étant juste le passage rapide de paramètres en utilisant les références. En réalité, placer des **const** dans les méthodes est une très bonne chose. Il ne faut pas le vivre comme une corvée de plus, mais comme une façon de préciser sa pensée : "suis-je ou non en train d'ajouter une méthode qui modifie l'objets?". Le compilateur va ensuite vérifier pour nous la cohérence de ce **const** avec tout le reste. Ceci a deux effets importants :

- Découverte de bugs à la compilation. (On pensait qu'un objet n'était pas modifié et il l'est.)
- Optimisation du programme².

—

La fin du chapitre peut être considérée comme difficile. Il est toutefois recommandé de la comprendre, même si la maîtrise et la mise en application de ce qui s'y trouve est laissée aux plus avancés.

—

10.7 Destructeur

Lorsqu'un objet meurt, une autre de ses méthodes est appelée : le *destructeur*.

Le destructeur :

- est appelé quand l'objet meurt.
- porte le nom de la classe précédé de `~`.
- comme les constructeurs, n'a pas de type.
- n'a pas de paramètres (Il n'y a donc qu'un seul destructeur par classe.)

2. Lorsque le compilateur sait qu'un objet reste constant pendant une partie du programme, il peut éviter d'aller le relire à chaque fois. Le `const` est donc une information précieuse pour la partie optimisation du compilateur.

Un exemple sera plus parlant. Rajoutons un destructeur au programme de la section 10.3 :

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

class obj {
public:
    obj();
    ~obj();
};

obj::obj() {
    cout << "obj_";
}

obj::~~obj() {
    cout << "~_";
}

void f(obj d) {
}

obj g() {
    obj e;
    cout << 6 << "_";
    return e;
}

int main()
{
    cout << 0 << "_";
    obj a;
    cout << 1 << "_";
    for (int i=2;i<=4;i++) {
        obj b;
        cout << i << "_";
    }
    f(a);
    cout << 5 << "_";
    a=g();
    return 0;
}
```

Il affiche maintenant :

```
0 obj 1 obj 2 ~ obj 3 ~ obj 4 ~ ~ 5 obj 6 ~ ~ ~
```

Repérez bien à quel moment les objets sont détruits. Constatez aussi qu'il y a plus d'appels au destructeur (7) qu'au constructeur (5) : nous n'avons pas encore parlé du constructeur pour les objets qui sont construits par copie...

10.8 Destructeurs et tableaux

Le destructeur est appelé pour tous les éléments du tableau. Ainsi,

```
1   if (a==b) {
2       obj t[10];
3       ...
4   }
```

appellera 10 fois le constructeur vide en ligne 2 et dix fois le destructeur en ligne 4. Dans le cas d'un tableau dynamique, c'est au moment du **delete[]** que les destructeurs sont appelés (avant la désallocation du tableau!).

```
if (a==b) {
    obj* t=new obj[n]; // n appels à obj()
    ...
    delete[] t;        // n appels à ~obj()
}
```

Attention : il est possible d'écrire **delete t** sans les []. C'est une erreur! Cette syntaxe est réservée à une autre utilisation du **new/delete**. L'utiliser ici a pour conséquence de bien désallouer le tas, mais d'oublier d'appeler les destructeurs sur les **t[i]**

10.9 Constructeur de copie

Voyons enfin ce fameux constructeur. Il n'a rien de mystérieux. Il s'agit d'un constructeur prenant en paramètre un autre objet, en général en référence constante.

Le constructeur de copie :

- Se déclare : `obj::obj(const obj& o);`
- Est utilisé évidemment par :
`obj a;`
`obj b(a); // b à partir de a`
- Mais aussi par :
`obj a;`
`obj b=a; // b à partir de a, synonyme de b(a)`
à ne pas confondre avec :
`obj a,b;`
`b=a; // ceci n'est pas un constructeur!`
- Et aussi pour construire les paramètres des fonctions et leur valeur de retour.

Notre programme exemple est enfin complet. En rajoutant :

```
obj::obj(const obj& o) {
    cout << "copy_";
}
```

il affiche :

```
0 obj 1 obj 2 ~ obj 3 ~ obj 4 ~ copy ~ 5 obj 6 copy ~ ~ ~
```

Nous avons enfin autant d'appels (7) aux constructeurs qu'au destructeur !

Il reste malgré tout à savoir une chose sur ce constructeur, dont nous comprendrons l'importance par la suite :

Lorsqu'il n'est pas programmé explicitement, le constructeur par copie recopie tous les champs de l'objet à copier dans l'objet construit.

Remarquez aussi que lorsqu'on définit soi-même un constructeur, le constructeur vide par défaut n'existe plus mais le constructeur de copie par défaut existe toujours !

10.10 Affectation

Il reste en fait une dernière chose qu'il est possible de reprogrammer pour un objet : l'affectation. Si l'affectation n'est pas reprogrammée, alors elle se fait naturellement par recopie des champs. Pour la reprogrammer, on a recours à l'opérateur `=`. Ainsi `a=b`, se lit `a.operator=(b)` si jamais celui-ci existe. Rajoutons donc :

```
void obj::operator=(const obj&o) {
    cout << "=_" ;
}
```

à notre programme, et il affiche :

```
0 obj 1 obj 2 ~ obj 3 ~ obj 4 ~ copy ~ 5 obj 6 copy ~ = ~ ~
```

On raffine en général un peu. L'instruction `a=b=c`; entre trois entiers marche pour deux raisons :

- Elle se lit `a=(b=c)`;
- L'instruction `b=c` affecte `c` à `b` et retourne la valeur de `c`

Pour pouvoir faire la même chose entre trois objets, on reprogrammera plutôt l'affectation ainsi :

```
obj obj::operator=(const obj&o) {
    cout << "=_" ;
    return o;
}
```

```
...
obj a,b,c;
a=b=c; // OK car a=(b=c)
```

ou même ainsi, ce qui dépasse nos connaissances actuelles, mais que nous préconisons car cela évite de recopier un objet au moment du **return** :

```
const obj& obj::operator=(const obj&o) {
    cout << "=_" ;
    return o;
}
...
obj a,b,c;
a=b=c; // OK car a=(b=c)
```

Un dernier conseil :

Attention à ne pas abuser ! Il n'est utile de reprogrammer le constructeur par copie et l'opérateur d'affectation que lorsqu'on veut qu'ils fassent autre chose que leur comportement par défaut^a !

a. Contrairement au constructeur vide, qui, lui, n'existe plus dès qu'on définit un autre constructeur, et qu'il est donc en général indispensable de reprogrammer, même pour reproduire son comportement par défaut

10.11 Objets avec allocation dynamique

Tout ce que nous venons de voir est un peu abstrait. Nous allons enfin découvrir à quoi ça sert. Considérons le programme suivant :

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

class vect {
    int n;
    double *t;
public:
    void alloue(int N);
    void libere();
};

void vect::alloue(int N) {
    n=N;
    t=new double[n];
}

void vect::libere() {
    delete[] t;
}

int main()
{
    vect v;
    v.alloue(10);
    ...
    v.libere();
    return 0;
}
```

10.11.1 Construction et destruction

Il apparaît évidemment que les constructeurs et les destructeurs sont là pour nous aider :


```

#include <iostream>
using namespace std;

class vect {
    int n;
    double *t;
public:
    vect(int N);
    ~vect();
};

vect::vect(int N) {
    n=N;
    t=new double[n];
}

vect::~~vect() {
    delete[] t;
}

int main()
{
    vect v(10);
    ...
    return 0;
}

```

Grâce aux constructeurs et au destructeur, nous pouvons enfin laisser les allocations et les désallocations se faire toutes seules !

10.11.2 Problèmes !

Le malheur est que cette façon de faire va nous entraîner assez loin pour des débutants. Nous allons devoir affronter deux types de problèmes.

Un problème simple

Puisqu'il n'y a qu'un seul destructeur pour plusieurs constructeurs, il va falloir faire attention à ce qui se passe dans le destructeur. Rajoutons par exemple un constructeur vide :

```

vect::vect() {
}

```

alors la destruction d'un objet créé à vide va vouloir désallouer un champ `t` absurde. Il faudra donc faire, par exemple :

```

vect::vect() {
    n=0;
}

```

```
vect::~vect() {
    if (n!=0)
        delete[] t;
}
```

Des problèmes compliqués

Le programme suivant ne marche pas :

```
int main()
{
    vect v(10),w(10);
    w=v;
    return 0;
}
```

Pourquoi ? Parce que l'affectation par défaut recopie les champs de *v* dans ceux de *w*. Du coup, *v* et *w* se retrouvent avec les mêmes champs *t* ! Non seulement ils iront utiliser les mêmes valeurs, d'où certainement des résultats faux, mais en plus *une même zone du tas va être désallouée deux fois, tandis qu'une autre ne le sera pas*³ !

Il faut alors reprogrammer l'affectation, ce qui n'est pas trivial. On décide en général de réallouer la mémoire et de recopier les éléments du tableau :

```
const vect& vect::operator=(const vect& v) {
    if (n!=0)
        delete[] t; // On se desalloue si necessaire
    n=v.n;
    if (n!=0) {
        t=new double[n]; // Reallocation et recopie
        for (int i=0;i<n;i++)
            t[i]=v.t[i];
    }
    return v;
}
```

Cette version ne marche d'ailleurs pas si on fait *v=v* car alors *v* est désalloué avant d'être recopié dans lui-même, ce qui provoque une lecture dans une zone qui vient d'être désallouée⁴.

10.11.3 Solution !

Des problèmes identiques se posent pour le constructeur de copie... Ceci dit, en factorisant le travail à faire dans quelques petites fonctions privées, la solution n'est pas si compliquée. Nous vous la soumettons en bloc. Elle peut même servir de schéma pour la plupart des objets similaires⁵ :

3. Ne pas désallouer provoque évidemment des *fuites de mémoire*. Désallouer deux fois provoque dans certains cas une erreur. C'est le cas en mode Debug sous Visual, ce qui aide à repérer les bugs !

4. Il suffit de rajouter un test (*&v==this*) pour repérer ce cas, ce qui nous dépasse un petit peu...

5. Ceci n'est que le premier pas vers une série de façon de gérer les objets. Doit-on recopier les tableaux ? Les partager en faisant en sorte que le dernier utilisateur soit chargé de désallouer ? Etc, etc.

```

1  #include <iostream>
2  using namespace std;
3
4  class vect {
5      // champs
6      int n;
7      double *t;
8      // fonctions privées
9      void alloc(int N);
10     void kill();
11     void copy(const vect& v);
12 public:
13     // constructeurs "obligatoires"
14     vect();
15     vect(const vect& v);
16     // destructeur
17     ~vect();
18     // affectation
19     const vect& operator=(const vect& v);
20     // constructeurs supplémentaires
21     vect(int N);
22 };
23
24 void vect::alloc(int N) {
25     n=N;
26     if (n!=0)
27         t=new double[n];
28 }
29
30 void vect::kill() {
31     if (n!=0)
32         delete[] t;
33 }
34
35 void vect::copy(const vect& v) {
36     alloc(v.n);
37     for (int i=0;i<n;i++) // OK même si n==0
38         t[i]=v.t[i];
39 }
40
41 vect::vect() {
42     alloc(0);
43 }
44
45 vect::vect(const vect& v) {
46     copy(v);
47 }
48

```

```

49 vect::~~vect() {
50     kill();
51 }
52
53 const vect& vect::operator=(const vect& v) {
54     if (this!=&v) {
55         kill();
56         copy(v);
57     }
58     return v;
59 }
60
61 vect::vect(int N) {
62     alloc(N);
63 }
64
65 // Pour tester constructeur de copie
66 vect f(vect a) {
67     return a;
68 }
69 // Pour tester le reste
70 int main()
71 {
72     vect a,b(10),c(12),d;
73     a=b;
74     a=a;
75     a=c;
76     a=d;
77     a=f(a);
78     b=f(b);
79     return 0;
80 }

```

10.12 Fiche de référence

Fiche de référence (1/4)		
Boucles	...	}
— do { ... } while(!ok);	— for(int i=1, j=10; j>i; i=i+2, j=j-3) ...	Clavier
— int i=1; while(i<=100) { ... i=i+1; }	— for (int i=...) { for (int j=...) { //saute cas i==j if (i==j) continue; }	— Debug : F5 
— for(int i=1; i<=10; i++)	...	— Step over : F10 
		— Step inside : F11 
		— Indent : Ctrl+A, Ctrl+I
		— Step out : Maj+F11 
		— Gest. tâches : Ctrl+Maj+Ech

Fiche de référence (2/4)		
Structures — struct Point { double x,y; Color c; }; ... Point a; a.x=2.3; a.y=3.4; a.c=Red; Point b={1,2.5,Blue}; — Une structure est un objet entièrement public (→ cf objets!)	const int m=12; void f() { n=10; // OK int i=m; // OK ... — Conversion : int i=int(x),j; float x=float(i)/j; — Pile/Tas	— Surcharge : int hasard(int n); int hasard(int a, int b); double hasard(); — Opérateurs : vect operator+(vect A,vect B) { ... } ... vect C=A+B; — Pile des appels — Itératif/Récuratif
	Fonctions — Définition : int plus(int a,int b){ int c=a+b; return c; } void affiche(int a) { cout << a << endl; } — Déclaration : int plus(int a,int b); — Retour : int signe(double x) { if (x<0) return -1; if (x>0) return 1; return 0; } void afficher(int x, int y) { if (x<0 y<0) return; if (x>=w y>=h) return; DrawPoint(x,y,RED); } — Appel : int f(int a) { ... } int g() { ... } ... int i=f(2),j=g(); — Références : void swap(int& a, int& b){ int tmp=a; a=b;b=tmp; } ... int x=3,y=2; swap(x,y);	— Références constantes (pour un passage rapide) : void f(const obj& x){ ... } void g(const obj& x){ f(x); // OK } Tableaux — Définition : double x[5],y[5]; for(int i=0;i<5;i++) y[i]=2*x[i]; — const int n=5; int i[n],j[2*n]; — Initialisation : int t[4]={1,2,3,4}; string s[2]={"ab","c"}; — Affectation : int s[3]={1,2,3},t[3]; for (int i=0;i<3;i++) t[i]=s[i]; — En paramètre : void init(int t[4]) { for(int i=0;i<4;i++) t[i]=0; } void init(int t[], int n) { for(int i=0;i<n;i++) t[i]=0; }

Fiche de référence (3/4)		
Taille variable : <pre>int* t=new int[n]; ... delete[] t;</pre> En paramètre (suite) : <pre>void f(int* t,int n){ t[i]=... } void alloue(int*& t){ t=new int[n]; }</pre> 2D: <pre>int A[2][3]; A[i][j]=...; int A[2][3]= {{1,2,3},{4,5,6}}; void f(int A[2][2]);</pre> 2D dans 1D : <pre>int A[2*3]; A[i+2*j]=...;</pre> Taille variable (suite) : <pre>int *t,*s,n;</pre> Objets <pre>struct obj { int x; // champ int f(); // méthode int g(int y); }; int obj::f() { int i=g(3); // mon g int j=x+i; // mon x return j; } ... int main() { obj a; a.x=3; int i=a.f(); } class obj { int x,y; void a_moi(); public: int z; void pour_tous(); void autre(obj A); }; void obj::a_moi() { x=..; // OK</pre>	<pre>..=y; // OK z=..; // OK } void obj::pour_tous() { x=..; // OK a_moi(); // OK } void autre(obj A) { x=A.x; // OK A.a_moi(); // OK } ... int main() { obj A,B; A.x=..; //NON A.z=..; //OK A.a_moi(); //NON A.pour_tous(); //OK A.autre(B); //OK } class obj { obj operator+(obj B); }; ... int main() { obj A,B,C; C=A+B; // C=A.operator+(B) } Méthodes constantes : void obj::f() const{ ... } void g(const obj& x){ x.f(); // OK } Constructeur : class point { int x,y; public: point(int X,int Y); }; point::point(int X, int Y){ x=X; y=Y; } ... point a(2,3); Constructeur vide : obj::obj() { ...</pre>	<pre>} ... obj a;</pre> — Objets temporaires : <pre>vec vec::operator+(vec b) { return vec(x+b.x, y+b.y); } ... c=vec(1,2) +f(vec(2,3));</pre> — Destructeur : <pre>obj::~~obj() { ... }</pre> — Constructeur de copie : <pre>obj::obj(const obj& o){ ... }</pre> Utilisé par : - obj b(a); - obj b=a; //mieux que obj b;b=a; - paramètres des fonctions - valeur de retour — Affectation : <pre>obj& obj::operator=(const obj&o){ ... return *this; }</pre> — Objets avec allocation dynamique automatique : cf section 10.11 Compilation séparée — #include "vect.h", aussi dans vect.cpp — Fonctions : déclarations dans le .h, définitions dans le .cpp — Types : définitions dans le .h — Ne déclarer dans le .h que les fonctions utiles. — #pragma once au début du fichier. — Ne pas trop découper...

Fiche de référence (4/4)**Tests**

```

— Comparaison :
  == != < > <= >=
— Négation : !
— Combinaisons : && ||
— if (i==0) j=1;
— if (i==0) j=1;
  else      j=2;

— if (i==0) {
    j=1;
    k=2;
  }
— bool t=(i==0);
  if (t)
    j=1;
— switch (i) {
  case 1:
    ...;
    ...;
    break;
  case 2:
  case 3:
    ...;
    break;
  default:
    ...;
}

```

Entrées/Sorties

```

— #include <iostream>
  using namespace std;
  ...
  cout <<"I="<<i<<endl;
  cin >> i >> j;

```

Divers

```

— i++;
  i--;
  i-=2;
  j+=3;
— j=i%n; // Modulo
— #include <cstdlib>
  ...
  i=rand()%n;
  x=rand()/
    double(RAND_MAX);
— #include <ctime>
  // Un seul appel
  srand((unsigned int)
    time(0));
— #include <cmath>
  double sqrt(double x);
  double cos(double x);
  double sin(double x);
  double acos(double x);

```

```

— #include <string>
  using namespace std;
  string s="hop";
  char c=s[0];
  int l=s.size();
— #include <ctime>
  s=double(clock())
    /CLOCKS_PER_SEC;
— #include <cmath>
  double pi=M_PI;

```

Imagine++

— Voir documentation...

Conseils

— Nettoyer en quittant.
 — Erreurs et warnings : cliquer.
 — Indenter.
 — Ne pas laisser de warning.
 — Utiliser le debugueur.
 — Faire des fonctions.
 — Tableaux : pas pour transcrire une formule mathématique !
 — Faire des structures.
 — Faire des fichiers séparés.
 — Le .h doit suffire à l'utilisateur (qui ne doit pas regarder le .cpp)
 — Ne pas abuser du récursif.
 — Ne pas oublier delete.
 — Compiler régulièrement.
 — #include <cassert>
 ...
 assert(x!=0);
 y=1/x;
 — Faire des objets.
 — Ne pas toujours faire des objets !
 — Penser interface / implémentation / utilisation.

Erreurs fréquentes

— Pas de définition de fonction dans une fonction !
 — int q=r=4; // NON!
 — if (i=2) // NON!
 if i==2 // NON!
 if (i==2) then // NON!
 — for(int i=0,i<100,i++)
 // NON!
 — int f() {...}
 int i=f; // NON!

```

— double x=1/3; // NON!
  int i,j;
  x=i/j; // NON!
  x=double(i/j); //NON!
— double x[10],y[10];
  for(int i=1;i<=10;i++)
    y[i]=2*x[i]; //NON
— int n=5;
  int t[n]; // NON
— int f()[4] { // NON!
  int t[4];
  ...
  return t; // NON!
}
  int t[4]; t=f();
— int s[3]={1,2,3},t[3];
  t=s; // NON!
— int t[2];
  t={1,2}; // NON!
— struct Point {
    double x,y;
  } // NON!
— Point a;
  a={1,2}; // NON!
— #include "tp.cpp"//NON
— int f(int t[][]); //NON
  int t[2,3]; // NON!
  t[i,j]=...; // NON!
— int* t;
  t[1]=...; // NON!
— int* t=new int[2];
  int* s=new int[2];
  s=t; // On perd s!
  delete[] t;
  delete[] s; //Déjà fait
— int *t,s; // s est int
  // non int*
  t=new int[n];
  s=new int[n]; // NON!
— class vec {
    int x,y;
  public:
    ...
  };
  ...
  vec a={2,3}; // NON
— vec v=vec(1,2); //NON
  vec v(1,2); // OUI
— obj* t=new obj[n];
  delete t; // manque []

```


Chapitre 11

Chaînes de caractères, fichiers

Nous commençons avec ce chapitre un tour de tout ce qui est utile et même souvent indispensable et que nous n'avons pas encore vu : chaînes de caractères, fichiers, plus quelques fonctionnalités utiles. Encore une fois, nous ne verrons pas tout de manière exhaustive, mais les fonctions les plus couramment utilisées.

—

Vous en connaissez suffisamment pour réaliser de nombreux programmes. Ce qui vous manque en général ici, c'est la pratique. Après avoir affronté les exercices tout faits, vous réalisez que, livrés à vous-même, il vous est difficile de vous en sortir. Alors lancez-vous ! Tentez de programmer l'un des projets proposés sur la page Web du cours. Vous constaterez rapidement qu'il vous manque aussi quelques fonctions ou types de variables usuels. Ce chapitre est là pour y remédier...

11.1 Chaînes de caractères

Les *chaînes de caractères* sont les variables stockant des suites de caractères, c'est-à-dire du texte. Nous les avons déjà rencontrées :

```
#include <string>
using namespace std;
...
string s="hop";
char c=s[0];
int l=s.size();
```

Complétons :

1. Les chaînes peuvent être comparées. C'est l'ordre alphabétique qui est évidemment utilisé :

```
if (s1==s2) ...
if (s1!=s2) ...
if (s1<s2) ...
if (s1>s2) ...
if (s1>=s2) ...
if (s1<=s2) ...
```

2. On peut chercher un caractère dans un chaîne :

```
size_t i=s.find('h'); // position de 'h' dans s?
size_t j=s.find('h',3); // position de 'h' dans s
                        // à partir de la position 3,
                        // en ignorant s[0] à s[2]
```

- Attention c'est le type `size_t`¹ qui est utilisé et non **int**. Considérez-le comme un entier mais pour lequel C++ choisit lui-même sur combien d'octets il faut le mémoriser...
- Si le caractère n'est pas trouvé, `find` retourne `string::npos` (une constante, dont la valeur importe peu).

3. On peut aussi chercher une sous-chaîne :

```
size_t i=s.find("hop"); // où est "hop" dans s?
size_t j=s.find("hop",3); // où est "hop" dans s à partir
                        // de la position 3?
```

4. Ajouter une chaîne à la fin d'une autre :

```
string a="comment";
string b="ça_va,_les_amis?";
string txt=a+"_"+b;
```

5. Extraire une sous chaîne :

```
string s1="un_deux_trois";
string s2=string(s1,3,4); // sous chaîne de longueur 4
                        // commençant en s1[3] (ici "deux")
```

6. Attention : la récupération d'une string au clavier coupe la chaîne si l'on appuie sur la touche "Entrée" mais aussi au premier espace rencontré. Ainsi, si l'on tape "bonjour les amis", le programme :

```
string s;
cin >> s; // Jusqu'à "Entrée" ou un espace
```

récupérera "bonjour" comme valeur de `s` (et éventuellement "les" puis "amis" si l'on programme d'autres `cin>>t...`). Pour récupérer la ligne complète, espaces compris, il faudra faire un

```
getline(cin,s); // Toute la ligne jusqu'à "Entrée"
```

On pourra éventuellement préciser un autre caractère que la fin de ligne :

```
getline(cin,s,':'); // Tout jusqu'à un ':' (non compris)
```

7. Convertir une string en une chaîne au format C : le C mémorise ses chaînes dans des tableaux de caractères terminés par un 0. Certaines fonctions prennent encore en paramètre un **char*** ou un **const char***². Il faudra alors leur passer `s.c_str()` pour convertir une variable `s` de type `string` (cf section 11.2.2).

1. En réalité, il faut utiliser le type `string::size_type`.

2. Nous n'avons pas encore vu le rôle de **const** avec les tableaux.

```
string s="hop_hop";
const char *t=s.c_str();
```

Vous trouverez d'autres fonctions dans l'aide en ligne de votre environnement de développement, ou tout simplement proposées par celui-ci quand vous utiliserez les string.

11.2 Fichiers

11.2.1 Principe

Pour lire et écrire dans un fichier, on procède exactement comme avec cout et cin. On crée simplement une variable de type ofstream pour écrire dans un fichier, ou de type ifstream pour lire...

