LA PROGRAMMATION POUR...

☑ les élèves ingénieurs

☑ *. . . ou les collégiens*

☑ débutants

☑ *. . . ou confirmés*

Cours de l'École des ponts - 2007/2008
Renaud Keriven
CERTIS - ENPC
keriven@certis.enpc.fr

Version électronique et programmes téléchargeables sur
`http://certis.enpc.fr/~keriven/Info/`

Recommandation :

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

A méditer :

– *"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 ? . . . 9
1.2 Comment apprendre ? . . . 10
1.2.1 Choix du langage . . . 10
1.2.2 Choix de l'environnement . . . 10
1.2.3 Principes et conseils . . . 11

2 Bonjour, Monde ! **13**
2.1 L'ordinateur . . . 15
2.1.1 Le micro-processeur . . . 15
2.1.2 La mémoire . . . 17
2.1.3 Autres Composants . . . 18
2.2 Système d'exploitation . . . 20
2.3 La Compilation . . . 21
2.4 L'environnement de programmation . . . 22
2.4.1 Noms de fichiers . . . 22
2.4.2 Debuggeur . . . 23
2.4.3 TP . . . 23

3 Premiers programmes **25**
3.1 Tout dans le `main()` ! . . . 25
3.1.1 Variables . . . 25
3.1.2 Tests . . . 29
3.1.3 Boucles . . . 31
3.1.4 Récréations . . . 32
3.2 Fonctions . . . 34
3.2.1 Retour . . . 37
3.2.2 Paramètres . . . 38
3.2.3 Passage par référence . . . 39
3.2.4 Portée, Déclaration, Définition . . . 42
3.2.5 Variables locales et globales . . . 43
3.2.6 Surcharge . . . 44
3.3 TP . . . 44
3.4 Fiche de référence . . . 45

4 Les tableaux **47**
4.1 Premiers tableaux . . . 47
4.2 Initialisation . . . 49
4.3 Spécificités des tableaux . . . 50

4.3.1 Tableaux et fonctions . . . 50
4.3.2 Affectation . . . 52
4.4 Récréations . . . 53
4.4.1 Multi-balles . . . 53
4.4.2 Avec des chocs ! . . . 55
4.4.3 Mélanger les lettres . . . 57
4.5 TP . . . 59
4.6 Fiche de référence . . . 59

5 Les structures **63**
5.1 Révisions . . . 63
5.1.1 Erreurs classiques . . . 63
5.1.2 Erreurs originales . . . 63
5.1.3 Conseils . . . 64
5.2 Les structures . . . 65
5.2.1 Définition . . . 65
5.2.2 Utilisation . . . 66
5.3 Récréation : TP . . . 67
5.4 Fiche de référence . . . 68

6 Plusieurs fichiers ! **71**
6.1 Fichiers séparés . . . 72
6.1.1 Principe . . . 72
6.1.2 Avantages . . . 73
6.1.3 Utilisation dans un autre projet . . . 74
6.1.4 Fichiers d'en-têtes . . . 74
6.1.5 A ne pas faire... . . . 76
6.1.6 Implémentation . . . 77
6.1.7 Inclusions mutuelles . . . 77
6.2 Opérateurs . . . 78
6.3 Récréation : TP suite et fin . . . 79
6.4 Fiche de référence . . . 79

7 La mémoire **83**
7.1 L'appel d'une fonction . . . 83
7.1.1 Exemple . . . 83
7.1.2 Pile des appels et débuggeur . . . 85
7.2 Variables Locales . . . 87
7.2.1 Paramètres . . . 87
7.2.2 La pile . . . 87
7.3 Fonctions récursives . . . 88
7.3.1 Pourquoi ça marche ? . . . 88
7.3.2 Efficacité . . . 89
7.4 Le tas . . . 90
7.4.1 Limites . . . 91
7.4.2 Tableaux de taille variable . . . 91
7.4.3 Essai d'explication . . . 92
7.5 L'optimiseur . . . 93
7.6 TP . . . 93

7.7 Fiche de référence . . . 94
7.8 Examens sur machine . . . 96

8 Allocation dynamique **97**
8.1 Tableaux bidimensionnels . . . 97
8.1.1 Principe . . . 97
8.1.2 Limitations . . . 97
8.1.3 Solution . . . 98
8.2 Allocation dynamique . . . 99
8.2.1 Pourquoi ça marche ? . . . 100
8.2.2 Erreurs classiques . . . 101
8.2.3 Conséquences . . . 101
8.3 Structures et allocation dynamique . . . 103
8.4 Boucles et `continue` . . . 105
8.5 TP . . . 106
8.6 Fiche de référence . . . 106

9 Premiers objets **109**
9.1 Philosophie . . . 109
9.2 Exemple simple . . . 110
9.3 Visibilité . . . 112
9.4 Exemple des matrices . . . 112
9.5 Cas des opérateurs . . . 114
9.6 Interface . . . 116
9.7 Protection . . . 117
9.7.1 Principe . . . 117
9.7.2 Structures vs Classes . . . 119
9.7.3 Accesseurs . . . 119
9.8 TP . . . 120
9.9 Fiche de référence . . . 120

10 Constructeurs et Destructeurs **125**
10.1 Le problème . . . 125
10.2 La solution . . . 126
10.3 Cas général . . . 126
10.3.1 Constructeur vide . . . 126
10.3.2 Plusieurs constructeurs . . . 128
10.3.3 Tableaux d'objets . . . 129
10.4 Objets temporaires . . . 130
10.5 TP . . . 131
10.6 Références Constantes . . . 131
10.6.1 Principe . . . 131
10.6.2 Méthodes constantes . . . 133
10.7 Destructeur . . . 134
10.8 Destructeurs et tableaux . . . 136
10.9 Constructeur de copie . . . 136
10.10 Affectation . . . 137
10.11 Objets avec allocation dynamique . . . 138
10.11.1 Construction et destruction . . . 138

10.11.2 Problèmes ! . . . 139
10.11.3 Solution ! . . . 140
10.12 Fiche de référence . . . 142
10.13 Devoir écrit . . . 145

11 En vrac... 147
11.1 Chaînes de caratères . . . 147
11.2 Fichiers . . . 149
11.2.1 Principe . . . 149
11.2.2 Chaînes et fichiers . . . 150
11.2.3 Objets et fichiers . . . 150
11.3 Valeurs par défaut . . . 151
11.3.1 Principe . . . 151
11.3.2 Utilité . . . 152
11.3.3 Erreurs fréquentes . . . 152
11.4 Accesseurs . . . 152
11.4.1 Référence comme type de retour . . . 153
11.4.2 Utilisation . . . 153
11.4.3 `operator()` . . . 154
11.4.4 Surcharge et méthode constante . . . 154
11.4.5 "inline" . . . 155
11.5 Assertions . . . 157
11.6 Types énumérés . . . 157
11.7 Fiche de référence . . . 158

12 En vrac (suite) ... 163
12.1 Opérateur binaires . . . 163
12.2 Valeur conditionnelle . . . 164
12.3 Boucles et `break` . . . 165
12.4 Variables statiques . . . 166
12.5 `const` et tableaux . . . 167
12.6 `template` . . . 167
12.6.1 Principe . . . 167
12.6.2 `template` et fichiers . . . 169
12.6.3 Classes . . . 169
12.6.4 STL . . . 172
12.7 Fiche de référence . . . 174
12.8 Devoir final . . . 179

A Travaux Pratiques 181
A.1 L'environnement de programmation . . . 181
A.1.1 Bonjour, Monde ! . . . 181
A.1.2 Premières erreurs . . . 183
A.1.3 Debugger . . . 185
A.1.4 Deuxième programme . . . 186
A.1.5 S'il reste du temps . . . 186
A.1.6 Installer Visual Studio chez soi . . . 186
A.2 Variables, boucles, conditions, fonctions . . . 187
A.2.1 Premier programme avec fonctions . . . 187

A.2.2 Premier programme graphique avec la CLGraphics . . . 187
A.2.3 Jeu de Tennis . . . 189
A.3 Tableaux . . . 191
A.3.1 Mastermind Texte . . . 191
A.3.2 Mastermind Graphique . . . 193
A.4 Structures . . . 195
A.4.1 Etapes . . . 195
A.4.2 Aide . . . 197
A.4.3 Théorie physique . . . 198
A.5 Fichiers séparés . . . 200
A.5.1 Fonctions outils . . . 200
A.5.2 Vecteurs . . . 200
A.5.3 Balle à part . . . 201
A.5.4 Retour à la physique . . . 201
A.6 Les tris . . . 203
A.6.1 Mélanger un tableau . . . 203
A.6.2 Tris quadratiques . . . 204
A.6.3 Quicksort . . . 204
A.6.4 Gros tableaux . . . 205
A.7 Images . . . 207
A.7.1 Allocation . . . 207
A.7.2 Tableaux statiques . . . 207
A.7.3 Tableaux dynamiques . . . 208
A.7.4 Charger un fichier . . . 208
A.7.5 Fonctions . . . 208
A.7.6 Structure . . . 209
A.7.7 Suite et fin . . . 209
A.8 Premiers objets et dessins de fractales . . . 210
A.8.1 Le triangle de Sierpinski . . . 210
A.8.2 Une classe plutôt qu'une structure . . . 211
A.8.3 Changer d'implémentation . . . 211
A.8.4 Le flocon de neige . . . 211
A.9 Tron . . . 212
A.9.1 Serpent . . . 212
A.9.2 Tron . . . 213
A.9.3 Graphismes . . . 213

B Examens 215
B.1 Examen sur machine 2003 : énoncé . . . 215
B.1.1 Crible d'Ératosthène . . . 215
B.1.2 Calcul de π par la méthode de Monte Carlo . . . 215
B.1.3 Serpent . . . 217
B.2 Examen sur machine 2004 : énoncé . . . 220
B.2.1 Calcul de l'exponentielle d'un nombre complexe . . . 220
B.2.2 Compression RLE . . . 220
B.3 Examen sur machine 2005 : énoncé . . . 223
B.3.1 Construction du Modèle 3D . . . 223
B.3.2 Projection : du 3D au 2D . . . 223
B.3.3 Affichage à l'écran . . . 224

B.3.4 Animation du tétraèdre . . . 224
B.3.5 Un modèle plus élaboré . . . 225
B.4 Examen sur machine 2006 : énoncé . . . 226
B.4.1 Voyageur de commerce par recuit simulé . . . 226
B.4.2 Travail demandé . . . 226
B.5 Devoir écrit 2003 : énoncé . . . 227
B.5.1 Tableau d'exécution . . . 227
B.5.2 Grands entiers . . . 228
B.5.3 Constructeurs . . . 228
B.6 Devoir écrit 2004 : énoncé . . . 231
B.6.1 Tableau d'exécution . . . 231
B.6.2 Constructeurs . . . 232
B.6.3 Le compte est bon . . . 234
B.7 Devoir écrit 2006 : énoncé . . . 237
B.7.1 Enoncé – Tours de Hanoi . . . 237
B.8 Devoir final 2003 : énoncé . . . 240
B.8.1 Tableau d'exécution . . . 240
B.8.2 Erreurs . . . 241
B.8.3 Qu'affiche ce programme ? . . . 242
B.8.4 Le jeu du Pendu . . . 244
B.8.5 Programme mystère . . . 245
B.9 Devoir final 2004 : énoncé . . . 247
B.9.1 Erreurs . . . 247
B.9.2 Qu'affiche ce programme ? . . . 248
B.9.3 Chemins dans un graphe . . . 251
B.9.4 Tours de Hanoï . . . 252
B.9.5 Table de hachage . . . 255
B.10 Devoir final 2005 : énoncé . . . 257
B.10.1 Erreurs à corriger . . . 257
B.10.2 Qu'affiche ce programme ? . . . 257
B.10.3 Tableau d'exécution . . . 259
B.10.4 Résolveur de Sudoku . . . 260
B.11 Devoir final 2006 : énoncé . . . 263
B.11.1 Erreurs à corriger . . . 263
B.11.2 Qu'affiche ce programme ? . . . 264
B.11.3 Tableau d'exécution . . . 266
B.11.4 Huit dames . . . 268

C La CLGraphics 271

D Fiche de référence finale 273

Chapitre 1

Préambule

Note : Ce premier chapitre maladroit 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 faillait 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 renfermement[5] 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 :
  - 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.

[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.

    - oublie qu'il a le plus souvent appris seul quand il était plus jeune, en programmant des choses simples et ludiques[7]. Il devrait donc faire venir ses élèves à la programmation par le côté ludique, et non avec les mêmes sempiternels exemples[8].
- 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 modernes[9]. 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 logiciels[11] et en oublie qu'il pourrait lui aussi en créer. En conclusion, collégiens ou ingénieurs, débutants ou passionées, **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.

[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 !

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 aie été conçu, entre autres, dans un soucis de simplification du C++[12], nous préfèrerons 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 débutant[13]. 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 efficace[14].
3. C++ est souvent incontournable dans certains milieux, par exemple en finance.
4. Enfin, certains aspects pratiques et pourtant simples de C++ ont disparu dans Java[15].

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 prônons aucun en particulier[16]. Ceci dit, pour des raisons pédagogiques, nous pensons

[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.

[15]Les opérateurs par exemple.

[16]L'idéal est en fait d'avoir les deux "sous la main".

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, debuggeur, 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.

Sans arrière pensée de principe, nous avons opté pour l'environnement de Microsoft, Visual Studio, à la fois simple et puissant. Il est le plus utilisé des produits commerciaux. Il en existe quelques équivalents gratuits sous linux, mais pas encore suffisamment aboutis pour nous faire hésiter. C'est donc le choix de Visual Studio et ce choix seul qui est la raison de l'utilisation de Windows au détriment de linux... Mieux encore, il existe maintenant une version de Visual gratuite : Visual Express. 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.

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**[17] (de la machine et de son programme). Que programmer soit

[17] 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 !

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.

5. **Debugger**. Souvent, la connaissance du debuggeur (l'outil pour rechercher les bugs) est négligée et son apprentissage est repoussé au stade avancé. Il s'agit pourtant d'un outil essentiel 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 :

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    cout << "Hello, World!" << endl;
7    return 0;
8  }
```

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 automatiquement[4] quand le pro-

[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*. A 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()` 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 ?)

gramme *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 :

[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 calculer[10], 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 accessible[11] 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'un 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 soit[12] et quelle que soit sa vitesse[13], 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 mémoire[14] 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.

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 :

[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).

```
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
```

Sous l'environnement Visual Studio que nous utiliserons, 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 nombres[15] correspond évidemment à une instruction précise. C'est tout de suite moins compréhensible[16] ! 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]
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...

[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 !

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 octet[19] correspond à un nombre binaire de 8 bits[20], 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 adresse[22]. Ainsi, notre fameux `c=3*a+2*b;` devient en mémoire :

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

[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 agés 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.

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 elles stockées quelque part en mémoire sur un nombre d'octets suffisant[23] 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 main[24]. Bref, on a en plus[25] :

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

où les octets $a_1, ..., a_4$ combinés donnent l'entier `a` sur 32 bits. Certains processeurs (dits *little-endians* décident $a = a_1a_2a_3a_4$, d'autres (*big-endians*) 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 !

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 rapide[26] 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

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

l'ordinateur est éteint mais qui en contre-partie ne peut être modifiée[27]. 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 fois[28] ! 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-cassettes[29], disquettes[30], disques durs[31] ou bandes magnétiques. Aujourd'hui, on peut rajouter les CD, DVD, les cartes mémoire, les "clés USB", etc, etc.

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 !

[27]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 règlage sont en CMOS. Attention à l'usure de la pile !

[28]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 !

[29]Très lent et très peu fiable, mais le quotidien des ordinateurs personnels.

[30]Le luxe. Un lecteur de 40Ko coûtait 5000F !

[31]Les premiers étaient de véritables moteurs de voiture, réservés aux importants centres de calcul.

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 données[32], d'autres des programmes[33], d'autres encore contenaient eux-mêmes des fichiers[34].
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)[35], enfin avec plusieurs processeurs par machine.

Pour gérer tout cela, s'est dégagé le concept de *système d'exploitation*[36]. 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 utilisateurs[37], mais son rôle le plus délicat est de gérer les programmes (ou tâches ou *process*) en train de s'exécuter. Il doit pour cela essentiellement faire face à deux problèmes[38] :

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.

[32]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.

[33]On parle de fichier *exécutable*...

[34]Les répertoires.

[35]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*) !

[36]Operating System

[37]Espérons qu'un jour les utilisateurs ne seront pas eux-aussi des périphériques !

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

Lorsqu'un process à 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 mémoire[39].

Avec l'arrivée des systèmes d'exploitation, les fichiers exécutables ont du s'adapter pour de nombreuse 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 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 mesure[40] : 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 virtuel[41], 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**[42]. Un *langage compilé* est alors à opposer à un *langage interprété*. Dans le cas du

[39] Il se contente de modifier anarchiquement ses données, ce qui est déjà pas mal !

[40] même s'il est parfois pré-traité pour accélérer l'exécution.

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

[42] 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

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`, `min()`, etc. L'ensemble de ces instructions constitue ce qu'on appelle une *bibliothèque*[43].
- 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.**

2.4 L'environnement de programmation

L'environnement de programmation est le logiciel permettant de programmer. Dans notre cas il s'agit de Visual Studio Express. 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 *debuggeur* 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. 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

Sous Windows, l'*extension* (le suffixe) sert à se repérer dans les types de fichier :

- Un fichier source C++ se terminera par `.cpp`[44].
- Un fichier objet sera en `.obj`
- Un fichier exécutable en `.exe`

Nous verrons aussi plus loin dans le cours :

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 !

[43] 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.

[44] Un fichier en `.c` sera considéré comme du C. Différence avec linux : un fichier en `.C` sera aussi traité comme du C et non comme du C++ !

- 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) : fichier `.lib` ou `.dll`

2.4.2 Debuggeur

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 *debuggeur*[45].

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 debuggeur peut aussi inspecter les appels de fonctions, espionner la modification d'une variable, etc.

2.4.3 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 Visual sur votre ordinateur (lien `http://certis.enpc.fr/~keriven/CertisLibs` mentionné à la fin du TP). Voici donc ce qu'il faut retenir du TP :

[45] Débogueur (F) !

1. **Toujours travailler en local (bureau ou disque D :) et sauvegarder sur le disque partagé (Z :)**
2. **Type de projet utilisé :** `CertisLibs Project`
3. **Nettoyer ses solutions 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 debuggeur.**
9. **Touches utiles :**

F7	=		=	**Build**
Ctrl+F5	=		=	**Start without debugging**
Ctrl+F7	=		=	**Compile only**
F5	=		=	**Start debugging**
Maj+F5	=		=	**Stop**
F10	=		=	**Step over**
F11	=		=	**Step inside**
Ctrl+K,Ctrl+F	=			**Indent selection**

—

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

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 fois[1] ! 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 programme[2].

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 déjà rencontré les `int` qui sont le plus souvent aujourd'hui stockés sur quatre octets, soit

[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 !

32 bits, et pouvant prendre $2^{32} = 4294967296$ valeurs possibles[3]. Par convention, les `int` stockent les nombres entiers relatifs[4], avec autant de nombres négatifs que de nombres positifs[5], soit, dans le cas de 32 bits[6], de -2147483648 à 2147483647 suivant une certaine correspondance avec le binaire[7].

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!
```

Dans ce programme :

---

[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 : $0 = 000...000, 1 = 000...001, 2147483647 = 011...111, -2147483648 = 100...000, -2147483647 = 100..001, -2 = 111...110, -1 = 111...111$

- 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 affiche[10] :

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

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 :

> **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 :

[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 sa 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

```
1        int i;
2        i=j; // Erreur: j n'existe pas encore!
3        int j=2;
4        if (j>1) {
5            int k=3;
6            j=k;
7        }
8        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 :

```
1        int i=3;            // Entier relatif
2        double x=12.3;      // Nombre réel (double précision)
3        char c='A';         // Caractère
4        string s="hop";     // Chaîne de caractères
5        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 caractères[11]. Enfin, les booléens sont des variables qui valent vrai (`true`) ou faux (`false`).