1. Voici comment faire :

```
#include <fstream>
using namespace std;
...
ofstream f("hop.txt");
f << 1 << ' ' << 2.3 << ' ' << "salut" << endl;
f.close();

ifstream g("hop.txt");
int i;
double x;
string s;
g >> i >> x >> s;
g.close();
```

2. Il est bon de vérifier que l'ouverture s'est bien passée. Une erreur fréquente est de préciser un mauvais nom de fichier : le fichier n'est alors pas ouvert.

```
ifstream g("../data/hop.txt");
if (!g.is_open()) {
    cout << "help!" << endl;
    return 1;
}
```

(Attention, toujours utiliser le slash /, portable, et non le backslash \ même sous Windows). On peut aussi avoir besoin de savoir si on est arrivé au bout du fichier :

```
do {
    ...
} while (!(g.eof()));
```

3. Une fonction (en fait macro du préprocesseur) utile de Imagine++ (dans Common) est srcPath, qui remplace un chemin relatif en chemin absolu en faisant précéder le chemin par l'emplacement du dossier contenant le fichier source. Ainsi, le fichier sera trouvé quel que soit le dossier courant dans lequel est lancé le programme. Ainsi si notre dossier source est /home/pascal/Test/,

```
ifstream g(srcPath("hop.txt"));
```

cherchera le fichier `/home/pascal/Test/hop.txt`, même si notre programme se trouve dans le dossier `build`. L'équivalent pour le type `string` est `stringSrcPath`.

4. Un fichier peut s'ouvrir après construction :

```
ofstream f;
f.open("hop.txt");
...
```

5. Moins fréquent, mais très utile à connaître : on peut écrire dans un fichier directement la suite d'octets en mémoire qui correspond à une variable ou un tableau. Le fichier est alors moins volumineux, l'écriture et la lecture plus rapides (pas besoin de traduire un nombre en une suite de caractères ou l'inverse !)

```
double x[10];
double y;
ofstream f("hop.bin", ios::binary);
f.write((const char*)x, 10 * sizeof(double));
f.write((const char*)&y, sizeof(double));
f.close();
...
ifstream g("hop.bin", ios::binary);
g.read((char*)x, 10 * sizeof(double));
g.read((const char*)&y, sizeof(double));
g.close();
```

Attention à ne pas oublier le "mode d'ouverture" `ios::binary`

11.2.2 Chaînes et fichiers

1. Pour ouvrir un fichier, il faut préciser le nom avec une chaîne au format C. D'où la conversion...

```
void lire(string nom) {
    ifstream f(nom.c_str()); // Conversion obligatoire ...
    ...
}
```

2. Pour lire une chaîne avec des espaces, même chose qu'avec `cin` :

```
getline(g, s);
getline(g, s, ':');
```

3. Enfin, un peu technique mais très pratique : les `stringstream` qui sont des chaînes simulant des fichiers virtuels. On les utilise notamment pour convertir une chaîne en nombre ou l'inverse :

```
#include <sstream>
using namespace std;

string s="12";
```

```

stringstream f;
int i;
// Chaîne vers entier
f << s; // On écrit la chaîne
f >> i; // On relit un entier! (i vaut 12)
i++;
// Entier vers chaîne
f.clear(); // Ne pas oublier si on a déjà utilisé f
f << i; // On écrit un entier
f >> s; // On relit une chaîne (s vaut "13")

```

11.2.3 Objets et fichiers

Le grand intérêt des << et >>³ est la possibilité de les redéfinir pour des objets ! C'est technique, mais il suffit de recopier ! Voici comment :

```

struct point {
    int x,y;
};

ostream& operator<<(ostream& f, const point& p) {
    f << p.x << ' ' << p.y; // ou quoi que ce soit d'autre!
                             // (on a décidé ici d'écrire les deux
                             // coordonnées séparées par un espace...)
    return f;
}

istream& operator>>(istream& f, point& p) {
    f >> p.x >> p.y; // ou quoi que ce soit d'autre!
    return f;
}

...
point p;
cin >> p;
cout << p;
ofstream f("../hop.txt");
f << p;
...
ifstream g("../hop.txt");
g >> p;

```

3. Ils ont l'air un peu pénibles à utiliser pour le programmeur habitué au `printf` et `scanf` du C. On voit ici enfin leur puissance !

11.3 Valeurs par défaut

11.3.1 Principe

Souvent utile ! On peut donner des valeurs par défaut aux derniers paramètres d'une fonction, valeurs qu'ils prendront s'ils ne sont pas précisés à l'appel :

```
void f(int a,int b=0,int c=0) {
    // ...
}

void g() {
    f(12);      // Appelle f(12,0,0);
    f(10,2);    // Appelle f(10,2,0);
    f(1,2,3);   // Appelle f(1,2,3);
}
```

S'il y a déclaration puis définition, on ne précise les valeurs par défaut que dans la déclaration :

```
void f(int a,int b=0); // déclaration

void g() {
    f(12);    // Appelle f(12,0);
    f(10,2);  // Appelle f(10,2);
}

void f(int a,int b) { // ne pas re-préciser ici le b par défaut...
    // ...
}
```

11.3.2 Utilité

En général, on part d'une fonction :

```
int f(int a,int b) {
    ...
}
```

Puis, on veut lui rajouter un comportement spécial dans un certain cas :

```
int f(int a,int b,bool special) {
    ...
}
```

Plutôt que de transformer tous les anciens appels à f (.,.) en f (.,., **false**), il suffit de faire :

```
int f(int a,int b,bool special=false) {
    ...
}
```

pour laisser les anciens appels inchangés, et uniquement appeler f (.,., **true**) dans les futurs cas particuliers qui vont se présenter.

11.3.3 Erreurs fréquentes

Voici les erreurs fréquentes lorsqu'on veut utiliser des valeurs par défaut :

1. Vouloir en donner aux paramètres au milieu de la liste :

```
void f(int a,int b=3,int c) { // NON! Les derniers paramètres
                             // Pas ceux du milieu!
}
```

2. Engendrer des problèmes de surcharge :

```
void f(int a) {
    ...
}
void f(int a,int b=0) { // Problème de surcharge!
    ...                // On ne saura pas résoudre f(1)
}
```

11.4 Accesseurs

Voici, en cinq étapes, les points utiles à connaître pour faire des accesseurs pratiques et efficaces.

11.4.1 Référence comme type de retour

Voici une erreur souvent rencontrée, qui fait hurler ceux qui comprennent ce qui se passe :

```
int i; // Variable globale
int f() {
    return i;
}
...
f()=3; // Ne veut rien dire (pas plus que 2=3)
```

On ne range pas une valeur dans le retour d'une fonction, de même qu'on n'écrit pas $2=3$! En fait, si ! C'est possible. Mais uniquement si la fonction retourne une référence, donc un "lien" vers une variable :

```
int i; // Variable globale
int& f() {
    return i;
}
...
f()=3; // OK! Met 3 dans i!
```

Attention : apprendre ça à un débutant est très dangereux. En général, il se dépêche de commettre l'horreur suivante :

```
int& f() {
    int i; // Var. locale
```

```

    return i; // référence vers une variable qui va mourir!
              // C'EST GRAVE!
}
...
f()=3; // NON!!! Le i n'existe plus. Que va-t-il se passer?!

```

11.4.2 Utilisation

Même si un objet n'est pas une variable globale, un champ de cet objet ne meurt pas en sortant d'une de ses méthodes ! On peut, partant du programme :

```

class point {
    double x[N];
public:
    void set(int i, double v);
};
void point::set(int i, double v) {
    x[i]=v;
}
...
point p;
p.set(1,2.3);

```

le transformer en :

```

class point {
    double x[N];
public:
    double& element(int i);
};
double& point::element(int i) {
    return x[i];
}
...
point p;
p.element(1)=2.3;

```

11.4.3 operator()

Etape suivante : ceci devient encore plus utile quand on connaît **operator()** qui permet de redéfinir les parenthèses :

```

class point {
    double x[N];
public:
    double& operator()(int i);
};
double& point::operator()(int i) {
    return x[i];
}

```



```

}
...
point p;
p(1)=2.3; // Joli , non?

```

Notez que l'on peut passer plusieurs paramètres, ce qui est utile par exemple pour les matrices :

```

class mat {
    double x[M*N];
public:
    double& operator()(int i,int j);
};
double& mat::operator()(int i,int j) {
    return x[i+M*j];
}
...
mat A;
A(1,2)=2.3;

```

11.4.4 Surcharge et méthode constante

Nous sommes maintenant face à un **problème** : le programme précédent ne permet pas d'écrire :

```

void f(mat& A) {
    A(1,1)=2; // OK
}
void f(const mat& A) {
    double x=A(1,1); // NON! Le compilateur ne sait pas que
                     // cette ligne ne modifiera pas A!
}

```

car la méthode **operator()** n'est pas constante. Il y a heureusement une **solution** : programmer deux accesseurs, en profitant du fait qu'entre une méthode et une méthode constante, il y a surcharge possible, même si elles ont les mêmes paramètres ! Cela donne :

```

class mat {
    double x[M*N];
public:
    // Même nom, mêmes paramètres, mais l'une est 'const'!
    // Donc surcharge possible
    double& operator()(int i,int j);
    double operator()(int i,int j) const;
};
double mat::operator()(int i,int j) const {
    return x[i+M*j];
}
double& mat::operator()(int i,int j) {
    return x[i+M*j];
}

```

```

}
void f(mat& A) {
    A(1,1)=2; // OK, appelle le premier operator()
}
void f(const mat& A) {
    double x=A(1,1); // OK, appelle le deuxième
}

```

11.4.5 "inline"

Principe

Dernière étape : *appeler une fonction et récupérer sa valeur de retour est un mécanisme complexe, donc long*. Appeler $A(i,j)$ au lieu de faire $A.x[i+M*j]$ est une grande perte de temps : on passe plus de temps à appeler la fonction $A.operator()(i,j)$ et à récupérer sa valeur de retour, qu'à exécuter la fonction elle-même ! *Cela pourrait nous conduire à retourner aux structures en oubliant les classes !⁴*

Il existe un moyen de supprimer ce mécanisme d'appel en faisant en sorte que le corps de la fonction soit recopié dans le code appelant lui-même. Pour cela, il faut déclarer la fonction **inline**. Par exemple :

```

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

...
double y=sqr(z-3);

```

fait exactement comme si on avait écrit $y=(z-3)(z-3)$, sans qu'il n'y ait d'appel de fonction !

Précautions

Bien comprendre ce qui suit :

- Une fonction **inline** est recompilée à chaque ligne qui l'appelle, ce qui **ralentit la compilation et augmente la taille du programme** !
- **inline** est donc réservé aux fonctions courtes pour lesquelles l'appel est pénalisant par rapport au corps de la fonction !
- Si la fonction était déclarée dans un `.h` et définie dans un `.cpp`, il faut maintenant la **mettre entièrement dans le .h** car l'utilisateur de la fonction a besoin de la définition pour remplacer l'appel de la fonction par son corps !
- Pour pouvoir exécuter les fonctions pas à pas sous débogueur, les fonctions **inline** sont compilées comme des fonctions normales en mode Debug. Seul le mode Release profitera donc de l'accélération.

4. Les programmeurs C pourraient aussi être tentés de programmer des "macros" (ie. des raccourcis avec des `#define`, ce que nous n'avons pas appris à faire). Celles-ci sont moins puissantes que les **inline** car elles ne vérifient pas les types, ne permettent pas d'accéder aux champs privés, etc. Le programmeur C++ les utilisera avec parcimonie !

Cas des méthodes

Dans le cas d'une méthode, il faut bien penser à la mettre dans le fichier `.h` si la classe était définie en plusieurs fichiers. C'est le moment de révéler ce que nous gardions caché :

Il est possible de DÉFINIR UNE MÉTHODE ENTIÈREMENT DANS LA DÉFINITION DE LA CLASSE, au lieu de seulement l'y déclarer puis placer sa définition en dehors de celle de la classe. Cependant, ceci n'est pas obligatoire^a, ralentit la compilation et va à l'encontre de l'idée qu'il faut masquer le contenu des méthodes à l'utilisateur d'une classe. C'est donc RÉSERVÉ AUX PETITES FONCTIONS, en général de type `inline`.

^a. Contrairement à ce qu'il faut faire en Java ! Encore une source de mauvaises habitudes pour le programmeur Java qui se met à C++...

Voici ce que cela donne en pratique :

```
class mat {
    double x[M*N];
public:
    inline double& operator()(int i, int j) {
        return x[i+M*j];
    }
    inline double operator()(int i, int j) const {
        return x[i+M*j];
    }
};
```

11.5 Assertions

Rappelons l'existence de la fonction `assert()` vue en 7.6. Il ne faut pas hésiter à s'en servir car elle facilite la compréhension du code (répond à la question "quels sont les présupposés à ce point du programme ?") et facilite le diagnostic des erreurs. Sachant qu'elle ne coûte rien en mode Release (car non compilée), il ne faut pas se priver de l'utiliser. Voici par exemple comment rendre sûrs nos accesseurs :

```
#include <cassert>
```

```
class mat {
    double x[M*N];
public:
    inline double& operator()(int i, int j) {
        assert(i >= 0 && i < M && j >= 0 && j < N);
        return x[i+M*j];
    }
    inline double operator()(int i, int j) const {
        assert(i >= 0 && i < M && j >= 0 && j < N);
        return x[i+M*j];
    }
};
```

```
};
```

11.6 Types énumérés

C'est une bonne idée de passer par des constantes pour rendre un programme plus lisible :

```
const int nord=0, est=1, sud=2, ouest=3;
void avance(int direction);
```

mais il est maladroit de faire ainsi ! Il vaut mieux connaître l'existence des *types énumérés* :

```
enum Dir {nord, est, sud, ouest};
void avance(Dir direction);
```

Il s'agit bien de définir un nouveau type, qui, en réalité, masque des entiers. Une précision : on peut forcer certaines valeurs si besoin. Comme ceci :

```
enum Code {C10=200,
           C11=231,
           C12=240,
           C13, // Vaudra 241
           C14}; // " 242
```

—

Voilà. C'est tout pour aujourd'hui ! Nous continuerons au prochain chapitre. Il est donc temps de retrouver notre célèbre fiche de référence...

11.7 Fiche de référence

Fiche de référence (1/5)		
Boucles — do { ... } while(!ok); — int i=1; while(i<=100) { ... i=i+1; } — for(int i=1; i<=10; i++)	... — for(int i=1, j=10; j>i; i=i+2, j=j-3) ... — for (int i=...) { for (int j=...) { //saute cas i==j if (i==j) continue; } ...	} Clavier — Debug : F5  — Step over : F10  — Step inside : F11  — Indent : Ctrl+A, Ctrl+I — Step out : Maj+F11  — Gest. tâches : Ctrl+Maj+Ech

Fiche de référence (2/5)		
Structures — struct Point { double x,y; Color c; }; ... Point a; a.x=2.3; a.y=3.4; a.c=Red; Point b={1,2.5,Blue}; — Une structure est un objet entièrement public (→ cf objets!)	bool t=true; float y=1.2f; unsigned int j=4; signed char d=-128; unsigned char d=25; complex<double> z(2,3); — Variables globales : int n; const int m=12; void f() { n=10; // OK int i=m; // OK ... — Conversion : int i=int(x),j; float x=float(i)/j; — Pile/Tas — Type énuméré : enum Dir{N,E,S,W}; void avance(Dir d);	...; ...; break; case 2: case 3: ...; break; default: ...; }
Variables — Définition : int i; int k,l,m; — Affectation : i=2; j=i; k=l=3; — Initialisation : int n=5,o=n; — Constantes : const int s=12; — Portée : int i; // i=j; interdit! int j=2; i=j; // OK! if (j>1) { int k=3; j=k; // OK! } //i=k; interdit! — Types : int i=3; double x=12.3; char c='A'; string s="hop";	Tests — Comparaison : == != < > <= >= — Négation : ! — Combinaisons : && — if (i==0) j=1; — if (i==0) j=1; else j=2; — if (i==0) { j=1; k=2; } — bool t=(i==0); if (t) j=1; — switch (i) { case 1:	Conseils — Nettoyer en quittant. — Erreurs et warnings : cliquer. — Indenter. — Ne pas laisser de warning. — Utiliser le debugueur. — Faire des fonctions. — Tableaux : pas pour transcrire une formule mathématique ! — Faire des structures. — Faire des fichiers séparés. — Le .h doit suffire à l'utilisateur (qui ne doit pas regarder le .cpp) — Ne pas abuser du récursif. — Ne pas oublier delete. — Compiler régulièrement. — #include <cassert> ... assert(x!=0); y=1/x; — Faire des objets. — Ne pas toujours faire des objets ! — Penser interface / implémentation / utilisation.

Fiche de référence (3/5)		
Fonctions — Définition : <pre>int plus(int a,int b){ int c=a+b; return c; } void affiche(int a) { cout << a << endl; }</pre> — Déclaration : <pre>int plus(int a,int b);</pre> — Retour : <pre>int signe(double x) { if (x<0) return -1; if (x>0) return 1; return 0; } void afficher(int x, int y) { if (x<0 y<0) return; if (x>=w y>=h) return; DrawPoint(x,y,RED); } — Appel : <pre>int f(int a) { ... } int g() { ... } ... int i=f(2),j=g();</pre> — Références : <pre>void swap(int& a, int& b){ int tmp=a; a=b;b=tmp; } ... int x=3,y=2; swap(x,y);</pre> — Surcharge : <pre>int hasard(int n); int hasard(int a, int b); double hasard();</pre> — Opérateurs : <pre>vect operator+(vect A,vect B) { ...</pre> </pre>	<pre> } ... vect C=A+B; — Pile des appels — Itératif/Récurusif — Références constantes (pour un passage rapide) : void f(const obj& x){ ... } void g(const obj& x){ f(x); // OK } — Valeurs par défaut : void f(int a,int b=0); void g() { f(12); // f(12,0); f(10,2); // f(10,2); } void f(int a,int b) { // ... } — Inline (appel rapide) : inline double sqr(double x) { return x*x; } ... double y=sqr(z-3); — Référence en retour : int i; // Var. globale int& f() { return i; } ... f()=3; // i=3!</pre>	<pre> for (int i=0;i<3;i++) t[i]=s[i]; — En paramètre : — void init(int t[4]) { for(int i=0;i<4;i++) t[i]=0; } — void init(int t[], int n) { for(int i=0;i<n;i++) t[i]=0; } — Taille variable : int* t=new int[n]; ... delete[] t; — En paramètre (suite) : — void f(int* t,int n){ t[i]=... } — void alloue(int*& t){ t=new int[n]; } — 2D : int A[2][3]; A[i][j]=...; int A[2][3]= {{1,2,3},{4,5,6}}; void f(int A[2][2]); — 2D dans 1D : int A[2*3]; A[i+2*j]=...; — Taille variable (suite) : int *t,*s,n;</pre>
	Tableaux — Définition : <pre>— double x[5],y[5]; for(int i=0;i<5;i++) y[i]=2*x[i]; — const int n=5; int i[n],j[2*n];</pre> — Initialisation : <pre>int t[4]={1,2,3,4}; string s[2]={"ab","c"};</pre> — Affectation : <pre>int s[3]={1,2,3},t[3];</pre>	Compilation séparée — #include "vect.h", aussi dans vect.cpp — Fonctions : déclarations dans le .h, définitions dans le .cpp — Types : définitions dans le .h — Ne déclarer dans le .h que les fonctions utiles. — #pragma once au début du fichier. — Ne pas trop découper...

Fiche de référence (4/5)

Objets

```

— struct obj {
    int x;    // champ
    int f();  // méthode
    int g(int y);
};
int obj::f() {
    int i=g(3); // mon g
    int j=x+i;  // mon x
    return j;
}
...
int main() {
    obj a;
    a.x=3;
    int i=a.f();
}
— class obj {
    int x,y;
    void a_moi();
public:
    int z;
    void pour_tous();
    void autre(obj A);
};
void obj::a_moi() {
    x=..;    // OK
    ..=y;    // OK
    z=..;    // OK
}
void obj::pour_tous(){
    x=..;    // OK
    a_moi(); // OK
}
void autre(obj A) {
    x=A.x;    // OK
    A.a_moi(); // OK
}
...
int main() {
    obj A,B;
    A.x=..;    //NON
    A.z=..;    //OK
    A.a_moi(); //NON
    A.pour_tous(); //OK
    A.autre(B); //OK
}
— class obj {
    obj operator+(obj B);
};
...
int main() {
    obj A,B,C;
    C=A+B;
    // C=A.operator+(B)
}
— Méthodes constantes :
void obj::f() const{
    ...
}

```

```

void g(const obj& x){
    x.f(); // OK
}
— Constructeur :
class point {
    int x,y;
public:
    point(int X,int Y);
};
point::point(int X,
              int Y){
    x=X;
    y=Y;
}
...
point a(2,3);
— Constructeur vide :
obj::obj() {
    ...
}
...
obj a;
— Objets temporaires :
vec vec::operator+(
                    vec b) {
    return vec(x+b.x,
              y+b.y);
}
...
c=vec(1,2)
    +f(vec(2,3));
— Destructeur :
obj::~~obj() {
    ...
}
— Constructeur de copie :
obj::obj(const obj& o){
    ...
}
Utilisé par :
- obj b(a);
- obj b=a;
//mieux que obj b;b=a;
- paramètres des fonctions
- valeur de retour
— Affectation :
obj& obj::operator=(
                    const obj&o){
    ...
    return *this;
}
— Objets avec allocation dynamique automatique : cf section 10.11
— Accesseurs :
class mat {

```

```

    double *x;
public:
    double& operator()
        (int i,int j){
        assert(i>=0 ...);
        return x[i+M*j];
    }
    double operator()
        (int i,int j)const{
        assert(i>=0 ...);
        return x[i+M*j];
    }
    ...

```

Divers

```

— i++;
i--;
i-=2;
j+=3;
— j=i%n; // Modulo
— #include <cstdlib>
...
i=rand()%n;
x=rand()/
    double(RAND_MAX);
— #include <ctime>
// Un seul appel
srand((unsigned int)
    time(0));
— #include <cmath>
double sqrt(double x);
double cos(double x);
double sin(double x);
double acos(double x);
— #include <string>
using namespace std;
string s="hop";
char c=s[0];
int l=s.size();
if (s1==s1) ...
if (s1!=s2) ...
if (s1<s2) ...
size_t i=s.find('h'),
    j=s.find('h',3);
k=s.find("hop");
l=s.find("hop",3);
a="comment";
b="ça va?";
txt=a+" "+b;
s1="un deux trois";
s2=string(s1,3,4);
getline(cin,s);
getline(cin,s,':');
const char *t=s.c_str();

```

Fiche de référence (5/5)		
<pre>#include <ctime> s=double(clock()) /CLOCKS_PER_SEC; #include <cmath> double pi=M_PI;</pre>	<pre>f << s; f >> i; // Entier vers chaîne f.clear(); f << i; f >> s;</pre>	<pre>— Point a; a={1,2}; // NON! — #include "tp.cpp"//NON — int f(int t[][]);//NON int t[2,3]; // NON! t[i,j]=...; // NON!</pre>
Entrées/Sorties <pre>— #include <iostream> using namespace std; ... cout <<"I="<<i<<endl; cin >> i >> j; — #include <fstream> using namespace std; ofstream f("hop.txt"); f << 1 << ' ' << 2.3; f.close(); ifstream g("hop.txt"); if (!g.is_open()) { return 1; } int i; double x; g >> i >> x; g.close(); — do { ... } while (!g.eof()); — ofstream f; f.open("hop.txt"); — double x[10],y; ofstream f("hop.bin", ios::binary); f.write((const char*)x, 10*sizeof(double)); f.write((const char*)&y, sizeof(double)); f.close(); ifstream g("hop.bin", ios::binary); g.read((char*)x, 10*sizeof(double)); g.read((const char*)&y, sizeof(double)); g.close(); — string s; ifstream f(s.c_str()); — #include <sstream> using namespace std; stringstream f; // Chaîne vers entier</pre>	<pre>— ostream& operator<<(ostream& f, const point&p){ f<<p.x<<' '<<p.y; return f; } istream& operator>>(istream& f,point& p){ f>>p.x>>p.y; return f; }</pre> Erreurs fréquentes <pre>— Pas de définition de fonction dans une fonction ! — int q=r=4; // NON! — if (i=2) // NON! if i==2 // NON! if (i==2) then // NON! — for(int i=0,i<100,i++) // NON! — int f() {...} int i=f; // NON! — double x=1/3; // NON! int i,j; x=i/j; // NON! x=double(i/j); //NON! — double x[10],y[10]; for(int i=1;i<=10;i++) y[i]=2*x[i];//NON — int n=5; int t[n]; // NON — int f()[4] { // NON! int t[4]; ... return t; // NON! } int t[4]; t=f(); — int s[3]={1,2,3},t[3]; t=s; // NON! — int t[2]; t={1,2}; // NON! — struct Point { double x,y; } // NON!</pre>	<pre>— int* t; t[1]=...; // NON! — int* t=new int[2]; int* s=new int[2]; s=t; // On perd s! delete[] t; delete[] s;//Déjà fait — int *t,s;// s est int // non int* t=new int[n]; s=new int[n];// NON! — class vec { int x,y; public: ... }; ... vec a={2,3}; // NON — vec v=vec(1,2);//NON vec v(1,2); // OUI — obj* t=new obj[n]; delete t; // manque [] — //NON! void f(int a=2,int b); — void f(int a,int b=0); void f(int a);// NON! — Ne pas tout mettre inline! — int f() { ... } f()=3; // HORREUR! — int& f() { int i; return i; } f()=3; // NON!</pre>
		Imagine++ <pre>— Voir documentation...</pre>

Chapitre 12

Fonctions et classes paramétrées (templates)

Nous continuons dans ce chapitre un inventaire de diverses choses utiles. Parmi elles, les structures de données de la *STL* (Standard Template Library) nécessiteront la compréhension des `template`. Nous aborderons donc cet aspect intéressant du C++.

—

12.1 `template`

12.1.1 Principe

Considérons la fonction classique pour échanger deux variables :

```
void echange(int& a, int& b) {  
    int tmp;  
    tmp=a;  
    a=b;  
    b=tmp;  
}  
  
...  
int i, j;  
...  
echange(i, j);
```

Si nous devons maintenant échanger deux variables de type **double**, il faudrait ré-écrire une autre fonction `echange()`, identique aux définitions de type `près`. Heureusement, le C++ offre la possibilité de définir une fonction avec un *type générique*, un peu comme un type variable, que le compilateur devra "*instancier*" au moment de l'appel de la fonction en un type précis. Cette "*programmation générique*" se fait en définissant un "**template**" :

```
// Echange deux variables de n'importe quel type T  
template <typename T>  
void echange(T& a, T& b) {  
    T tmp;
```

```

    tmp=a;
    a=b;
    b=tmp;
}

...
int a=2,b=3;
double x=2.1,y=2.3;
echange(a,b); // "instance" T en int
echange(x,y); // "instance" T en double
...

```

Autre exemple :

```

// Maximum de deux variables (a condition que operator >() existe
// pour le type T)
template <typename T>
T maxi(T a,T b) {
    return (a>b)?a:b;
}

```

La déclaration **typename** T précise le type générique. On peut en préciser plusieurs :

```

// Cherche e1 dans le tableau tab1 et met
// dans e2 l'element de tab2 de meme indice
// Renvoie false si non trouvé
template <typename T1,typename T2>
bool cherche(T1 e1,T2& e2,const T1* tab1,const T2* tab2, int n) {
    for (int i=0;i<n;i++)
        if (tab1[i]==e1) {
            e2=tab2[i];
            return true;
        }
    return false;
}

...
string noms[3]={"jean","pierre","paul"};
int ages[3]={21,25,15};
...
string nm="pierre";
int ag;
if (cherche(nm,ag,noms,ages,3))
    cout << nm << " _a_ " << ag << " _ans " << endl;
...

```

12.1.2 **template** et fichiers

Il faut bien comprendre que

Le compilateur ne fabrique pas une fonction "magique" qui arrive à travailler sur plusieurs types! Il crée en réalité autant de fonctions qu'il y a d'utilisations de la fonction générique avec des types différents (ie. d'instanciations)

Pour ces raisons :

1. Faire des fonctions **template** ralentit la compilation et augmente la taille des programmes.
2. On ne peut plus mettre la déclaration dans un fichier d'en-tête et la définition dans un fichier `.cpp`, car tous les fichiers utilisateurs doivent connaître aussi la définition. Du coup, la règle est de **tout mettre dans le fichier d'en-tête**¹.

12.1.3 Classes

Il est fréquent qu'une définition de classe soit encore plus utile si elle est générique. C'est possible. Mais attention ! Dans le cas des fonctions, c'est le compilateur qui détermine tout seul quels types sont utilisés. Dans le cas des classes, c'est l'utilisateur qui doit préciser en permanence avec la syntaxe `obj<type>` le type utilisé :

// Paire de deux variables de type T

```
template <typename T>
class paire {
    T x[2];
public:
    // constructeurs
    paire();
    paire(T A,T B);
    // accesseurs
    T operator()(int i) const;
    T& operator()(int i);
};
```

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

```
template <typename T>
paire<T>::paire(T A,T B) {
    x[0]=A; x[1]=B;
}
```

```
template <typename T>
T paire<T>::operator()(int i) const {
    assert(i==0 || i==1);
}
```

1. Ceci est gênant et va à l'encontre du principe consistant à mettre les déclarations dans le `.h` et à masquer les définitions dans le `.cpp`. Cette remarque a déjà été formulée pour les fonctions `inline`. Le langage prévoit une solution avec le mot clé `export`, mais les compilateurs actuels n'implémentent pas encore cette fonctionnalité !

```

    return x[i];
}

template <typename T>
T& paire<T>::operator()(int i) {
    assert(i==0 || i==1);
    return x[i];
}

...
paire<int> p(1,2),r;
int i=p(1);
paire<double> q;
q(1)=2.2;
...

```

Dans le cas de la classe très simple ci-dessus, on aura recours aux fonctions **inline** vues en 11.4.5 :

```

// Paire de deux variables de type T
// Fonctions courtes et rapides en inline
template <typename T>
class paire {
    T x[2];
public:
    // constructeurs
    inline paire() {}
    inline paire(T A,T B) { x[0]=A; x[1]=B; }
    // accesseurs
    inline T operator()(int i) const {
        assert(i==0 || i==1);
        return x[i];
    }
    inline T& operator()(int i) {
        assert(i==0 || i==1);
        return x[i];
    }
};

```

Lorsque plusieurs types sont génériques, on les sépare par une virgule :

```

// Paire de deux variables de types différents
template <typename S,typename T>
class paire {
public:
    // Tout en public pour simplifier
    S x;
    T y;
    // constructeurs
    inline paire() {}
    inline paire(S X,T Y) { x=X; y=Y; }
};

```

```

...
paire<int ,double> P(1 ,2.3);
paire<string ,int> Q;
Q.x="pierre";
Q.y=25;
...

```

Enfin, on peut aussi rendre générique le choix d'un entier :

```

// n-uplet de variables de type T
// Attention: chaque nuplet<T,N> sera un type différent
template <typename T, int N>
class nuplet {
    T x[N];
public:
    // accesseurs
    inline T operator()(int i) const {
        assert(i>=0 && i<N);
        return x[i];
    }
    inline T& operator()(int i) {
        assert(i>=0 && i<N);
        return x[i];
    }
};

...
nuplet<int ,4> A;
A(1)=3;
nuplet<string ,2> B;
B(1)="pierre";
...

```

Les fonctions doivent évidemment s'adapter :

```

template <typename T, int N>
T somme(nuplet<T,N> u) {
    T s=u(0);
    for (int i=1;i<N;i++)
        s+=u(i);
    return s;
}

...
nuplet<double ,3> C;
...
cout << somme(C) << endl;
...

```

Au regard de tout ça, on pourrait être tenté de mettre des **template** partout. Et bien, non !

Les templates sont délicats à programmer, longs à compiler, etc. Il ne faut pas en abuser ! Il vaut mieux plutôt commencer des classes ou des fonctions sans template. On ne les rajoute que lorsqu'apparaît le besoin de réutiliser l'existant avec des types différents. Et répétons-le encore une fois : le compilateur crée une nouvelle classe ou une nouvelle fonction à chaque nouvelle valeur (instanciation) des types ou des entiers génériques ^a.

^a. Les nuplets ci-dessus, n'ont donc rien-à-voir avec des tableaux de taille variables. Tout se passe comme si on avait programmé des tableaux de taille constante pour plusieurs valeurs de la taille.

12.1.4 STL

Les **template** sont délicats à programmer, mais pas à utiliser. Le C++ offre un certain nombre de fonctions et de classes utilisant les **template**. Cet ensemble est communément désigné sous le nom de STL (Standard Template Library). Vous en trouverez la documentation complète sous Visual ou à défaut sur Internet. Nous exposons ci-dessous quelques exemples qui devraient pouvoir servir de point de départ et faciliter la compréhension de la documentation.

Des fonctions simples comme min et max sont définies de façon générique :

```
int i=max(1,3);
double x=min(1.2,3.4);
```

Attention : une erreur classique consiste à appeler max(1,2,3) : le compilateur l'interprète comme le max d'un **int** et d'un **double** ce qui provoque une erreur ! Il faut taper max(1,2,3).

Les complexes sont eux-aussi génériques, laissant variable le choix du type de leurs parties réelle et imaginaire :

```
#include <complex>
using namespace std;
...
complex<double> z1(1.1,3.4), z2(1,0), z3;
z3=z1+z2;
cout << z3 << endl;
double a=z3.real(), b=z3.imag();
double m=abs(z3); // module
double th=arg(z3); // argument
```

Les couples sont aussi offerts par la STL :

```
pair<int, string> P(2, "hop");
P.first=3;
P.second="hop";
```

Enfin, un certain nombre de structures de données sont fournies et s'utilisent suivant un même schéma. Voyons l'exemple des listes :

```
#include <list>
using namespace std;
...
list<int> l; // l=[]
```

```

l.push_front(2); // l=[2]
l.push_front(3); // l=[3,2]
l.push_back(4); // l=[3,2,4]
l.push_front(5); // l=[5,3,2,4]
l.push_front(2); // l=[2,5,3,2,4]

```

Pour désigner un emplacement dans une liste, on utilise un *itérateur*. Pour désigner un emplacement en lecture seulement, on utilise un *itérateur constant*. Le '*' sert ensuite à accéder à l'élément situé à l'emplacement désigné par l'itérateur. Seule difficulté : le type de ces itérateurs est un peu compliqué à taper² :

```

list<int>::const_iterator it;
it=l.begin(); // Pointe vers le début de la liste
cout << *it << endl; // affiche 2
it=l.find(3); // Pointe vers l'endroit ou se trouve
               // le premier 3
if (it!=l.end())
    cout << "3_est_dans_la_liste" << endl;
list<int>::iterator it2;
it2=l.find(3); // Pointe vers l'endroit ou se trouve
               // le premier 3
*it=6; // maintenant l=[2,5,6,2,4]

```

Les itérateurs servent également à parcourir les listes (d'où leur nom!) :

```

// Parcourt et affiche une liste
template <typename T>
void affiche(list<T> l) {
    cout << "[";
    for (list<T>::const_iterator it=l.begin(); it!=l.end(); it++)
        cout << *it << ' ';
    cout << "]" << endl;
}

// Remplace a par b dans une liste
template <typename T>
void remplace(list<T>& l, T a, T b) {
    for (list<T>::iterator it=l.begin(); it!=l.end(); it++)
        if (*it==a)
            *it=b;
}

...
affiche(l);
remplace(l,2,1); // maintenant l=[1,5,3,1,4]
...

```

Enfin, on peut appeler des algorithmes comme le tri de la liste :

```

l.sort();
affiche(l);

```

2. Nous n'avons pas vu comment définir de nouveaux types cachés dans des classes ! C'est ce qui est fait ici...

Sur le même principe que les listes, vous trouverez dans la STL :

- Les piles ou stack (Last In First Out).
- Les files ou queue (First In First Out).
- Les ensembles ou set (pas deux fois le même élément).
- Les vecteurs ou vector (tableaux de taille variable).
- Les tas ou heap (arbres binaires de recherche).
- Les tables ou map (table de correspondance clé/valeur).
- Et quelques autres encore...

Le reste de ce chapitre regroupe quelques notions utiles mais non fondamentales. Elles vous serviront probablement plus pour comprendre des programmes déjà écrits que dans vos propres programmes.

12.2 Opérateurs binaires

Parmi les erreurs classiques, il y a évidemment celle qui consiste à remplacer

```
if (i==0)
...
```

par

```
if (i=0) // NON!!!
...
```

qui range 0 dans i puis considère 0 comme un booléen, c'est à dire **false**. Une autre erreur fréquente consiste à écrire

```
if (i==0 & j==2) // NON!!!
...
```

au lieu de

```
if (i==0 && j==2)
...
```

Cette erreur n'est possible que parce que & existe. Il s'agit de opérateur binaire "ET" sur des entiers. Il est défini ainsi : effectuer a&b revient à considérer l'écriture de a et de b en binaire puis à effectuer un "ET" bit par bit (avec la table 1&1 donne 1 ; 1&0, 0&1 et 0&0 donnent 0). Par exemple : 13&10 vaut 8 car en binaire 1101&1010 vaut 1000.

Il existe ainsi toute une panoplie d'opérateurs binaires :

symbole	utilisation	nom	résultat	exemple
&	a&b	et	1&1=1, 0 sinon	13&10=8
	a b	ou	0 0=0, 1 sinon	13 10=15
^	a^b	ou exclusif	1^0=0^1=1, 0 sinon	13^10=7
>>	a>>n	décalage à droite	décale les bits de a n fois vers la droite et comble à gauche avec des 0 (les n premiers de droite sont perdus)	13>>2=3
<<	a<<n	décalage à gauche	décale les bits de a n fois vers la gauche et comble à droite avec des 0	5<<2=20
~	~a	complément	~1=0, ~0=1	~13=-14

Remarques :

- Ces instructions sont particulièrement rapides car simples pour le processeur.
- Le fait que a^b existe est aussi source de bugs (**il ne s'agit pas de la fonction puissance !**)
- Le résultat de $\sim$ dépend en fait du type : si par exemple i est un entier non signé sur 8 bits valant 13, alors $\sim i$ vaut 242, car ~ 00001101 vaut 11110010 .

En pratique, tout cela ne sert pas à faire joli ou savant, mais à manipuler les nombres bit par bit. Ainsi, il arrive souvent qu'on utilise un **int** pour mémoriser un certain nombre de propriétés en utilisant le moins possible de mémoire avec la convention que la propriété n est vraie ssi le n^{eme} bit de l'entier est à 1. Un seul entier de 32 bits pourra par ainsi mémoriser 32 propriétés là où il aurait fallu utiliser 32 variables de type **bool**. Voici comment on utilise les opérateurs ci-dessus pour manipuler les bits en question :

$i =(1<<n)$	passé à 1 le bit n de i
$i \&=\sim(1<<n)$	passé à 0 le bit n de i
$i \wedge=(1<<n)$	inverse le bit n de i
if ($i \&(1<<n)$)	vrai ssi le bit n de i est à 1

Il existe aussi d'autres utilisations fréquentes des opérateurs binaires, non pour des raisons de gain de place, mais pour des raisons de rapidité :

$(1<<n)$	vaut 2^n (sinon il faudrait faire int (pow(2.,n)) !)
$(i>>1)$	calcule $i/2$ rapidement
$(i>>n)$	calcule $i/2^n$ rapidement
$(i \& 255)$	calcule $i \% 256$ rapidement (idem pour toute puissance de 2)

12.3 Valeur conditionnelle

Il arrive qu'on ait à choisir entre deux valeurs en fonction du résultat d'un test. Une construction utile est alors :

`( test ) ? val1 : val2`

qui vaut `val1` si `test` est vrai et `val2` sinon. Ainsi

```
if (x>y)
    maxi=x;
else
    maxi=y;
```

pourra être remplacé par :

```
maxi=(x>y)? x : y;
```

Il ne faut pas abuser de cette construction sous peine de programme illisible !

12.4 Boucles et **break**

Nous avons déjà rencontré à la section 8.4 l'instruction `continue` qui *saute la fin d'une boucle et passe au tour d'après*. Très utile aussi, la commande `break` *sort de la boucle en ignorant tout ce qu'il restait à y faire*. Ainsi le programme :

```
bool arreter=false ;
for (int i=0;i<N && !arreter ;i++) {
    A;
    if (B)
        arreter=true ;
    else {
        C;
        if (D)
            arreter=true ;
        else {
            E;
        }
    }
}
```

devient de façon plus lisible et plus naturelle :

```
for (int i=0;i<N;i++) {
    A;
    if (B)
        break ;
    C;
    if (D)
        break ;
    E;
}
```

Questions récurrentes des débutants :

1. `break` ne sort pas d'un `if` !

```
if (...) {
    ...;
    if (...)
        break ; // NON!!! Ne sort pas du if! (mais éventuellement
                // d'un for qui serait autour...)
    ...
}
```

2. `break` ne sort que de la boucle courante, pas des boucles autour :

```
for (int i=0;i<N;i++) {
    ...
    for (int j=0;j<M;j++) {
        ...
        if (...)
            break ; // termine la boucle en j et passe donc
```

```

// en ligne 10 (pas en ligne 12)
    ...
    }
    ...
}
...

```

3. `break` et `continue` marchent évidemment avec `while` et `do ...while` de la même façon qu'avec `for`.

12.5 Variables statiques

Il arrive souvent qu'on utilise une variable globale pour mémoriser de façon permanente une valeur qui n'intéresse qu'une seule fonction :

```

// Fonction random qui appelle srand() toute seule
// au premier appel...
bool first=true;
double random() {
    if (first) {
        first=false;
        srand((unsigned int)time(0));
    }
    return double(rand())/RAND_MAX;
}

```

Le danger est alors que tout le reste du programme voie cette variable globale et l'utilise ou la confonde avec une autre variable globale. Il est possible de *cacher cette variable dans la fonction* grâce au mot clé **static** placé devant la variable :

```

// Fonction random qui appelle srand() toute seule
// au premier appel... avec sa variable globale
// masquée à l'intérieur
double random() {
    static bool first=true; // Ne pas oublier static!
    if (first) {
        first=false;
        srand((unsigned int)time(0));
    }
    return double(rand())/RAND_MAX;
}

```

Attention : il s'agit bien d'une variable globale et non d'une variable locale. Une variable locale mourrait à la sortie de la fonction, ce qui dans l'exemple précédent donnerait un comportement non désiré !

NB : Il est aussi possible de cacher une variable globale dans une classe, toujours grâce à **static**. Nous ne verrons pas comment et renvoyons le lecteur à la documentation du C++.

12.6 **const** et tableaux

Nous avons vu malgré nous **const char *** comme paramètre de certaines fonctions (ouverture de fichier par exemple). Il nous faut donc l'expliquer : *il ne s'agit pas d'un pointeur de **char** qui serait constant mais d'un pointeur vers des **char** qui sont constants !* Il faut donc retenir que :

placé devant un tableau, **const** signifie que ce sont les éléments du tableau qui ne peuvent être modifiés.

Cette possibilité de préciser qu'un tableau ne peut être modifié est d'autant plus importante qu'un tableau est toujours passé en référence : sans le **const**, on ne pourrait assurer cette préservation des valeurs :

```
void f(int t[4]) {
    ...
}

void g(const int t[4]) {
    ...
}

void h(const int* t, int n) {
    ...
}

...
int a[4];
f(a);    // modifie peut-être a[]
g(a);    // ne modifie pas a[]
h(a,4);  // ne modifie pas a[]
...
```

12.7 Fiche de référence

Fiche de référence (1/6)		
Boucles — do { ... } while(!ok); — int i=1; while(i<=100) { ... i=i+1; } — for(int i=1;i<=10;i++) ... — for(int i=1,j=10;j>i; i=i+2,j=j-3) ... — for (int i=...) for (int j=...) { //saute cas i==j if (i==j) continue; ... } — for (int i=...) { ... if (t[i]==s){ // quitte boucle break; } ... }	— if (i==0) j=1; — if (i==0) j=1; else j=2; — if (i==0) { j=1; k=2; } — bool t=(i==0); if (t) j=1; — switch (i) { case 1: ...; ...; break; case 2: case 3: ...; break; default: ...; } — mx=(x>y)?x:y;	void afficher(int x, int y) { if (x<0 y<0) return; if (x>=w y>=h) return; DrawPoint(x,y,RED); } — Appel: int f(int a) { ... } int g() { ... } ... int i=f(2),j=g(); — Références : void swap(int& a, int& b){ int tmp=a; a=b;b=tmp; } ... int x=3,y=2; swap(x,y); — Surcharge : int hasard(int n); int hasard(int a, int b); double hasard(); — Opérateurs : vect operator+(vect A,vect B) { ... } ... vect C=A+B; — Pile des appels — Itératif/Récuratif — Références constantes (pour un passage rapide) : void f(const obj& x){ ... } void g(const obj& x){ f(x); // OK }
Clavier — Debug : F5  — Step over : F10  — Step inside : F11  — Indent : Ctrl+A, Ctrl+I — Step out : Maj+F11  — Gest. tâches : Ctrl+Maj+Ech Tests — Comparaison : == != < > <= >= — Négation : ! — Combinaisons : &&	Fonctions — Définition : int plus(int a,int b){ int c=a+b; return c; } void affiche(int a) { cout << a << endl; } — Déclaration : int plus(int a,int b); — Retour : int signe(double x) { if (x<0) return -1; if (x>0) return 1; return 0; }	

Fiche de référence (2/6)		
Valeurs par défaut : <pre>void f(int a,int b=0); void g() { f(12); // f(12,0); f(10,2); // f(10,2); } void f(int a,int b) { // ... }</pre> Inline (appel rapide) : <pre>inline double sqr(double x) { return x*x; } ... double y=sqr(z-3);</pre> Référence en retour : <pre>int i; // Var. globale int& f() { return i; } ... f(); // i=3!</pre> Structures <pre>— struct Point { double x,y; Color c; }; ... Point a; a.x=2.3; a.y=3.4; a.c=Red; Point b={1,2.5,Blue};</pre> — Une structure est un objet entièrement public (→ cf objets!) Variables — Définition : <pre>int i; int k,l,m;</pre> — Affectation : <pre>i=2; j=i; k=l=3;</pre> — Initialisation : <pre>int n=5,o=n;</pre> — Constantes : <pre>const int s=12;</pre>	— Portée : <pre>int i; // i=j; interdit! int j=2; i=j; // OK! if (j>1) { int k=3; j=k; // OK! } //i=k; interdit!</pre> — Types : <pre>int i=3; double x=12.3; char c='A'; string s="hop"; bool t=true; float y=1.2f; unsigned int j=4; signed char d=-128; unsigned char d=25; complex<double> z(2,3);</pre> — Variables globales : <pre>int n; const int m=12; void f() { n=10; // OK int i=m; // OK ... }</pre> — Conversion : <pre>int i=int(x),j; float x=float(i)/j;</pre> — Pile/Tas — Type énuméré : <pre>enum Dir{N,E,S,W}; void avance(Dir d);</pre> — Variables statiques : <pre>int f() { static bool once=true; if (once) { once=false; ... } ... }</pre> Objets <pre>— struct obj { int x; // champ</pre>	<pre>int f(); // méthode int g(int y); }; int obj::f() { int i=g(3); // mon g int j=x+i; // mon x return j; } ... int main() { obj a; a.x=3; int i=a.f(); ... }</pre> — class obj { <pre> int x,y; void a_moi(); public: int z; void pour_tous(); void autre(obj A); }; void obj::a_moi() { x=..; // OK ..=y; // OK z=..; // OK } void obj::pour_tous(){ x=..; // OK a_moi(); // OK } void autre(obj A) { x=A.x; // OK A.a_moi(); // OK } ... int main() { obj A,B; A.x=..; //NON A.z=..; //OK A.a_moi(); //NON A.pour_tous(); //OK A.autre(B); //OK ... }</pre> — class obj { <pre> obj operator+(obj B); }; ... int main() { obj A,B,C; C=A+B; // C=A.operator+(B)</pre>

Fiche de référence (3/6)		
Méthodes constantes : <pre>void obj::f() const{ ... } void g(const obj& x){ x.f(); // OK }</pre> Constructeur : <pre>class point { int x,y; public: point(int X,int Y); }; point::point(int X, int Y){ x=X; y=Y; } ... point a(2,3);</pre> Constructeur vide : <pre>obj::obj() { ... } ... obj a;</pre> Objets temporaires : <pre>vec vec::operator+(vec b) { return vec(x+b.x, y+b.y); } ... c=vec(1,2) +f(vec(2,3));</pre> Destructeur : <pre>obj::~~obj() { ...</pre>	<pre>} Constructeur de copie : obj::obj(const obj& o){ ... } Utilisé par : - obj b(a); - obj b=a; //mieux que obj b;b=a; - paramètres des fonctions - valeur de retour Affectation : obj& obj::operator=(const obj&o){ ... return *this; } Objets avec allocation dynamique automatique : cf section 10.11 Accesseurs : class mat { double *x; public: double& operator() (int i,int j){ assert(i>=0 ...); return x[i+M*j]; } double operator() (int i,int j)const{ assert(i>=0 ...); return x[i+M*j]; } ...</pre> Compilation séparée — #include "vect.h", aussi dans vect.cpp	— Fonctions : déclarations dans le .h, définitions dans le .cpp — Types : définitions dans le .h — Ne déclarer dans le .h que les fonctions utiles. — #pragma once au début du fichier. — Ne pas trop découper...
		STL — min,max,... — complex<double> z; — pair<int,string> p; p.first=2; p.second="hop"; — #include<list> using namespace std; ... list<int> l; l.push_front(1); ... — if(l.find(3)!=l.end()) ... — list<int>::const_iterator it; for (it=l.begin(); it!=l.end();it++) s+= *it; — list<int>::iterator it for (it=l.begin(); it!=l.end();it++) if (*it==2) *it=4; — stack, queue, heap, map, set, vector...