Voici, pour information, quelques types supplémentaires :

```
6        float y=1.2f;              // Nombre réel simple précision
7        unsigned int j=4;          // Entier naturel
8        signed char d=-128;        // Entier relatif un octet
9        unsigned char d=254;       // Entier naturel un octet
10       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 `float` valent au plus `FLT_MAX` (ici, environ `3.4e+38`[12]) et que leur valeur la plus petite strictement positive est `FLT_MIN` (ici, environ `1.2e-38`), de même que pour les `double` les constantes `DBL_MAX` et `DBL_MIN` valent ici environ `1.8e+308` et `2.2e-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 complexes[13].

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

[12] **Coin des collégiens :** 10^{38} ou `1e+38` vaut 1 suivi de 38 zéros, 10^{-38} ou `1e-38` vaut 0.000...01 avec 37 zéros avant le 1. En compliquant : `3.4e+38` vaut 34 suivis de 37 zéros (38 chiffres après le 3) et `1.2e-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 parait important

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 :

```
1       if (i==0) // i est-il nul?
2           cout << "i est nul" << endl;
3       ...
4       if (i>2)  // i est-il plus grand que 2?
5           j=3;
6       else
7           j=5;  // Si on est ici, c'est que i<=2
8       ...
9       // Cas plus compliqué!
10      if (i!=3 || (j==2 && k!=3) || !(i>j && i>k)) {
11          // Ici, i est différent de 3 ou alors
12          // j vaut 2 et k est différent de 3 ou alors
13          // on n'a pas i plus grand a la fois de j et de k
14          cout << "Une première instruction" << endl;
15          cout << "Une deuxième instruction" << endl;
16      }
```

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

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

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

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.

[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 !

multiple. Enfin, le *mot clé* default, à placer en dernier, correspond aux cas non précisés. Le programme suivant[16] 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;
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 précédent ceci est équivalent à[17] :

```
11         if (c=='a')
12             cout << "Vous avez tapé 'a'!" << endl;
```

[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>.

[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 connaîtrons de C++, plus nous devrons 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.

```
13          else if (c=='f') {
14              cout << "Vous avez tapé 'f'. Au revoir!" << endl;
15              fini=true;
16          } else if (c=='e' || c=='i' || c=='o' || c=='u' || c=='y')
17              cout << "Vous avez tapé une autre voyelle!" << endl;
18          else
19              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` !

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: ";
9           cin >> i;
10      } while (i<1 || i>10); // On 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 :

```
1       int i=1;
2       while (i<=100) {
3           cout << i << endl;
4           i=i+1;
5       }
```

Enfin, on a crée 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 :

```
1       int i;
2       for (i=1;i<=100;i=i+1) {
3           cout << i << endl;
4       }
```

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 :

```
1       for (int i=1;i<=100;i++)
2           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] :

```
1       for (int i=1,j=100;j>i;i=i+2,j=j-3)
2           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;
```

[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.

```
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 modulo[21].

C'est évidemment plus intéressant, surtout à programmer, 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
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 triche!" << 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 C pour les instructions graphiques...)

[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.

```
1   #include <CL/Graphics/Graphics.h>
2   using namespace CL::Graphics;
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      Terminate();
26      return 0;
27  }
```

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 fatiguante.

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

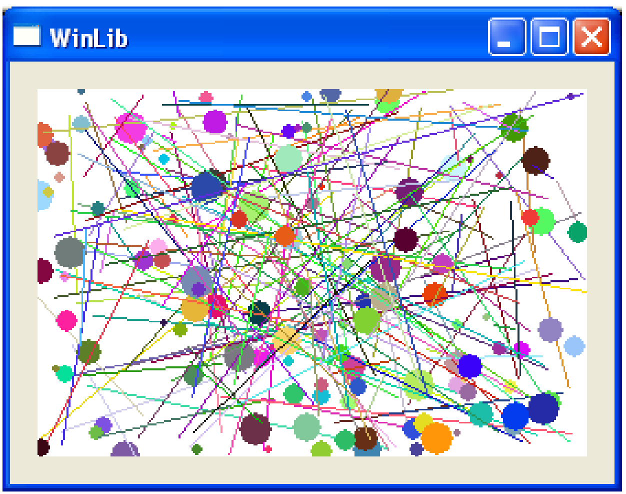


Fig. 3.1 – Traits et cercles au hasard...

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 <CL/Graphics/Graphics.h>
2    using namespace CL::Graphics;
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;
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       Terminate();
23       return 0;
24   }
```

La première chose qui choque[22], 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 !

Bref, notre programme devient :

```
6   // Nombre entre 0 et n-1
7   int hasard(int n)
8   {
9       return rand()%n;
10  }
11
12  int main()
13  {
14      const int w=300,h=200;
15      OpenWindow(w,h);
16      for (int i=0;i<150;i++) {
17          int x1=hasard(w),y1=hasard(h); // Point initial
18          int x2=hasard(w),y2=hasard(h); // Point final
19          Color c=Color(hasard(256),hasard(256),hasard(256));
20          DrawLine(x1,y1,x2,y2,c); // Tracé de segment
21          int xc=hasard(w),yc=hasard(h); // Centre du cercle
22          int rc=hasard(w/20);  // Rayon
23          Color cc=Color(hasard(256),hasard(256),hasard(256));
24          FillCircle(xc,yc,rc,cc); // Cercle
25      }
26      Terminate();
27      return 0;
28  }
```

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

[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...

des nombres au hasard avec une autre fonction[23], 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 :

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 :

```
1   void dis_bonjour_a_la_dame(string nom_de_la_dame) {
2       cout << "Bonjour, Mme " << nom_de_la_dame << "!" << endl;
3   }
4   ...
5     dis_bonjour_a_la_dame("Germaine");
6     dis_bonjour_a_la_dame("Fitzgerald");
7   ...
```

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 :

```
1   int signe_avec_un_seul_return(double x) {
2       int s;
3       if (x==0)
4           s=0;
5       else if (x<0)
6           s=-1;
7       else
8           s=1;
9       return s;
10  }
11
12  int signe_plus_simple(double x) {
13      if (x<0)
14          return -1;
15      if (x>0) // Notez l'absence de else, devenu inutile!
16          return 1;
17      return 0;
```

[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()`.

[24]Contrairement à certains vieux langages, comme le Pascal

```
18  }
```

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

```
1   void telephoner_avec_un_seul_return(string nom) {
2       if (j_ai_le_telephone) {
3           if (mon telephone_marche) {
4               if (est_dans_l_annuaire(nom)) {
5                   int numero=numero_telephone(nom);
6                   composer(numero);
7                   if (ca_decroche) {
8                       parler();
9                       raccrocher();
10                  }
11              }
12          }
13      }
14  }
15  void telephoner_plus_simple(string nom) {
16      if (!j_ai_le_telephone)
17          return;
18      if (!mon telephone_marche)
19          return;
20      if (!est_dans_l_annuaire(nom))
21          return;
22      int numero=numero_telephone(nom);
23      composer(numero);
24      if (!ca_decroche)
25          return;
26      parler();
27      raccrocher();
28  }
```

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 :

```
1   // Nombre entre a et b
2   int hasard2(int a,int b)
3   {
4       return a+(rand()%(b-a+1));
5   }
6
7   // Nombre entre 0 et 1
8   double hasard3()
9   {
10      return rand()/double(RAND_MAX);
11  }
```

```
12  ...
13      int a=hasard2(1,10);
14      double x=hasard3();
15  ...
```

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`[a]`(i/j);`

[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 :

```
1  void triple(int x) {
2      x=x*3;
3  }
4  ...
5      int a=2;
6      triple(a);
7      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 :

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

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

```
1   void echanger1(int x,int y) {
2       int t=x;
3       x=y;
4       y=t;
5   }
6   void echanger2(int& x,int& y) {
7       int t=x;
8       x=y;
9       y=t;
10  }
11  ...
12      int a=2,b=3;
13      echanger1(a,b);
14      cout << a << " " << b << " ";
15      echanger2(a,b);
16      cout << a << " " << b << endl;
17  ...
```

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 :

```
1   void un_point(int& x, int& y) {
2       x=...;
3       y=...;
4   }
5   ...
6       int a,b;
7       un_point(a,b);
8   ...
```

Ainsi, notre programme de dessin aléatoire deviendrait :

```
1   #include <CL/Graphics/Graphics.h>
2   using namespace CL::Graphics;
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(hasard(256),hasard(256),hasard(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         Terminate();
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 :

```
1   void f()
2   {
3       g(); // Erreur: g() inconnue
4   }
5
6   void g() {
7       f();
8   }
```

puisque les deux fonctions on 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 :

```
1   void g();  // Déclaration de g
2
3   void f()
4   {
5       g();   // OK: fonction déclarée
6   }
7
8   void g() { // Définition de g
9       f();
10  }
```

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 :

```
1  void f()
2  {
3      int x;
4      x=3;
5  }
6
7  void g() {
8      int y;
9      y=x; // Erreur: x inconnu
10 }
```

Si vraiment deux fonctions doivent utiliser 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

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* :

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   {
4       return rand()%n;
5   }
6   // Nombre entre a et b
7   int hasard(int a,int b)
8   {
9       return a+(rand()%(b-a+1));
10  }
11  // Nombre entre 0 et 1
12  double hasard()
13  {
14      return rand()/double(RAND_MAX);
15  }
16  ...
17     int i=hasard(3);  // entre 0 et 2
18     int j=hasard(2,4) // entre 2 et 4
19     double k=hasard();// entre 0 et 1
20  ...
```

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).

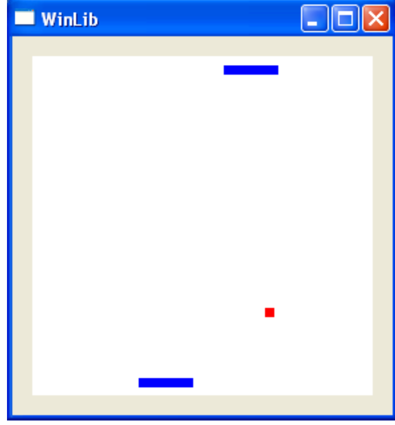


Fig. 3.2 – Mini tennis...

3.4 Fiche de référence

Nous commençons maintenant à nous fabriquer une "*fiche de référence*" qui servira d'aide mémoire lorsque l'on est devant la machine. Nous la compléterons après chaque chapitre. Elle contient ce qui est vu pendant le chapitre, mais aussi pendant le TP correspondant. Les nouveautés par rapport à la fiche précédente seront en rouge. La fiche finale est en annexe D.

Fiche de référence (1/2)

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=254;
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);
int i,j;
double x=double(i)/j;
```

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:
    ...;
}
```

Boucles

–
```
do {
  ...
} while (!ok);
```
–
```
int i=1;
while (i<=100) {
   ...
   i=i+1;
}
```
–
```
for (int i=1;i<=100;i++)
  ...
```
–
```
for (int i=1,j=100;j>i;
     i=i+2,j=j-3)
  ...
```

Fiche de référence (2/2)

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();
```

Divers

– `i++;`
  `i--;`
  `i-=2;`
  `j+=3;`
– `j=i%n; // Modulo`
– `#include <cstdlib>`
```
...
i=rand()%n;
x=rand()/double(RAND_MAX);
```

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;
double x;
x=i/j; // NON!
x=double(i/j); //NON!
```

CLGraphics

– Voir documentation...

Clavier

– Build : F7
– Start : Ctrl+F5
– Compile : Ctrl+F7
– Debug : F5
– Stop : Maj+F5
– Step over : F10
– Step inside : F11
– Indent : Ctrl+K,Ctrl+F

Conseils

– Travailler en local
– `CertisLibs Project`
– Nettoyer en quittant.
– Erreurs et warnings : cliquer.
– Indenter.
– Ne pas laisser de warning.
– Utiliser le debuggeur.
– 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 :

```
1   int x1,y1,u1,v1; // Balle 1
2   int x2,y2,u2,v2; // Balle 2
3   int x3,y3,u3,v3; // Balle 3
4   int x4,y4,u4,v4; // Balle 4
5   ...
6   BougeBalle(x1,y1,u1,v1);
7   BougeBalle(x2,y2,u2,v2);
8   BougeBalle(x3,y3,u3,v3);
9   BougeBalle(x4,y4,u4,v4);
10  ...
```

pourra avantageusement être remplacé par :

```
1   int x[4],y[4],u[4],v[4]; // Balles
2   ...
3   for (int i=0;i<4;i++)
4           BougeBalle(x[i],y[i],u[i],v[i]);
5   ...
```

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 :

```
1 double x[100],y[100],z[100];
2 ...
3 ... // ici, les x[i] et y[i] prennent des valeurs
4 ...
5 for (int i=0;i<100;i++)
6     z[i]=x[i]+y[i];
7 ...
8 ... // ici, on utilise les z[i]
9 ...
```

Il y deux choses essentielles à retenir.

1. D'abord, que

   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<=99`, ce qui est plus lisible. Mais attention à ne pas mettre `i<=100` !

2. Ensuite, qu'

   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 à

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 donnée[1], 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 ca!
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);
```

ce qui épargnera, à la machine, un tableau (donc de la mémoire et des calculs), et à vous des bugs (donc vos nerfs !).

4.2 Initialisation

Tout comme une variable, un tableau peut être initialisé :

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

Attention, la syntaxe utilisée pour l'initialisation ne marche pas pour une affectation[2] :

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

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

[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 comme toujours comme les autres variables[3]...

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 :

```
1   void affiche(int s[4]) {
2       for (int i=0;i<4;i++)
3           cout << s[i] << endl;
4   }
5   ...
6   int t[4]={1,2,3,4};
7   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 soucis 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;
16  affecte1(a,0);      // a ne sera pas mis à 0
```

[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.

```
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 somme1(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=somme1(a,b); // ERREUR
22  somme2(a,b,c); // OK
```

Enfin, et c'est utilisé tout le temps,

Une fonction n'est pas obligée de travailler sur une seule taille de tableau... mais il est impossible de retrouver la taille d'un tableau !

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 :

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

ne marche pas et on est obligé de faire :

```
1  int s[4]={1,2,3,4},t[4];
2  for (int i=0;i<4;i++)
3      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
16  ...
```

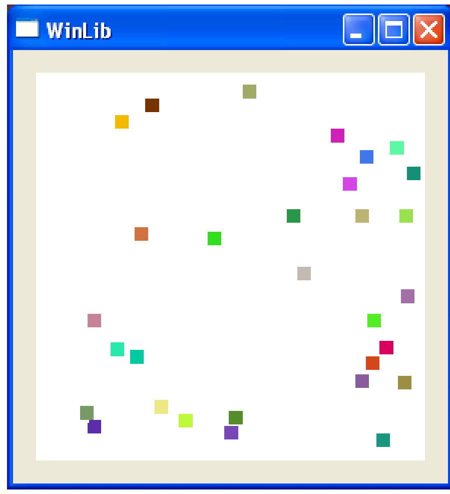


FIG. 4.1 – Des balles qui rebondissent... (momentanément figées ! Allez sur la page du cours pour un programme animé !)

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 la CLGraphics annexe C).

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

```
1   #include <CL/Graphics/Graphics.h>
2   using namespace CL::Graphics;
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        Terminate();
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 le `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, par exemple sur le très didactique laboratoire virtuel `labo.ntic.org`. 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 reconvertir 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       Terminate();
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 }
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();
```

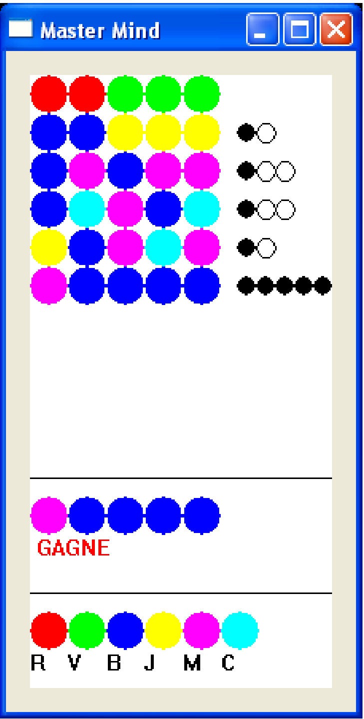


FIG. 4.2 – Master mind...

```
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 :
```
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=254;
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);
int i,j;
double x=double(i)/j;
```

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:
    ...;
}
```

Tableaux

– Définition :

  –
```
double x[10],y[10];
for (int i=0;i<10;i++)
   y[i]=2*x[i];
```
  –
```
const int n=5;
int i[n],j[2*n]; // OK
```
– Initialisation :
```
int t[4]={1,2,3,4};
string s[2]={"ab","cd"};
```
– Affectation :
```
int s[4]={1,2,3,4},t[4];
for (int i=0;i<4;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;
}
```

Boucles

–
```
do {
   ...
} while (!ok);
```
–
```
int i=1;
while (i<=100) {
   ...
   i=i+1;
}
```
–
```
for (int i=1;i<=100;i++)
   ...
```
–
```
for (int i=1,j=100;j>i;
     i=i+2,j=j-3)
   ...
```

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();
```

Fiche de référence (2/2)

Divers

```
– i++;
  i--;
  i-=2;
  j+=3;
– j=i%n; // Modulo
– #include <cstdlib>
  ...
  i=rand()%n;
  x=rand()/double(RAND_MAX);

– #include <ctime>
  ...
  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();
```

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;
  double x;
  x=i/j; // NON!
  x=double(i/j); //NON!
– double x[10],y[10];
  for (int i=1;i<=10;i++)
      // NON!
      y[i]=2*x[i];
– int n=5;
  int t[n]; // NON
– int f()[4] { // NON!
      int t[4];
      ...
      return t; // NON!
  }
  ...
  int t[4]
  t=f();
– int s[4]={1,2,3,4},t[4];
  t=s; // NON!
– int t[2];
  t={1,2}; // NON!
```

CLGraphics

– Voir documentation...

Clavier

– Build : F7
– Start : Ctrl+F5
– Compile : Ctrl+F7
– Debug : F5
– Stop : Maj+F5
– Step over : F10
– Step inside : F11
– Indent : Ctrl+K,Ctrl+F

Conseils

– Travailler en local
– `CertisLibs Project`
– Nettoyer en quittant.
– Erreurs et warnings : cliquer.
– Indenter.
– Ne pas laisser de warning.
– Utiliser le debuggeur.
– Faire des fonctions.
– Tableaux : quand c'est utile ! (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 ont fait de même pour les variables, 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 un peu 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)
     ...
```

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 debuggeur.

[1] Ne me faîtes 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 !

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.

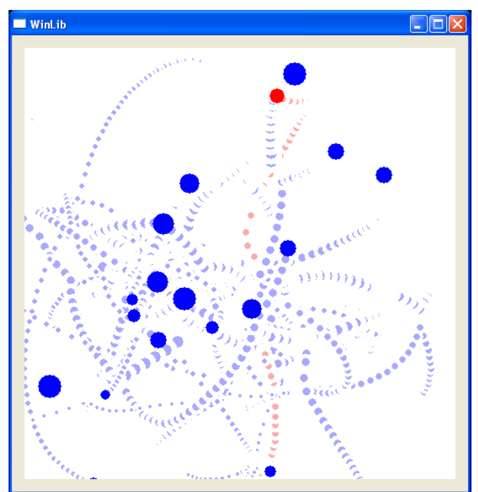
5.2.2 Utilisation

Les structures se manipulent comme les autres types[2]. 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;
    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();
...
```

[2]D'ailleurs, nous avions 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 !



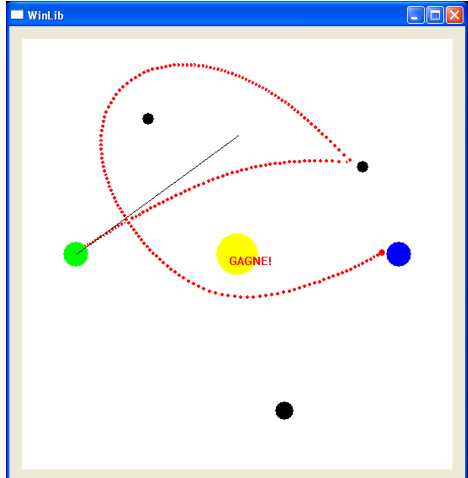


FIG. 5.1 – Corps célestes et duel...

Attention, tout comme pour les tableaux, la syntaxe utilisée pour l'initialisation ne marche pas pour une affectation[4] :

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

D'ailleurs, répétons-le :

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

5.3 Récréation : TP

Nous pouvons maintenant aller faire le TP donné en annexe A.4 afin de mieux comprendre les structures. Nous ferons même des tableaux de structures[5] ! Nous obtiendrons un projectile naviguant au milieu des étoiles puis un duel dans l'espace (figure 5.1) !

[4] La situation s'améliorera avec les objets.

[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 !

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 (1/2)

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=254;
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);
int i,j;
double x=double(i)/j;
```

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:
    ...;
}
```

Boucles

–
```
do {
    ...
} while (!ok);
```
–
```
int i=1;
while (i<=100) {
    ...
    i=i+1;
}
```
–
```
for (int i=1;i<=100;i++)
    ...
```
–
```
for (int i=1,j=100;j>i;
     i=i+2,j=j-3)
    ...
```

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();
```

Fiche de référence (2/2)

Tableaux

- Définition :
  - ```
    double x[10],y[10];
    for (int i=0;i<10;i++)
         y[i]=2*x[i];
    ```
  - ```
    const int n=5;
    int i[n],j[2*n]; // OK
    ```
- Initialisation :
  ```
  int t[4]={1,2,3,4};
  string s[2]={"ab","cd"};
  ```
- Affectation :
  ```
  int s[4]={1,2,3,4},t[4];
  for (int i=0;i<4;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;
    }
    ```

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};
  ```

Divers

- ```
  i++;
  i--;
  i-=2;
  j+=3;
  ```
- ```
  j=i%n; // Modulo
  ```
- ```
  #include <cstdlib>
  ...
  i=rand()%n;
  x=rand()/double(RAND_MAX);
  ```
- ```
  #include <ctime>
  ...
  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();
  ```

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

CLGraphics

- Voir documentation...