Fiche de référence (4/6)**Template**

— Fonctions :

```
// A mettre dans le
// fichier qui utilise
// ou dans un .h
template <typename T>
T maxi(T a,T b) {
    ...
}
...
// Le type est trouvé
// tout seul!
maxi(1,2); //int
maxi(.2,.3); //double
maxi("a","c");//string
```

— Objets :

```
template <typename T>
class paire {
    T x[2];
public:
    paire() {}
    paire(T a,T b) {
        x[0]=a;x[1]=b;
    }
    T add() const;
};
...
template <typename T>
T paire<T>::add() const {
    return x[0]+x[1];
}
...
// Le type doit être
// précisé!
paire<int> a(1,2);
int s=a.somme();
paire<double> b;
...
```

— Multiples :

```
template <typename T,
          typename S>
class hop {
    ...
};
...
hop<int,string> A;
...
```

— Entiers :

```
template <int N>
```

```
class hop {
    ..
};
...
hop<3> A;
...
```

Entrées/Sorties

```
— #include <iostream>
using namespace std;
...
cout <<"I="<<i<<endl;
cin >> i >> j;

— #include <fstream>
using namespace std;
ofstream f("hop.txt");
f << 1 << ' ' << 2.3;
f.close();
ifstream g("hop.txt");
if (!g.is_open()) {
    return 1;
}
int i;
double x;
g >> i >> x;
g.close();

— do {
    ...
} while (!(g.eof()));

— ofstream f;
f.open("hop.txt");

— double x[10],y;
ofstream f("hop.bin",
           ios::binary);
f.write((const char*)x,
        10*sizeof(double));
f.write((const char*)&y,
        sizeof(double));
f.close();
ifstream g("hop.bin",
           ios::binary);
g.read((char*)x,
        10*sizeof(double));
g.read((const char*)&y,
        sizeof(double));
g.close();

— string s;
ifstream f(s.c_str());

— #include <sstream>
using namespace std;
```

```
stringstream f;
// Chaîne vers entier
f << s;
f >> i;
// Entier vers chaîne
f.clear();
f << i;
f >> s;
```

```
— ostream& operator<<(
    ostream& f,
    const point&p){
    f<<p.x<<' '<<p.y;
    return f;
}
istream& operator>>(
    istream& f,point& p){
    f>>p.x>>p.y;
    return f;
}
```

Conseils

- Nettoyer en quittant.
- Erreurs et warnings : cliquer.
- Indenter.
- Ne pas laisser de warning.
- Utiliser le debugueur.
- Faire des fonctions.
- Tableaux : pas pour transcrire une formule mathématique !
- Faire des structures.
- Faire des fichiers séparés.
- Le .h doit suffire à l'utilisateur (qui ne doit pas regarder le .cpp)
- Ne pas abuser du récursif.
- Ne pas oublier delete.
- Compiler régulièrement.
- #include <cassert>


```
...
assert(x!=0);
y=1/x;
```
- Faire des objets.
- Ne pas toujours faire des objets !
- Penser interface / implémentation / utilisation.

Fiche de référence (5/6)		
Divers — i++; i--; i-=2; j+=3; — j=i%n; // Modulo — #include <cstdlib> ... i=rand()%n; x=rand()/ double(RAND_MAX); — #include <ctime> // Un seul appel srand((unsigned int) time(0)); — #include <cmath> double sqrt(double x); double cos(double x); double sin(double x); double acos(double x); — #include <string> using namespace std; string s="hop"; char c=s[0]; int l=s.size(); if (s1==s1) ... if (s1!=s2) ... if (s1<s2) ... size_t i=s.find('h'), j=s.find('h',3); k=s.find("hop"); l=s.find("hop",3); a="comment"; b="ça va?"; txt=a+" "+b; s1="un deux trois"; s2=string(s1,3,4); getline(cin,s); getline(cin,s,':'); const char *t=s.c_str(); — #include <ctime> s=double(clock()) /CLOCKS_PER_SEC; — #include <cmath> double pi=M_PI; — Opérateurs binaires and: a&b or: a b xor: a^b right shift: a>>n left shift: a<<n complement: ~a exemples:	set(i,1): i =(1<<n) reset(i,1): i&=~(1<<n) test(i,1): if (i&(1<<n)) flip(i,1): i^=(1<<n) Erreurs fréquentes — Pas de définition de fonction dans une fonction! — int q=r=4; // NON! — if (i=2) // NON! if i==2 // NON! if (i==2) then // NON! — for(int i=0,i<100,i++) // NON! — int f() {...} int i=f; // NON! — double x=1/3; // NON! int i,j; x=i/j; // NON! x=double(i/j); //NON! — double x[10],y[10]; for(int i=1;i<=10;i++) y[i]=2*x[i]; //NON — int n=5; int t[n]; // NON — int f()[4] { // NON! int t[4]; ... return t; // NON! } int t[4]; t=f(); — int s[3]={1,2,3},t[3]; t=s; // NON! — int t[2]; t={1,2}; // NON! — struct Point { double x,y; } // NON! — Point a; a={1,2}; // NON! — #include "tp.cpp"//NON — int f(int t[][]); //NON int t[2,3]; // NON! t[i,j]=...; // NON! — int* t; t[1]=...; // NON! — int* t=new int[2]; int* s=new int[2]; s=t; // On perd s! delete[] t; delete[] s; //Déjà fait	— int *t,s; // s est int // non int* t=new int[n]; s=new int[n]; // NON! — class vec { int x,y; public: ... }; ... vec a={2,3}; // NON — vec v=vec(1,2); //NON vec v(1,2); // OUI — obj* t=new obj[n]; delete t; // manque [] — //NON! void f(int a=2,int b); — void f(int a,int b=0); void f(int a); // NON! — Ne pas tout mettre inline! — int f() { ... } f()=3; // HORREUR! — int& f() { int i; return i; } f()=3; // NON! — if (i>0 & i<n) // NON if (i<0 i>n) // NON — if (...) { ... if (...) break; // Non, // boucles seulement } — for (i ...) for (j ...) { ... if (...) break; //NON, quitte // juste boucle j } — int i; double x; j=max(i,0); //OK y=max(x,0); //NON! // 0.0 et non 0: max // est un template STL Imagine++ — Voir documentation...

Fiche de référence (6/6)**Tableaux**

— Définition :

```
— double x[5], y[5];
  for(int i=0; i<5; i++)
    y[i]=2*x[i];

— const int n=5;
  int i[n], j[2*n];
```

— Initialisation :

```
int t[4]={1,2,3,4};
string s[2]={"ab", "c"};
```

— Affectation :

```
int s[3]={1,2,3}, t[3];
for (int i=0; i<3; i++)
  t[i]=s[i];
```

— En paramètre :

```
— void init(int t[4]) {
  for(int i=0; i<4; i++)
```

```
  t[i]=0;
```

```
}
```

```
— void init(int t[],
           int n) {
  for(int i=0; i<n; i++)
    t[i]=0;
}
```

— Taille variable :

```
int* t=new int[n];
...
delete[] t;
```

— En paramètre (suite) :

```
— void f(int* t, int n){
  t[i]=...
}
```

```
— void alloue(int*& t){
  t=new int[n];
}
```

— 2D :

```
int A[2][3];
A[i][j]=...;
int A[2][3]=
  {{1,2,3},{4,5,6}};
void f(int A[2][2]);
```

— 2D dans 1D :

```
int A[2*3];
A[i+2*j]=...;
```

— Taille variable (suite) :

```
int *t, *s, n;
```

— En paramètre (fin) :

```
void f(const int* t,
      int n) {
  ...
  s+=t[i]; // OK
  ...
  t[i]=...; // NON!
}
```

Annexe A

Travaux Pratiques

Note : les corrigés seront disponibles sur la page web du cours après chaque TP.

A.1 L'environnement de programmation

A.1.1 Bonjour, Monde !

1. *Connexion :*

Se connecter sous Windows ou Linux.

2. *Projet :*

Télécharger l'archive `Tp1_Initial.zip` sur la page du cours, la décompresser sur le bureau. Le fichier `CMakeLists.txt` décrit le projet. La ligne qui nous intéresse est la suivante :

```
add_executable(Tp1 Tp1.cpp)
```

Elle indique que le programme s'appellera `Tp1` (`Tp1.exe` sous Windows) et que le code source pour le construire est dans le fichier `Tp1.cpp`.

- (a) Visual (donc sous Windows) n'est pas capable d'interpréter le fichier `CMakeLists.txt`, il faut utiliser le programme `Cmake` au préalable. Lancer donc ce programme, et aller chercher comme répertoire "source code" `Tp1_Initial` par le bouton "Browse source...". C'est une bonne idée de séparer les fichiers générés des sources : sélectionner dans la ligne suivante un nouveau répertoire "Build". Cliquer "Generate" et sélectionner le bon générateur (Visual Studio 2008, sur les machines de l'École). Si tout s'est bien passé, on peut fermer `Cmake` et ouvrir le fichier "solution" (extension `.sln`) dans le répertoire Build, ce qui lance "Visual Studio"
 - (b) Sous Linux, lancer Kdevelop, le menu "Open Project" et choisir le `CMakeLists.txt`. Les choix proposés par défaut (en particulier le mode "Debug") sont corrects. Kdevelop comprend le format `CMakeLists` et est capable de lancer `Cmake` lui-même.
 - (c) QtCreator (Linux, Windows ou Mac) connaît également `Cmake`, donc procédez comme avec Kdevelop.
3. *Programmation :*
- (a) Rajouter `cout << "Hello" << endl;` sur la ligne avant `return 0;`

4. Génération :

- (a) Dans la fenêtre "Solution explorer" de Visual Studio, rechercher et afficher le fichier `Tp1.cpp`.
- (b) "Build/Build solution", ou "F7" ou bouton correspondant.
- (c) Vérifier l'existence d'un fichier `Tp1` (`Tp1.exe` sous Windows) dans Build/`Tp1` (Build/`Tp1/Debug` sous Windows).

5. Exécution :

- (a) Lancer le programme (sous Visual) avec "Debug/Start Without Debugging" (ou "Ctrl+F5" ou bouton correspondant). Il faut d'abord lui préciser quel programme lancer (clic droit sur le projet `Tp1` dans l'explorateur de solution, "Sélectionner comme projet de démarrage") Une fenêtre à fonds noir "console" s'ouvre, dans laquelle le programme s'exécute, et la fenêtre console reste ouverte jusqu'à l'appui d'une touche du clavier.
- (b) Vérifier qu'on a en fait créé un programme indépendant qu'on peut lancer dans une fenêtre de commande :
 - Essayer de le lancer depuis le gestionnaire de fichiers standard : le programme se referme tout de suite !
 - Dans les menus Windows : "Démarrer/Exécuter"
 - "Ouvrir: cmd"
 - Taper "D:",¹ puis
"cd \Documents and Settings\login\Bureau\Tp1\Tp1\Debug"
(apprenez à profiter de la complétion automatique avec la touche TAB).
 - Vérifier la présence de `Tp1.exe` avec la commande "dir".
 - Taper "`Tp1`".
- (c) Sous Kdevelop (Linux), il faut commencer par aller dans le menu "Configure launches", ajouter avec le bouton "+" et sélectionner l'exécutable. On peut alors le lancer avec le bouton "Execute".

Touche utile (Visual) : Ctrl+F5 =  = Start without debugging
--

6. Fichiers :

On a déjà suivi la création des fichiers principaux au fur et à mesure. Constatons la présence de `Tp1.obj` qui est la compilation de `Tp1.cpp` (que le linker a ensuite utilisé pour créer `Tp1.exe`). Voir aussi la présence de nombreux fichiers de travail. Quelle est la taille du répertoire de Build de `Tp1` (clic droit + propriétés) ?

7. Nettoyage :

Supprimer les fichiers de travail et les résultats de la génération avec "Build / Clean solution" puis fermer Visual Studio. Quelle est la nouvelle taille du répertoire ?

1. Sur certaines machines, il faut en fait aller sur C :, vérifiez en regardant où est installé votre projet

8. *Compression :*

Sous Windows, en cliquant à droite sur le répertoire `Tp1`, fabriquer une archive comprimée `Tp1.zip` (ou `Tp1.7z` suivant la machine). Attention il faut quitter Visual Studio avant de compresser. Il peut sinon y avoir une erreur ou certains fichiers trop importants peuvent subsister.

Il faut quitter Visual Studio avant de compresser.

Une telle archive, comprenant tous les fichiers nécessaires à la compilation, sera le format utilisé pour déposer vos exercices et rendus de TP sur Educnet.

Notez bien qu'avec Cmake nous avons deux dossiers :

- Le dossier *source* contenant les fichiers `Tp1.cpp` et `CMakeLists.txt` ;
- Le dossier *build* que vous avez choisi au démarrage de Cmake.

Le plus important est le premier, puisque le deuxième peut toujours être régénéré avec Cmake. N'envoyez à votre enseignant que le répertoire source, il recompilera lui-même. Votre *build* lui est probablement inutile car il n'utilise pas le même système que vous. Ainsi, quand vous avez terminé un projet ou un TP, n'hésitez pas à nettoyer en supprimant votre dossier *build*, mais gardez précieusement votre dossier *source*, c'est celui-ci qui représente le résultat de votre travail. Bien que Cmake autorise d'utiliser un même répertoire pour les deux, c'est à éviter, pour bien séparer les sources et les fichiers générés automatiquement. Pour comprendre un peu comment tous ces outils se coordonnent, on peut se reporter à l'annexe C du polycopié.

A.1.2 Premières erreurs

1. *Modifier le programme :*

Modifier le programme en changeant par exemple la phrase affichée.

- (a) Tester une nouvelle génération/exécution. Vérifier que Visual Studio sauve le fichier automatiquement avant de générer.
- (b) Modifier à nouveau le programme. Tester directement une exécution. Visual Studio demande automatiquement une génération !

Lancer directement une exécution sauve et génère automatiquement. Attention toutefois de ne pas confirmer l'exécution si la génération s'est mal passée.

2. *Erreurs de compilation*

Provoquer, constater et apprendre à reconnaître quelques erreurs de compilation :

- (a) `includ` au lieu de `include`
- (b) `iostrem` au lieu de `iostream`
- (c) Oublier le `;` après `std`
- (d) `inte` au lieu de `int`
- (e) `cou` au lieu de `cout`
- (f) Oublier les guillemets `"` fermant la chaîne `"Hello_..."`
- (g) Rajouter une ligne `i=3;` avant le **return**.

A ce propos, il est utile de découvrir que :

Double-cliquer sur un message d'erreur positionne l'éditeur sur l'erreur.

3. Erreur de linker

Il est un peu tôt pour réussir à mettre le linker en erreur. Il est pourtant indispensable de savoir différencier ses messages de ceux du compilateur. En général, le linker indiquera une erreur s'il ne trouve pas une fonction ou des variables parce qu'il manque un fichier objet ou une bibliothèque. C'est aussi une erreur s'il trouve deux fois la même fonction...

- (a) Rajouter une ligne `f (2) ;` avant le **return** et faire `Ctrl+F7`. C'est pour l'instant une erreur de compilation.
- (b) Corriger l'erreur de compilation en rajoutant une ligne (pour l'instant "magique")
`void f(int i);` avant la ligne avec `main`. Compiler sans linker : il n'y a plus d'erreur. Générer le programme : le linker constate l'absence d'une fonction `f ( )` utilisée par la fonction `main()` qu'il ne trouve nulle part.

4. Indentations :

Avec toutes ces modifications, le programme ne doit plus être correctement "indenté". C'est pourtant essentiel pour une bonne compréhension et repérer d'éventuelle erreur de parenthèses, accolades, etc. Le menu `Edit/Advanced` fournit de quoi bien indenter.

Pour repérer des erreurs, toujours bien indenter.
Ctrl+K, Ctrl+F (Visual) = Ctrl+I (QtCreator) = indenter la zone sélectionnée.
Ctrl+A, Ctrl+K, Ctrl+F (Visual) = Ctrl+A, Ctrl+I (QtCreator) = tout indenter.

5. Warnings du compilateur

En modifiant le `main()`, provoquer les warnings suivants :²

- (a) `int i;`
`i=2.5;`
`cout << i << endl;`
 Exécuter pour voir le résultat.
- (b) `int i;`
`i=4;`
`if (i=3) cout << "salut" << endl;`
 Exécuter !
- (c) `int i, j;`
`j=i;`
 Exécuter (répondre "abandonner" !)
- (d) Provoquer le warning inverse : variable déclarée mais non utilisée.

2. Certains de ces warnings ne se manifestent qu'en niveau d'exigence le plus élevé : pour le mettre en place, clic droit sur le projet dans la fenêtre de gauche, menu "Propriétés", onglet C++, sélectionner le "warning level" 4 (3, moins exigeant, est le défaut).

- (e) Ajouter `exit`; comme première instruction de `main`. Appeler une fonction en oubliant les arguments arrive souvent! Exécuter pour voir. Corriger en mettant `exit (0)`; . Il y a maintenant un autre warning. Pourquoi? (La fonction `exit ()` quitte le programme en urgence!)

Il est très formellement déconseillé de laisser passer des warnings! Il faut les corriger au fur et à mesure. Une option du compilateur propose même de les considérer comme des erreurs!

A.1.3 Debugger

Savoir utiliser le debugueur est essentiel. Il doit s'agir du premier réflexe en présence d'un programme incorrect. C'est un véritable moyen d'investigation, plus simple et plus puissant que de truffer son programme d'instructions supplémentaires destinées à espionner son déroulement.

1. Taper le `main` suivant.

```
int main()
{
    int i, j, k;
    i=2;
    j=3*i;
    if (j==5)
        k=3;
    else
        k=2;
    return 0;
}
```

2. Pour pouvoir utiliser le debugueur avec QtCreator ou Kdevelop, il faut compiler en mode Debug. Cela se fait en modifiant une variable de `Cmake`. Sous QtCreator, aller dans l'onglet Projects et ajouter comme argument :

```
-DCMAKE_BUILD_TYPE=Debug
```

Cela écrit dans le fichier `CMakeCache.txt` généré par `Cmake`.

3. Lancer le programme "sous le debugueur" avec `Build/Start` ou `F5` ou le bouton correspondant. Que se passe-t-il?
4. Placer un "point d'arrêt" en cliquant dans la colonne de gauche à la hauteur de la ligne `i=2`; puis relancer le debugueur.
5. Avancer en "Step over" avec `F10` ou le bouton correspondant et suivre les valeurs des variables (dans la fenêtre spéciale ou en plaçant (sans cliquer!) la souris sur la variable).
6. A tout moment, on peut interrompre l'exécution avec `Stop` ou `Maj+F5`. Arrêter l'exécution avant d'atteindre la fin de `main` sous peine de partir dans la fonction qui a appelé `main`!

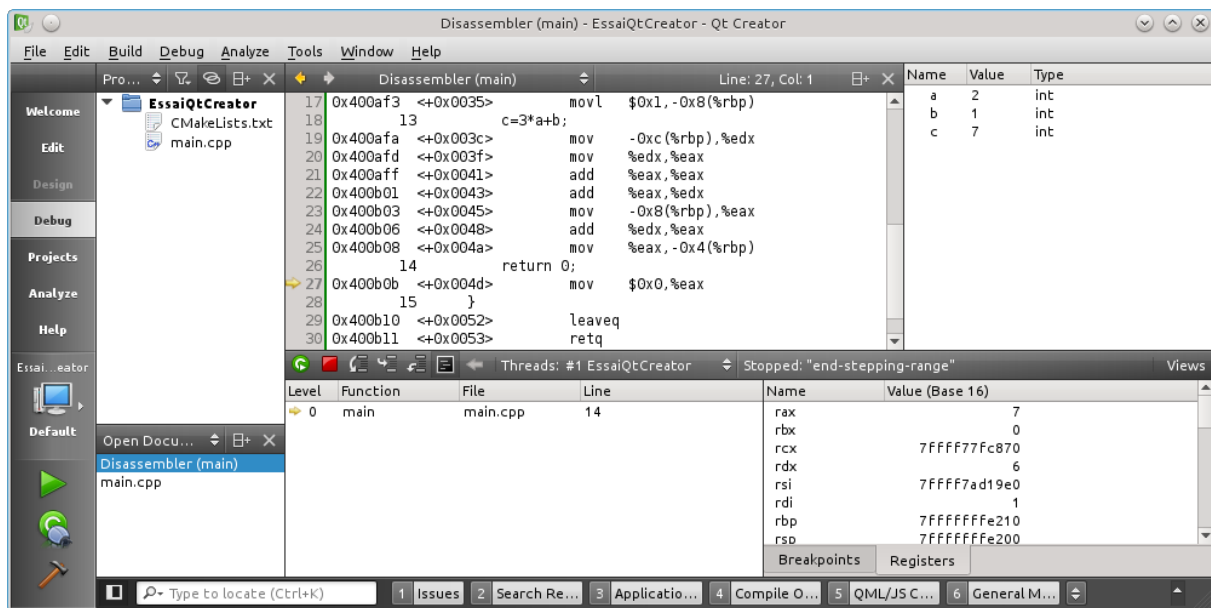


FIGURE A.1 – QtCreator montrant le code machine et les registres

7. Placer un deuxième point d'arrêt et voir que F5 redémarre le programme jusqu'au prochain point d'arrêt rencontré.
8. Ajouter `i=max(j,k);` avant le **return**. Utiliser "Step into" ou F11 ou le bouton correspondant quand le curseur est sur cette ligne. Constater la différence avec F10.
9. Enfin, pour voir à quoi ressemble du code machine, exécuter jusqu'à un point d'arrêt puis faire Debug/Windows/Disassembly. On peut aussi dans ce même menu voir les registres du micro-processeur. Il arrive qu'on se retrouve dans la fenêtre "code machine" sans l'avoir demandé quand on debugge un programme pour lequel on n'a plus le fichier source. Cet affichage est en fait très utile pour vérifier ce que fait le compilateur et voir s'il optimise bien.
10. Pour faire l'étape précédente sous QtCreator, on peut sélectionner l'option "Operate by Instruction" du menu Build . Les registres sont visibles en allant dans le menu Window puis Views. Voir Figure A.1.

	F5	=		=	Debug
Touches utiles :	F10	=		=	Step over
	F11	=		=	Step inside

A.1.4 S'il reste du temps

Téléchargez le programme supplémentaire `Tp1_Final.zip` sur la page du cours (<http://imagine.enpc.fr/~monasse/Info>), jouez avec...et complétez-le !

A.1.5 Installer Imagine++ chez soi

Allez voir sur <http://imagine.enpc.fr/~monasse/Imagine++>. Installez sur votre ordinateur portable et essayez de refaire ce TP avec QtCreator.

A.2 Variables, boucles, conditions, fonctions

A.2.1 Premier programme avec fonctions

1. *Récupérer le programme exemple :*

Télécharger l'archive `Tp2.zip` sur la page du cours, la décompresser sur le bureau et ouvrir la solution dans Visual. Étudier le projet `Hop` dont voici les sources :

```
1 #include <iostream>
2 using namespace std;
3
4 int plus(int a,int b)
5 {
6     int c;
7     c=a+b;
8     return c;
9 }
10
11 void triple1(int a)
12 {
13     a=a*3;
14 }
15
16 void triple2(int& a)
17 {
18     a=a*3;
19 }
20
21 int main()
22 {
23     int i , j =2,k;
24     i=3;
25     k=plus(i , j );
26     triple1(i);
27     triple2(i);
28     return 0;
29 }
```

2. *Debugger :*

Exécuter le programme pas à pas et étudier la façon dont les variables changent. Souvenez-vous que si vous n'utilisez pas Visual, vous devez utiliser Cmake avec l'option `-DCMAKE_BUILD_TYPE=Debug`.

A.2.2 Premier programme graphique avec Imagine++

Dans ce TP et les suivants, nous utiliserons la librairie graphique d'Imagine++ (cf annexe du polycopié). Elle permet de gérer très simplement le fenêtrage, le dessin, et les entrées-sorties clavier et souris.

1. *Programme de départ :*

Etudier le programme du projet `Tennis` dont voici le source :

```

1 #include <Imagine/Graphics.h>
2 using namespace Imagine;
3 ...
4
5 //////////////////////////////////////
6 // Fonction principale
7 int main()
8 {
9     // Ouverture de la fenetre
10    openWindow(256,256);
11    // Position et vitesse de la balle
12    int xb=128,
13        yb=20,
14        ub=2,
15        vb=3;
16    // Boucle principale
17    while (true) {
18        // Affichage de la balle
19        fillRect(xb-3,yb-3,7,7,Red);
20        // Temporisation
21        millisleep(20);
22        // Effacement de la balle
23        fillRect(xb-3,yb-3,7,7,White);
24        // Rebond
25        if (xb+ub>253)
26            ub=-ub;
27        // Mise a jour de la position de la balle
28        xb+=ub;
29        yb+=vb;
30    }
31    endGraphics();
32    return 0;
33 }
```

Ne pas s'intéresser à la fonction `Clavier()`

Générer puis exécuter la solution. Que se passe-t-il ?

2. *Aide de Visual Studio* A tout moment, la touche `F1` permet d'accéder à la documentation du mot clé situé sous le curseur. Tester cette fonctionnalité sur les mots-clés `if`, `while` et `return`.

Touche utile : `F1` = Accéder à la documentation

3. *Comprendre le fonctionnement du programme :*

Identifier la boucle principale du programme. Elle se décompose ainsi :

- (a) Affichage de la balle
- (b) Temporisation de quelques millisecondes pour que la balle ne se déplace pas trop vite
- (c) Effacement de la balle
- (d) Gestion des rebonds
- (e) Mise à jour des coordonnées de la balle

Pourquoi la ligne comportant l’instruction **while** suscite-t-elle un warning ? A quoi sert la condition formulée par l’instruction **if** ?