Clavier

- Build : F7
- Start : Ctrl+F5
- Compile : Ctrl+F7
- Debug : F5
- Stop : Maj+F5
- Step over : F10
- Step inside : F11
- Indent : Ctrl+K,Ctrl+F

Conseils

- Travailler en local
- `CertisLibs Project`
- Nettoyer en quittant.
- Erreurs et warnings : cliquer.
- Indenter.
- Ne pas laisser de warning.
- Utiliser le debuggeur.
- Faire des fonctions.
- Tableaux : quand c'est utile ! (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 passage[1] *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)

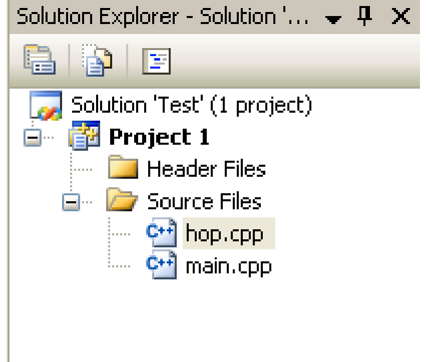


Fig. 6.1 – Plusieurs fichiers sources...

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

6.1 Fichiers séparés

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 :

- Ouvrir le menu `Project/Add New Item` ou faire `Ctrl+Maj+A` ou cliquer sur
- Choisir l'ajout d'un fichier 'Visual C++/Code/C++ file' et en préciser le nom (sans qu'il soit besoin de préciser `.cpp` dans le nom)

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 ce 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. Malheureusement[2] :

> **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 :

- Fichier `hop.cpp` :

```
1   // Définitions
2   void f(int x) {
3       ...
4   }
5   int g() {
6       ...
7   }
8   // Autres fonctions
9   ...
```

- Fichier `main.cpp` :

```
1   // Déclarations
2   void f(int x);
3   int g();
```

[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...

```
4   ...
5   int main() {
6       ...
7       // Utilisation
8       int a=g();
9       f(a);
10      ...
```

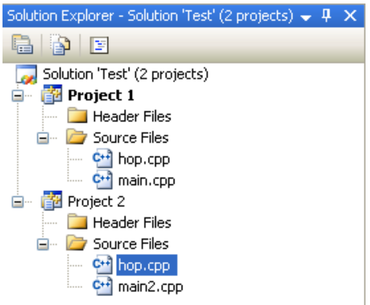
Nous pourrions aussi évidemment déclarer dans `hop.cpp` certaines fonctions de `main.cpp` pour pouvoir les utiliser. Attention toutefois : si des fichiers s'utilisent de façon croisée, c'est peut-être que nous sommes en train de ne pas découper les sources convenablement.

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 objet[3] ! Donc changer quelques lignes dans un fichier n'entraînera pas la compilation de tout le programme mais seulement du fichier concerné[4].

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.



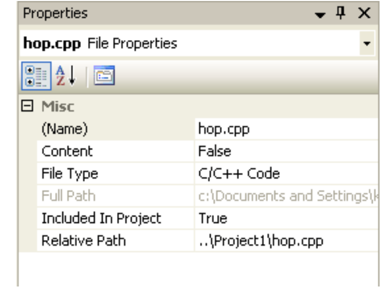


FIG. 6.2 – Même source dans deux projets

[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. Visual gère automatiquement ces dépendances (C'est plus compliqué, mais possible aussi, sous linux...)

[4]En fait, Visual est même capable de ne recompiler que certaines parties du fichier quand les modifications apportées sont simples...

6.1.3 Utilisation dans un autre projet

Pour utiliser dans un projet 2 un fichier source d'un projet 1, il suffit de rajouter le source en question dans la liste des sources du projet 2 ! Pour cela, après avoir sélectionné le projet 2 :

- Choisir le menu `Project/Add Existing Item` (ou `Alt+Maj+A` ou)
- Sélectionner le fichier source.

ou plus simplement :

- Avec la souris, glisser/déplacer le fichier entre les deux projets[5] en maintenant la touche `Ctrl` enfoncée (un + apparaît).

On obtient le résultat figure 6.2.

Attention : il faut bien comprendre que

le fichier ainsi partagé reste dans le répertoire du premier projet

comme le confirme l'affichage de ses propriétés (à droite figure 6.2). Dans notre exemple, chaque projet a son propre fichier principal, `main.cpp` ou `main2.cpp`, mais il y a un seul `hop.cpp`. Modifier le contenu de ce fichier aura des conséquences sur les deux projets à la fois. C'est ce que nous voulions !

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 principal les utilisant. Nous pouvons mieux faire[6]. Pour cela, il est 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*. La plupart des lignes commençant par un `#` lui seront destinées. Nous en verrons d'autres. Attention : jusqu'ici nous utilisions une forme légèrement différente : `#include <nom>`, qui va chercher le fichier `nom` dans les répertoires des bibliothèques C++[7].

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-tête[8]. Pour créer ce fichier, faire comme pour le source, mais en choisissant "Visual C++/Code/Header file" au lieu de "Visual C++/Code/C++ file". Voila ce que cela donne :

- Fichier `hop.cpp` :

```
1  // Définitions
2  void f(int x) {
```

[5] On peut aussi glisser/déplacer le fichier depuis l'explorateur Windows

[6] Toujours le moindre effort !...

[7] Les fichiers d'en-tête `iostream`, etc. sont parfois appelés en-têtes *système*. Leur nom ne se termine pas toujours par `.h` (voir après)

[8] `.h` comme header.

```
3       ...
4   }
5   int g() {
6       ...
7   }
8   // Autres fonctions
9   ...
```

– Fichier `hop.h` :

```
1   // Déclarations
2   void f(int x);
3   int g();
```

– Fichier `main.cpp` du premier projet :

```
1   #include "hop.h"
2   ...
3   int main() {
4       ...
5       // Utilisation
6       int a=g();
7       f(a);
8       ...
```

– Fichier `main2.cpp` du deuxième projet (il faut préciser l'emplacement complet de l'en-tête, qui se trouve dans le répertoire du premier projet[9]) :

```
1   #include "../Project1/hop.h"
2   ...
3   int main() {
4       ...
5       // Utilisation
6       f(12);
7       int b=g();
8       ...
```

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

– Fichier `hop.cpp` :

```
1   #include "hop.h"
2   ...
3   // Définitions
4   void f(int x) {
5       ...
6   }
7   int g() {
8       ...
9   }
10  // Autres fonctions
11  ...
```

[9]On peut aussi préciser au compilateur une liste de répertoires où il peut aller chercher les fichiers d'en-tête. Utiliser pour cela les propriétés du projet, option `"C/C++ / General / Additional Include Directories"` en précisant `"../Project1"` comme répertoire. Après quoi, un `#include "hop.h"` suffit.

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` :

```
1  // Types
2  struct Vecteur {
3     double x,y;
4  };
5  // Déclarations
6  double norme(Vecteur V);
7  Vecteur plus(Vecteur A,Vecteur B);
```

– Fichier `vect.cpp` :

```
1  #include "vect.h" // Fonctions et types
2  // Définitions
3  double norme(Vecteur V) {
4     ...
5  }
6  Vecteur plus(Vecteur A,Vecteur B) {
7     ...
8  }
9  // Autres fonctions
10 ...
```

– Fichier `main.cpp` du premier :

```
1  #include "vect.h"
2  ...
3  int main() {
4      ...
5      // Utilisation
6      Vecteur C=plus(A,B);
7      double n=norme(C);
8      ...
```

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 apparaître[10].

[10]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...

2. ne jamais inclure un fichier séparé lui-même ! C'est généralement une "grosse bêtise"[11]. 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 programmées, 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 pourrons 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. A 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 une phénomène d'inclusions sans fin qui provoque une erreur[12]. 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

`#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.

[11] 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)

[12] Les pré-processeurs savent heureusement détecter ce cas de figure.

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 vecteur[13] 2D et en implémente l'addition :

```
1   struct vect {
2       double x,y;
3   };
4   vect plus(vect m,vect n) {
5       vect p={m.x+n.x,m.y+n.y};
6       return p;
7   }
8   int main() {
9       vect a={1,2},b={3,4};
10      vect c=plus(a,b);
11      return 0;
12  }
```

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

```
1   struct vect {
2       double x,y;
3   };
4   vect operator+(vect m,vect n) {
5       vect p={m.x+n.x,m.y+n.y};
6       return p;
7   }
8   int main() {
9       vect a={1,2},b={3,4};
10      vect c=a+b;
11      return 0;
12  }
```

Nous pouvons aussi définir un produit par un scalaire, un produit scalaire[14], etc[15].

```
15  // Produit par un scalaire
16  vect operator*(double s,vect m) {
17      vect p={s*m.x,s*m.y};
18      return p;
19  }
20  // Produit scalaire
21  double operator*(vect m,vect n) {
```

[13] **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 !

[14] 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

[15] 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.

```
22      return m.x*n.x+m.y*n.y;
23  }
24  int main() {
25      vect a={1,2},b={3,4};
26      vect c=2*a;
27      double s=a*b;
28      return 0;
29  }
```

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 :
```
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=254;
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);
int i,j;
double x=double(i)/j;
```

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:
    ...;
}
```

Fiche de référence (2/3)

Boucles

- `do {`
```
    ...
  } while (!ok);
```
- `int i=1;`
```
  while (i<=100) {
    ...
    i=i+1;
  }
```
- `for (int i=1;i<=100;i++)`
```
    ...
```
- `for (int i=1,j=100;j>i;`
```
      i=i+2,j=j-3)
    ...
```

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;
```

Tableaux

- Définition :
  - `double x[10],y[10];`
```
    for (int i=0;i<10;i++)
        y[i]=2*x[i];
```
  - `const int n=5;`
```
    int i[n],j[2*n]; // OK
```
- Initialisation :
```
int t[4]={1,2,3,4};
string s[2]={"ab","cd"};
```
- Affectation :
```
int s[4]={1,2,3,4},t[4];
for (int i=0;i<4;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;
    }
```

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"`, y compris 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>`
```
  ...
  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();
```

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

CLGraphics

- Voir documentation...

Clavier

- Build : F7
- Start : Ctrl+F5
- Compile : Ctrl+F7
- Debug : F5
- Stop : Maj+F5
- Step over : F10
- Step inside : F11
- Indent : Ctrl+K,Ctrl+F
- Add New It. : Ctrl+Maj+A
- Add Exist. It. : Alt+Maj+A

Conseils

- Travailler en local
- `CertisLibs Project`
- Nettoyer en quittant.
- Erreurs et warnings : cliquer.
- Indenter.
- Ne pas laisser de warning.
- Utiliser le debuggeur.
- Faire des fonctions.
- Tableaux : quand c'est utile ! (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 pourrons alors mieux comprendre les variables locales, comment marche exactement l'appel d'une fonction, les fonctions récursives, etc. Après cela, nous pourrons 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 indiçant 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	2	?	?	23	3	[*reste*]						
10	23	2	?	?	23	3	[*reste*]	?					
11	23	2	?	?	23	3	[*reste*]	7					
12	23	2	?	2	23	3	[*reste*]	7					
13a	23	2	?	2	23	3	[*reste*]	7					
4	23	2	?	2	23	3	[*reste*]	7		23	3	7	2
5	23	2	?	2	23	3	[*reste*]	7		23	3	7	2
7	23	2	?	2	23	3	[*reste*]	7					
13b	23	2	?	2	23	3	[*reste*]	7					
14	23	2	?	2	23	3	[*reste*]	7	7				
15	23	2	?	2					7				
24b	23	2	7	2					7				
25	23	2	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 des appels**. Pour mieux comprendre cette pile, nous allons utiliser le debuggeur. Avant cela, précisons ce qu'un informaticien entend par *pile*.

[1] par exemple 24a et 24b

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

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 out[3]). 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 débuggeur

Observons donc la figure 7.1 obtenue en lançant notre programme d'exemple sous debuggeur. 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 en haut de la pile. Plus profond dans la pile, nous voyons que `main()` a été appelé par une fonction `mainCRTStartup()` elle même appelée par un étrange `kernel32.dll`[4]. Vérifiez aussi les variables et le fait qu'elles valent n'importe quoi (ici -858993460 !) 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 progressée** et nous en sommes toujours à la ligne 5. Simplement, Visual offre la possibilité en double-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 ligne 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()`.

(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 sortant[5] (Maj-F11) pour relancer jusqu'à sortir de `verifie()`,

[3] Dernier rentré, premier sorti.

[4] Ces deux fonctions sont respectivement : (i) la fonction que le compilateur crée pour faire un certain nombre de choses avant et après `main()` et (ii) la partie de Windows qui lance le programme lui-même.

[5] Step Out ou Maj-F11 ou . Notez aussi 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 "exécuter jusqu'à cette ligne" ()

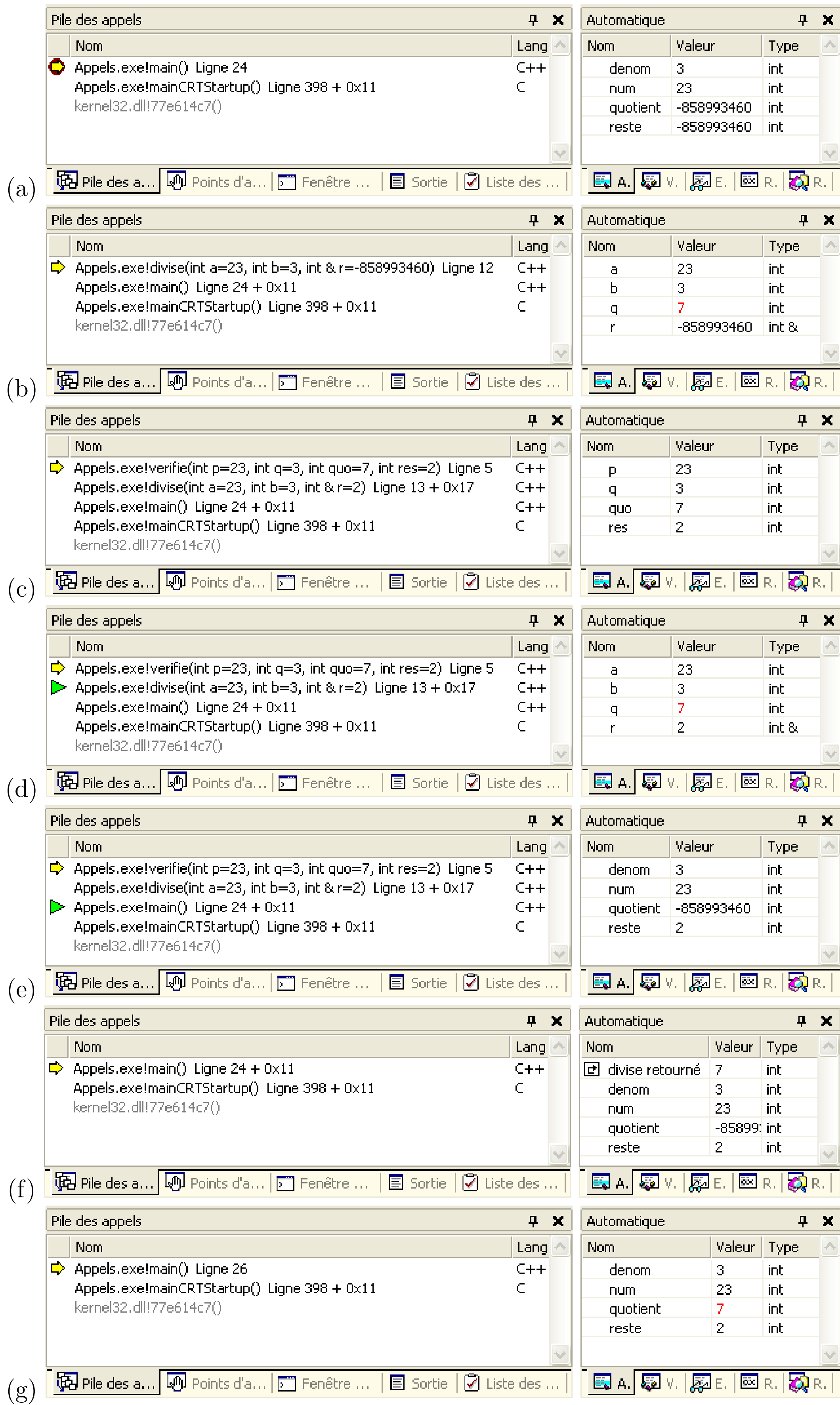


Fig. 7.1 – Appels de fontions

pile	variable	valeur	pile	variable	valeur	pile	variable	valeur
	place libre			place libre			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	...		pris par les	...		pris par les	...
	les fonctions	...		les fonctions	...		les fonctions	...
	avant main()	...		avant main()	...		avant main()	...

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

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.

7.2.2 La pile

Les variables locales (et donc les paramètres) ne sont pas mémorisées à des adresses fixes en mémoire[6], 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 mémoire[7] :

[6] Souvenons nous du chapitre 2.
[7] Et permettra de faire des fonctions récursives, cf section suivante !

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 factorielle[8]. 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-même[9].

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 :

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				

[8]**Coin des collégiens :** La factorielle d'un nombre entier n s'écrit $n!$ et vaut $n! = 1 \times 2 \times ... \times n$.
[9]Le fait de pouvoir mettre des `return` au milieu des fonctions est ici bien commode!

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→	$\mathtt{n}_{fact1(1)}$	1
	$\mathtt{n}_{fact1(2)}$	2
	$\mathtt{n}_{fact1(3)}$	3

ce que l'on peut aisément vérifier avec le débuggeur. 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 prête[10]. 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 appels[11]. Mais la vraie raison qui fait qu'on évite parfois le récursif est qu'

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'appel[12]. Dans le cas d'une fonction récursive, on essaie donc si c'est nécessaire d'écrire une version *dérécursivée* (ou *itérative*) de la fonction. Pour notre factorielle, cela donne :

[10] C'est une erreur classique de débutant que de vouloir abuser du récursif.
[11] Sous Visual, il s'arrête pour $n = 5000$ environ.
[12] Nous verrons dans un autre chapitre les fonctions `inline` qui répondent à ce problème.

```
1  // Version itérative
2  int fact2(int n)
3  {
4      int f=1;
5      for (int i=2;i<=n;i++)
6          f*=i;
7      return f;
8  }
```

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 dérécursivée :

```
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  }
```

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'ordinateur[13] : 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**[14] :

> 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), rajouter 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 entraine 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 :

> 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 remplit(int t[], int n)
5   {
6       for (int i=0;i<n;i++)
7           t[i]=i+1;
```

[13] 500000x4 soit 2Mo seulement !

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

```
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      remplit(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      remplit(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'ordinateur[15], 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

[15]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.

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.

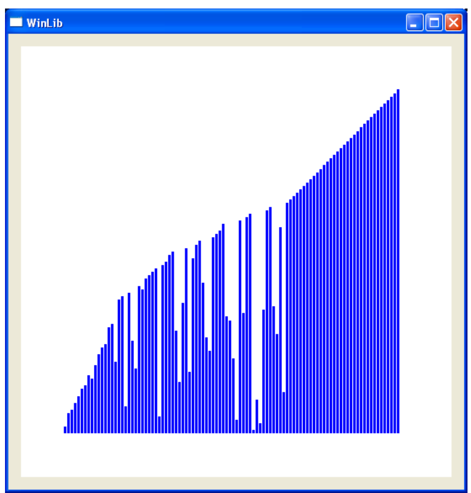
Il y a plusieurs façons de traduire en langage machine un source C++. Le résultat de la compilation peut donc être différent d'un compilateur à l'autre. Au moment de compiler, on peut aussi rechercher à produire un exécutable le plus rapide possible : on dit que le compilateur *optimise* le code. En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

Jusqu'ici, nous utilisions 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.
- Savoir que les modes Debug et Release créent des fichiers objets et exécutables dans des répertoires différents et donc que **lorsqu'on nettoie les solutions, il faut le faire pour chacun des deux modes**.