4. *Gestion de tous les rebonds :*

Compléter le programme afin de gérer tous les rebonds. Par exemple, il faut inverser la vitesse horizontale *ub* quand la balle va toucher les bords gauche ou droit de la fenêtre.

5. *Variables globales :*

Doubler la hauteur de la fenêtre. Modifier la taille de la balle. Cela nécessite de modifier le code à plusieurs endroits. Aussi, à la place de valeurs numériques “en dur”, il vaut mieux définir des variables. Afin de simplifier et bien que ça ne soit pas toujours conseillé, utiliser des variables globales constantes. Pour cela, insérer tout de suite après les deux lignes d’include le code suivant

```
const int width = 256;    // Largeur de la fenetre
const int height = 256;  // Hauteur de la fenetre
const int ball_size = 3;  // Rayon de la balle
```

et reformuler les valeurs numériques du programmes à l’aide de ces variables. Le mot clé **const** indique que ces variables ne peuvent être modifiées après leur initialisation. Essayer de rajouter la ligne `width=300;` au début de la fonction `main` et constater que cela provoque une erreur de compilation.

6. *Utilisation de fonctions :*

La balle est dessinée deux fois au cours de la boucle, la première fois en rouge et la deuxième fois en blanc pour l’effacer. Ici le dessin de la balle ne nécessite qu’une ligne mais cela pourrait être beaucoup plus si sa forme était plus complexe. Aussi, pour que le programme soit mieux structuré et plus lisible, et que le code comporte le moins possible de duplications, regrouper l’affichage de la balle et son effacement dans une fonction `DessineBalle` définie avant la fonction `main` :

```
void DessineBalle(int x,int y,Color col) {
    ...
}
```

De même, définir une fonction

```
void BougeBalle(int &x,int &y,int &u,int &v)
pour gérer les rebonds et le déplacement de la balle.
```

A.2.3 Jeu de Tennis

Nous allons rendre ce programme plus ludique en y ajoutant deux raquettes se déplaçant horizontalement en haut et en bas de l’écran, et commandées par les touches du clavier.

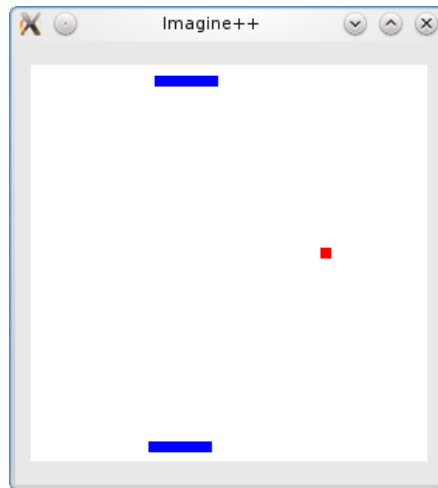


FIGURE A.2 – Mini tennis...

1. *Affichage des raquettes :*

Ajouter dans la fonction `main` des variables `xr1,yr1,xr2,yr2` dédiées à la position des deux raquettes. Puis définir une fonction `DessineRaquette` en prenant modèle sur `DessineBalle`. Placer les appels de ces fonctions aux endroits appropriés dans la boucle principale.

2. *Gestion du clavier :*

La gestion du clavier est réalisée pour vous par la fonction `Clavier` dont nous ignorerons le contenu pour l'instant. Cette fonction nous permet de savoir directement si une des touches qui nous intéressent (q et s pour le déplacement de la première raquette, k et l pour la deuxième) sont enfoncées ou non. Cette fonction, `Clavier(int& sens1, int& sens2)`, retourne dans `sens1` et `sens2`, les valeurs 0, -1 ou 1 (0 : pas de déplacement, -1 : vers la gauche, 1 : vers la droite).

3. *Déplacement des raquettes :*

Coder le déplacement d'une raquette dans une fonction

void BougeRaquette(int &x, int sens)

puis appeler cette fonction dans la boucle principale pour chacune des deux raquettes. Evidemment, faire en sorte que les raquettes ne puissent sortir de la fenêtre.

4. *Rebonds sur les raquettes :*

S'inspirer de la gestion des rebonds de la balle. Ici il faut non seulement vérifier si la balle va atteindre le bas ou le haut de l'écran mais aussi si elle est assez proche en abscisse de la raquette correspondante.

5. *Comptage et affichage du score :*

Modifier la fonction `BougeBalle` afin de comptabiliser le score des deux joueurs et l'afficher dans la console.

6. *Pour ceux qui ont fini :*

Lors d'un rebond sur la raquette, modifier l'inclinaison de la trajectoire de la balle en fonction de la vitesse de la raquette ou de l'endroit de frappe.

Vous devriez avoir obtenu un programme ressemblant à celui de la figure A.2.

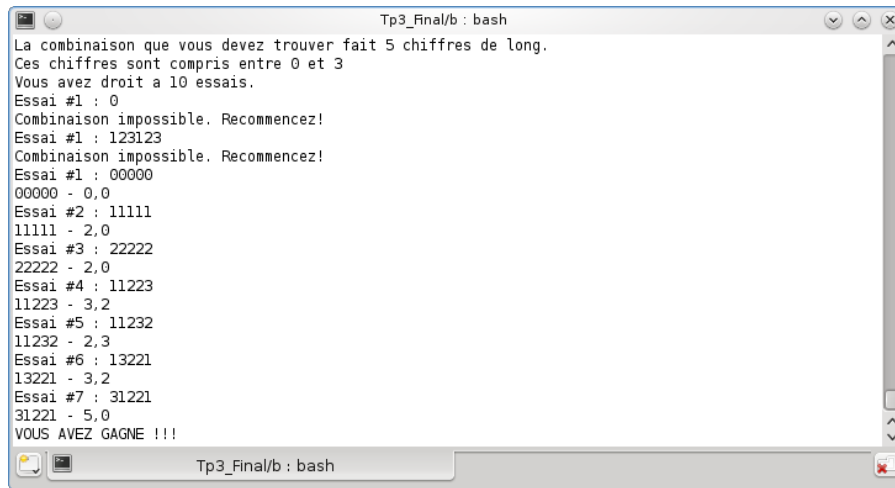


FIGURE A.3 – Master mind à la console...

A.3 Tableaux

Dans ce TP, nous allons programmer un jeu de Mastermind, où l'utilisateur doit deviner une combinaison générée aléatoirement par l'ordinateur. Le joueur dispose d'un nombre déterminé d'essais. A chaque essai d'une combinaison, l'ordinateur fournit deux indices : le nombre de pions correctement placés et le nombre de pions de la bonne couleur mais incorrectement positionnés.

A.3.1 Mastermind Texte

1. *Récupérer la solution de départ :*

Télécharger l'archive `Tp3_Initial.zip` sur la page du cours, la décompresser dans un répertoire faisant apparaître les noms des deux élèves (en cas de binôme) et ouvrir le projet `MasterMind`. Etudier ce projet.

2. *Représenter une combinaison :*

Nous prendrons ici une combinaison de 5 pions de 4 couleurs différentes. La couleur d'un pion sera représentée par un entier compris entre 0 et 3. Pour une combinaison de 5 pions, nous allons donc utiliser un tableau de 5 entiers.

```
int combin[5]; // tableau de 5 entiers
```

3. *Afficher une combinaison :*

Programmer une fonction permettant d'afficher une combinaison donnée à l'écran. La manière la plus simple de faire consistera à faire afficher les différents chiffres de la combinaison sur une même ligne les uns à la suite des autres.

4. *Générer une combinaison aléatoirement :*

En début de partie, l'ordinateur doit générer aléatoirement une combinaison à faire deviner à l'utilisateur. Nous allons pour cela utiliser les fonctions déclarées dans le fichier `cstdlib`, notamment la fonction `rand()` permettant de générer un nombre au hasard entre 0 et `RAND_MAX`. Afin d'obtenir un nombre entre 0 et `n`, on procédera de la manière suivante :

```
x = rand()%n;
```

Pour que la séquence de nombres générée ne soit pas la même d'une fois sur l'autre, il est nécessaire d'initialiser le générateur avec une graine variable. La manière la plus simple de procéder consiste à utiliser l'heure courante. La fonction `time()` déclarée dans le fichier `ctime` permet de l'obtenir.

En fin de compte, la fonction suivante nous permet donc de générer une combinaison :

```
#include <cstdlib>
#include <ctime>
using namespace std;

void genereCombinaison(int combin[5])
{
    for (int i=0; i<5; ++i)
        combin[i] = rand()%4;    // appels au generateur
}
...
srand((unsigned int)time(0)); // initialisation
genereCombinaison(combin);
```

5. Changer la complexité du jeu :

Rapidement, vous allez devenir des experts en Mastermind. Vous allez alors vouloir augmenter la difficulté. Il suffira alors d'allonger la longueur de la combinaison, ou d'augmenter le nombre de couleurs possibles. Cela vous est d'ores et déjà très facile si vous avez pensé à définir une constante globale pour chacune de ces deux grandeurs. Si ce n'est pas le cas, il est grand temps de le faire. Définissez par exemple :

```
const int nbcases = 5;    // longueur de la combinaison
const int nbcouleur = 4;  // nombre de couleurs différentes
```

Reprenez le code que vous avez déjà écrit en utilisant ces constantes. Il est très important de stocker les paramètres constants dans des variables, cela fait gagner beaucoup de temps lorsque l'on veut les modifier.

6. Saisie d'une combinaison au clavier :

La fonction suivante, que nous vous demanderons d'admettre, saisit une *chaîne de caractères* (string) au clavier et remplit le tableau `combi[]` avec les chiffres que les `nbcases` premiers caractères de la chaîne représentent.

```
void getCombinaison(int combi[nbcases])
{
    cout << "Votre_essai:_";
    string s;
    cin >> s;
    for (int i=0; i<nbcases; i++)
        combi[i] = s[i] - '0';
}
```

Attention, le `cin` ne fonctionne pas depuis le terminal intégré de QtCreator (d'ailleurs la fenêtre s'appelle *Application Output*, ne supporte pas l'Input), il faut lui dire

d'utiliser un terminal externe. Aller dans l'onglet Projects, rubrique Run et cliquer le bouton "Run in Terminal".

Dans le cadre de notre Mastermind, il s'agit de modifier cette fonction pour qu'elle contrôle que la chaîne rentrée est bien de bonne taille et que les chiffres sont bien entre 0 et `nbcouleur-1`. L'essai devra être redemandé jusqu'à ce que la combinaison soit valide. On utilisera entre autres la fonction `s.size()` qui retourne la taille de la chaîne `s` (la syntaxe de cette fonction sera comprise plus tard dans le cours...)

Une petite explication sur `s[i] - '0'`. Les caractères (type `char`) sont en fait une valeur numérique, le code ASCII. '0' donne le code ASCII du caractère zéro. On peut le vérifier en faisant par exemple

```
cout << (int)'0' << endl;
```

(Il faut convertir en entier, sinon on lui demande d'afficher un `char`, et donc il affiche le caractère associé au code ASCII, 0!) Cette valeur est 48. Il se trouve que les chiffres de 0 à 9 ont des codes ASCII consécutifs : '0'=48, '1'=49, '9'=57. Le type `string` de la variable `s` se comporte comme un tableau de `char`. Ainsi `s[i] - '0'` vaut 0 si `s[i]` est le caractère 0, '1'-'0'=1 si c'est le caractère 1, etc.

7. Traitement d'une combinaison :

Il faudrait maintenant programmer une fonction comparant une combinaison donnée avec la combinaison à trouver. Cette fonction devrait renvoyer deux valeurs : le nombre de pions de la bonne valeur bien placés, puis, dans les pions restant, le nombre de pions de la bonne valeur mais mal placés.

Par exemple, si la combinaison à trouver est 02113 :

00000 : 1 pion bien placé (0xxxx), 0 pion mal placé (xxxxx)
 20000 : 0 pion bien placé (xxxxx), 2 pions mal placés (20xxx)
 13133 : 2 pions bien placés (xx1x3), 1 pion mal placé (1xxxx)
 13113 : 3 pions bien placés (xx113), 0 pion mal placé (xxxxx)
 12113 : 4 pions bien placés (x2113), 0 pion mal placé (xxxxx)

...

Pour commencer et pouvoir tout de suite tester le jeu, programmer une fonction renvoyant uniquement le nombre de pions bien placés.

8. *Boucle de jeu* : Nous avons maintenant à notre disposition toutes les briques nécessaires³, il n'y a plus qu'à les assembler pour former un jeu de mastermind. Pensez par ailleurs à ajouter la détection de la victoire (quand tous les pions sont bien placés), et celle de la défaite (quand un nombre limite d'essais a été dépassé).
9. *Version complète* : Compléter la fonction de traitement d'une combinaison pour qu'elle renvoie également le nombre de pions mal placés.

A.3.2 Mastermind Graphique

Le jeu de Mastermind que nous venons de réaliser reste malgré tout très peu convivial. Nous allons y remédier en y ajoutant une interface graphique.

3. même si la fonction de traitement d'une combinaison est pour l'instant incomplète

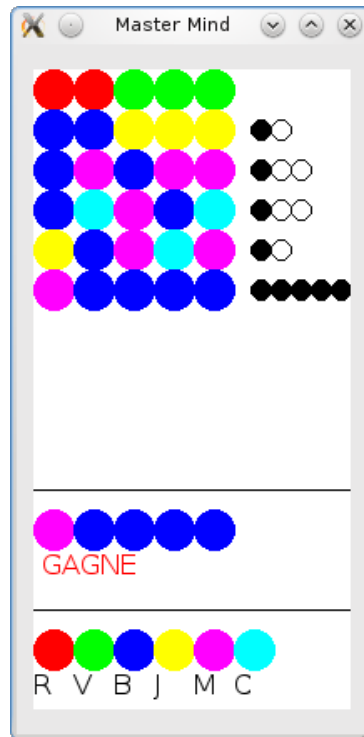


FIGURE A.4 – Master mind graphique...

1. *Etude du projet de départ :*

Passer dans le projet `MastermindGraphique`⁴

Les fonctions graphiques sont déjà définies. Elles fonctionnent selon un principe de division de la fenêtre graphique en lignes. La fonction :

```
void afficheCombinaison(int combi[nbcases], int n);
```

permet d'afficher la combinaison *combi* sur la ligne *n*. Au début du programme, on laisse en haut de la fenêtre graphique autant de lignes libres que le joueur a d'essais pour y afficher le déroulement du jeu. On affiche en bas de la fenêtre graphique un mini mode d'emploi qui résume les correspondances entre touches et couleurs.

2. *Mastermind graphique :*

Réinsérer dans ce projet les fonctions de génération aléatoire d'une combinaison et de comparaison de deux comparaisons écrites précédemment. Puis reprogrammer la boucle principale du jeu en utilisant l'affichage graphique.

3. *Ultime amélioration :*

On souhaite pouvoir effacer une couleur après l'avoir tapée, au cas où l'on se serait trompé. Etudier les fonctions

```
int Clavier();
```

4. Sous Visual, penser à le définir comme projet de démarrage pour que ce soit lui qui se lance à l'exécution (son nom doit apparaître en gras dans la liste des projets). Sous QtCreator, sélectionnez-le

avec l'icône représentant un écran sur la gauche de la fenêtre.



et

```
void getCombinaison(int [], int);
```

La première prend déjà en compte la touche Retour arrière, mais pas la seconde qui la considère comme une erreur de frappe. Modifier cette dernière en conséquence.

A.4 Structures

Avertissement : Dans ce TP, nous allons faire évoluer des corps soumis à la gravitation, puis leur faire subir des chocs élastiques. Il s'agit d'un long TP qui nous occupera plusieurs séances. En fait, le TP suivant sera une réorganisation de celui-ci. Les sections 11 et 15 ne sont données que pour les élèves les plus à l'aise et ne seront abordées qu'en deuxième semaine. En section A.4.2 sont décrites quelques-unes des fonctions à utiliser, et en A.4.3 leur justification physique.

A.4.1 Etapes

Mouvement de translation

1. *Pour commencer, étudier le projet* :
Télécharger le fichier `TP4.zip` sur la page habituelle, le décompresser et lancer votre environnement de développement. Parcourir le projet, en s'attardant sur les variables globales et la fonction `main` (inutile de regarder le contenu des fonctions déjà définies mais non utilisées). Le programme fait évoluer un point (x, y) selon un mouvement de translation constante (v_x, v_y) , et affiche régulièrement un disque centré en ce point. Pour ce faire, afin de l'effacer, on retient la position du disque au dernier affichage (dans `ox` et `oy`) ; par ailleurs, deux instructions commençant par `noRefresh` sont placées autour des instructions graphiques afin d'accélérer l'affichage.
2. *Utiliser une structure* :
Modifier le programme de façon à utiliser une structure `Balle` renfermant toute l'information sur le disque (position, vitesse, rayon, couleur).
3. *Fonctions d'affichage* :
Créer (et utiliser) une fonction **void** `AfficheBalle(Balle D)` affichant le disque `D`, et une autre **void** `EffaceBalle(Balle D)` l'effaçant.
4. *Faire bouger proprement le disque* :
Pour faire évoluer la position du disque, remplacer les instructions correspondantes déjà présentes dans `main` par un appel à une fonction qui modifie les coordonnées d'une `Balle`, en leur ajoutant la vitesse de la `Balle` multipliée par un certain pas de temps défini en variable globale ($dt = 1$ pour l'instant).

Gravitation

5. *Évolution par accélération* :
Créer (et utiliser) une fonction qui modifie la vitesse d'une `Balle` de façon à lui faire subir une attraction constante $a_x = 0$ et $a_y = 0.0005$. Indice : procéder comme précédemment, c'est-à-dire ajouter $0.0005 * dt$ à v_y ...
6. *Ajouter un soleil* :
On souhaite ne plus avoir une gravité uniforme. Ajouter un champ décrivant la masse à la structure `Balle`. Créer un soleil (de type `Balle`), jaune, fixe (ie de vitesse nulle) au milieu de la fenêtre, de masse 10 et de rayon 4 pixels (la masse de la planète qui bouge étant de 1). L'afficher.

7. *Accélération gravitationnelle :*

Créer (et utiliser à la place de la gravitation uniforme) une fonction qui prend en argument la planète et le soleil, et qui fait évoluer la position de la planète. Rappel de physique : l'accélération à prendre en compte est $-G m_S / r^3 \vec{r}$, avec ici $G = 1$ (Vous aurez sans doute besoin de la fonction **double** sqrt(**double** x), qui retourne la racine carrée de x). Ne pas oublier le facteur dt ... Faire tourner et observer. Essayez diverses initialisations de la planète (par exemple $x = \text{largeur}/2$, $y = \text{hauteur}/3$, $v_x = 1$, $v_y = 0$). Notez que l'expression de l'accélération devient très grande lorsque r s'approche de 0 ; on prendra donc garde à ne pas utiliser ce terme lorsque r devient trop petit.

8. *Initialisation aléatoire :*

Créer (et utiliser à la place des conditions initiales données pour le soleil) une fonction initialisant une `Balle`, sa position étant dans la fenêtre, sa vitesse nulle, son rayon entre 5 et 15, et sa masse valant le rayon divisé par 20. Vous aurez probablement besoin de la fonction `Random`...

9. *Des soleils par milliers...*

Placer 10 soleils aléatoirement (et en tenir compte à l'affichage, dans le calcul du déplacement de l'astéroïde...).

10. *Diminuer le pas de temps de calcul :*

Afin d'éviter les erreurs dues à la discrétisation du temps, diminuer le pas de temps dt , pour le fixer à 0.01 (voire à 0.001 si la machine est assez puissante). Régler la fréquence d'affichage en conséquence (inversement proportionnelle à dt). Lancer plusieurs fois le programme.

Chocs élastiques simples11. *Faire rebondir l'astéroïde :*

Faire subir des chocs élastiques à l'astéroïde à chaque fois qu'il s'approche trop d'un soleil, de façon à ce qu'il ne rentre plus dedans (fonction `ChocSimple`), et rétablir dt à une valeur plus élevée, par exemple 0.1 (modifier la fréquence d'affichage en conséquent). Pour savoir si deux corps sont sur le point d'entrer en collision, utiliser la fonction `Collision`.

Jeu de tir

(figure A.5 droite)

12. *Ouvrir un nouveau projet :*

Afin de partir dans deux voies différentes et travailler proprement, nous allons ajouter un nouveau projet `Imagine++`, appelé `Duel`, dans cette même solution. Recopier (par exemple par copier/coller) intégralement le contenu du fichier `Tp4.cpp` dans un fichier `Duel.cpp`. Une fois cette copie faite, modifier le fichier `CMakeLists.txt` en ajoutant deux lignes indiquant que l'exécutable `Duel` dépend de `Duel.cpp` et utilise la bibliothèque `Graphics` d'`Imagine++`.

13. *À vous de jouer !*

Transformer le projet `Duel`, à l'aide des fonctions qui y sont déjà présentes, en un jeu de tir, à deux joueurs. Chacun des deux joueurs a une position fixée, et divers soleils sont placés aléatoirement dans l'écran. Chaque joueur, à tour de rôle, peut

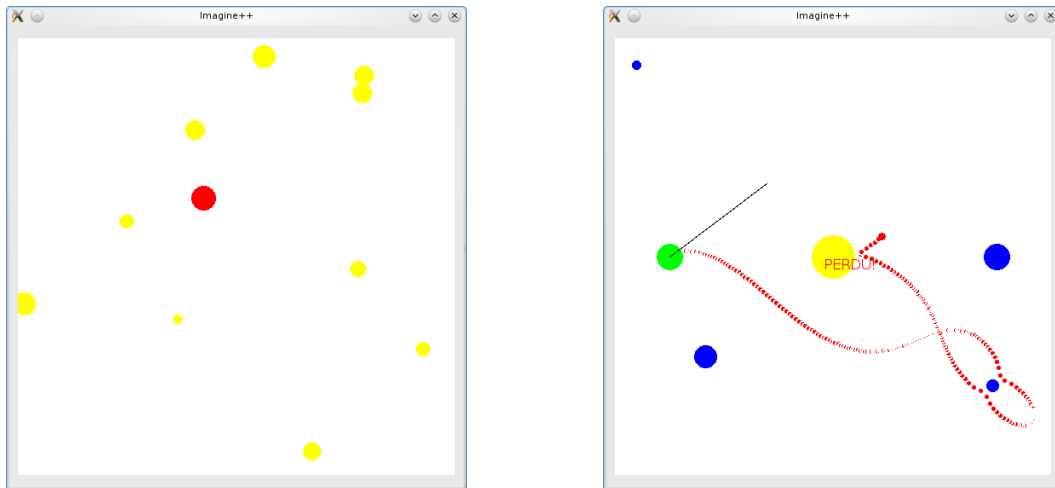


FIGURE A.5 – Corps célestes et jeu de tir...

lancer une `Balle` avec la vitesse initiale de son choix, la balle subissant les effets de gravitation des divers soleils, et disparaissant au bout de 250 pas de temps d’affichage. Le gagnant est le premier qui réussit à atteindre l’autre... Conseils pratiques : positionner symétriquement les joueurs par rapport au centre, de préférence à mi-hauteur en laissant une marge d’un huitième de la largeur sur le côté ; utiliser la fonction `GetMouse` pour connaître la position de la souris ; en déduire la vitesse désirée par le joueur en retranchant à ces coordonnées celles du centre de la boule à lancer, et en multipliant par un facteur 0.00025.

14. *Améliorations :*

Faire en sorte qu’il y ait systématiquement un gros soleil au centre de l’écran (de masse non nécessairement conséquente) afin d’empêcher les tirs directs.

15. *Initialisation correcte :*

Modifier la fonction de placement des soleils de façon à ce que les soleils ne s’intersectent pas initialement, et qu’ils soient à une distance minimale de 100 pixels des emplacements des joueurs.

Chocs élastiques

(figure A.5 gauche)

16. *Tout faire évoluer, tout faire rebondir :*

On retourne dans le projet `Gravitation`. Tout faire bouger, y compris les soleils. Utiliser, pour les chocs élastiques, la fonction `Chocs` (qui fait rebondir les deux corps). Faire en sorte que lors de l’initialisation les soleils ne s’intersectent pas.

A.4.2 Aide

Fonctions fournies :

void `InitRandom()` ;

est à exécuter une fois avant le premier appel à `Random`.

double Random(**double** a, **double** b);

renvoie un double aléatoirement entre a et b (compris). Exécuter *une fois* InitRandom(); avant la première utilisation de cette fonction.

void ChocSimple(**double** x, **double** y, **double** &vx, **double** &vy, **double** m, **double** x2, **double** y2, **double** vx2, **double** vy2);

fait rebondir la première particule, de coordonnées (x, y), de vitesse (vx, vy) et de masse m, sur la deuxième, de coordonnées (x2, y2) et de vitesse (vx2, vy2), sans déplacer la deuxième.

void Choc(**double** x, **double** y, **double** &vx, **double** &vy, **double** m, **double** x2, **double** y2, **double** &vx2, **double** &vy2, **double** m2);

fait rebondir les deux particules l'une contre l'autre.

bool Collision(**double** x1, **double** y1, **double** vx1, **double** vy1, **double** r1, **double** x2, **double** y2, **double** vx2, **double** vy2, **double** r2);

renvoie **true** si le corps de coordonnées (x1, y1), de vitesse (vx1, vy1) et de rayon r1 est sur le point d'entrer en collision avec le corps de coordonnées (x2, y2), de vitesse (vx2, vy2) et de rayon r2, et **false** sinon.