7.6 TP

Le TP que nous proposons en A.6 est une illustration des fonctions récursive et des tableaux de taille variable. Il consiste en la programmation de quelques façons de trier des données : tri à bulle, Quicksort, etc. Afin de rendre le tout plus attrayant, le tri sera visualisé graphiquement et chronométré (figure 7.3).



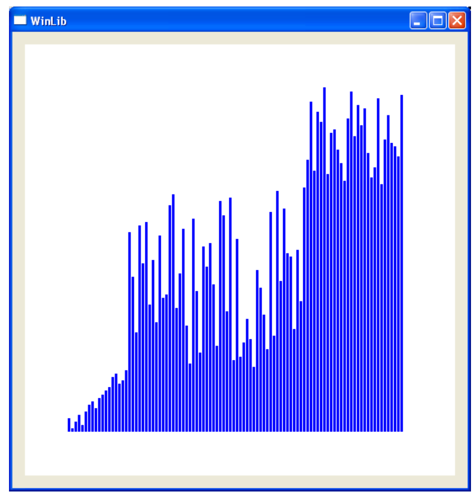


FIG. 7.3 – Deux tris en cours d'exécution : tri à bulle et Quicksort...

7.7 Fiche de référence

Fiche de référence (1/3)

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=254;
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);
int i,j;
double x=double(i)/j;
```
– Pile/Tas

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:
    ...;
}
```

Boucles

–
```
do {
   ...
} while (!ok);
```
–
```
int i=1;
while (i<=100) {
   ...
   i=i+1;
}
```
–
```
for (int i=1;i<=100;i++)
   ...
```
–
```
for (int i=1,j=100;j>i;
     i=i+2,j=j-3)
   ...
```

Fiche de référence (2/3)

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/Récursif

Tableaux

- Définition :
  - `double x[10],y[10];`
    ```
    for (int i=0;i<10;i++)
         y[i]=2*x[i];
    ```
  - `const int n=5;`
    `int i[n],j[2*n]; // OK`
- Initialisation :
```
int t[4]={1,2,3,4};
string s[2]={"ab","cd"};
```
- Affectation :
```
int s[4]={1,2,3,4},t[4];
for (int i=0;i<4;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"`, y compris 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>
  ...
  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

- Build : F7
- Start : Ctrl+F5
- Compile : Ctrl+F7
- Debug : F5
- Stop : Maj+F5
- Step over : F10
- Step inside : F11
- Indent : Ctrl+K,Ctrl+F
- Add New It. : Ctrl+Maj+A
- Add Exist. It. : Alt+Maj+A
- Step out : Maj+F11
- Run to curs. : Click droit
- Complétion : Alt+→

Fiche de référence (3/3)

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

CLGraphics

– Voir documentation...

Conseils

– Travailler en local
– `CertisLibs Project`
– Nettoyer en quittant.
– Erreurs et warnings : cliquer.
– Indenter.
– Ne pas laisser de warning.
– Utiliser le debuggeur.
– Faire des fonctions.
– Tableaux : quand c'est utile ! (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.
– Debug/Release : nettoyer les deux.

7.8 Examens sur machine

Nous vous conseillons aussi de vous confronter aux examens proposés en annexe. Vous avez toutes les connaissances nécessaires.

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 dynamique[1] 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];
```

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.

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

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++)
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 lignes

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 $T(i + mj)$. Cela donne :

```
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 $\texttt{i}^{eme}$ élément. Lorsque `t` est un pointeur d'`int`, `t[i]` désigne la variable `int` stockée `i` places[2] 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 :
   - 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      ...
```

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

```
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*s,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 matrices[3] et leur produit :

```
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;
```

[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.

```
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++) {
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).

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'il 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).

FIG. 8.1 – 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.7 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 8.1).

8.6 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<=100;i++)
     ...
– for (int i=1,j=100;j>i;
       i=i+2,j=j-3)
     ...
– for (int i=...)
    for (int j=...) {
      // saute le cas i==j
      if (i==j)
        continue;
      ...
    }
```

Fiche de référence (2/3)

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=254;
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);
int i,j;
double x=double(i)/j;
```
– Pile/Tas

Clavier

– Build : F7
– Start : Ctrl+F5
– Compile : Ctrl+F7
– Debug : F5
– Stop : Maj+F5
– Step over : F10
– Step inside : F11
– Indent : Ctrl+K,Ctrl+F
– Add New It. : Ctrl+Maj+A
– Add Exist. It. : Alt+Maj+A
– Step out : Maj+F11
– Run to curs. : Click droit
– Complétion : Alt+→
– 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/Récursif

Tableaux

– Définition :
  – `double x[10],y[10];`
```
for (int i=0;i<10;i++)
    y[i]=2*x[i];
```
  – `const int n=5;`
```
int i[n],j[2*n]; // OK
```
– Initialisation :
```
int t[4]={1,2,3,4};
string s[2]={"ab","cd"};
```
– Affectation :
```
int s[4]={1,2,3,4},t[4];
for (int i=0;i<4;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;
```

Fiche de référence (3/3)

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"`, y compris 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...

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>
  ...
  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;
  ```
- ```
  #define _USE_MATH_DEFINES
  #include <cmath>
  double pi=M_PI;
  ```

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;
  double x;
  x=i/j; // NON!
  x=double(i/j); //NON!
  ```
- ```
  double x[10],y[10];
  for (int i=1;i<=10;i++)
        // NON!
        y[i]=2*x[i];
  ```
- ```
  int n=5;
  int t[n]; // NON
  ```
- ```
  int f()[4] { // NON!
      int t[4];
      ...
      return t; // NON!
  }
  ...
  int t[4]
  t=f();
  ```
- ```
  int s[4]={1,2,3,4},t[4];
  t=s; // NON!
  ```
- ```
  int t[2];
  t={1,2}; // NON!
  ```
- ```
  struct Point {
     double x,y;
  } // NON!
  ```
- ```
  Point a;
  a={1,2}; // NON!
  ```
- `#include "vect.cpp"// NON!`
- ```
  void 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; // Déja fait!
  ```
- ```
  int *t,s;// s est int
           // et non int* !
  t=new int[n];
  s=new int[n];// NON!
  ```

CLGraphics

- Voir documentation...

Conseils

- Travailler en local
- `CertisLibs Project`
- Nettoyer en quittant.
- Erreurs et warnings : cliquer.
- Indenter.
- Ne pas laisser de warning.
- Utiliser le debuggeur.
- Faire des fonctions.
- Tableaux : quand c'est utile ! (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.
- Debug/Release : nettoyer les deux.
  ```

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 aspects[1]*. 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 liés. 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 propres [3]. 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 :

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

[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.

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 là 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 à certains moment en C++ et que les programmeurs venant de Java mettent partout en se trompant d'ailleurs sur son type. Vous n'en n'aurez en général pas 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 en train
3                 // de s'exécuter
4     int j=x+i;  // champ x de l'objet dont la méthode f() est en train de
5                 //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);

// ========= définitions (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.detruit();
    B.detruit();
    A.detruit();
    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 `operator`*op*`(objB B)`, alors `A`*op*`B` 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
                                     // double n'est pas un objet!
```

car cela reviendrait à définir une méthode pour le type `double`, qui n'est pas un objet[4]. 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, donc 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...

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);
```

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

```
    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 moyen[5] 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 sur, 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.

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 des 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!
```

[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 sure et efficace, comme il en existe déjà en C++ standard ou dans des bibliothèques complémentaires disponibles sur le WEB. Etc, etc.

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 débuter 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 avions 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 méthode[6].

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 :

```
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++)
```

[6]sans compter qu'ils les déclarent souvent comme en C avec d'inutiles `typedef`. Mais bon, ceci ne devrait pas vous concerner !

```
      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]` !

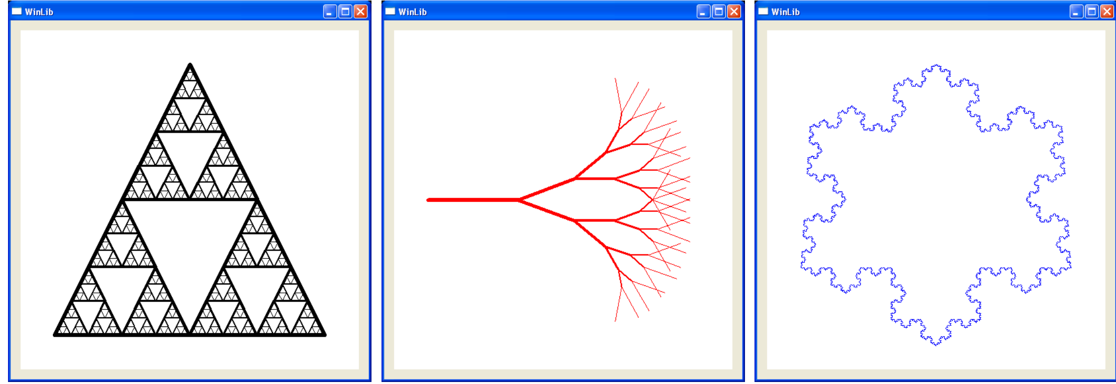


FIG. 9.1 – Fractales

9.8 TP

Vous devriez maintenant pouvoir faire le TP en A.8 qui dessine quelques courbes fractales (figure 9.1) en illustrant le concept d'objet..

9.9 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<=100;i++)
     ...
- for (int i=1,j=100;j>i;
        i=i+2,j=j-3)
     ...
- for (int i=...)
     for (int j=...) {
       // saute le cas i==j
       if (i==j)
         continue;
       ...
     }
```

Clavier

- Build : F7
- Start : Ctrl+F5
- Compile : Ctrl+F7
- Debug : F5
- Stop : Maj+F5
- Step over : F10
- Step inside : F11
- Indent : Ctrl+K,Ctrl+F
- Add New It. : Ctrl+Maj+A
- Add Exist. It. : Alt+Maj+A
- Step out : Maj+F11
- Run to curs. : Click droit
- Complétion : Alt+→
- Gest. tâches : Ctrl+Maj+Ech

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=254;
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);
int i,j;
double x=double(i)/j;
```
– Pile/Tas

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/Récursif

Tableaux

– Définition :

  – ```
    double x[10],y[10];
    for (int i=0;i<10;i++)
        y[i]=2*x[i];
    ```
  – ```
    const int n=5;
    int i[n],j[2*n]; // OK
    ```

– Initialisation :
```
int t[4]={1,2,3,4};
string s[2]={"ab","cd"};
```
– Affectation :
```
int s[4]={1,2,3,4},t[4];
for (int i=0;i<4;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 !)
  ```

Fiche de référence (3/4)

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 une_autre(obj A);
  };
  void obj::a_moi() {
    x=..;   // OK
    ..=y;   // OK
    z=..;   // OK
  }
  void obj::pour_tous() {
    x=..;    // OK
    a_moi(); // OK
  }
  void 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
  ```
- ```
  class obj {
    obj operator+(obj B);
  };
  ...
  int main() {
    obj A,B,C;
    C=A+B;
    // C=A.operator+(B)
  ```

Compilation séparée

- `#include "vect.h"`, y compris 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...

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;
  double x;
  x=i/j; // NON!
  x=double(i/j); //NON!
  ```
- ```
  double x[10],y[10];
  for (int i=1;i<=10;i++)
       // NON!
       y[i]=2*x[i];
  ```
- ```
  int n=5;
  int t[n]; // NON
  ```
- ```
  int f()[4] { // NON!
      int t[4];
      ...
      return t; // NON!
  }
  ...
  int t[4]
  t=f();
  ```
- ```
  int s[4]={1,2,3,4},t[4];
  t=s; // NON!
  ```
- ```
  int t[2];
  t={1,2}; // NON!
  ```
- ```
  struct Point {
     double x,y;
  } // NON!
  ```
- ```
  Point a;
  a={1,2}; // NON!
  ```
- `#include "vect.cpp"// NON!`
- ```
  void 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; // Déja fait!
  ```
- ```
  int *t,s;// s est int
           // et non int* !
  t=new int[n];
  s=new int[n];// NON!
  ```

Fiche de référence (4/4)

Divers

- `i++;`
  `i--;`
  `i-=2;`
  `j+=3;`
- `j=i%n; // Modulo`
- `#include <cstdlib>`
  `...`
  `i=rand()%n;`
  `x=rand()/double(RAND_MAX);`
- `#include <ctime>`
  `...`
  `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;`
- `#define _USE_MATH_DEFINES`
  `#include <cmath>`
  `double pi=M_PI;`

CLGraphics

- Voir documentation...

Conseils

- Travailler en local
- `CertisLibs Project`
- Nettoyer en quittant.
- Erreurs et warnings : cliquer.
- Indenter.
- Ne pas laisser de warning.
- Utiliser le debuggeur.
- Faire des fonctions.
- Tableaux : quand c'est utile ! (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.
- Debug/Release : nettoyer les deux.
- Faire des objets.
- Ne pas toujours faire des objets !
- Penser interface / implémentation / 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 privée[1] !

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 crée[a].

[a]Ce qui explique qu'il n'est pas besoin de lui préciser un type de retour.

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 contruit :

```
  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 crée sans rien préciser, c'est le *construteur 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 soit 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 bon essai de construire chaque u[i]
                              // avec point(1,2), mais ça 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 soit 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 avions 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'écrira plutôt :

```
point point::operator+(point b) {
  return point(x+b.x,y+b.y);
}
...
  c=point(1,2)+f(point(2,3));
```

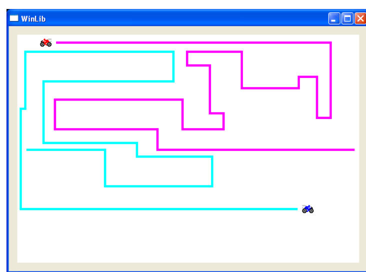


Fig. 10.1 – Jeu de Tron.

10.5 TP

Nous pouvons faire une pause et aller faire le TP que nous proposons en A.9. 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 recopie 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 *demander 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 complète[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 programme[2].

—

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 << "dest ";
}

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 dest obj 3 dest obj 4 dest dest 5 obj 6 dest dest dest
```

Repérez bien à quel moment les objets sont détruits. Constatez aussi qu'il y a des appels au destructeur pour les objets qui sont construits par copie et pour lesquels nous n'avons pas encore parlé du constructeur...

10.8 Destructeurs et tableaux

Le destructeur est appelé pour tous les éléments du tableau. Ainsi,

```
10      if (a==b) {
11          obj t[10];
12          ...
13      }
```

appellera 10 fois le constructeur vide en ligne 11 et dix fois le destructeur en ligne 13. Dans le cas d'un tableau dynamique, c'est au moment du `delete[]` que les destructeurs sont appelés (avant la désallocation du tableau !).

```
10      if (a==b) {
11          obj* t=new obj[n]; // n appels à obj()
12          ...
13          delete[] t;        // n appels à ~obj();
14      }
```

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 dest obj 3 dest obj 4 dest copy dest 5 obj 6 copy dest
dest dest
```

Nous avons enfin autant d'appels 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 dest obj 3 dest obj 4 dest copy dest 5 obj 6 copy dest
= dest dest
```

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 désallouée[4].

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 similaires[5] :

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

class vect {
    // champs
```

[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 charger de désallouer ? Etc, etc.

```
    int n;
    double *t;
    // fonctions privées
    void alloc(int N);
    void kill();
    void copy(const vect& v);
public:
    // constructeurs "obligatoires"
    vect();
    vect(const vect& v);
    // destructeur
    ~vect();
    // affectation
    const vect& operator=(const vect& v);
    // constructeurs supplémentaires
    vect(int N);
};

void vect::alloc(int N) {
    n=N;
    if (n!=0)
        t=new double[n];
}

void vect::kill() {
    if (n!=0)
        delete[] t;
}

void vect::copy(const vect& v) {
    alloc(v.n);
    for (int i=0;i<n;i++) // OK même si n==0
        t[i]=v.t[i];
}

vect::vect() {
    alloc(0);
}

vect::vect(const vect& v) {
    copy(v);
}

vect::~vect() {
    kill();
}

const vect& vect::operator=(const vect& v) {
    if (this!=&v) {
```

```
        kill();
        copy(v);
    }
    return v;
}

vect::vect(int N) {
    alloc(N);
}

// Pour tester constructeur de copie
vect f(vect a) {
    return a;
}
// Pour tester le reste
int main()
{
    vect a,b(10),c(12),d;
    a=b;
    a=a;
    a=c;
    a=d;
    a=f(a);
    b=f(b);
    return 0;
}
```

10.12 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<=100;i++)
    ...
```
-
```
for (int i=1,j=100;j>i;
     i=i+2,j=j-3)
    ...
```
-
```
for (int i=...)
  for (int j=...) {
    // saute le cas i==j
    if (i==j)
      continue;
    ...
  }
```

Clavier

- Build : F7
- Start : Ctrl+F5
- Compile : Ctrl+F7
- Debug : F5
- Stop : Maj+F5
- Step over : F10
- Step inside : F11
- Indent : Ctrl+K,Ctrl+F
- Add New It. : Ctrl+Maj+A
- Add Exist. It. : Alt+Maj+A
- Step out : Maj+F11
- Run to curs. : Click droit
- Complétion : Alt+→
- Gest. tâches : Ctrl+Maj+Ech

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 (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=254;
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);
int i,j;
double x=double(i)/j;
```
– Pile/Tas

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/Récursif

– Références constantes (pour un passage rapide) :
```
void f(const obj& x){
  ...
}
void g(const obj& x){
  f(x); // OK
}
```

Tableaux

– Définition :
  –
```
double x[10],y[10];
for (int i=0;i<10;i++)
     y[i]=2*x[i];
```
  –
```
const int n=5;
int i[n],j[2*n]; // OK
```
– Initialisation :
```
int t[4]={1,2,3,4};
string s[2]={"ab","cd"};
```
– Affectation :
```
int s[4]={1,2,3,4},t[4];
for (int i=0;i<4;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;
```

Fiche de référence (3/4)

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 une_autre(obj A);
};
void obj::a_moi() {
  x=..;   // OK
  ..=y;   // OK
  z=..;   // OK
}
void obj::pour_tous() {
  x=..;    // OK
  a_moi(); // OK
}
void 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
```
–
```
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 :
```
point point::operator+(
                 point b) {
  return point(x+b.x,
               y+b.y);
}
...
  c=point(1,2)
    +f(point(2,3));
```
– Destructeur :
```
obj::~obj() {
  ...
}
```
– Constructeur de copie :
```
obj::obj(const obj& o) {
  ...
}
```
Utilisé par :
- `obj b(a);`
- `obj b=a;`
`//Différent de obj b;b=a;`
- paramètres des fonctions
- valeur de retour

– Affectation :
```
const obj& obj::operator=
           (const obj&o) {
  ...
  return o;
}
```
– Objets avec allocation dynamique automatique : cf section 10.11

Compilation séparée

– `#include "vect.h"`, y compris 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...

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;
```

Fiche de référence (4/4)

Divers

- `i++;`
  `i--;`
  `i-=2;`
  `j+=3;`
- `j=i%n; // Modulo`
- `#include <cstdlib>`
  `...`
  `i=rand()%n;`
  `x=rand()/double(RAND_MAX);`
- `#include <ctime>`
  ```
  ...
  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;
  ```
- ```
  #define _USE_MATH_DEFINES
  #include <cmath>
  double pi=M_PI;
  ```

CLGraphics

- Voir documentation...

Conseils

- Travailler en local
- `CertisLibs Project`
- Nettoyer en quittant.
- Erreurs et warnings : cliquer.
- Indenter.
- Ne pas laisser de warning.
- Utiliser le debuggeur.
- Faire des fonctions.
- Tableaux : quand c'est utile ! (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.
- Debug/Release : nettoyer les deux.
- 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;
  double x;
  x=i/j; // NON!
  x=double(i/j); //NON!
  ```
- ```
  double x[10],y[10];
  for (int i=1;i<=10;i++)
        // NON!
        y[i]=2*x[i];
  ```
- ```
  int n=5;
  int t[n]; // NON
  ```
- ```
  int f()[4] { // NON!
      int t[4];
      ...
      return t; // NON!
  }
  ...
  int t[4]
  t=f();
  ```
- ```
  int s[4]={1,2,3,4},t[4];
  t=s; // NON!
  ```
- ```
  int t[2];
  t={1,2}; // NON!
  ```
- ```
  struct Point {
     double x,y;
  } // NON!
  ```
- ```
  Point a;
  a={1,2}; // NON!
  ```
- `#include "vect.cpp"// NON!`
- ```
  void 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; // Déja fait!
  ```
- ```
  int *t,s;// s est int
               // et non int* !
  t=new int[n];
  s=new int[n];// NON!
  ```
- ```
  class point {
    int x,y;
  public:
    ...
  };
  ...
    point a={2,3}; // NON!
  ```
- Oublier de redéfinir le constructeur vide.
- ```
  point p=point(1,2);// NON!
  point p(1,2);       // OUI!
  ```
- ```
  obj* t=new obj[n];
  ...
  delete t; // oubli de []
  ```

10.13 Devoir écrit

Vous pouvez maintenant vous confronter aux devoirs écrits proposés en annexe, par exemple B.5 et B.6. Vous avez toutes les connaissances nécessaires...

Chapitre 11

En vrac...

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, etc. 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 caratères

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 êtres comparées. C'est l'ordre alphabétique qui est évidemment utilisé :

   ```
   if (s1==s1) ...
   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, ie. en ignorant s[0], s[1] et 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 `-1`.

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 à partir
                          // 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).

```
string s="hop hop";
const char *t=s.c_str();
```

Vous trouverez d'autres fonctions dans l'aide en ligne de Visual, ou tout simplement proposées par Visual quand vous utiliserez les `string`.

[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.

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. Voic 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;
    }
```

   On peut aussi avoir besoin de savoir si on est arrivé au bout du fichier :

```
    do {
      ...
    } while (!(g.eof());
```

3. Un fichier peut s'ouvrir après construction :

```
   ofstream f;
   f.open("hop.txt");
   ...
```

4. 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;
};
```

[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 !

```
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;
```

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! operator() n'est pas une méthode constante
                 // 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) {
```

[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 !

```
    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 debuggeur, les fonctions `inline` sont compilées comme des fonctions normales en mode Debug. Seul le mode Release profitera donc de l'accélération.

Cas des méthodes

Dans le cas d'une méthode, il faut bien penser à la mettre dans le ficher `.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

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 ne disparaît à la compilation en mode Release. C'est une fonction peu connue des débutants, et c'est bien dommage ! 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 (1/4)

Boucles

- ```
  do {
    ...
  } while (!ok);
  ```
- ```
  int i=1;
  while (i<=100) {
    ...
    i=i+1;
  }
  ```
- ```
  for (int i=1;i<=100;i++)
    ...
  ```
- ```
  for (int i=1,j=100;j>i;
       i=i+2,j=j-3)
    ...
  ```
- ```
  for (int i=...)
    for (int j=...) {
      // saute le cas i==j
      if (i==j)
        continue;
      ...
    }
  ```

Clavier

- Build : F7
- Start : Ctrl+F5
- Compile : Ctrl+F7
- Debug : F5
- Stop : Maj+F5
- Step over : F10
- Step inside : F11
- Indent : Ctrl+K,Ctrl+F
- Add New It. : Ctrl+Maj+A
- Add Exist. It. : Alt+Maj+A
- Step out : Maj+F11
- Run to curs. : Click droit
- Complétion : Alt+→
- Gest. tâches : Ctrl+Maj+Ech

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 !)

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=254;
  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);
  int i,j;
  double x=double(i)/j;
  ```
- Pile/Tas
- Type énuméré :
  ```
  enum Dir{nord,est,
           sud,ouest};
  void avance(Dir d);
  ```

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:
      ...;
  }
  ```

Conseils

- Travailler en local
- `CertisLibs Project`
- Nettoyer en quittant.
- Erreurs et warnings : cliquer.
- Indenter.
- Ne pas laisser de warning.
- Utiliser le debuggeur.
- Faire des fonctions.
- Tableaux : quand c'est utile ! (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.
- Debug/Release : nettoyer les deux.
- Faire des objets.
- Ne pas toujours faire des objets !
- Penser interface / implémentation / utilisation.
  ```

Fiche de référence (2/4)

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/Récursif

– 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[10],y[10];
  for (int i=0;i<10;i++)
       y[i]=2*x[i];
  ```
  –
  ```
  const int n=5;
  int i[n],j[2*n]; // OK
  ```
– Initialisation :
```
int t[4]={1,2,3,4};
string s[2]={"ab","cd"};
```
– Affectation :
```
int s[4]={1,2,3,4},t[4];
for (int i=0;i<4;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;
```

Compilation séparée

– `#include "vect.h"`, y compris 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 (3/4)

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 une_autre(obj A);
  };
  void obj::a_moi() {
    x=..;   // OK
    ..=y;   // OK
    z=..;   // OK
  }
  void obj::pour_tous() {
    x=..;    // OK
    a_moi(); // OK
  }
  void 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
  ```
- ```
  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 :
  ```
  point point::operator+(
                  point b) {
    return point(x+b.x,
                 y+b.y);
  }
  ...
    c=point(1,2)
      +f(point(2,3));
  ```
- Destructeur :
  ```
  obj::~obj() {
    ...
  }
  ```
- Constructeur de copie :
  ```
  obj::obj(const obj& o) {
    ...
  }
  ```
  Utilisé par :
  - `obj b(a);`
  - `obj b=a;`
  `//Différent de obj b;b=a;`
  - paramètres des fonctions
  - valeur de retour
- Affectation :
  ```
  const obj& obj::operator=
              (const obj&o) {
    ...
    return o;
  }
  ```
- Objets avec allocation dynamique automatique : cf section 10.11
- Accesseurs :
  ```
  class mat {
     double *x;
  public:
   inline double& operator()
              (int i,int j) {
     assert(i>=0 ...);
     return x[i+M*j];
   }
   inline 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>
  ...
  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');
  size_t j=s.find('h',3);
  size_t i=s.find("hop");
  size_t j=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 <cassert>
  ...
   assert(x!=0);
   y=1/x;
  ```
- ```
  #include <ctime>
  s=double(clock())
          /CLOCKS_PER_SEC;
  ```
- ```
  #define _USE_MATH_DEFINES
  #include <cmath>
  double pi=M_PI;
  ```
  ```

Fiche de référence (4/4)

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 nom;
  ifstream f(nom.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;
  }
  ```

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;
  double x;
  x=i/j; // NON!
  x=double(i/j); //NON!
  ```
- ```
  double x[10],y[10];
  for (int i=1;i<=10;i++)
       // NON!
       y[i]=2*x[i];
  ```
- ```
  int n=5;
  int t[n]; // NON
  ```
- ```
  int f()[4] { // NON!
      int t[4];
      ...
      return t; // NON!
  }
  ...
  int t[4]
  t=f();
  ```
- ```
  int s[4]={1,2,3,4},t[4];
  t=s; // NON!
  ```
- ```
  int t[2];
  t={1,2}; // NON!
  ```
- ```
  struct Point {
      double x,y;
  } // NON!
  ```
- ```
  Point a;
  a={1,2}; // NON!
  ```
- `#include "vect.cpp"// NON!`
- ```
  void 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; // Déja fait!
  ```
- ```
  int *t,s;// s est int
           // et non int* !
  t=new int[n];
  s=new int[n];// NON!
  ```
- ```
  class point {
    int x,y;
  public:
    ...
  };
  ...
    point a={2,3}; // NON!
  ```
- Oublier de redéfinir le constructeur vide.
- ```
  point p=point(1,2);// NON!
  point p(1,2);      // OUI!
  ```
- ```
  obj* t=new obj[n];
  ...
  delete t; // oubli de []
  ```
- ```
  //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!
  ```

CLGraphics

- Voir documentation...

Chapitre 12

En vrac (suite) ...

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++*

—

Toujours sous forme d'un inventaire, le début de ce chapitre sera un peu en vrac, mais nous nous limiterons toujours à quelques aspects pratiques du C++ souvent utilisés donc souvent rencontrés dans les programmes des autres ! La fin du chapitre est plus utile et plus fondamentale : nous y abordons les `template`, ou *programmation générique.*

12.1 Opérateur 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 :

<table>
<tr><th>symbole</th><th>utilisation</th><th>nom</th><th>résultat</th><th>exemple</th></tr>
<tr><td>&</td><td>a&b</td><td>et</td><td>1&1=1, 0 sinon</td><td>13&10=8</td></tr>
<tr><td>|</td><td>a|b</td><td>ou</td><td>0|0=0, 1 sinon</td><td>13|10=15</td></tr>
<tr><td>^</td><td>a^b</td><td>ou exclusif</td><td>1^0=0^1=1, 0 sinon</td><td>13^10=7</td></tr>
<tr><td>>></td><td>a>>n</td><td>décalage à droite</td><td>décale les bits de a n fois vers la droite et comble à gauche avec des 0 (les n premiers de droite sont perdus)</td><td>13>>2=3</td></tr>
<tr><td><<</td><td>a<<n</td><td>décalage à gauche</td><td>décale les bits de a n fois vers la gauche et comble à droite avec des 0</td><td>5<<2=20</td></tr>
<tr><td>~</td><td>~a</td><td>complément</td><td>~1=0, ~0=1</td><td>~13=-14</td></tr>
</table>

Remarques :
- Ces instructions sont particulièrement rapides car simples pour le processeur.
- Les fait que `a^b` existe est aussi source de bugs (**il ne s'agit pas le la fonction puissance !**)
- Le résultat de `~` dépend en fait du type : si par exemple `i` est un entier non signé sur 8 bits valant `13`, alors `~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 :

<table>
<tr><td>i|=(1<<n)</td><td>passe à 1 le bit n de i</td></tr>
<tr><td>i&=~(1<<n)</td><td>passe à 0 le bit n de i</td></tr>
<tr><td>i^=(1<<n)</td><td>inverse le bit n de i</td></tr>
<tr><td>if (i&(1<<n))</td><td>vrai ssi le bit n de i est à 1</td></tr>
</table>

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)

12.2 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 !

12.3 Boucles et `break`

Nous avons déjà rencontré à la section 8.4 la commande `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 :

```
1   for (int i=0;i<N;i++) {
2       ...
3       for (int j=0;j<M;j++) {
4           ...
5           if (...)
6               break; // termine la boucle en j et passe donc
7                      // en ligne 10 (pas en ligne 12)
8           ...
9       }
10      ...
11  }
12  ...
```

3. `break` et `continue` marchent évidemment avec `while` et `do` ... `while` de la même façon qu'avec `for`.

12.4 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++.

12.5 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[]
  ...
```

12.6 template

12.6.1 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 devions 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); // "instancie" T en int
    echange(x,y); // "instancie" 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;
    ...
```

12.6.2 `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].

12.6.3 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>
```

[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 !

```
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);
    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.

12.6.4 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 taper[2] :

```
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>
```

[2]Nous n'avons pas vu comment définir de nouveaux types cachés dans des classes ! C'est ce qui est fait ici...

```
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);
```

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...

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<=100;i++)
     ...
- for (int i=1,j=100;j>i;
       i=i+2,j=j-3)
     ...
- for (int i=...)
    for (int j=...) {
      // saute le cas i==j
      if (i==j)
        continue;
      ...
    }
- for (int i=...) {
    ...
    if (t[i]==s){
      ...
      // quitte la boucle
      break;
    }
    ...
  }
```

Clavier
- Build : F7
- Start : Ctrl+F5
- Compile : Ctrl+F7
- Debug : F5
- Stop : Maj+F5
- Step over : F10
- Step inside : F11
- Indent : Ctrl+K,Ctrl+F
- Add New It. : Ctrl+Maj+A
- Add Exist. It. : Alt+Maj+A
- Step out : Maj+F11
- Run to curs. : Click droit
- Complétion : Alt+→
- Gest. tâches : Ctrl+Maj+Ech

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:
     ...;
  }
- mx=(x>y)?x:y;
```

Fiche de référence (2/6)

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/Récursif

– 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!
```

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 !)

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=254;
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);
int i,j;
double x=double(i)/j;
```
– Pile/Tas

– Type énuméré :
```
enum Dir{nord,est,
         sud,ouest};
void avance(Dir d);
```
– Variables statiques :
```
int f() {
 static bool first=true;
 if (first) {
  first=false;
  ...
 }
 ...
}
```

Fiche de référence (3/6)

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 une_autre(obj A);
};
void obj::a_moi() {
  x=..;    // OK
  ..=y;    // OK
  z=..;    // OK
}
void obj::pour_tous() {
  x=..;     // OK
  a_moi(); // OK
}
void 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
```
–
```
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 :
```
point point::operator+(
                  point b) {
  return point(x+b.x,
               y+b.y);
}
...
  c=point(1,2)
    +f(point(2,3));
```
– Destructeur :
```
obj::~obj() {
  ...
}
```
– Constructeur de copie :
```
obj::obj(const obj& o) {
  ...
}
```
Utilisé par :
- `obj b(a);`
- `obj b=a;`
`//Différent de obj b;b=a;`
- paramètres des fonctions
- valeur de retour

– Affectation :
```
const obj& obj::operator=
            (const obj&o) {
  ...
  return o;
}
```
– Objets avec allocation dynamique automatique : cf section 10.11

– Accesseurs :
```
class mat {
   double *x;
public:
 inline double& operator()
            (int i,int j) {
     assert(i>=0 ...);
     return x[i+M*j];
  }
  inline double operator()
        (int i,int j)const {
     assert(i>=0 ...);
     return x[i+M*j];
  }
  ...
```

Compilation séparée

– `#include "vect.h"`, y compris 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 l'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 somme()const;
};
...
template <typename T>
T paire<T>::somme()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;
  hop<5> B;
  ...
```

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 nom;
ifstream f(nom.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

- Travailler en local
- `CertisLibs Project`
- Nettoyer en quittant.
- Erreurs et warnings : cliquer.
- Indenter.
- Ne pas laisser de warning.
- Utiliser le debuggeur.
- Faire des fonctions.
- Tableaux : quand c'est utile ! (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.
- Debug/Release : nettoyer les deux.
- 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>
  ...
  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');
  size_t j=s.find('h',3);
  size_t i=s.find("hop");
  size_t j=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 <cassert>
  ...
   assert(x!=0);
   y=1/x;
– #include <ctime>
  s=double(clock())
          /CLOCKS_PER_SEC;
– #define _USE_MATH_DEFINES
  #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;
  double x;
  x=i/j; // NON!
  x=double(i/j); //NON!
– double x[10],y[10];
  for (int i=1;i<=10;i++)
       // NON!
       y[i]=2*x[i];
– int n=5;
  int t[n]; // NON
– int f()[4] { // NON!
      int t[4];
      ...
      return t; // NON!
  }
  ...
  int t[4]
  t=f();
– int s[4]={1,2,3,4},t[4];
  t=s; // NON!
– int t[2];
  t={1,2}; // NON!
– struct Point {
      double x,y;
  } // NON!
– Point a;
  a={1,2}; // NON!
– #include "vect.cpp"// NON!
– void 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; // Déja fait!
– int *t,s;// s est int
            // et non int* !
  t=new int[n];
  s=new int[n];// NON!
– class point {
    int x,y;
  public:
    ...
  };
  ...
    point a={2,3}; // NON!
```

– Oublier de redéfinir le constructeur vide.

```
– point p=point(1,2);// NON!
  point p(1,2);       // OUI!
– obj* t=new obj[n];
  ...
  delete t; // oubli de []
– //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! Pour les
      // boucles seulement!
  }
– for (i ...)
   for (j ...) {
    ...
    if (...)
     break;//NON! Ne quitte
     // Que la boucle en j!
– int i;
  double x;
  ...
  j=max(i,0);//OK
  y=max(x,0);//NON! Utiliser
  // 0.0 et non 0 (max est
  // un template (STL)...)
```

CLGraphics

– Voir documentation...

Fiche de référence (6/6)

Tableaux

– Définition :
  - ```
    double x[10],y[10];
    for (int i=0;i<10;i++)
         y[i]=2*x[i];
    ```
  - ```
    const int n=5;
    int i[n],j[2*n]; // OK
    ```

– Initialisation :
```
int t[4]={1,2,3,4};
string s[2]={"ab","cd"};
```

– Affectation :
```
int s[4]={1,2,3,4},t[4];
for (int i=0;i<4;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!
}
```

12.8 Devoir final

Vous pouvez enfin vous confronter aux devoirs complets proposés en annexe, par exemple B.8 et B.9.

Annexe A

Travaux Pratiques

A.1 L'environnement de programmation

A.1.1 Bonjour, Monde !

1. *Connexion :*
   Se connecter sous Windows. Problèmes possibles : redémarrer si linux, mauvais domaine Windows ("se connecter à :"), mot de passe oublié, redémarrer si réseau planté ! Rappel : pour changer son mot de passe, utiliser `http://nfs2`.
2. *Répertoires :*
   Identifier (sous "Poste de travail") les deux disques locaux (`C:` et `D:`) et les disques réseau (`H:` et `Z:`). On travaillera en local sur le bureau qui se trouve en réalité dans `D:\Documents and Settings\login\Bureau`.

   > **Tout essai de travail directement dans H : ou Z : est cause de lenteur, de saturation du réseau et peut entraîner une perte de fichier si vos *quotas disque* sont atteints.**

   On évitera par contre de saturer le bureau en déplaçant ses programmes dans `Z:` en fin de séance. Le principe est donc le suivant :
   (a) Connexion.
   (b) Copie éventuelle d'un travail précédent de `Z:` sur le bureau.
   (c) Travail sur le bureau.
   (d) Recopie du travail vers `Z:` (PS : `Z:` est sauvegardé par les responsables de la DIT, pas `H:`)
   (e) Déconnexion.
3. *Projet :*
   Créer un projet `Tp1` sur le bureau.

   > **Visual C++ voudra travailler dans "Mes Documents". Pour ne pas surcharger les transferts, on travaillera plutôt par aller-retour entre Z : et le bureau.**

   (a) Lancer "`Microsoft Visual C++ Express 2005`"
   (b) Menu "`File/New/Project`"
   (c) "`Name: Tp1`", "`Location: Desktop`".
   (d) "`Type: CertisLibs Project`",

4. Vérifier avec l'explorateur qu'un répertoire `Tp1\Tp1` a bien été crée sur le bureau.
5. *Configuration :*
   Faire apparaître dans Visual :
   (a) La "toolbar" `Build`
   (b) L'icône de lancement correspondant à la commande `Start Without Debugging`
6. *Programmation :*
   (a) Rajouter `cout << "Hello" << endl;` sur la ligne avant `return 0;`
7. *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` (en réalité `Tp1.exe`) dans `Tp1\Tp1\Debug`.

Touche utile : F7 = = Build

8. *Exécution :*
   (a) Lancer le programme avec `"Debug/Start Without Debugging"` (ou `"Ctrl+F5"` ou bouton correspondant). Une fenêtre "console" s'ouvre, le programme s'exécute dedans, et la fenêtre console attend l'appui sur une touche pour se refermer.
   (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 l'explorateur Windows : le programme se referme tout de suite !
   – Dans les menus Windows : `"Démarrer/Exécuter"`
   – `"Ouvrir: cmd"`
   – Taper `"D:"`, puis `"cd \Documents and Settings\nom_du_binome\Bureau\Tp1\Tp1\Deb` (compléter les noms avec la touche `TAB`).
   – Vérifier la présence de `Tp1.exe` avec la commande `"dir"`.
   – Taper `"Tp1"`.

Touche utile : Ctrl+F5 = = Start without debugging

9. *Fichiers :*
   On a déjà suivi la création des fichiers principaux au fur et à mesure. Constater 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 de Visual Studio. Quelle est la taille du répertoire `Tp1` (click droit + propriétés) ?
10. *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 ?
11. *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 comprimer. Il peut sinon y avoir une erreur ou certains fichiers trop importants peuvent subsister.

La politique de quotas fait qu'il est indispensable de nettoyer ses solutions quand on quitte. C'est essentiel aussi pour pouvoir envoyer ses exercices à son responsable. De plus, il faut quitter Visual Studio avant de comprimer.

12. *Envoi :*
Envoyer le fichier par mail à son responsable et indiquant bien le nom du binôme.
13. *Sauvegarde :*
Recopier aussi l'archive comprimée sur son compte (disque Z :)

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.

   (c) Faire une nouvelle modification. Tester une simple compilation (`Build/Compile` ou `Ctrl+F7`). Cette étape crée juste `Tp1.obj` à partir de `Tp1.cpp` sans aller jusqu'à `Tp1.exe`. Vérifier dans la fenêtre de commande avec `dir` que le fichier `Tp1.exe` est resté antérieur à `Tp1.cpp` et `Tp1.obj`. Vérifier qu'une génération ne lance plus ensuite que l'édition des liens sans avoir besoin de recompiler.

Touche utile : Ctrl+F7 = = Compile current file (without linking)

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 le `"` 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 sera en 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()` mais qu'il ne trouve nulle part.
   (c) Revenir à un programme qui marche et taper `mai` au lieu de `main`. En déduire le nom de la fonction qui appelle `main()` sans qu'on n'en soit conscient !
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.
Touche utile : Ctrl+K,Ctrl+F = indenter la zone sélectionnée.
Touche utile : Ctrl+A,Ctrl+K,Ctrl+F = tout indenter.

5. *Warnings du compilateur*
En modifiant le `main()`, provoquer les warnings suivants :
   (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.
   (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 debuggeur 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. Lancer le programme "sous le debuggeur" avec `Build/Start` ou `F5` ou le bouton correspondant. Que se passe-t'il ?
3. Placer un "point d'arrêt" en cliquant dans la colonne de gauche à la hauteur de la ligne `i=2;` puis relancer le debuggeur.
4. 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).
5. 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` !
6. Placer un deuxième point d'arrêt et voir que `F5` redémarre le programme jusqu'au prochain point d'arrêt rencontré.
7. 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`.
8. 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.

Touches utiles :					
	F5	=		=	**Debug**
	Maj+F5	=		=	**Stop**
	F10	=		=	**Step over**
	F11	=		=	**Step inside**

A.1.4 Deuxième programme

1. Ajouter un nouveau projet à la solution. Trouvez bien comment ne pas créer de nouvelle solution !.
2. Que fait `Build` ? Et `Start Without Debugging` ? Pourquoi ? Utiliser `Project/Set as startup project` pour exécuter le nouveau projet. L'exécuter. Le lancer aussi depuis une fenêtre console. Et depuis l'explorateur Windows.

   Question finale : donnez à nouveau votre programme à votre responsable (nettoyer, sortir de Visual, comprimer, envoyer par mail)

A.1.5 S'il reste du temps

Télécharger le programme supplémentaire `Tp1Supplement.zip` sur la page du cours (`http://certis.enpc.fr/~keriven/Info`), jouer avec... et le compléter !

A.1.6 Installer Visual Studio chez soi

Allez voir sur `http://certis.enpc.fr/~keriven/CertisLibs`.

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. Etudier 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.

A.2.2 Premier programme graphique avec la CLGraphics

Dans ce TP et les suivants, nous utiliserons la librairie graphique CLGraphics (cf annexe du polycopié). La CLGraphics 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
2   #include <CL/Graphics/Graphics.h>
```

```
3  using namespace CL::Graphics;
4  ...
5
6  ///////////////////////////////////////////////////////////
7  // Fonction principale
8  int main()
9  {
10     // Ouverture de la fenetre
11     OpenWindow(256,256);
12     // Position et vitesse de la balle
13     int xb=128,
14         yb=20,
15         ub=2,
16         vb=3;
17     // Boucle principale
18     while (true) {
19         // Affichage de la balle
20         FillRect(xb-3,yb-3,7,7,Red);
21         // Temporisation
22         MilliSleep(20);
23         // Effacement de la balle
24         FillRect(xb-3,yb-3,7,7,White);
25         // Rebond
26         if (xb+ub>253)
27             ub=-ub;
28         // Mise a jour de la position de la balle
29         xb+=ub;
30         yb+=vb;
31     }
32     Terminate();
33     return 0;
34 }
```