A.4.3 Théorie physique

NB : Cette section n'est donnée que pour expliquer le contenu des fonctions pré-programmées fournies avec l'énoncé. Elle peut être ignorée en première lecture.

Accélération

La somme des forces exercées sur un corps A est égale au produit de sa masse par l'accélération de son centre de gravité.

$$\sum_i \vec{F}_{i/A} = m_A \vec{a}_{G(A)}$$

Gravitation universelle

Soient deux corps A et B. Alors A subit une force d'attraction

$$\vec{F}_{B/A} = -Gm_A m_B \frac{1}{d_{A,B}^2} \vec{u}_{B \rightarrow A}.$$

Chocs élastiques

Soient A et B deux particules rentrant en collision. Connaissant tous les paramètres avant le choc, comment déterminer leur valeur après ? En fait, seule la vitesse des particules reste à calculer, puisque dans l'instant du choc, les positions ne changent pas.

Durant un choc dit *élastique*, trois quantités sont conservées :

1. la quantité de mouvement $\vec{P} = m_A \vec{v}_A + m_B \vec{v}_B$
2. le moment cinétique $M = m_A \vec{r}_A \times \vec{v}_A + m_B \vec{r}_B \times \vec{v}_B$ (qui est un réel dans le cas d'un mouvement plan).

3. l'énergie cinétique $E_c = \frac{1}{2}m_A v_A^2 + \frac{1}{2}m_B v_B^2$.

Ce qui fait 4 équations pour 4 inconnues.

Résolution du choc

On se place dans le référentiel du centre de masse. On a alors, à tout instant :

1. $\vec{P} = 0$ (par définition de ce référentiel), d'où $m_A \vec{v}_A = -m_B \vec{v}_B$.
2. $M = (\vec{r}_A - \vec{r}_B) \times m_A \vec{v}_A$, d'où, en notant $\Delta \vec{r} = \vec{r}_A - \vec{r}_B$, $M = \Delta \vec{r} \times m_A \vec{v}_A$.
3. $2E_c = m_A(1 + \frac{m_A}{m_B})v_A^2$.

La constance de E_c nous informe que dans ce repère, la norme des vitesses est conservée, et la constance du moment cinétique que les vitesses varient parallèlement à $\Delta \vec{r}$. Si l'on veut que les vitesses varient effectivement, il ne nous reste plus qu'une possibilité : multiplier par -1 la composante des $\vec{v}_i$ selon $\Delta \vec{r}$. Ce qui fournit un algorithme simple de rebond.

Décider de l'imminence d'un choc

On ne peut pas se contenter, lors de l'évolution pas à pas des coordonnées des disques, de décider qu'un choc aura lieu entre t et $t + dt$ rien qu'en estimant la distance entre les deux disques candidats à la collision à l'instant t , ni même en prenant en plus en considération cette distance à l'instant $t + dt$, car, si la vitesse est trop élevée, un disque peut déjà avoir traversé l'autre et en être ressorti en $t + dt$... La solution consiste à expliciter le minimum de la distance entre les disques en fonction du temps, variant entre t et $t + dt$.

Soit $N(u) = (\vec{r}_A(u) - \vec{r}_B(u))^2$ le carré de la distance en question. On a :

$$N(u) = (\vec{r}_A(t) - \vec{r}_B(t) + (u - t)(\vec{v}_A(t) - \vec{v}_B(t)))^2$$

Ce qui donne, avec des notations supplémentaires :

$$N(u) = \Delta \vec{r}(t)^2 + 2(u - t)\Delta \vec{r}(t) \cdot \Delta \vec{v}(t) + (u - t)^2 \Delta \vec{v}(t)^2$$

La norme, toujours positive, est minimale au point u tel que $\partial_u N(u) = 0$, soit :

$$(t_m - t) = -\frac{\Delta \vec{r}(t) \cdot \Delta \vec{v}(t)}{\Delta \vec{v}(t)^2}$$

Donc :

1. si $t_m < t$, le minimum est atteint en t ,
2. si $t < t_m < t + dt$, le minimum est atteint en t_m ;
3. sinon, $t + dt < t_m$, le minimum est atteint en $t + dt$.

Ce qui nous donne explicitement et simplement la plus petite distance atteinte entre les deux corps entre t et $t + dt$.

A.5 Fichiers séparés

Nous allons poursuivre dans ce TP les simulations de gravitation et de chocs élastiques entamées la semaine dernière, en séparant dans différents fichiers les différentes fonctions et structures utilisées.

1. *De bonnes bases :*

Reprenez le travail en cours de la semaine dernière. Si vous avez été au bout de celui-ci ou au moins jusqu'à la question 7 incluse, faites-en une sauvegarde séparée pour pouvoir vous y reporter si les travaux entrepris dans ce TP ont tendance à tout casser.

A.5.1 Fonctions outils

2. *Un fichier de définitions...*

Ajouter un nouveau fichier source nommé `Tools.cpp` au projet. Y placer les fonctions fournies à l'avance au début du TP4 (`InitRandom`, `Random`, `Choc`, `ChocSimple` et `Collision`), en les retirant de `Gravitation.cpp`. Ne pas oublier les lignes suivantes, que l'on pourra retirer de `Gravitation.cpp` :

```
#include <cstdlib>
#include <ctime>
using namespace std;
```

Attention de ne pas placer le nouveau fichier dans le "build directory" de CMake. Vous devez bien placer ce fichier dans le dossier des sources (le même que `Gravitation.cpp`). N'oubliez pas de modifier les arguments de `add_executable` du `CMakeLists.txt` pour y ajouter `Tools.cpp`. Il faut alors relancer Cmake. Sous Visual Studio, il devrait constater le changement du `CMakeLists.txt` et vous proposer de recharger le projet. Acceptez sa proposition, il relance Cmake en coulisses. Sous QtCreator, relancez Cmake (depuis QtCreator).

3. *...et un fichier de déclarations*

Ajouter un nouveau fichier d'en-tête nommé `Tools.h`. Inclure la protection contre la double inclusion vue en cours (**#pragma once**). Y placer les déclarations des fonctions mises dans `Tools.cpp`, ainsi que la définition de `dt`, en retirant celle-ci de `main`. Rajouter au début de `Tools.cpp` et de `Gravitation.cpp` un

```
#include "Tools.h"
```

A.5.2 Vecteurs

4. *Structure Vector :*

Créer dans un nouveau fichier `Vector.h` une structure représentant un vecteur du plan, avec deux membres de type **double**. Ne pas oublier le mécanisme de protection contre la double inclusion. Déclarer (et non définir) les opérateurs et fonction suivants :

```
Vector operator+(Vector a, Vector b);           // Somme
Vector operator-(Vector a, Vector b);           // Différence
```



```
double norme2(Vector a); // Norme euclidienne
Vector operator*(Vector a, double lambda); // Mult. scalaire
Vector operator*(double lambda, Vector a); // Mult. scalaire
```

5. *Fonctions et opérateurs sur les Vector :*

Créer un nouveau fichier `Vector.cpp`. Mettre un **#include** du fichier d'en-tête correspondant et définir les opérateurs qui y sont déclarés (Rappel : `sqrt` est défini dans le fichier d'en-tête système `<cmath>`; ne pas oublier non plus le **using namespace std**; qui permet d'utiliser cette fonction). Astuce : une fois qu'une version de **operator*** est définie, la deuxième version peut utiliser la première dans sa définition...

6. *Vecteur vitesse et vecteur position :*

Systématiquement remplacer dans `Gravitation.cpp` les vitesses et positions par des objets de type `Vector` (y compris dans la définition de la structure `Balle`). Utiliser autant que possible les opérateurs et fonction définis dans `Vector.cpp`.

A.5.3 Balle à part

7. *Structure Balle :*

Déplacer la structure `Balle` dans un nouveau fichier d'en-tête `Balle.h`. Puisque `Balle` utilise les types `Vector` et `Color`, il faut aussi ajouter ces lignes :

```
#include <Image/Graphics.h>
using namespace Image;

#include "Vector.h"
```

8. *Fonctions associées :*

Déplacer toutes les fonctions annexes prenant des `Balle` en paramètres dans un nouveau fichier `Balle.cpp`. Il ne devrait plus rester dans `Gravitation.cpp` d'autre fonction que `main`. Déclarer dans `Balle.h` les fonctions définies dans `Balle.cpp`. Ajouter les **#include** nécessaires dans ce dernier fichier et dans `Gravitation.cpp` et faire les adaptations nécessaires (par exemple, si des fonctions utilisent `largeur` ou `hauteur`, comme ces constantes ne sont définies que dans `Gravitation.cpp`, il faut les passer en argument...)

A.5.4 Retour à la physique

9. *Des soleils par milliers... :*

Placer 10 soleils aléatoirement (et en tenir compte à l'affichage, dans le calcul du déplacement de l'astéroïde...).

10. *Diminuer le pas de temps de calcul :*

Afin d'éviter les erreurs dues à la discrétisation du temps, diminuer le pas de temps `dt`, pour le fixer à 0.01 (voire à 0.001 si la machine est assez puissante). Régler la fréquence d'affichage en conséquence (inversement proportionnelle à `dt`). Lancer plusieurs fois le programme.

Chocs élastiques simples

11. *Faire rebondir l'astéroïde :*

Faire subir des chocs élastiques à l'astéroïde à chaque fois qu'il s'approche trop d'un soleil, de façon à ce qu'il ne rentre plus dedans (fonction `ChocSimple`), et rétablir dt à une valeur plus élevée, par exemple 0.1 (modifier la fréquence d'affichage en conséquent). Pour savoir si deux corps sont sur le point d'entrer en collision, utiliser la fonction `Collision`.

Jeu de tir

12. *Ouvrir un nouveau projet :*

Afin de partir dans deux voies différentes et travailler proprement, ajouter un nouveau projet appelé `Duel`, dans cette même solution. On ajoute un dossier `Duel` au même niveau que `Gravitation` et on modifie le `CMakeLists.txt`.

13. *Ne pas refaire deux fois le travail :*

Comme nous aurons besoins des mêmes fonctions dans ce projet que dans le projet `Gravitation`, ajouter au projet (sans en créer de nouveaux !) les fichiers `Vector.h`, `Vector.cpp`, `Balle.h`, `Balle.cpp`, `Tools.h`, `Tools.cpp`. Les fichiers sont les *mêmes* que dans le projet `Gravitation`, ils ne sont pas recopiés. Mettre au début de `Duel.cpp` (fichier à placer dans le répertoire `Duel`) les **#include** correspondants. Essayer de compiler `Duel.cpp`. Comme le compilateur n'arrive pas à trouver les fichiers inclus, qui ne sont pas dans le même répertoire, il faut lui indiquer où les trouver :

```
#include ".. / Gravitation / Tools.h"
```

Pour le `CMakeLists.txt` du répertoire `Duel`, inspirez-vous de celui de `Gravitation`.

14. *À vous de jouer !*

Transformer le projet `Duel`, à l'aide des fonctions définies auparavant, en un jeu de tir, à deux joueurs. Chacun des deux joueurs a une position fixée, et divers soleils sont placés aléatoirement dans l'écran. Chaque joueur, à tour de rôle, peut lancer une `Balle` avec la vitesse initiale de son choix, la balle subissant les effets de gravitation des divers soleils, et disparaissant au bout de 250 pas de temps d'affichage. Le gagnant est le premier qui réussit à atteindre l'autre... Conseils pratiques : positionner symétriquement les joueurs par rapport au centre, de préférence à mi-hauteur en laissant une marge d'un huitième de la largeur sur le côté ; utiliser la fonction `GetMouse` pour connaître la position de la souris ; en déduire la vitesse désirée par le joueur en retranchant à ces coordonnées celles du centre de la boule à lancer, et en multipliant par un facteur 0.00025.

15. *Améliorations :*

Faire en sorte qu'il y ait systématiquement un gros soleil au centre de l'écran (de masse non nécessairement conséquente) afin d'empêcher les tirs directs.

16. *Initialisation correcte :*

Modifier la fonction de placement des soleils de façon à ce que les soleils ne s'intersectent pas initialement, et qu'ils soient à une distance minimale de 100 pixels des emplacements des joueurs.

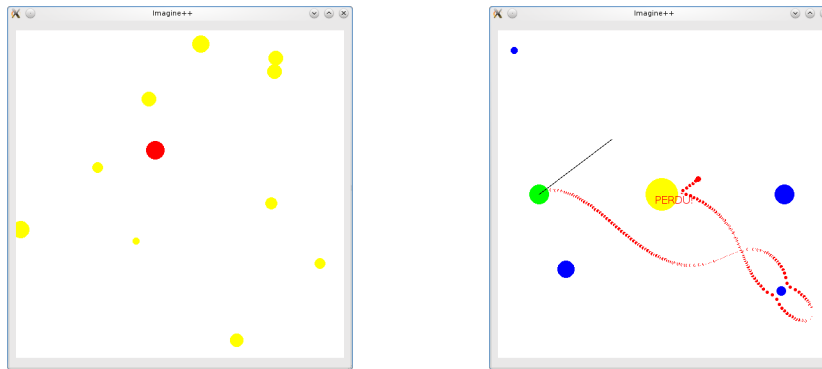


FIGURE A.6 – Corps célestes et jeu de tir...

Chocs élastiques

17. *Tout faire évoluer, tout faire rebondir :*

On retourne dans le projet *Gravitation*. Tout faire bouger, y compris les soleils. Utiliser, pour les chocs élastiques, la fonction `Chocs` (qui fait rebondir les deux corps). Faire en sorte que lors de l'initialisation les soleils ne s'intersectent pas.

A.6 Images



FIGURE A.7 – Deux images et différents traitements de la deuxième (négatif, flou, relief, déformation, contraste et contours).

Dans ce TP, nous allons jouer avec les tableaux bidimensionnels statiques (mais stockés dans des tableaux 1D) puis dynamiques. Pour changer de nos passionnantes matrices, nous travaillerons avec des images (figure A.7).

A.6.1 Allocation

1. *Récupérer le projet :*
Télécharger le fichier `Tp7_Initial.zip` sur la page habituelle, le décompresser et lancer Visual C++.
2. *Saturer la mémoire :*
Rien à voir avec ce qu'on va faire après mais il faut l'avoir fait une fois... Faire, dans une boucle infinie, des allocations de 1000000 entiers sans désallouer et regarder la taille du process grandir. (Utiliser `Ctrl+Shift+Echap` pour accéder au gestionnaire de tâches). Compiler en mode Release pour utiliser la "vraie" gestion du tas (Le mode Debug utilise une gestion spécifique qui aide à trouver les bugs et se comporte différemment...)

A.6.2 Tableaux statiques

3. *Niveaux de gris :*
Une image noir et blanc est représentée par un tableau de pixels de dimensions constantes $W=300$ et $H=200$. Chaque pixel (i, j) est un `byte` (entier de 0 à 255) allant de 0 pour le noir à 255 pour le blanc. L'origine est en haut à gauche, i est

l'horizontale et j la verticale. Dans un tableau de `byte` mono-dimensionnel t de taille $W \times H$ mémorisant le pixel (i, j) en $t[i + W \times j]$:

- Stocker une image noire et l'afficher avec `putGreyImage(0, 0, t, W, H)`.
- Idem avec une image blanche.
- Idem avec un dégradé du noir au blanc (attention aux conversions entre `byte` et **double**).
- Idem avec $t(i, j) = 128 + 128 \sin(4\pi i/W) \sin(4\pi j/H)$ (cf figure A.7). Utiliser

```
#define _USE_MATH_DEFINES
#include <cmath>
```

pour avoir les fonctions et les constantes mathématiques : `M_PI` vaut π .

4. Couleurs :

Afficher, avec `putColorImage(0, 0, r, g, b, W, H)`, une image en couleur stockée dans trois tableaux r, g et b (rouge, vert, bleu). Utiliser la fonction `click()` pour attendre que l'utilisateur clique avec la souris entre l'affichage précédent et ce nouvel affichage.

A.6.3 Tableaux dynamiques

5. Dimensions au clavier :

Modifier le programme précédent pour que W et H ne soient plus des constantes mais des valeurs entrées au clavier. Ne pas oublier de désallouer.

A.6.4 Charger un fichier

6. Image couleur :

La fonction `loadColorImage(srcPath("ppd.jpg"), r, g, b, W, H)` ; charge le fichier "ppd.jpg" qui est dans le répertoire contenant les sources (`srcPath`), alloue elle-même les tableaux r, g, b , les remplit avec les pixels de l'image, et affecte aussi W et H en conséquence. Attention : ne pas oublier de désallouer les tableaux r, g, b avec `delete[]` après usage.

- Charger cette image et l'afficher. Ne pas oublier les désallocations.

7. Image noir et blanc :

La fonction `loadGreyImage(srcPath("ppd.jpg"), t, W, H)` fait la même chose mais convertit l'image en noir et blanc. Afficher l'image en noir et blanc...

A.6.5 Fonctions

8. Découper le travail :

On ne garde plus que la partie noir et blanc du programme. Faire des fonctions pour allouer, détruire, afficher et charger les images :

```
byte* AlloueImage(int W, int H);
void DetruitImage(byte *I);
void AfficheImage(byte* I, int W, int H);
byte* ChargeImage(char* name, int &W, int &H);
```

9. *Fichiers :*

Créer un `image.cpp` et un `image.h` en conséquence...

A.6.6 Structure10. *Principe :*

Modifier le programme précédent pour utiliser une structure :

```
struct Image {
    byte* t;
    int w,h;
};
```

`AlloueImage()` et `ChargeImage()` pourront retourner des `Image`.

11. *Indépendance :*

Pour ne plus avoir à savoir comment les pixels sont stockés, rajouter :

```
byte Get(Image I, int i, int j);
void Set(Image I, int i, int j, byte g);
```

12. *Traitements :*

Ajouter dans `main.cpp` différentes fonctions de modification des images

```
Image Negatif(Image I);
Image Flou(Image I);
Image Relief(Image I);
Image Contours(Image I, double seuil);
Image Deforme(Image I);
```

et les utiliser :

- (a) `Negatif` : changer le noir en blanc et vice-versa par une transformation affine.
- (b) `Flou` : chaque pixel devient la moyenne de lui-même et de ses 8 voisins. Attention aux pixels du bords qui n'ont pas tous leurs voisins (on pourra ne pas moyenner ceux-là et en profiter pour utiliser l'instruction `continue`!).
- (c) `Relief` : la dérivée suivant une diagonale donne une impression d'ombres projetées par une lumière rasante.
 - Approcher cette dérivée par différence finie : elle est proportionnelle à $I(i+1, j+1) - I(i-1, j-1)$.
 - S'arranger pour en faire une image allant de 0 à 255.
- (d) `Contours` : calculer par différences finies la dérivée horizontale $d_x = (I(i+1, j) - I(i-1, j))/2$ et la dérivée verticale d_y , puis la norme du gradient $|\nabla I| = \sqrt{d_x^2 + d_y^2}$ et afficher en blanc les points où cette norme est supérieure à un seuil.
- (e) `Deforme` : Construire une nouvelle image sur le principe $J(i, j) = I(f(i, j))$ avec f bien choisie. On pourra utiliser un sinus pour aller de 0 à $W-1$ et de 0 à $H-1$ de façon non linéaire.

A.6.7 Suite et fin

13. S'il reste du temps, s'amuser :

- Rétrécir une image.
- Au lieu du négatif, on peut par exemple changer le contraste. Comment ?

A.7 Premiers objets et dessins de fractales

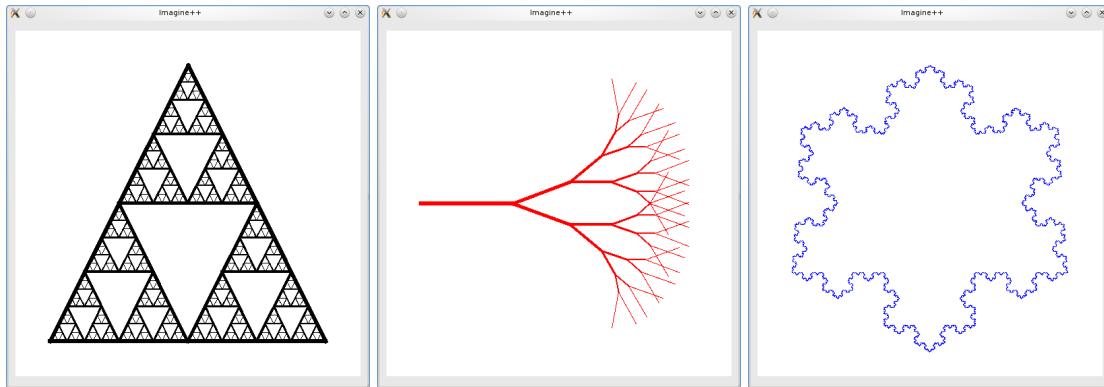


FIGURE A.8 – Fractales...

Dans ce TP, nous allons nous essayer à la programmation objet. Nous allons transformer une structure vecteur en une classe et l'utiliser pour dessiner des courbes fractales (figure A.8).

A.7.1 Le triangle de Sierpinski

1. *Récupérer le projet :*

Télécharger le fichier `Tp7_Initial.zip` sur la page habituelle, le décompresser et lancer votre IDE. Etudier la structure `Vector` définie dans les fichiers `Vector.cpp` et `Vector.h`.

2. *Interfaçage avec Imagine++ :*

La structure `Vector` ne comporte pas de fonction d'affichage graphique. Ajouter dans `main.cpp` des fonctions `drawLine` et `drawTriangle` prenant des `Vector` en paramètres. Il suffit de rebondir sur la fonction

```
void drawLine(int x1,int y1,int x2,int y2,const Color& c,int pen_w)
```

d'Imagine++. Le dernier paramètre contrôle l'épaisseur du trait.

3. *Triangle de Sierpinski :*

C'est la figure fractale choisie par l'ENPC pour son logo. La figure ci-dessous illustre sa construction.

Écrire une fonction récursive pour dessiner le triangle de Sierpinski. Cette fonc-

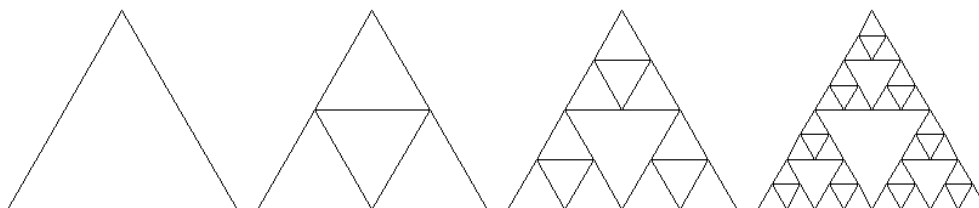


FIGURE A.9 – Construction du triangle de Sierpinski.

tion prendra en paramètres les trois points du triangle en cours et l'épaisseur du trait. Les trois sous-triangles seront dessinés avec un trait plus fin. **Ne pas oublier la condition d'arrêt de la récursion !**

Utiliser cette fonction dans le main en lui fournissant un triangle initial d'épaisseur 6.

A.7.2 Une classe plutôt qu'une structure

4. *Classe vecteur :*

Transformer la structure `Vector` en une classe. Y incorporer toutes les fonctions et les opérateurs. Passer en public le strict nécessaire. Faire les modifications nécessaires dans `main.cpp`.

5. *Accesseurs pour les membres :*

Rajouter des accesseurs en lecture et en écriture pour les membres, et les utiliser systématiquement dans le programme principal. L'idée est de cacher aux utilisateurs de la classe `Vector` les détails de son implémentation.

6. *Dessin récursif d'un arbre :*

Nous allons maintenant dessiner un arbre. Pour cela il faut partir d'un tronc et remplacer la deuxième moitié de chaque branche par deux branches de même longueur formant un angle de 20 degrés avec la branche mère. La figure ci-dessous illustre le résultat obtenu pour différentes profondeurs de récursion.

Écrire une fonction récursive pour dessiner une telle courbe. Vous aurez besoin

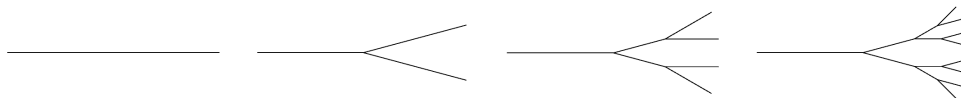


FIGURE A.10 – Construction de l'arbre.

de la fonction `Rotate` de la classe `Vector`.

A.7.3 Changer d'implémentation

7. *Deuxième implémentation :*

Modifier l'implémentation de la classe `Vector` en remplaçant les membres **double** `x,y`; par un tableau **double** `coord[2]`; Quelles sont les modifications à apporter dans `main.cpp`?

8. *Vecteurs de dimension supérieure :*

L'avantage de cette dernière implémentation est qu'elle se généralise aisément à des vecteurs de dimension supérieure. Placer une constante globale `DIM` égale à 2 au début de `Vector.h` et rendre la classe `Vector` indépendante de la dimension.