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 de duplications possible, 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.

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

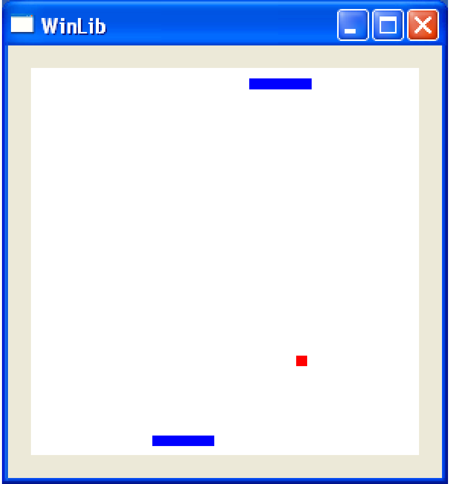


Fig. A.1 – Mini tennis...

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.

A la fin de la séance
- **Envoyez la solution nettoyée et compressée au responsable de votre groupe**
- **Pensez à sauvegarder votre travail sur Z :**

Vous devriez avoir obtenu un programme ressemblant à celui de la figure A.1.

```
Défilement "c:\Users\Renaud\TeX\Enseignement INFO\InfoENPC\2004\TPs\03\Tp3_final\Ma...
La combinaison que vous devez trouver fait 5 chiffres de long.
Ces chiffres sont compris entre 0 et 3
Vous avez droit a 10 essais.
Essai #1 : 0
Combinaison impossible. Recommencez!
Essai #1 : 123123
Combinaison impossible. Recommencez!
Essai #1 : 12345
Combinaison impossible. Recommencez!
Essai #1 : 00000
00000 - 0,0
Essai #2 : 11111
11111 - 1,0
Essai #3 : 02221
02221 - 3,0
Essai #4 : 33221
33221 - 3,2
Essai #5 : 32321
32321 - 5,0
VOUS AVEZ GAGNE !!!
Press any key to continue
```

FIG. A.2 – 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 et ouvrir la solution `MasterMind` dans Visual Studio. Etudier le projet `Mastermind`.
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])
{
    srand(unsigned int(time(0))); // initialisation
                                  //   du generateur
    for (int i=0; i<5; ++i)
        combin[i] = rand()%4;   // appels au generateur
}
```

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 nbcoul = 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';
}
```

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 `nbcoul-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...)

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 nécessaires[1], 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.

1. *Etude du projet de départ :*
   Passer dans le projet `Mastermind Graphique`. 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).

   Les fonctions graphiques sont déjà définies. Elle 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();
   ```

[1] même si la fonction de traitement d'une combinaison est pour l'instant incomplète

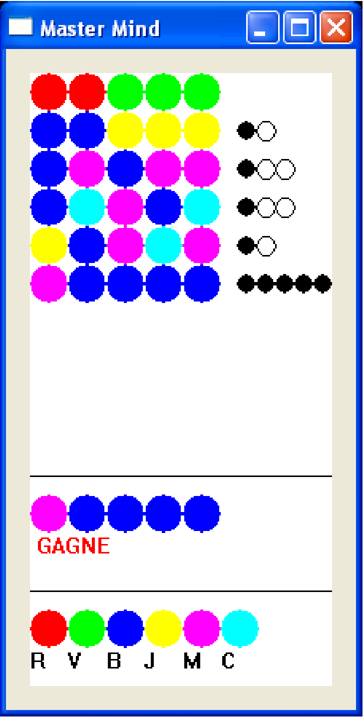


FIG. A.3 – Master mind graphique ...

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 la fin de la séance
- **Nettoyez et envoyez votre programme sous forme ZIP ou RAR à votre enseignant.**
- **Pensez à sauvegarder votre travail sur Z :**

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 deux semaines. 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 Visual C++. 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 (`vx`, `vy`), 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 $-G\, m_S/r^3\ \overrightarrow{r}$, 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 $x =$ `largeur`/2, $y =$ `hauteur`/3, $v_x = 1$, $v_y = 0$).

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 dûes 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 conséquent (inversement proportionnelle à dt). Lancer plusieurs fois le programme.

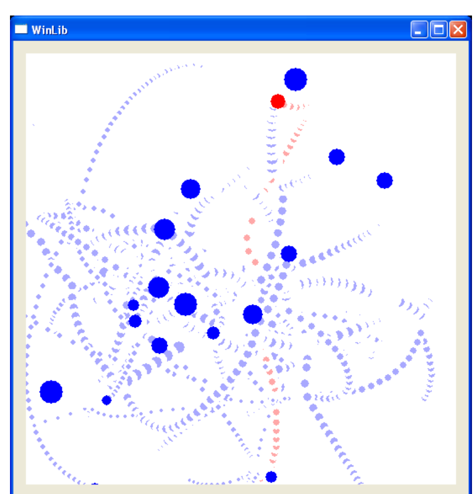
Chocs élastiques simples

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

(à droite figure A.4)

12. *Ouvrir un nouveau projet :*
    Afin de partir dans deux voies différentes et travailler proprement, ajouter un nouveau projet (WinLib5), appelé `Duel`, dans cette même solution, et recopier (par exemple par copier/coller) intégralement le contenu du fichier `main.cpp` du projet `Gravitation`.
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 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.



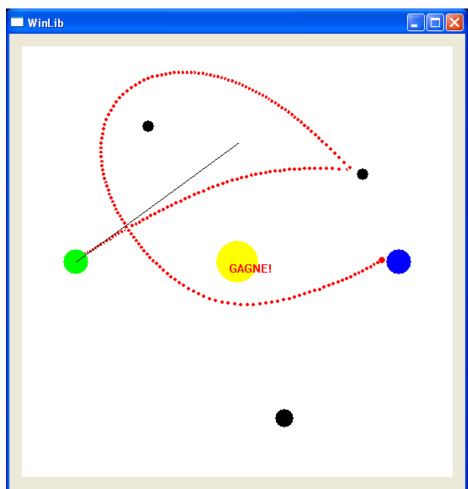


FIG. A.4 – Corps célestes et jeu de tir...

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

(à gauche figure A.4)

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.

$$\sum_i \overrightarrow{F}_{i/A} = m_A \overrightarrow{a}_{G(A)}$$

Gravitation universelle

Soient deux corps A et B. Alors A subit une force d'attraction

$$\overrightarrow{F}_{B/A} = -G m_A m_B \frac{1}{d_{A,B}^2} \overrightarrow{u}_{B\to A}.$$

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 $\overrightarrow{P} = m_A \overrightarrow{v}_A + m_B \overrightarrow{v}_B$
2. le moment cinétique $M = m_A \overrightarrow{r}_A \times \overrightarrow{v}_A + m_B \overrightarrow{r}_B \times \overrightarrow{v}_B$ (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. $\overrightarrow{P} = 0$ (par définition de ce référentiel), d'où $m_A \overrightarrow{v}_A = -m_B \overrightarrow{v}_B$.
2. $M = (\overrightarrow{r}_A - \overrightarrow{r}_B) \times m_A \overrightarrow{v}_A$, d'où, en notant $\Delta\overrightarrow{r} = \overrightarrow{r}_A - \overrightarrow{r}_B$, $M = \Delta\overrightarrow{r} \times m_A \overrightarrow{v}_A$.
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 : mutliplier 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 :*
   Télécharger le fichier Tp5.zip sur la page habituelle, le décompresser et lancer Visual C++. Le projet Gravitation contient une solution partielle au TP4 (jusqu'à la question 7, incluse). Si vous avez été plus loin et/ou si vous préférez réutiliser votre propre solution, vous pouvez quitter Visual C++, remplacer le fichier Gravitation.cpp par celui que vous aurez récupéré dans votre TP4, et relancer Visual C++.

A.5.1 Fonctions outils

2. *Un fichier de définitions...*
   Ajouter un nouveau fichier source nommé Tools.cpp au projet avec "Fichier / Ajouter un nouvel élément / Fichier C++". 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 :

```
#include <cstdlib>
#include <ctime>
using namespace std;
```

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 de deux vecteurs
Vector operator-(Vector a,Vector b);      // Différence de deux vecteurs
double norme2(Vector a);                  // Norme euclidienne d'un vecteur
Vector operator*(Vector a,double lambda); // Multiplication par un scalaire
Vector operator*(double lambda,Vector a); // Multiplication par un 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 <CL/Graphics/Graphics.h>
using namespace CL::Graphics;

#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

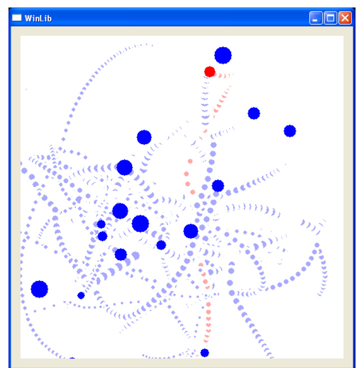
9. *Des soleils par milliers... :*
Placer 10 soleils aléatoirement (et en tenir compte à l'affichage, dans le calcul du déplacement de l'astéroïde...).
10. *Diminuer le pas de temps de calcul :*
Afin d'éviter les erreurs dues à la discrétisation du temps, diminuer le pas de temps dt, pour le fixer à 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.



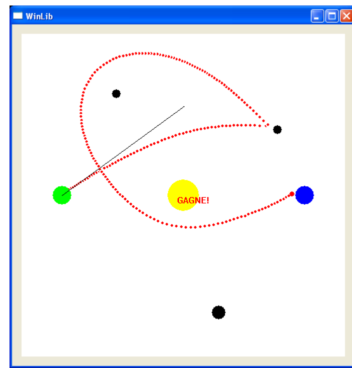


FIG. A.5 – Corps célestes et jeu de tir...

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 du `Duel.cpp` 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"` par exemple).
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.

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.

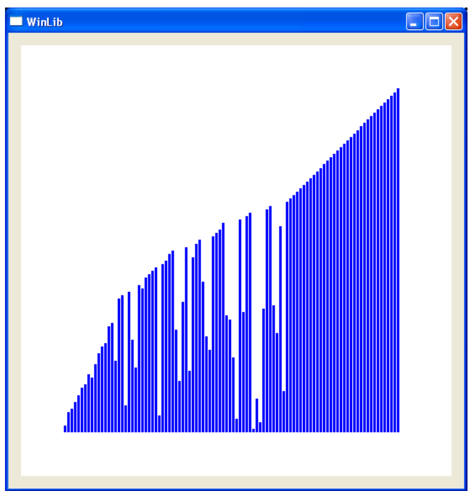
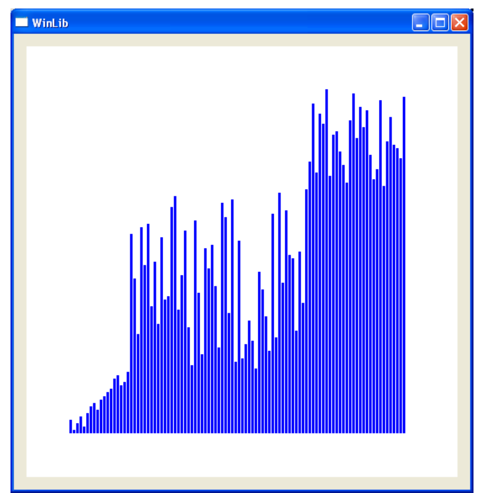



Fig. A.6 – Deux tris en cours d'exécution : tri à bulle et Quicksort...

A.6 Les tris

Dans ce TP, nous allons programmer quelques algorithmes de tri d'éléments dans un tableau. Pour que cela soit interactif, et que l'on puisse voir comment se passent chacun des tris, une interface graphique a été programmée, qui affiche le tableau et permet de visualiser les opération qu'on réalise dessus (figure A.6).

A.6.1 Mélanger un tableau

1. *Pour commencer, étudier le projet :*
   Télécharger le fichier `Tp6.zip` sur la page habituelle, le décompresser et lancer Visual C++. Le projet est séparé entre le fichier `main.cpp`, dans lequel on programmera les algorithmes de tri, et le couple (`tools.cpp`, `tools.h`), qui gère l'interface graphique et quelques fonctions utiles à la comparaison des différents tris.
   Ouvrir `tools.h` pour découvrir les fonctions de cette interface graphique, puis `main.cpp` pour voir comment elles sont utilisées (les lignes commentées sont là pour montrer comment vous utiliserez les fonctions `melange_tableau` et `tri_selection` que vous allez programmer).
   Exécuter le projet. Pour l'instant, il ne fait qu'initialiser un tableau et l'afficher.
2. *Accès à un tableau, échange de 2 valeurs*
   Pour que les opérations que vous effectuerez sur les tableaux soient affichées automatiquement, il faudra que vous utilisiez uniquement les fonctions `valeur` et `echange` déclarée dans `tools.h` (ne pas accéder directement au tableau avec `T[i]`).
   Entraînez-vous à les utiliser dans la fonction `main()`, en accédant à une valeur du tableau `T0`, et en permutant 2 de ses éléments.
3. *Mélanger un tableau*
   Une fonction déclarée dans `tools.h` n'existe pas encore dans `tools.cpp` : la fonction `void melange_tableau (double T[], int taille)` qui comme son nom l'indique mélange un tableau. Ajoutez-la (dans `tools.cpp` bien sûr)
   Idée : le mélange le plus rapide (et le plus efficace) consiste à parcourir le tableau une fois, et à permuter chacun des éléments avec un autre choisi au hasard (utiliser

la fonction `int random(int a)` définie dans `tools.cpp` pour tirer un entier entre `0` et `a-1`).

A.6.2 Tris quadratiques

Les 3 algorithmes de tri qui suivent trient un tableau en temps $0(n^2)$, c'est à dire que le temps pour trier un tableau est proportionnel au carré de la taille de ce tableau. Si vous avez peu de temps, vous pouvez ne faire qu'un ou deux des trois, pour pouvoir toucher également au tri Quicksort.

4. *Tri sélection*
   C'est le tri le plus naïf. Il consiste à parcourir le tableau une première fois pour trouver le plus petit élément, mettre cet élément au début (par une permutation), parcourir une seconde fois le tableau (de la 2ème à la dernière case) pour trouver le second plus petit élément, le placer en 2ème position, et ainsi de suite...
   Programmer la fonction `void tri_selection(double T[], int taille)`. Vous pouvez alors décommenter les lignes commentées dans `main()` et exécuter.
5. *Tri insertion*
   En général, c'est à peu près l'algorithme qu'utilise un être humain pour trier un paquet de cartes, des fiches...
   Il consiste à ajouter un à un les éléments non triés au bon endroit parmis ceux qui sont déjà triés : si nécessaire on échange les 2 premiers éléments du tableau pour les mettre dans l'ordre, puis (si nécessaire) on déplace le 3ème élément vers la gauche, par des échanges de proche en proche, jusqu'à ce qu'il soit à la bonne position par rapport aux 2 premiers, puis le 4ème, et ainsi de suite...
   Programmer `void tri_insertion(double T[], int taille)` et regarder comment ça se passe.
6. *Tri à bulle*
   Le tri à bulle consiste à parcourir n fois le tableau, et à chaque fois qu'on est sur un élément, on l'échange avec son voisin de droite si ce dernier est plus petit que lui. Programmer `tri_bulle(double T[], int taille)` et regarder comment ça se passe. Constater qu'on n'est pas obligé de parcourir tout le tableau à chaque fois et améliorer la fonction.

A.6.3 Quicksort

L'algorithme :
le tri Quicksort adopte la stratégie *"diviser pour régner"* qui consiste à réduire le problème du tri d'un tableau de taille n aux tris de 2 tableaux de taille $\frac{n}{2}$:
on choisit un élément dans le tableau (on le prend en général au hasard, mais par commodité on prendra ici le premier élément du tableau) qu'on appelle **pivot**. On sépare ensuite les autres éléments entre ceux inférieurs au pivot et ceux supérieurs au pivot. Il n'y a plus qu'à trier alors ces deux moitiés.

7. *Pivot*
   – Créer une fonction `int pivot(double T[], int taille)` qui prend comme pivot le premier élément du tableau, échange les éléments du tableau de manière à ce qu'on aie d'abord les éléments inférieurs au pivot, puis le pivot, puis les éléments supérieurs au pivot, et qui renvoie la nouvelle position du pivot.

- Ne pas oublier d'exécuter la fonction pour vérifier qu'elle marche bien !
- Idée :
  - on utilise 2 index qui parcourent le tableau, le premier à partir du 2ème élément (le 1er étant le pivot) et avançant vers la droite, le second à partir du dernier élément et avançant vers la gauche
  - le premier index s'arrête sur le premier élément supérieur au pivot qu'il rencontre, et le second index, sur le premier élément inférieur au pivot qu'il rencontre
  - on échange alors les 2 éléments, et les 2 index continuent d'avancer, et ainsi de suite
  - quand les 2 index se rencontrent, à gauche de l'intersection tous les éléments sont inférieurs au pivot, et à droite ils sont supérieurs
  - on échange le pivot (en 1ère position) avec le dernier des éléments inférieurs pour obtenir ce qu'on désire

8. *Fonctions récursives*
   Le principe même de la statégie *diviser pour régner* implique que la fonction qui effectue le tri quicksort va s'appeler elle-même pour trier une sous-partie du tableau. En pratique, on va utiliser 2 arguments `debut` et `fin` qui indiquent qu'on ne réalise le tri que entre les indices `debut` et `fin`.
   Changer la fonction `pivot` en lui ajoutant ces 2 arguments, et en ne la faisant effectivement travailler que entre ces 2 indices (par exemple, le pivot est initialement à l'indice `debut`)

9. *Quicksort*
   On peut maintenant écrire une fonction
   `void quicksort_recursif(double T[], int taille, int debut, int fin)`,
   qui contient l'algorithme, ainsi que la fonction
   `void quicksort(double  T[], int taille)`,
   qui ne fait qu'appeler cette dernière, mais qui sera celle qu'on utilisera dans `main()` (car plus simple).

A.6.4 Gros tableaux

Maintenant qu'on a vu graphiquement comment marchent ces algorithmes, il est intéressant de les faire fonctionner et de les comparer sur des tableaux de grande taille.

10. *Tableaux de taille variable*
    Si on veut tester nos algorithmes sur des tableaux de grande taille, il faut utiliser des tableaux de taille variable. Remplacer toutes les lignes de type `double t[taille];` par `double *t = new double[taille]`, et à la fin des fonctions où se trouvent ces déclarations, ajouter la ligne `delete[] t;`.
    D'autre part il faut désactiver l'affichage :
    `init_tools(512, false)`.

11. *Nombre de lectures et d'écriture*
    Pour comparer plus rigoureusement 2 algorithmes, on peut comparer le nombre de lectures du tableau et le nombre d'écriture dans le tableau.
    Créer 2 variables globales dans `tools.cpp` et modifier les fonctions `init_tri`, `valeur`, `echange` et `fin_tri` pour initialiser, compter et afficher le nombre de lec-

tures et d'écritures. **Au fait, en combien d'opérations s'effectue en moyenne Quicksort ?**

12. *Temps de calcul*
    Il est intéressant également d'avoir les temps de calcul exacts. Pour cela, on peut enregistrer dans une nouvelle variable globale `timer0` le temps qu'il est avant le tri :
    `timer0 = double(clock())/CLOCKS_PER_SEC ;`
    et la retrancher au temps qu'il est après le tri (modifier les fonctions `init_tri` et `fin_tri` pour faire ce calcul et l'afficher).
13. *Mode Release*
    On peut changer le mode de compilation en le passant de *Debug* à *Release*. Vérifier que l'exécution des algorithmes est effectivement bien plus rapide en mode *Release* !

A.7 Images

Fig. 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.7.1 Allocation

1. *Récupérer le projet :*
   Télécharger le fichier `Tp7.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.7.2 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*H` mémorisant le pixel `(i,j)` en `t[i+W*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.7.3 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.7.4 Charger un fichier

6. *Image couleur :*
La fonction `LoadColorImage("../ppd.jpg",r,g,b,W,H);` charge le fichier `"../ppd.jpg"` qui est dans le répertoire au dessus du projet, 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 : les tableaux `r,g,b` ne doivent pas être désalloués avec `delete` mais avec la fonction spéciale `DeleteImage()`.
– Charger cette image et l'afficher. Ne pas oublier les désallocations.

7. *Image noir et blanc :*
La fonction `LoadGreyImage("../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.7.5 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 :

```
void AlloueImage(byte* &I,int W,int H);
void DetruitImage(byte *I);
void AfficheImage(byte* I,int W,int H);
void ChargeImage(char* name,byte* &I,int &W,int &H);
```

9. *Fichiers :*
Créer un `image.cpp` et un `image.h` en conséquence...

A.7.6 Structure

10. *Principe :*
Modifier le programme précédent pour utiliser une structure :

```
struct Image {
  byte* t;
  int w,h;
};
```

`AlloueImage()` et `ChargeImage()` pourront retourner des `Image` au lieu de faire un passage par référence.

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 vise-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.7.7 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.8 Premiers objets et dessins de fractales

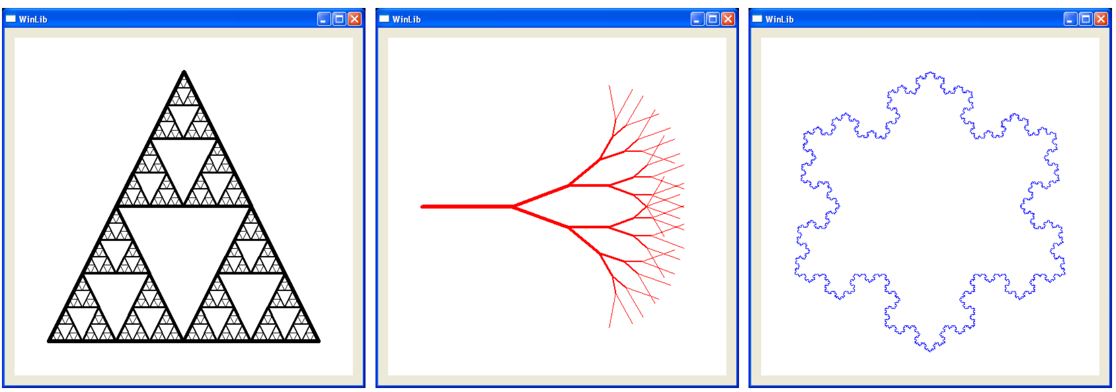


Fig. A.8 – Fractales...

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.8.1 Le triangle de Sierpinski

1. *Récupérer le projet :*
   Télécharger le fichier `Tp8.zip` sur la page habituelle, le décompresser et lancer Visual C++. Etudier la structure `Vector` définie dans les fichiers `Vector.cpp` et `Vector.h`.
2. *Interfaçage avec la WinLib :*
   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 &col,int pen_width)`
   de la WinLib. 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.
   Ecrire une fonction récursive pour dessiner le triangle de Sierpinski. Cette fonction

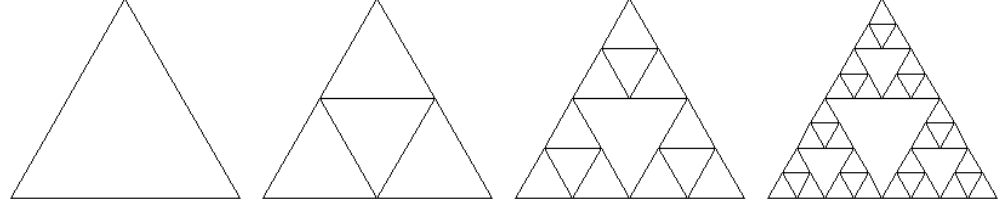


Fig. A.9 – Construction du triangle de Sierpinski.

prendra en paramètres les trois points du triangle en cours et l'epaisseur 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.8.2 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 membre :*
   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.
   Ecrire une fonction récursive pour dessiner une telle courbe. Vous aurez besoin de

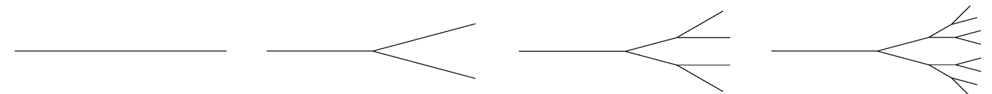


Fig. A.10 – Construction de l'arbre.

la fonction `Rotate` de la classe `Vector`.

A.8.3 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.8.4 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.
   Ecrire une fonction récursive pour dessiner une 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.

FIG. A.11 – Construction de la courbe de Koch.

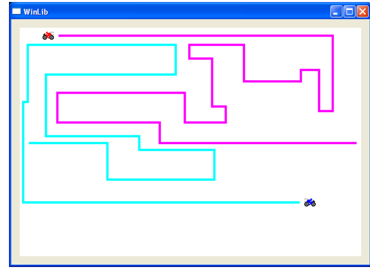


FIG. A.12 – Jeu de Tron.

A.9 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.

A.9.1 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 nmax 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 5 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 $x- > (x+1)\%4$ 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 $x- > (x-1)\%4$ 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 $p- > p+dir[d]$ 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.9.2 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 `Z`, `Q`, `S` et `D`. La fonction `Clavier()` renverra donc les entiers `int('Z')`, `int('Q')`, `int('S')` et `int('D')`. 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.9.3 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` de la CLGraphics, 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

Examens

B.1 Examen sur machine 2003 : énoncé

B.1.1 Crible d'Ératosthène

Rappel : *un entier naturel est dit premier s'il n'est divisible que par 1 et lui-même, et qu'il est différent de 1.*

Le but de cet exercice est de dresser par ordre croissant la liste des nombres premiers. On utilisera pour cela le crible d'Ératosthène, qui repose essentiellement sur le fait que les diviseurs éventuels d'un entier sont plus petits que lui. La méthode consiste à parcourir dans l'ordre croissant la liste des nombres entiers candidats (par exemple initialement tous les nombres de 2 à 999, si l'on se restreint aux entiers inférieurs à 1000), et, à chaque nombre rencontré, à retirer de la liste des nombres candidats tous les multiples de celui-ci. Une fois la liste parcourue, il ne restera que les nombres premiers.

On recherche les nombres premiers inférieurs ou égaux à n, n étant un nombre entier supérieur à 2, intialement fixé à 100. Plutôt que de gérer une liste d'entiers et d'en enlever des nombres, on travaillera avec un tableau de n booléens avec la convention que la i^{e} case du tableau contiendra `true` si i est premier, et `false` sinon.

1. Partir d'un projet "console" (`Win 32 Basic Console`).
2. Dans la fonction `main`, créer le tableau de booléens et le remplir avec des `true`.
3. Créer une fonction `multiples` qui prend en argument le tableau, sa taille et un entier a, et qui tourne à `false` les éléments du tableau correspondants à des multiples de a (excepté a, bien sûr).
4. Utiliser la fonction `multiples` de façon appropriée dans la fonction `main`, dans une boucle parcourant le tableau, afin d'exécuter le crible d'Ératosthène. Il est inutile de considérer les multiples des nombres qui ne sont pas premiers (à ce propos ne pas oublier que 1 consitue un cas particulier).
5. Afficher le nombre de nombres premiers inférieurs ou égaux à 211, ainsi que les nombres premiers en question.
6. Vérifier que l'on peut s'arrêter à $\sqrt{n}$.

B.1.2 Calcul de π par la méthode de Monte Carlo

On désigne par *méthode de Monte Carlo* une méthode de résolution d'un problème mathématique à l'aide de suites de nombres aléatoires convergeant vers le résultat, en

référence aux nombreux casinos monégasques. Le méthode de Monte Carlo classique pour calculer une valeur approchée de π consiste à tirer aléatoirement un grand nombre n de fois des points (x, y) du plan avec $-1 \leqslant x \leqslant 1$, $-1 \leqslant y \leqslant 1$ puis de déterminer quelle est la proportion p_n de ces points qui sont situés dans le disque unité. Comme la surface de ce dernier vaut π, on peut montrer que $lim_{n \to +\infty} p_n = \frac{\pi}{4}$. Le but de cet exercice est de programmer ce calcul et une visualisation graphique de celui-ci.

Fonctions utiles

– *Nombres pseudo-aléatoires*

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

void srand(unsigned int seed); // Initialise le gestionnaire de nombres
                               // pseudo-aléatoires

int rand(); // Retourne un nombre pseudo-aléatoire entre 0 et RAND_MAX
            // inclus
```

– *Temps*

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

long time(...); // time(0) renvoie le nombre de secondes écoulées
                // depuis le 1er janvier 1970
```

– *WinLib*

```
#include <win>
using namespace Win;

void DrawPoint(int x,int y,const Color& col); // Affiche un pixel à
                                              // l'écran

void DrawCircle(int xc,int yc,int r,const Color& col); // Affiche un cercle
                                                       // centré en (xc,yc),
                                                       // de rayon r
                                                       // et de couleur col
```

1. Ajouter un nouveau projet WinLib nommé `MonteCarlo` à la solution et le définir cmme projet de démarrage.
2. Définir dans `main.cpp` une constante globale `taille_fenetre` et l'utiliser dans l'appel à `OpenWindow`. Dans tout ce qui suit, la fenêtre sera carrée, représentant l'ensemble des points dont les coordonnées sont entre -1 et 1.
3. Ajouter au projet deux fichiers `Point.h` et `Point.cpp` dans lesquels vous déclarerez et définirez à l'endroit approprié :

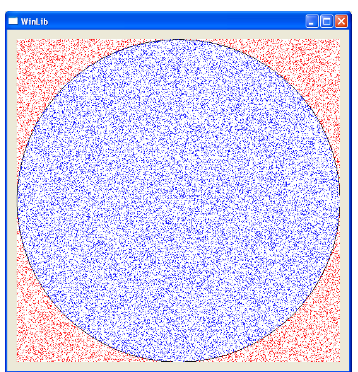
   (a) une structure PointPlan représentant des points du plan (il faut en particulier pouvoir avoir des coordonnées entre -1 et 1)

   (b) une fonction GenerePoint retournant un PointPlan dont les coordonnées sont tirées aléatoirement entre -1 et 1. *Ne pas oublier qu'en C++, la division de deux entiers est la division entière !*

   (c) une fonction AffichePoint, prenant en argument la taille de la fenêtre d'affichage, une couleur et un PointPlan, et qui affiche ce dernier à l'écran dans la couleur précisée (*penser à renormaliser !*).

4. Appeler la procédure d'initialisation du gestionnaire de nombre pseudo-aléatoires au début de `main` avec `srand(unsigned int(time(0)));`. Penser aux `#include` et `using namespace` correspondants.
5. Dessiner à l'écran le cercle unitaire au début de `main.cpp`.
6. Définir une constante globale `nb_iterations` dans `main.cpp` représentant le nombre d'itérations effectuées. Construire une boucle effectuant `nb_iterations` fois les opérations suivantes :
   (a) Générer un point pseudo-aléatoire
   (b) Incrémenter une variable `compteur` si ce point est dans le disque unitaire (c'est-à-dire si $x^2+y^2 \leqslant 1$). On rajoutera une fonction retournant le carré de la norme d'un point.
   (c) Afficher ce point à l'écran, en bleu s'il est dans le disque, en rouge sinon.
7. Afficher à la fin du programme la valeur de π déduite.
8. Varier `nb_iterations` afin d'avoir un bon compromis temps d'affichage / précision de la valeur calculée. Que constate-t-on ? Pourquoi ? (Répondre en commentaire du programme)

B.1.3 Serpent



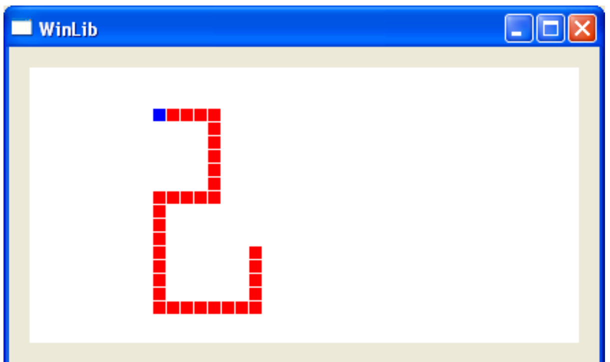


FIG. B.1 – A gauche : calcul de π. A droite : un serpent.

Le but de cet exercice est de programmer un serpent qui se déplace sur un terrain rectangulaire en évitant de se mordre lui-même. Pour cela structures et tableaux seront utiles. Ne pas oublier de compiler et tester à chaque étape... NB : certaines questions sont indépendantes.

1. *Structure :* Partir d'un projet WinLib. Créer une structure `point` mémorisant deux coordonnées entières `x` et `y`.
2. *Dessin :* On va dessiner un serpent de déplaçant sur un terrain de dimensions `(w,h)`. Pour bien le visualiser, on l'affichera dans un fenêtre de dimensions `(w*z,h*z)` (Prendre par exemple `(w,h,z)=(40,30,10)`).

(a) Programmer une fonction `void dessine(point p,int zoom,Color c)` utilisant la fonction
`FillRect(x,y,w,h,c)` pour dessiner un carré plein de coin supérieur gauche `(p.x*zoom,p.y*zoom)`, de côté `zoom-1` et de couleur `c`.

(b) Ecrire une fonction `void dessine(point s[],int n,int zoom)` dessinant un serpent constitué des points `s[0]` à `s[n-1]`, `s[0]` étant la tête. On pourra tracer le corps du serpent en rouge et sa tête en bleu (figure B.1).

(c) Initialiser un serpent dans un tableau de dimension constante `n=10` et l'afficher.

3. *Divers :* programmer quatre fonctions utiles pour la suite :

   (a) Une fonction `bool operator==(point a,point b)` retournant vrai si les points `a` et `b` sont égaux, ce qui permettra d'écrire un test comme `if (p1==p2)`.

   (b) Une fonction `bool cherche(point s[],int n,point p)` retournant vrai si le point `p` se trouve dans le tableau `s` de taille `n`.

   (c) Une fonction `void decale(point s[],int n,point p)` qui décale le tableau `s` vers le haut (i.e. `s[i]=s[i-1]`) et range `p` dans `s[0]`. Cette fonction sera utile pour la suite !

   (d) Une fonction `point operator+(point a,point b)` calculant la somme de deux points (considérés comme des vecteurs).

4. *Avancer d'une case :* Le serpent peut avancer dans une des quatre directions. On mémorise la direction de son déplacement par un entier de 0 à 3, avec la convention `(0,1,2,3)=(est,sud,ouest,nord)`.

   (a) Créer et initialiser une variable globale `const point dir[4]` telle que `dir[d]` correspond à un déplacement dans la direction `d`. Ainsi, `d[0]={1,0}`, `d[1]={0,1}`, etc.

   (b) Ecrire et tester une fonction `void avance(point s[],int n,int d,int zoom)` utilisant ce qui précède pour :
   – Avancer le serpent `s,n` dans la direction `d`.
   – Afficher sa nouvelle position sans tout re-dessiner (effacer la queue et dessiner la tête).

5. *Avancer :* Ecrire les quatre fonctions suivantes :
   – `bool sort(point a,int w,int h)` qui retourne vrai si le point `a` est en dehors du terrain.
   – `void init_rand()` qui initialise le générateur aléatoire.
   – `void change_dir(int& d)` qui change aléatoirement `d` en `d-1` une fois sur 20, en `d+1` une fois sur 20 et ne modifie pas `d` les 18 autres fois, ce qui, compte tenu de la convention adoptée pour les directions, revient à tourner à droite ou à gauche de temps en temps. Attention à bien conserver `d` entre 0 et 3.
   – `bool ok_dir(point s[],int n,int d,int w,int h)` qui teste si le déplacement du serpent dans la direction `d` ne va pas le faire sortir.

   puis les utiliser pour faire avancer le serpent dans une boucle de 500 pas de temps, en vérifiant qu'il ne sort pas du terrain. Penser à rajouter un `MilliSleep()` si le déplacement est trop rapide (sur certains PC de l'ENPC, la WinLib ne sera pas à jour et l'affichage sera lent et saccadé... Ignorer.)

6. *Ne pas se manger :* modifier `ok_dir()` pour que le serpent ne se rentre pas dans lui-même.

7. *Escargot :* le serpent peut se retrouver coincé s'il s'enroule sur lui-même. Programmer une fonction `bool coince(point s[],int n,int w,int h)` qui teste si aucune direction n'est possible et modifier la boucle principale pour terminer le programme dans ce cas.
8. *Grandir* (question finale difficile) :
   – Changer le programme pour que le tableau `s` soit alloué dans le tas (`n` pourra alors être variable).
   – Un pas de temps sur 20, appeler, à la place de `avance()`, une fonction `void allonge()` qui avance dans le direction `d` **sans supprimer la queue**. Du coup, `s` doit être réalloué et `n` augmente de 1. Ne pas oublier de désallouer l'ancienne valeur de `s`. Bien gérer l'affichage, etc. Cette fonction sera déclarée `void allonge(point*& s,int& n, int d,int zoom)`.

B.2 Examen sur machine 2004 : énoncé

B.2.1 Calcul de l'exponentielle d'un nombre complexe

Le but de cet exercice est de calculer l'exponentielle d'un nombre complexe et de s'en servir pour calculer le sinus et le cosinus d'un angle.

1. Partir d'un projet "console" `Win 32 Basic Console`
2. Ajouter au projet deux fichiers `complexe.cpp` et `complexe.h` dans lesquels vous déclarerez et définirez à l'endroit approprié :
   (a) une structure `complexe` (et pas `complex` qui existe déjà) représentant un nombre complexe sous forme cartésienne (partie réelle, partie imaginaire)
   (b) les opérateurs `+`, `*` entre deux `complexe`
   (c) l'opérateur `/` définissant la division d'un `complexe` par un `double`
3. On souhaite approximer la fonction exponentielle en utilisant son développement en série entière :

$$e^z = \sum_{i=0}^{+\infty} \frac{z^i}{i!}$$

   Écrire une fonction `exponentielle`, prenant en argument un `complexe z` et un `int n`, et qui retourne la somme :

$$\sum_{i=0}^{n} \frac{z^i}{i!}$$

4. Écrire une fonction `cos_sin` qui renvoie le cosinus et le sinus d'un angle θ en utilisant le développement limité de $e^{i\,\theta}$ à l'ordre n (on passera donc, en plus de l'angle `theta`, l'entier `n` en argument). On rappelle que :

$$e^{i\,\theta} = \cos\theta + i\,\sin\theta$$

5. Tester la fonction `cos_sin` pour différentes valeurs de `theta` et `n`. Vérifier qu'avec $n = 15$ et $\theta = \frac{\pi}{6}$, on obtient une bonne approximation des valeurs du cosinus ($\frac{\sqrt{3}}{2} \approx 0.866025404$) et du sinus ($\frac{1}{2}$).

B.2.2 Compression RLE

Dans cet exercice nous allons implémenter l'une des plus anciennes méthodes de compression : le codage RLE (Run Length Encoding). Le principe consiste à détecter une donnée ayant un nombre d'apparitions consécutives qui dépasse un seuil fixe, puis à remplacer cette séquence par deux informations : un chiffre indiquant le nombre de répétitions et l'information à répéter. Aussi, cette méthode remplace une séquence par une autre beaucoup plus courte moyennant le respect du seuil (que nous fixerons, par simplicité, à 0). Elle nécessite la présence de répétitions relativement fréquentes dans l'information source à compresser.

Cette méthode présente peu d'avantages pour la compression de fichier texte. Par contre, sur une image, on rencontre régulièrement une succession de données de même valeur : des pixels de même couleur.

Sur une image monochrome (un fax par exemple), l'ensemble à compresser est une succession de symboles dans un ensemble à deux éléments. Les éléments peuvent être soit des 255 (pixel allumé=blanc), soit des 0 (pixel éteint=noir) ; il est relativement facile de compter alternativement une succession de 0 et de 255, et de la sauvegarder telle quelle.

source : 0 0 0 0 0 0 0 0 0 0 255 255 255 255 255 255 255 255 0 0 0 0 0 255 255 255 255
compression : 10 8 5 4

Cependant, une convention reste à prendre : savoir par quel pixel commencera la succession de chiffres. Nous considèrerons que le premier nombre représente une succession de 0. Si la source ne commence pas par des pixels noirs (un zéro) il faut alors commencer la chaine codante par un 0 pour indiquer l'absence de pixel noir.

Dans ce qui suit nous allons encoder une image binaire (constituée de 0 et de 255 stockés dans un tableau de `byte`) dans un tableau de `int`.

1. Partir d'un projet "Winlib" (`Winlib5 Project`).
2. Dans la fonction `main`,
   (a) Déclarer les variables de type `const int` w et h representant la largeur et la hauteur de l'image à encoder. Ces variables pourront être initialisées à la valeur 256.
   (b) Déclarer le tableau de `byte` $image_source$ de taille $w \times h$ (rappel : le pixel (i, j) est l'élément $i + j * w$ du tableau).
   (c) Déclarer le tableau de `byte` $image_decodee$ de taille $w \times h$ (rappel : le pixel (i, j) est l'élément $i + j * w$ du tableau).
   (d) Déclarer le tableau de `int` $image_encodee$ de taille $w \times h + 1$
3. Créer une fonction $affiche_image$ de déclaration :
   `void affiche_image( byte image[], int w, int h );`
   Cette fonction affiche l'image stockée dans *image* de largeur w et de hauteur h à l'aide de la fonction *PutGreyImage* de la Winlib.
4. Dans la fonction *main*, tester la fonction $affiche_image$ avec $image_source$, sans l'avoir initialisée.
5. Créer une fonction $remplir_rectangle$ de déclaration :