NB : la fonction `Rotate` et les accesseurs que nous avons écrits ne se généralisent pas directement aux dimensions supérieures. Les laisser tels quels pour l'instant...

A.7.4 Le flocon de neige

9. *Courbe de Koch :*

Cette courbe fractale s'obtient en partant d'un segment et en remplaçant le deuxième tiers de chaque segment par deux segments formant la pointe d'un triangle équilatéral.

Écrire une fonction récursive pour dessiner une courbe de Koch.



FIGURE A.11 – Construction de la courbe de Koch.

10. *Flocon de neige :*

Il s'obtient en construisant une courbe de Koch à partir de chacun des côtés d'un triangle équilatéral.

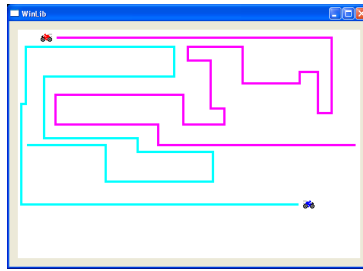


FIGURE A.12 – Jeu de Tron.

A.8 Tron

Dans ce TP, nous allons programmer le jeu TRON. Il s'agit d'un jeu à 2 joueurs, dans lequel chaque joueur pilote un mobile qui se déplace à vitesse constante et laisse derrière lui une trace infranchissable. Le premier joueur qui percute sa propre trace ou celle de son adversaire a perdu. Ce TP est assez ambitieux et s'approche d'un mini-projet. Il nous occupera plusieurs séances.

A.8.1 Serpent

Nous allons procéder en deux temps. D'abord programmer un jeu de Serpent à un joueur. Le programme `serpent.exe` vous donne une idée du résultat recherché. Dans ce jeu, le joueur pilote un Serpent qui s'allonge petit à petit (d'un élément tous les x tours, avec la convention que la longueur totale est bornée à $n_{\max}$ éléments). Il s'agit de ne pas se rentrer dedans ni de percuter les murs.

La solution de départ comporte deux fichiers, `utils.h` et `utils.cpp`, qui contiennent une structure `point` (qu'il faudra éventuellement étoffer de méthodes utiles) et une fonction destinée à récupérer les touches clavier pour l'interaction avec les joueurs.

Il s'agit ici de concevoir un objet `Serpent` doté des méthodes adéquates, plus une fonction `jeu_1p` exploitant les capacités du `Serpent` pour reproduire le comportement désiré. On pourra dans un premier temps ne pas gérer les collisions (avec le bord et avec lui-même), et ne les rajouter que dans un second temps. Votre travail se décompose en 6 étapes :

1. (*sur papier*) Définir l'interface de la classe `Serpent` (c'est-à-dire lister toutes les fonctionnalités nécessaires).
2. (*sur papier*) Réfléchir à l'implémentation de la classe `Serpent` : comment stocker les données ? comment programmer les différentes méthodes ? (lire en préliminaire les remarques du paragraphe suivant).
3. Dans un fichier `serpent.h`, écrire la déclaration de votre classe `Serpent` : ses membres, ses méthodes, ce qui est public, ce qui ne l'est pas.
4. Soumettre le résultat de vos réflexions à votre enseignant pour valider avec lui les choix retenus.

5. Implémenter la classe `Serpent` (c'est-à-dire programmer les méthodes que vous avez déclarées).
6. Programmer la fonction `jeu_1p` utilisant un `Serpent`.

Remarque : Dans le fichier `utils.h` sont définis :

1. 4 entiers `gauche`, `bas`, `haut`, `droite` de telle manière que :
 - (a) la fonction $x \rightarrow (x + 1) \% 4$ transforme `gauche` en `bas`, `bas` en `droite`, `droite` en `haut` et `haut` en `gauche` ; cette fonction correspond donc à un quart de tour dans le sens trigonométrique.
 - (b) la fonction $x \rightarrow (x - 1) \% 4$ transforme `gauche` en `haut`, `haut` en `droite`, `droite` en `bas` et `bas` en `gauche` ; cette fonction correspond donc à un quart de tour dans le sens des aiguilles d'une montre.
2. un tableau de 4 points `dir` de telle manière que, moyennant la définition d'une fonction permettant de faire la somme de deux points, la fonction $p \rightarrow p + dir[d]$ renvoie :
 - (a) pour `d=gauche` le point correspondant au décalage de `p` de 1 vers la gauche.
 - (b) pour `d=haut` le point correspondant au décalage de `p` de 1 vers la haut.
 - (c) pour `d=droite` le point correspondant au décalage de `p` de 1 vers la droite.
 - (d) pour `d=bas` le point correspondant au décalage de `p` de 1 vers la bas.

A.8.2 Tron

A partir du jeu de `Serpent` réalisé précédemment, nous allons facilement pouvoir implémenter le jeu `Tron`. Le programme `tron.exe` vous donne une idée du résultat recherché. Le principe de ce jeu est que chaque joueur pilote une moto qui laisse derrière elle une trace infranchissable. Le but est de survivre plus longtemps que le joueur adverse.

1. Passage à deux joueurs.

A partir de la fonction `jeu_1p`, créer une fonction `jeu_2p` implémentant un jeu de serpent à 2 joueurs. On utilisera pour ce joueur les touches `S`, `X`, `D` et `F`. La fonction `Clavier()` renverra donc les entiers `int('S')`, `int('X')`, `int('D')` et `int('F')`. Remarque : on ne gèrera qu'une touche par tour, soit un seul appel à la fonction `Clavier()` par tour.

2. Ultimes réglages

- (a) Gérer la collision entre les deux serpents.
- (b) Le principe de `Tron` est que la trace des mobiles reste. Pour implémenter cela, il suffit d'allonger nos serpents à chaque tour.

A.8.3 Graphismes

Petit bonus pour les rapides : nous allons voir comment gérer des graphismes un peu plus sympas que les rectangles uniformes que nous avons utilisés jusqu'ici. L'objectif est de remplacer le carré de tête par une image que l'on déplace à chaque tour.

Nous allons utiliser pour cela les `NativeBitmap` d'`Imagine++`, qui sont des images à affichage rapide. Pour charger une image dans une `NativeBitmap` on procède ainsi :

```
// Entiers passés par référence lors du chargement de l'image pour  
// qu'y soient stockées la largeur et la hauteur de l'image  
int w,h;  
// Chargement de l'image  
byte* rgb;  
loadColorImage("nom_fichier.bmp",rgb,w,h);  
// Déclaration de la NativeBitmap  
NativeBitmap ma_native_bitmap (w,h);  
// On place l'image dans la NativeBitmap  
ma_native_bitmap.setColorImage(0,0,rgb,w,h);
```

L'affichage d'une `NativeBitmap` à l'écran se fait alors avec la méthode :

```
void putNativeBitmap(int x, int y, NativeBitmap nb)
```

1. Remplacer dans le serpent l'affichage de la tête par l'affichage d'une image. On pourra utiliser les images `moto_blue.bmp` et `moto_red.bmp` fournies.
2. Utiliser l'image `explosion.bmp` lors de la mort d'un des joueurs.

Annexe B

Imagine++

Imagine++ est un ensemble de bibliothèques permettant de faire simplement du graphisme et de l'algèbre linéaire. Elles s'appuient pour cela sur les projets Qt (graphisme 2D et 3D) et Eigen (algèbre linéaire). Ceux-ci proposent une richesse de possibilités bien plus importantes que Imagine++ mais en contrepartie Imagine++ est plus simple à utiliser.

Imagine++ est un logiciel libre, vous pouvez donc l'utiliser et le distribuer à votre guise (et gratuitement!), mais si vous le distribuez sous une forme modifiée vous devez offrir selon les mêmes termes les sources de vos modifications. Une documentation complète est disponible sur la page Web d'Imagine++, détaillant l'installation et l'utilisation.

Pour utiliser un module, par exemple Images, un fichier source doit l'inclure

```
#include <Imagine/Images.h>
```

pour une compilation correcte, et votre fichier CMakeLists.txt doit comporter

```
ImagineUseModule(MonProgramme Images)
```

pour que l'édition de liens (link) réussisse.

Tout est englobé dans un namespace `Imagine`, donc si vous voulez éviter de préfixer les noms par `Imagine::` vous devez utiliser dans votre code

```
using namespace Imagine;
```

B.1 Common

Le module `Common` définit entre autres la classe `Color` codée par un mélange de rouge, vert et bleu, la quantité de chacun codée par un entier entre 0 et 255 :

```
Color noir  = Color(0,0,0);  
Color blanc = Color(255,255,255);  
Color rouge = Color(255,0,0);
```

Un certain nombre de constantes de ce type sont déjà définies : `BLACK`, `WHITE`, `RED`, `GREEN`, `BLUE`, `CYAN`, `MAGENTA`, `YELLOW`.

Un type `byte` (synonyme de `unsigned char`) est défini pour coder une valeur entière entre 0 et 255.

Très pratique, `srcPath` fait précéder la chaîne de caractère argument par le chemin complet du répertoire contenant le fichier source. L'équivalent pour un argument de type `string` est `stringSrcPath` :

```
const char* fichier = srcPath("mon_fichier.txt");
string s = "mon_fichier.txt";
s = stringSrcPath(s);
```

En d'autres termes, le fichier sera trouvé quel que soit l'emplacement de l'exécutable.

La classe template `FArray` s'utilise pour des tableaux de taille petite et connue à la compilation (allocation statique). Pour des tableaux de taille non connue à la compilation (allocation dynamique), utiliser `Array`. Pour les matrices et vecteurs, utiliser `FMatrix` et `FVector`, utilisant l'allocation statique comme indiqué par le préfixe `F` (*fixed*). Les équivalents dynamiques sont dans `LinAlg`.

B.2 Graphics

Le module `Graphics` propose du dessin en 2D et 3D. Les coordonnées 2D sont en pixel, l'axe des x est vers la droite et l'axe des y vers le bas (attention, ce n'est pas le sens mathématique usuel!). Le point (0,0) est donc le coin haut-gauche de la fenêtre (pour les fonctions de tracé) ou de l'écran (pour `openWindow`).

```
openWindow(500,500); // Taille de fenêtre
drawRect(10,10, 480,480,RED); // Coin haut-gauche (10,10),
                                // largeur 480, hauteur 480
drawLine(10,10, 490,490,BLUE); // Diagonale
Window w = openWindow(100,100); // Autre fenêtre
setActiveWindow(w); // Sélection pour les prochains tracés
drawString(10,10, "Du_texte", MAGENTA); // Mettre du texte
endGraphics(); // Attend un clic souris avant de fermer les fenêtres
```

Si on a beaucoup de dessins à faire à la suite et qu'on veut n'afficher que le résultat final (c'est plus esthétique), on encadre le code de tracé par :

```
noRefreshBegin();
...
noRefreshEnd();
```

Pour faire une animation, il est utile de faire une petite pause entre les images pour réguler la cadence :

```
milliSleep(50); // Temps en millisecondes
```

Attention cependant à ne pas intercaler une telle commande entre un `noRefreshBegin` et un `noRefreshEnd`, car rien ne s'afficherait pendant cette pause.

On peut charger une image (`loadGreyImage`, `loadColorImage`) ou sauvegarder (`saveGreyImage`, `saveColorImage`) dans un fichier. Attention, ces fonctions allouent de la mémoire qu'il ne faut pas oublier de libérer après usage.

```
byte* g;
int largeur, hauteur;
if (! loadGreyImage(srcPath("image.jpg"), g, largeur, hauteur)) {
    cerr << "Impossible_d'ouvrir_le_fichier_"
```



```

    << srcPath("image.jpg") << endl;
    exit(1);
}
// Dessine avec coin haut gauche en (0,0) :
putGreyImage(0, 0, g, largeur, hauteur);
delete [] g; // Ne pas oublier !

```

A noter `srcPath`, défini dans `Common`, qui indique de chercher dans le dossier contenant les fichiers source.

En fait, pour éviter de gérer soi-même la mémoire des images, il existe une classe dédiée à cela :

B.3 Images

Le module `Images` gère le chargement, la manipulation et la sauvegarde des images.

```

Image<byte> im; // Image en niveaux de gris
if (! load(im, srcPath("fichier_image.png"))) {
    cerr << "Impossible_d'ouvrir_le_fichier"
         << srcPath("fichier_image.png") << endl;
    exit(1);
}
display(im); // Dessine dans la fenêtre active
im(0,0)=128; // Met le pixel en gris
save(im, "fichier_image2.png"); // Sauvegarde l'image dans un fichier

```

Attention : la recopie et l'affectation (opérateur `=`) sont des opérations peu coûteuses qui en fait ne font qu'un lien entre les images, sans réelle copie. Ce qui fait que la modification de l'une affecte l'autre :

```

Image<Color> im1(100,100);
Image<Color> im2 = im1; // Constructeur par recopie
im1(10,10) = CYAN;
assert(im2(10,10) == CYAN); // im2 a été affectée

```

Pour faire une vraie copie plutôt qu'un lien, on utilise :

```

im2 = im1.clone();
im1(10,10) = CYAN; // N'affecte pas im2

```

Ainsi, si on passe une image comme paramètre d'une fonction, puisque c'est le constructeur par copie qui est appelé, tout se passe comme si on avait passé par référence :

```

void f(Image<Color> im) { // Passage par valeur
    im(10,10) = CYAN;
}

```

```

f(im1); // Modifie quand même le pixel (10,10)

```

Une erreur courante est de chercher à lire ou écrire à des coordonnées au-delà des bornes du tableau, typiquement une erreur dans un indice de boucle.

B.4 LinAlg

Le module `LinAlg` propose l'algèbre linéaire avec des classes matrice et vecteur.

```
Matrix<float> I(2,2); // Taille 2x2
I.fill(0.0f); // Matrice nulle
I(0,0)=I(1,1)=1.0f; // Matrice identité
cout << "det(I)=" << det(I) << endl; // Déterminant
```

Les opérateurs d'addition (matrice+matrice, vecteur+vecteur), soustraction (matrice-matrice, vecteur-vecteur) et multiplication (matrice*matrice, matrice*vecteur) sont bien sûr définis.

Comme pour les images, attention de ne pas sortir des bornes en accédant aux éléments des matrices et vecteurs !

Une fonction très utile est `linSolve` pour résoudre un système linéaire.

B.5 Installation

Examinons le fichier `CMakeLists.txt` d'un programme utilisant `Imagine++`.

```
cmake_minimum_required(VERSION 2.6)
find_package(Imagine REQUIRED)
```

```
project(Balle)
add_executable(Balle Balle.cpp)
ImagineUseModules(Balle Graphics)
```

Pour que le `find_package` fonctionne, il est parfois nécessaire de définir la variable `Image_DIR`. Cette commande va charger le fichier

```
/usr/share/Imagine++/CMake/ImagineConfig.cmake
```

quit contient des commandes `CMake`, dont `ImagineUseModules`. Cette dernière fait deux choses :

- Indique où chercher lors des `#include`, comme par exemple dans le dossier

```
/usr/share/Imagine++/include
```

Ainsi, l'instruction

```
#include "Imagine/Graphics.h"
```

inclura le fichier

```
/usr/share/Imagine++/include/Imagine/Graphics.h
```

- Indique qu'il faut lier la bibliothèque `libImageGraphics` à lors de l'édition de liens (link) ; elle se trouve dans

```
/usr/share/Imagine++/lib
```

Annexe C

Compilateur, CMake, Linker, Qt... Au secours !

D'accord, tout cela a l'air bien compliqué, et en effet ce n'est pas si simple. Néanmoins, il faut en passer par là, et nous allons voir que ce n'est pas si difficile à comprendre, malgré l'effroi que nous pouvons ressentir face à une telle accumulation d'outils. La lecture de cette annexe n'est pas nécessaire pour les utiliser, mais elle devrait aider à comprendre comment tout cela s'imbrique.

—

C.1 Compilation

C.1.1 Compilateur et linker

Commençons par le principe de base : pour passer du *code source*, c'est-à-dire des fichiers d'extension `.h` et `.cpp`¹, à un *programme exécutable*, il faut lancer le *build*, qu'on appelle souvent abusivement compilation, alors que cette compilation proprement dite n'est qu'une partie du processus². Celle-ci est composée en fait de deux phases, compilation (faite par le compilateur) suivie d'édition de liens (faite par le *linker*). Le rôle du linker est de rassembler les produits de la phase de compilation, les fichiers *objet* (d'extension `.o` ou `.obj`), et d'éventuelles *bibliothèques* (*libraries* en anglais, que les informaticiens francisent souvent à tort en *librairies*), qui ne sont en fait que des fichiers de type archive (comme un zip) de fichiers objet. Ceci est résumé par le schéma de la Figure C.1.

Plusieurs choses sont à remarquer :

- Le compilateur est lancé *indépendamment* sur chaque source `.cpp`, au point que ces compilations peuvent être lancées en parallèle. Une conséquence importante est que la modification d'un `.cpp` ne nécessite pas la recompilation des autres, d'où gain de temps du build.

1. Parfois on remplace l'extension `.h` par `.hpp` pour éviter de confondre avec les fichiers header du langage C. On rencontre aussi les conventions `.hxx` et `.cxx`, car le `+` pourrait être mal géré dans le nom d'un fichier par certains systèmes d'exploitation, et finalement le `x` n'est qu'un `+` tourné.

2. On fait donc là une synecdoque. Comme quoi faire de l'ingénierie ne prive pas d'avoir du goût pour les figures littéraires !

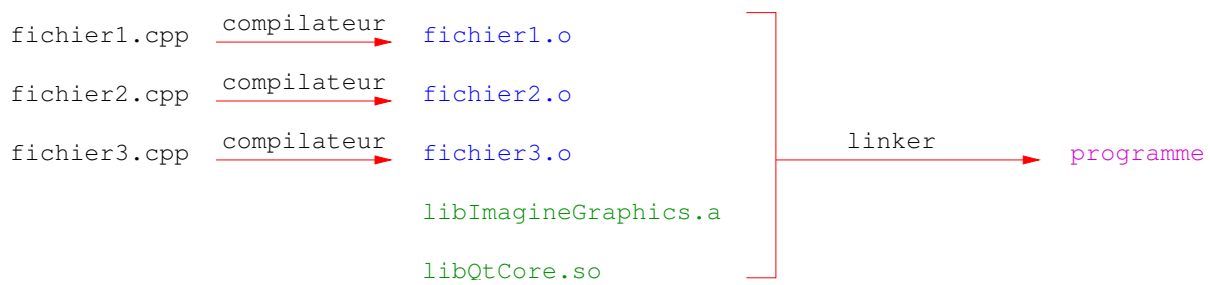


FIGURE C.1 – Phases de compilation et d'édition de liens. Les extensions sont les conventions des systèmes Unix (Linux et Mac), sous Windows les fichiers objet sont en `.obj`, les bibliothèques en `.lib` et le programme en `.exe`.

- Les fichiers *header* en `.h` ne sont *pas* passés directement au compilateur, mais ils le sont indirectement par les `.cpp` qui font des `#include` de ces `.h`. Comme plusieurs `.cpp` peuvent faire un `#include "fic1.h"`, ce fichier passe plusieurs fois par le compilateur. Il est donc essentiel que le compilateur ne génère pas de *symboles* dus à `fic1.h`, comme des variables globales (à éviter tant que possible de toute façon) ou des définitions de fonction, mais seulement des déclarations (de classes, de fonctions, de constantes)³. Sinon, lors de l'édition de liens, il va échouer en signalant : `Multiply defined symbol...`
- Le rôle de l'éditeur de liens est de vérifier que chaque fonction ou méthode de classe utilisée⁴ soit définie une et une seule fois, c'est-à-dire dans un unique `.o` ou bibliothèque. De plus, il vérifie que la fonction d'entrée du programme, le `main`, est bien défini également (une seule fois).
- Les bibliothèques se présentent sous deux formes : statiques (extension `.a`) et dynamiques (extension `.so`, comme `shared object`). Lors de l'édition de liens, les symboles des bibliothèques statiques utilisés sont inclus dans le programme résultant, alors que ceux des bibliothèques dynamiques ne le sont pas, l'éditeur de liens vérifie juste leur présence. L'avantage de ces dernières est que le programme est moins volumineux en mémoire, que plusieurs programmes utilisant la même bibliothèque ne nécessitent qu'un unique chargement en mémoire de cette bibliothèque lors de leur lancement, et que la correction d'une faille de sécurité ou d'un bug dans une bibliothèque dynamique ne nécessite pas de recompiler tous les programmes l'utilisant et de les redistribuer aux utilisateurs (pourvu qu'on prenne garde à seulement corriger ces bugs sans changer les arguments des fonctions, etc). La contrepartie, c'est que le programme exécutable ne peut pas se lancer directement car il est incomplet, il a besoin de trouver les bibliothèques dynamiques dont il dépend.

En fait, sous Windows, les bibliothèques dynamiques sont scindées en deux : le `.lib` sert lors de l'édition de liens (penser à lui comme à une sorte de header pour la bibliothèque) tandis que la partie `.dll` sert lors du lancement du programme. Le pro-

3. Les exceptions sont les fonctions `inline`, les méthodes de classes définies directement dans la classe, qui sont en fait aussi `inline`, et les fonctions ou méthodes `template`, qui sont très particulières car elles ne peuvent être réellement compilées que lorsque les types des paramètres `template` sont connus, on parle d'*instantiation* de ces templates.

4. Une fonction ou méthode *déclarée* mais non *appelée* ne provoque pas d'erreur mais est simplement considérée comme superflue par l'éditeur de liens.

gramme ne peut se lancer s'il ne trouve pas les `.dll` dont il a besoin⁵. On peut voir les bibliothèques dynamiques dont dépendent un programme exécutable avec les outils *ldd* (Linux), *otool* (Mac) et *depend.exe* (Windows).

C.2 En ligne de commande

Voici un exemple de ligne de compilation d'un fichier `test.cpp`, dont le fichier objet résultant s'appelle `test.o` :

```
/usr/bin/c++ -DQT_CORE_LIB -DQT_GUI_LIB -DQT_NO_DEBUG -DQT_OPENGL_LIB
-DQT_WIDGETS_LIB -O3 -DNDEBUG
-I/home/pascal/Ponts/svn/ImagineQt/Imagine/Common/src
-I/home/pascal/Ponts/svn/ImagineQt/Imagine/Graphics/src -I/usr/include/qt5
-I/usr/include/qt5/QtCore -I/usr/lib/x86_64-linux-gnu/qt5/mkspecs/linux-g++-64
-I/usr/include/qt5/QtWidgets -I/usr/include/qt5/QtGui
-I/usr/include/qt5/QtOpenGL
-DSRCDIR="/home/pascal/Ponts/svn/ImagineQt/Imagine/Graphics/test"
-o CMakeFiles/ImagineGraphicsTest.dir/test.o
-c /home/pascal/Ponts/svn/ImagineQt/Imagine/Graphics/test/test.cpp
```

Le compilateur s'appelle `c++`, les options `-D` font comme si `test.cpp` comportait un `#define` correspondant. L'option `-O3` indique l'optimisation du code niveau 3 (le plus élevé). Les options `-I` indiquent les dossiers où les `#include` doivent chercher les fichiers header s'ils ne les trouvent pas dans le dossier courant. Maintenant, l'édition de lien de ce seul fichier objet avec les bibliothèques dont il a besoin :

```
/usr/bin/c++ CMakeFiles/ImagineGraphicsTest.dir/test.o
-o ImagineGraphicsTest ../src/libImagineGraphics.a
/usr/lib/x86_64-linux-gnu/libQt5Core.so.5.2.1
/usr/lib/x86_64-linux-gnu/libQt5Widgets.so.5.2.1
/usr/lib/x86_64-linux-gnu/libQt5Gui.so.5.2.1
/usr/lib/x86_64-linux-gnu/libQt5OpenGL.so.5.2.1
-lGLU -lGL -lSM -lICE -lX11 -lXext
```

Le programme résultant s'appelle `ImagineGraphicsTest`, il utilise la bibliothèque statique `libImagineGraphics.a` et les bibliothèques dynamiques Qt, ainsi que les bibliothèques standard `libGLU.so`, `libGL.so`, etc. Ces dernières sont installées dans des emplacements connus du linker, donc on n'a pas besoin de lui dire où chercher, mais on aurait pu lui donner des indices avec des options `-L`, comme les `-I` de la compilation. En fait le programme `c++` lancé ainsi comprend qu'on est en train de faire de l'édition de liens et non une compilation, et appelle en fait le linker.

On voit donc que pour utiliser une bibliothèque, il faut (1) indiquer au compilateur où se trouvent ses fichiers header `.h`, (2) indiquer au linker où se trouvent ces bibliothèques.

5. Il cherche pour cela dans le dossier contenant le programme et dans la liste des dossiers indiqués par la variable d'environnement `PATH`. Sous Linux et Mac, la variable équivalente pour chercher les `.so` s'intitule `LD_LIBRARY_PATH`.

C.3 Make et CMake

On voit que les lignes ci-dessus sont complexes et la phase de build serait bien fastidieuse si on devait faire tout cela soi-même. A ce point entre en jeu l'automate de build nommé `make`. Celui-ci lit un fichier texte nommé `Makefile` qui décrit les dépendances entre fichiers et les règles pour régénérer les dépendances lorsque c'est nécessaire. Ainsi, un `Makefile` pour le programme ci-dessus indiquerait que le programme `ImagineGraphicsTest` dépend de `test.o` qui lui-même dépend de `test.cpp`. Mais cela ne s'arrête pas là, puisque `test.cpp` fait des `#include` de fichiers header `.h`, qui eux-mêmes peuvent dépendre d'autres `.h` par des `#include`, etc.

Les lignes de génération de `test.o` et de `ImagineGraphicsTest` sont celles de compilation et de link de l'exemple ci-dessus. Le programme `make` se fie au temps de modification des fichiers pour savoir ce qui est à jour, et ne nécessite aucune action, et ce qui ne l'est pas, et nécessite d'appliquer les règles de régénération : si un fichier *A*, dont dépend un fichier *B*, a été modifié plus récemment que *B* (ou que *B* n'existe pas), la règle de régénération de *B* est appliquée. Ainsi, l'unique commande `make` permet d'effectuer à la fois la compilation et le link. Lancer `make` deux fois de suite fait que le deuxième appel ne fait rien, puisque tous les fichiers sont à jour d'après leur heure de modification. C'est le comportement de moindre effort le plus raisonnable.

Pour trouver toutes les dépendances d'un fichier `.cpp`, il faut trouver ses `#include` et suivre ceux-ci transitivement. Le compilateur (en fait, la phase préliminaire de la compilation effectuée par le préprocesseur, à qui toutes les instructions en `#` sont adressées) en est tout à fait capable et écrire ces dépendances soi-même dans le `Makefile` est fastidieux et probablement difficile à maintenir exact lors du développement du code source. Pour compliquer les choses, le `make` sous Linux et Mac est le *GNU make* tandis que celui de Windows qui vient avec Visual Studio est *nmake*. Ils ne sont pas tout à fait compatibles, et comme de toutes façons les compilateurs sont différents et n'utilisent pas les mêmes options, il faudrait faire au moins deux `Makefile`. Pour éviter toute cette complexité, l'outil plus moderne *CMake*, une sorte de meta-make entre en scène. Ses avantages sont :

- Un format plus simple et uniforme pour tous les OS de meta-`Makefile` appelé `CMakeLists.txt`.
- Calcul automatique des dépendances.
- Possibilité (et c'est fortement conseillé) de placer les fichiers générés dans un dossier autre (`build directory`) que celui des sources (`source directory`), de manière à avoir une procédure de nettoyage très simple : simplement se débarrasser du `build directory`.

Le `CMakeLists.txt` pour l'exemple précédent est élémentaire :

```
find_package(Imagine REQUIRED)
add_executable(ImagineGraphicsTest test.cpp)
ImagineUseModules(ImagineGraphicsTest Graphics)
```

La première ligne indique qu'on a besoin d'utiliser `Imagine++`. Cela cherche un fichier `ImagineConfig.cmake` qui contient entre autres la commande `ImagineUseModules` (non standard dans *CMake*). La deuxième ligne indique que le programme à créer⁶ s'appelle `ImagineGraphicsTest(.exe` sous Windows) et qu'il ne dépend que du

6. pour créer une bibliothèque (dynamique par défaut) à la place, il suffit de remplacer par `add_library`

seul `test.cpp`. S'il y avait plusieurs `.cpp`, on les ajouterait à la ligne séparés par des espaces. On peut aussi ajouter les `.h`, bien que ceux-ci ne soient pas compilés directement : ils seront juste visibles dans la partie *headers* si on demande à CMake de créer un projet (voir les IDE). La dernière ligne indique que notre programme a besoin de la bibliothèque `Graphics` d'Imagine++, ce qui indique les chemins des headers et des bibliothèques. Le fait que cette bibliothèque dépende de Qt est automatiquement ajouté. Un exemple typique d'utilisation de CMake en ligne de commande sous Linux ou Mac :

```
mkdir Build ← Crée un dossier Build, le build directory
cd Build ← Va dans ce dossier
cmake .. ← Indique que le source directory est le parent (..)
make ← Il s'agit maintenant du make classique
```

Le lancement de `cmake` va créer un fichier `CMakeCache.txt` dans le build directory qui contient une liste de variables de CMake. Entre autres, on a `CMAKE_BUILD_TYPE` qui indique le degré d'optimisation à appliquer. Le degré maximum est le mode `Release`, tandis que le mode `Debug` est approprié pour un programme qu'on veut suivre ligne à ligne. On peut parfaitement modifier une de ces variables⁷ et relancer `cmake`. Le lancement de cette commande génère le fichier `Makefile` pour nous, qui sert alors de configuration au `make` classique.

À noter que la modification ultérieure du `CMakeLists.txt`, par exemple pour ajouter des fichiers source dans la ligne `add_executable`, ne nécessite pas de relancer explicitement `cmake`, car le `Makefile` généré retient que `CMakeCache.txt` est une dépendance de `CMakeLists.txt`. Donc le fait de lancer `make` directement suffit à relancer `cmake` avant d'effectuer le build normal.

Enfin, la force de CMake est qu'il connaît tous les IDE classiques (Visual Studio, Eclipse, Code::Blocks, etc) et est capable de générer des projets pour ceux-ci (cf Figure C.2). Absents de cette liste sont `Kdevelop` (Linux) et surtout `Qt Creator` (toutes plates-formes). En effet, ceux-ci gèrent eux-mêmes des projets CMake, donc on peut lancer `cmake` depuis leur interface.

L'intérêt de tout ceci est que l'utilisateur peut choisir l'IDE qui lui convient le mieux (ou aucun d'entre eux s'il préfère générer simplement des `Makefiles`) et qu'un seul fichier de configuration `CMakeLists.txt` assurera la portabilité du build sur toutes les plates-formes. Ensuite, c'est au programmeur de coder en respectant la norme du C++ pour un maximum de portabilité.

C.4 Utilisation d'un IDE

Le confort d'utilisation d'un IDE est notable : éditeur intégré, navigation facilitée dans le code, build intégré, debugger, gestion de versions (`git`), analyseur mémoire (`vagrind` sous Linux et Mac)... On peut très bien modifier le `CMakeLists.txt` depuis l'éditeur intégré de l'IDE et ne pas relancer `cmake` soi-même par la suite : CMake est suffisamment intelligent pour créer des projets qui savent quand il est nécessaire de relancer `cmake`. Du coup, une fois le projet créé, on n'a pas besoin de sortir de l'IDE.

Néanmoins, pour que tous ces outils fonctionnent en harmonie, il faut que les choses soient bien configurées. Par exemple, `QtCreator` doit savoir où trouver l'exé-

7. directement avec un éditeur de texte, ou alors avec les outils `ccmake` ou `cmake-gui` qui offrent une interface graphique

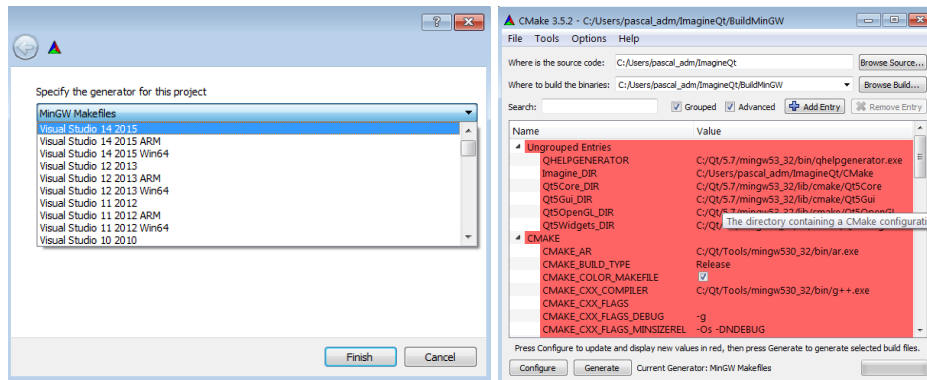


FIGURE C.2 – Création d'un projet avec choix de l'IDE et interaction avec les variables de CMake stockées dans `CMakeCache.txt` grâce à `cmake-gui`. Notons que CMake gère les différentes versions de Visual Studio, en 32 et 64 bits. Le compilateur MinGW étant installé (il vient avec l'installeur de Qt), on peut aussi créer des Makefiles comme sous Linux et Mac, sauf que le GNU `make` qui les interprète s'appelle `mingw32-make` et non `make`.

cutable `cmake`, ainsi que le compilateur, débbuger, etc. Enfin, il faut faire attention de garder CMake comme système de build, et décliner, quand on crée un nouveau fichier source, la proposition de l'IDE de l'intégrer au projet. Cela doit se faire au niveau de CMake, donc modifier le `CMakeLists.txt` pour cela.

Pour Imagine++, il faut que la commande `find_package(Imagine REQUIRED)` puisse trouver le fichier `ImagineConfig.cmake`. Pour cela, il faut l'aider en donnant le dossier d'installation dans la variable `Imagine_DIR`.

Lorsque plus rien ne semble marcher, il est sage de créer un nouveau dossier de build vierge et de travailler dans celui-ci. Cela peut s'avérer nécessaire après des changements majeurs dans un projet.

Enfin, notons que Qt *n'est pas* un IDE mais un ensemble de bibliothèques C++ multi-plates-formes, utilisées par Imagine++. Il se trouve que l'IDE QtCreator utilise ces bibliothèques Qt, et s'il a été installé en même temps que les bibliothèques Qt, il sait directement où trouver ces dernières, ce qui aide pour un projet Imagine++.

Annexe D

Fiche de référence finale

Fiche de référence (1/5)		
Variables — Définition : <pre>int i; int k,l,m;</pre> — Affectation : <pre>i=2; j=i; k=l=3;</pre> — Initialisation : <pre>int n=5,o=n;</pre> — Constantes : <pre>const int s=12;</pre> — Portée : <pre>int i; // i=j; interdit! int j=2; i=j; // OK! if (j>1) { int k=3; j=k; // OK! } //i=k; interdit!</pre> — Types : <pre>int i=3; double x=12.3; char c='A'; string s="hop"; bool t=true; float y=1.2f; unsigned int j=4; signed char d=-128; unsigned char d=25; complex<double> z(2,3);</pre> — Variables globales : <pre>int n; const int m=12; void f() { n=10; // OK</pre>	<pre>int i=m; // OK ...</pre> — Conversion : <pre>int i=int(x),j; float x=float(i)/j;</pre> — Pile/Tas	<pre>break; case 2: case 3: ...; break; default: ...; }</pre> — <code>mx=(x>y)?x:y;</code>
	— Type énuméré : <pre>enum Dir{N,E,S,W}; void avance(Dir d);</pre> — Variables statiques : <pre>int f() { static bool once=true; if (once) { once=false; ... } ... }</pre>	Boucles — <code>do {</code> <pre>... } while(!ok);</pre> — <code>int i=1;</code> <pre>while(i<=100) { ... i=i+1; }</pre> — <code>for(int i=1;i<=10;i++)</code> <pre>...</pre> — <code>for(int i=1,j=10;j>i;</code> <pre> i=i+2,j=j-3) ...</pre> — <code>for (int i=...)</code> <pre> for (int j=...) { //saute cas i==j if (i==j) continue; ... }</pre> — <code>for (int i=...)</code> <pre>{ ... if (t[i]==s){ // quitte boucle break; } ... }</pre>
Tests — Comparaison : <pre>== != < > <= >=</pre> — Négation : ! — Combinaisons : && — <code>if (i==0) j=1;</code> — <code>if (i==0) j=1;</code> <pre>else j=2;</pre> — <code>if (i==0) {</code> <pre> j=1; k=2; }</pre> — <code>bool t=(i==0);</code> <pre>if (t) j=1;</pre> — <code>switch (i) {</code> <pre>case 1: ...; ...;</pre>		

Fiche de référence (2/5)

Fonctions

— Définition :

```
int plus(int a,int b){
    int c=a+b;
    return c;
}
void affiche(int a) {
    cout << a << endl;
}
```

— Déclaration :

```
int plus(int a,int b);
```

— Retour :

```
int signe(double x) {
    if (x<0)
        return -1;
    if (x>0)
        return 1;
    return 0;
}
void afficher(int x,
              int y) {
    if (x<0 || y<0)
        return;
    if (x>=w || y>=h)
        return;
    DrawPoint(x,y,RED);
}
```

— Appel :

```
int f(int a) { ... }
int g() { ... }
...
int i=f(2),j=g();
```

— Références :

```
void swap(int& a,
          int& b){
    int tmp=a;
    a=b;b=tmp;
}
```

```
...
int x=3,y=2;
swap(x,y);
```

— Surcharge :

```
int hasard(int n);
int hasard(int a,
           int b);
double hasard();
```

— Opérateurs :

```
vect operator+(
    vect A,vect B) {
    ...
}
...
vect C=A+B;
```

— Pile des appels

— Itératif/Récurusif

— Références constantes (pour un passage rapide) :

```
void f(const obj& x){
    ...
}
void g(const obj& x){
    f(x); // OK
}
```

— Valeurs par défaut :

```
void f(int a,int b=0);
void g() {
    f(12); // f(12,0);
    f(10,2); // f(10,2);
}
void f(int a,int b) {
    // ...
}
```

— Inline (appel rapide) :

```
inline double
sqr(double x) {
    return x*x;
}
...
double y=sqr(z-3);
```

— Référence en retour :

```
int i; // Var. globale
int& f() {
    return i;
}
...
f()=3; // i=3!
```

Tableaux

— Définition :

```
— double x[5],y[5];
for(int i=0;i<5;i++)
    y[i]=2*x[i];
```

```
— const int n=5;
int i[n],j[2*n];
```

— Initialisation :

```
int t[4]={1,2,3,4};
string s[2]={"ab","c"};
```

— Affectation :

```
int s[3]={1,2,3},t[3];
for (int i=0;i<3;i++)
    t[i]=s[i];
```

— En paramètre :

```
— void init(int t[4]) {
    for(int i=0;i<4;i++)
```

```
t[i]=0;
```

```
}
```

```
— void init(int t[],
            int n) {
    for(int i=0;i<n;i++)
        t[i]=0;
}
```

— Taille variable :

```
int* t=new int[n];
...
delete[] t;
```

— En paramètre (suite) :

```
— void f(int* t,int n){
    t[i]=...
}
```

```
— void alloue(int*& t){
    t=new int[n];
}
```

— 2D :

```
int A[2][3];
A[i][j]=...;
int A[2][3]=
    {{1,2,3},{4,5,6}};
void f(int A[2][2]);
```

— 2D dans 1D :

```
int A[2*3];
A[i+2*j]=...;
```

— Taille variable (suite) :

```
int *t,*s,n;
```

— En paramètre (fin) :

```
void f(const int* t,
      int n) {
```

```
...
```

```
s+=t[i]; // OK
```

```
...
```

```
t[i]=...; // NON!
```

```
}
```

Structures

```
— struct Point {
    double x,y;
    Color c;
};
```

```
...
```

```
Point a;
a.x=2.3; a.y=3.4;
a.c=Red;
Point b={1,2.5,Blue};
```

```
— Une structure est un objet entièrement public (→ cf objets!)
```

Fiche de référence (3/5)

Objets

```

— struct obj {
    int x;    // champ
    int f(); // méthode
    int g(int y);
};
int obj::f() {
    int i=g(3); // mon g
    int j=x+i;  // mon x
    return j;
}
...
int main() {
    obj a;
    a.x=3;
    int i=a.f();
}

— class obj {
    int x,y;
    void a_moi();
public:
    int z;
    void pour_tous();
    void autre(obj A);
};
void obj::a_moi() {
    x=..;    // OK
    ..=y;    // OK
    z=..;    // OK
}
void obj::pour_tous(){
    x=..;    // OK
    a_moi(); // OK
}
void autre(obj A) {
    x=A.x;    // OK
    A.a_moi(); // OK
}
...
int main() {
    obj A,B;
    A.x=..;    //NON
    A.z=..;    //OK
    A.a_moi(); //NON
    A.pour_tous(); //OK
    A.autre(B); //OK
}

— class obj {
    obj operator+(obj B);
};
...
int main() {
    obj A,B,C;
    C=A+B;
    // C=A.operator+(B)
}

— Méthodes constantes :
void obj::f() const{
    ...
}

```

```

void g(const obj& x){
    x.f(); // OK
}

```

```

— Constructeur :
class point {
    int x,y;
public:
    point(int X,int Y);
};
point::point(int X,
              int Y){
    x=X;
    y=Y;
}
...
point a(2,3);

```

```

— Constructeur vide :
obj::obj() {
    ...
}
...
obj a;

— Objets temporaires :
vec vec::operator+(
                    vec b) {
    return vec(x+b.x,
              y+b.y);
}
...
c=vec(1,2)
   +f(vec(2,3));

```

```

— Destructeur :
obj::~~obj() {
    ...
}

— Constructeur de copie :
obj::obj(const obj& o){
    ...
}

```

Utilisé par :

- obj b(a);
- obj b=a;
- //mieux que obj b;b=a;
- paramètres des fonctions
- valeur de retour

```

— Affectation :
obj& obj::operator=(
                    const obj&o){
    ...
    return *this;
}

```

— Objets avec allocation dynamique automatique : cf section 10.11

```

— Accesseurs :
class mat {

```

```

    double *x;
public:
    double& operator()
        (int i,int j){
        assert(i>=0 ...);
        return x[i+M*j];
    }
    double operator()
        (int i,int j)const{
        assert(i>=0 ...);
        return x[i+M*j];
    }
    ...

```

Compilation séparée

- #include "vect.h", aussi dans vect.cpp
- Fonctions : déclarations dans le .h, définitions dans le .cpp
- Types : définitions dans le .h
- Ne déclarer dans le .h que les fonctions utiles.
- #pragma once au début du fichier.
- Ne pas trop découper...

STL

- min,max,...
- complex<double> z;
- pair<int,string> p;
 - p.first=2;
 - p.second="hop";
- #include<list>
 - using namespace std;
 - ...
 - list<int> l;
 - l.push_front(1);
 - ...
 - if(l.find(3)!=l.end())
 - ...
 - list<int>::
 - const_iterator it;
 - for (it=l.begin();
 it!=l.end();it++)
 s+= *it;
 - list<int>::iterator it
 - for (it=l.begin();
 it!=l.end();it++)
 if (*it==2)
 *it=4;
- stack, queue, heap,
 - map, set, vector...

Fiche de référence (4/5)

Entrées/Sorties

```

— #include <iostream>
using namespace std;
...
cout << "I=" << i << endl;
cin >> i >> j;
— #include <fstream>
using namespace std;
ofstream f("hop.txt");
f << 1 << ' ' << 2.3;
f.close();
ifstream g("hop.txt");
if (!g.is_open()) {
    return 1;
}
int i;
double x;
g >> i >> x;
g.close();
— do {
    ...
} while (!(g.eof()));
— ofstream f;
f.open("hop.txt");
— double x[10], y;
ofstream f("hop.bin",
    ios::binary);
f.write((const char*)x,
    10*sizeof(double));
f.write((const char*)&y,
    sizeof(double));
f.close();
ifstream g("hop.bin",
    ios::binary);
g.read((char*)x,
    10*sizeof(double));
g.read((const char*)&y,
    sizeof(double));
g.close();
— string s;
ifstream f(s.c_str());
— #include <sstream>
using namespace std;
stringstream f;
// Chaîne vers entier
f << s;
f >> i;
// Entier vers chaîne
f.clear();
f << i;
f >> s;
— ostream& operator<< (
    ostream& f,

```

```

    const point&p) {
        f<<p.x<<' ' << p.y;
        return f;
    }
istream& operator>>(
    istream& f, point& p) {
        f>>p.x>>p.y;
        return f;
    }

```

Template

— Fonctions :

```

// A mettre dans le
// fichier qui utilise
// ou dans un .h
template <typename T>
T maxi(T a, T b) {
    ...
}
...
// Le type est trouvé
// tout seul!
maxi(1,2); //int
maxi(.2,.3); //double
maxi("a","c"); //string

```

— Objets :

```

template <typename T>
class paire {
    T x[2];
public:
    paire() {}
    paire(T a, T b) {
        x[0]=a; x[1]=b;
    }
    T add() const;
};
...
template <typename T>
T paire<T>::add() const {
    return x[0]+x[1];
}
...
// Le type doit être
// précisé!
paire<int> a(1,2);
int s=a.somme();
paire<double> b;
...

```

— Multiples :

```

template <typename T,
    typename S>
class hop {
    ...

```

```

};
...
hop<int, string> A;
...
— Entiers :
template <int N>
class hop {
    ..
};
...
hop<3> A;
...

```

Conseils

- Nettoyer en quittant.
- Erreurs et warnings : cliquer.
- Indenter.
- Ne pas laisser de warning.
- Utiliser le debugueur.
- Faire des fonctions.
- Tableaux : pas pour transcrire une formule mathématique !
- Faire des structures.
- Faire des fichiers séparés.
- Le .h doit suffire à l'utilisateur (qui ne doit pas regarder le .cpp)
- Ne pas abuser du récursif.
- Ne pas oublier delete.
- Compiler régulièrement.
- #include <cassert>


```
...
assert(x!=0);
y=1/x;
```
- Faire des objets.
- Ne pas toujours faire des objets !
- Penser interface / implémentation / utilisation.

Clavier

- Debug : F5 
- Step over : F10 
- Step inside : F11 
- Indent : Ctrl+A, Ctrl+I
- Step out : Maj+F11 
- Gest. tâches : Ctrl+Maj+Ech

Fiche de référence (5/5)

Divers

```

— i++;
— i--;
— i-=2;
— j+=3;

— j=i%n; // Modulo

— #include <cstdlib>
...
i=rand()%n;
x=rand()/
  double(RAND_MAX);

— #include <ctime>
// Un seul appel
srand((unsigned int)
  time(0));

— #include <cmath>
double sqrt(double x);
double cos(double x);
double sin(double x);
double acos(double x);

— #include <string>
using namespace std;
string s="hop";
char c=s[0];
int l=s.size();
if (s1==s1) ...
if (s1!=s2) ...
if (s1<s2) ...
size_t i=s.find('h'),
  j=s.find('h',3);
k=s.find("hop");
l=s.find("hop",3);
a="comment";
b="ça va?";
txt=a+" "+b;
s1="un deux trois";
s2=string(s1,3,4);
getline(cin,s);
getline(cin,s,':');
const char *t=s.c_str();

— #include <ctime>
s=double(clock())
  /CLOCKS_PER_SEC;

— #include <cmath>
double pi=M_PI;

— Opérateurs binaires
  and :      a&b
  or :       a|b
  xor :      a^b
  right shift : a>>n
  left shift : a<<n
  complement : ~a

```

exemples :

```

set(i,1):  i|=(1<<n)
reset(i,1): i&=~(1<<n)
test(i,1): if (i&(1<<n))
flip(i,1): i^=(1<<n)

```

Erreurs fréquentes

— Pas de définition de fonction dans une fonction !

```

— int q=r=4; // NON!
— if (i=2) // NON!
  if i==2 // NON!
  if (i==2) then // NON!
— for(int i=0,i<100,i++)
  // NON!

— int f() {...}
  int i=f; // NON!

— double x=1/3; // NON!
  int i,j;
  x=i/j; // NON!
  x=double(i/j); //NON!

— double x[10],y[10];
  for(int i=1;i<=10;i++)
    y[i]=2*x[i]; //NON

— int n=5;
  int t[n]; // NON

— int f()[4] { // NON!
  int t[4];
  ...
  return t; // NON!
}
int t[4]; t=f();

— int s[3]={1,2,3},t[3];
  t=s; // NON!

— int t[2];
  t={1,2}; // NON!

— struct Point {
  double x,y;
} // NON!

— Point a;
  a={1,2}; // NON!

— #include "tp.cpp" //NON
— int f(int t[][]); //NON
  int t[2,3]; // NON!
  t[i,j]=...; // NON!

— int* t;
  t[1]=...; // NON!

— int* t=new int[2];
  int* s=new int[2];
  s=t; // On perd s!
  delete[] t;
  delete[] s; //Déjà fait

```

```

— int *t,s; // s est int
           // non int*
  t=new int[n];
  s=new int[n]; // NON!

— class vec {
  int x,y;
public:
  ...
};
...
  vec a={2,3}; // NON

— vec v=vec(1,2); //NON
  vec v(1,2); // OUI

— obj* t=new obj[n];
  delete t; // manque []

— //NON!
  void f(int a=2,int b);

— void f(int a,int b=0);
  void f(int a); // NON!

— Ne pas tout mettre inline!

— int f() {
  ...
}
f()=3; // HORREUR!

— int& f() {
  int i;
  return i;
}
f()=3; // NON!

— if (i>0 & i<n) // NON
  if (i<0 | i>n) // NON

— if (...) {
  ...
  if (...)
    break; // Non,
    // boucles seulement
}

— for (i ...)
  for (j ...) {
  ...
  if (...)
    break; //NON, quitte
    // juste boucle j

— int i;
  double x;
  j=max(i,0); //OK
  y=max(x,0); //NON!
  // 0.0 et non 0: max
  // est un template STL

```

Imagine++

— Voir documentation...