```
void remplir_rectangle(
        byte image[], int w, int h,
        int xul, int yul, int height, int width );
```

   Cette fonction dessine un rectangle blanc dans une image en mettant à 255 tous les pixels de l'image contenus dans le rectangle. L'image dans laquelle est dessiné le rectangle plein est donnée par le tableau *image* et les dimensions w et h de l'image. Le rectangle à remplir est donné par les coordonnées *xul* et *yul* de son coin supérieur gauche ainsi que par ses dimensions *height* et *width*.
6. Nous allons maintenant tester la fonction $remplir_rectangle$. Pour ce faire :
   (a) Remplir l'image avec des 0.
   (b) Utiliser la fonction $remplir_rectangle$ sur cette image pour y placer un rectangle de votre choix.
   (c) Afficher cette image à l'aide de $affiche_image$

7. Créer une fonction `RLE_encode` de déclaration :

```
void RLE_encode(
        byte source_image[], int w, int h,
        int compression[], int &comp_size );
```

   L'image à compresser est donnée par sa largeur w, sa hauteur h et par un tableau *source_image*. Le résultat de la compression est stocké dans le tableau *compression* et le nombre d'éléments utilisés dans ce tableau *comp_size*. (pour rappel, le tableau prévu pour recevoir l'image comprimée est déclarée de manière statique dans *main* et $w \times h + 1$ n'est qu'un majorant de la taille de l'image comprimée.)

8. Créer une fonction `RLE_decode` de déclaration :

```
void RLE_decode( int compression[], int comp_size,
        byte decomp_image[] );
```

   L'image à décompresser est donnée par un tableau *compression* et la taille des données dans ce tableau *comp_size*. Le resultat de la décompression est stocké dans le tableau *decomp_image*.

9. Dans la fonction `main` et à l'aide des fonctions précédemment définies :
   (a) Remplir dans le tableau `image_source` deux rectangles de votre choix.
   (b) Afficher cette image
   (c) Encoder *image_source* dans *image_encodee*
   (d) Decoder *image_encodee* dans *image_decodee*
   (e) A l'aide d'une boucle, vérifier que les deux images *image_source* et *image_decodee* sont identiques.
   (f) Afficher l'image décodée et valider la vérification précédente de manière visuelle.

10. Créer une fonction *remplir_diagonale* de déclaration :

```
void remplir_diagonale( byte image[], int w, int h );
```

   Cette fonction met tous les pixels (i, j) tels que $i == j$ de *image* à 255.

11. Tester la fonction *remplir_diagonale* de la même façon que la fonction *remplir_rectangle* a été testée.

12. On fixe $w = 256$ et $h = 128$
   (a) Remplir l'image source de zéros.
   (b) Remplir dans l'image source un rectangle dont le coin supérieur gauche est en $(20, 20)$ de dimensions $(80, 80)$
   (c) Remplir dans l'image source un second rectangle dont le coin supérieur gauche est en $(120, 10)$ de dimensions $(100, 100)$
   (d) Compresser l'image source et afficher la taille de l'image compressée *comp_size*
   (e) Remplir à nouveau l'image source de zéros.
   (f) Remplir dans l'image source la diagonale.
   (g) Compresser l'image source et afficher la taille de l'image compressée *comp_size*

13. On fixe $w = 1024$ et $h = 1024$
   (a) Un 'plantage' se produit.
   (b) Identifier le problème et modifier le programme pour qu'il fonctionne avec les valeurs de w et h données.

B.3 Examen sur machine 2005 : énoncé

Le but de ce problème est de créer et d'animer un modèle 3D et d'afficher un mouvement de cet objet vu par une caméra. Pour cela, nous procédons par étapes.

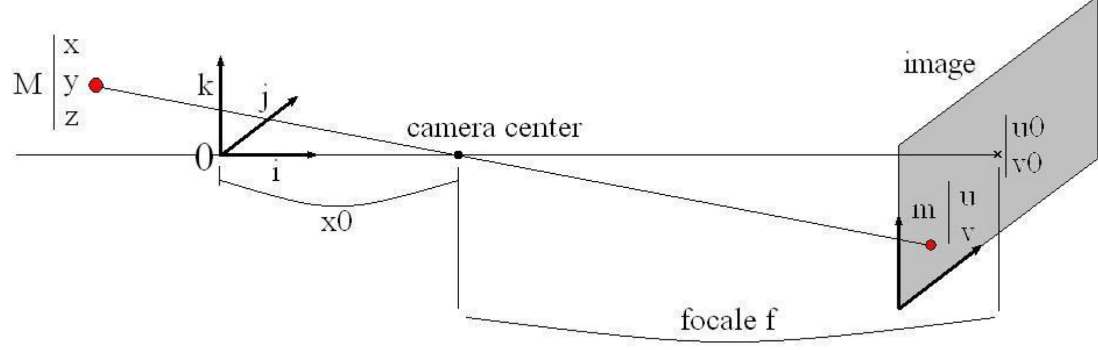
B.3.1 Construction du Modèle 3D

1. Travailler en local sous `D:\Info\` : créer un dossier `nom_prenom` dans `D:\Info\` (remplacer `nom_prenom` par son nom et son prénom !).
2. Créer une solution `Exam` dans `D:\Info\nom_prenom\` et ajouter un projet "Winlib" (`Winlib5 Projet`) de nom `logo` à la solution `Exam`.
3. Ajouter au projet deux fichiers `vect.cpp` et `vect.h` dans lesquels vous déclarerez et définirez à l'endroit approprié :
   (a) une structure `vect` représentant un vecteur de $\mathbf{R}^3$ (de composantes `double x,y,z` dans la base canonique (O,i,j,k) de $\mathbf{R}^3$),
   (b) un opérateur `+` entre deux `vect`.
4. On souhaite construire une structure représentant un tétraèdre. Ajouter au projet deux fichiers `tetra.cpp` et `tetra.h` et :
   (a) définir une structure `tetra` contenant 4 sommets (`vect M[4]`) et une couleur `Color c`,
   (b) déclarer et définir une fonction `regulier` ne prenant aucun argument et retournant un tétraèdre régulier de couleur rouge. Pour information, les quatre points suivant forment un tétraèdre régulier :
   $$\{1,0,0\},\{-\tfrac{1}{2},\tfrac{\sqrt{3}}{2},0\},\{-\tfrac{1}{2},-\tfrac{\sqrt{3}}{2},0\},\{0,0,\sqrt{2}\}.$$
   On rappelle que la fonction `sqrt` est définie dans `#include <cmath>`.

B.3.2 Projection : du 3D au 2D

Nous nous préoccupons maintenant de la projection 2D de ces représentations 3D. Ajouter au projet deux fichiers `camera.cpp` et `camera.h` dans lesquels vous déclarerez et définirez :

1. une structure `camera` contenant les variables suivantes : `int u0,v0` (le centre de l'image), `double x0` (l'éloignement), et `double f` (la focale). La figure ci-après schématise la représentation d'une caméra.



2. une fonction `projette` qui projette un point 3D (`vect M`) par l'intermédiaire d'une caméra (`camera c`). La fonction `projette` retourne un point 2D de type `Pixel`. Nous rappelons que le type `Pixel` est défini dans la WinLib :

```
struct Pixel{
    int x,y;
};
```

   Par conséquent, ne pas oublier d'ajouter les lignes `#include <win>` et `using namespace Win` dans le fichier `camera.h`. Les coordonnées `(u,v)` du projeté d'un point 3D de coordonnées `(x,y,z)` sont :

$$u = \text{c.u0} + \frac{\text{c.f * y}}{\text{c.x0-x}}$$
$$v = \text{c.v0} - \frac{\text{c.f * z}}{\text{c.x0-x}}.$$

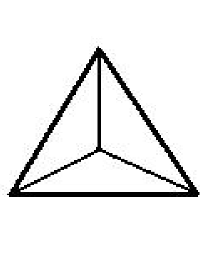
B.3.3 Affichage à l'écran

Nous sommes maintenant prêts pour afficher notre objet tétraèdre à l'écran.

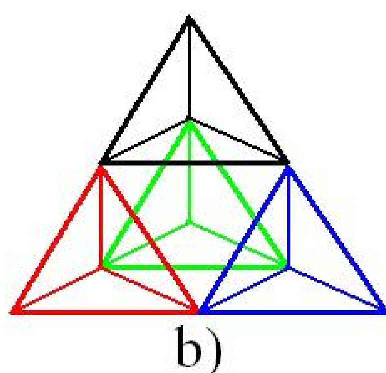
1. Dans les fichiers `tetra.cpp` et `tetra.h`, déclarer et définir une fonction `affiche` prenant en argument un `camera c`, un `tetra T`. Cette fonction dessine un tétraèdre `tetra T` dans sa couleur, vu par la camera `camera c`. Pour cela, on utilisera la fonction `projette` (déclarée et définie dans `camera.h` et `camera.cpp`), ainsi que la fonction `DrawLine` (définie dans la WinLib) pour tracer un segment entre deux `Pixel`. La définition de la fonction `DrawLine` est la suivante :

   `void DrawLine(const Pixel& p1,const Pixel& p2,const Color& col).`

   Le projeté d'un tétraèdre doit ressembler à la figure ci-dessous (figure-a) :



a)



b)

2. Dans le fichier `main.cpp`, créer un tétraèdre régulier et l'afficher à l'écran dans une image de taille 512 × 512. Pour cela, créer une caméra `camera C` de paramètres :

   `u0=256,v0=256`, `double x0=10`, et `double f=500`.

B.3.4 Animation du tétraèdre

Nous cherchons maintenant à animer notre modèle 3D.

1. Rotation d'un `vect` : dans `vect.h` et `vect.cpp`, ajouter une fonction `rotate` qui applique une rotation d'angle `double alpha` autour de l'axe Oz sur un vecteur

`vect a` et qui renvoie un `vect b`. Les fonctions `cos` et `sin` sont définies dans `#include <cmath>`.

$$\begin{aligned} \text{b.x} &= \cos(\alpha)\ \text{a.x} - \sin(\alpha)\ \text{a.y} \\ \text{b.y} &= \sin(\alpha)\ \text{a.x} + \cos(\alpha)\ \text{a.y} \\ \text{b.z} &= \text{a.z} \end{aligned}$$

2. Rotation d'un `tetra` : dans `tetra.h` et `tetra.cpp`, déclarer et définir une fonction `rotate` qui applique une rotation d'angle `double alpha` autour de l'axe Oz sur un `tetra` et qui retourne un `tetra`.
3. Première animation : dans `main.cpp`, animer le tétraèdre d'un mouvement de rotation sur place (pour l'instant sans ce soucier de l'effacer) : dans une boucle `for (int i=0;i<10000;i++)`, appliquer une rotation d'angle $\frac{i}{15}$ à chaque pas de temps. On pourra utiliser la fonction `MilliSleep(10)`.
4. Deuxième animation : maintenant, nous souhaitons afficher et effacer notre tétraèdre en mouvement :
   (a) dans les fichiers `tetra.cpp` et `tetra.h`, déclarer et définir une fonction `changeColor` prenant en argument un `tetra T` et un `Color c` et retournant une structure `tetra` (de couleur `Color c` et dont les sommets possèdent les mêmes coordonnées que celles de `tetra T`),
   (b) ajouter une fonction pour effacer un tétraèdre (utiliser les fonctions définies précédemment !).
   (c) dans `main.cpp`, animer le tétraèdre d'un mouvement de rotation en effaçant proprement à chaque pas de temps.

B.3.5 Un modèle plus élaboré

1. Dans le fichier `main.cpp`, créer un tableau `tetra tetras[4]` de 4 structures `tetra`. Initialiser les 4 tétraèdres du tableau de manière à produire la figure representée ci-dessus (figure-b). Pour cela, générer 4 tétraèdres "réguliers" et appliquer l'une des 4 translations suivantes sur chacun d'entre eux :

   $$\{1, 0, 0\}, \{-\tfrac{1}{2}, \tfrac{\sqrt{3}}{2}, 0\}, \{-\tfrac{1}{2}, -\tfrac{\sqrt{3}}{2}, 0\}, \{0, 0, \sqrt{2}\}.$$

   Définir pour cela une fonction `translate` pour générer le translaté d'un tétraèdre `tetra T` par un vecteur `vect t` (utiliser l'opérateur `+` de `vect`).
2. Finalement, nous allons animer notre objet `tetra tetras[4]` d'un mouvement complexe et afficher ce dernier dans la caméra `camera C`. A chaque pas de temps dans la boucle `for (int i=0;i<10000;i++)`, appliquer une rotation d'angle $\frac{i}{15}$ suivie par une translation de vecteur :
   `vect t={-12+8*cos(i/150.),8*sin(i/150.0),-3.0}`.

B.4 Examen sur machine 2006 : énoncé

B.4.1 Voyageur de commerce par recuit simulé

Soient n villes situées aux points $M_1, ..., M_n$, le problème du voyageur de commerce consiste à trouver un circuit fermé de longueur minimale passant par toutes les villes. Il s'agit donc de permuter les villes entre elles pour minimiser

$$l(M_1, ..., M_n) = \Sigma_{i=1}^{n-1} d(M_i, M_{i+1}) + d(M_n, M_1)$$

où $d(A, B)$ est la distance entre A et B. Une méthode classique pour trouver une solution approchée à ce problème est de procéder à un "recuit simulé" :

1. Partir d'un circuit quelconque C
2. Pour un "grand" nombre de fois
   (a) Modifier aléatoirement C en un circuit D
   (b) Si $l(D) < l(C)$ alors remplacer C par D
   (c) Sinon, remplacer C par D avec une probabilité $e^{-(l(D)-l(C))/T}$, où T est une constante à choisir.

B.4.2 Travail demandé

1. **Travailler en local dans** `D:\nom_prenom` ;
2. Y <u>créer</u> une solution `Examen` et lui <u>ajouter</u> un projet "Winlib" de nom `voyageur` ;
3. Un circuit sera mémorisé comme un tableau de `Pixel`[1] de taille constante `n` (valeur raisonnable : $n = 20$) ;
4. Faire et utiliser une fonction qui génère un circuit correspondant à n villes situées en des positions aléatoires dans la fenêtre ;
5. Faire et utiliser une fonction qui affiche ce circuit[2] ;
6. Faire et utiliser une fonction qui calcule la longueur de ce circuit ;
7. Faire et utiliser une fonction qui transforme un circuit en un autre par échange de deux villes choisies au hasard ;
8. Implémenter le recuit simulé sans l'étape (c). Afficher le circuit et sa longueur à chaque fois qu'il change. L'algorithme devrait rester coincé dans un minimum local ;
9. Rajouter l'étape (c). On rappelle que `double(rand())/RAND_MAX` est un nombre aléatoire entre 0 et 1. Ajuster T pour ne pas toujours remplacer C par D !
10. Choisir maintenant une valeur de T qui décroît en $1/\sqrt{t}$, où t est le nombre d'essais, de façon à accepter de plus en plus rarement un D plus long que C ;
11. Pour vraiment faire marcher l'algorithme, il faut programmer une nouvelle façon de transformer un circuit en un autre. La voici : choisir deux villes au hasard et retourner le chemin entre les deux villes, c'est-à-dire

    $$\text{transformer } (M_1, ..., M_n) \text{ en } (M_1, ..., M_{i-1}, M_j, M_{j-1}, ..., M_{i+1}, M_i, M_{j+1}, ..., M_n)$$

    Programmer cette nouvelle façon de faire ;
12. A la fin de l'examen, **ne pas vous déconnecter et attendre que le surveillant passe recupérer votre travail**.

---

[1] On rappelle que `Pixel` est une structure de la Winlib, contenant les champs `double x` et `double y`

[2] Les fonctions de la Winlib telles que "OpenWindow" ou "DrawLine" (confère Annexe C du poly) seront *très* utiles.

B.5 Devoir écrit 2003 : énoncé

B.5.1 Tableau d'exécution

Pour ce premier exercice, il s'agit d'exécuter pas à pas le programme suivant :

```
1   int hop(int x) {
2           x = x/2;
3           return x;
4   }
5
6   int hip(double& y) {
7           y = y/2;
8           return (int(y));
9   }
10
11  int test(double z) {
12          z = 2*hip(z) - z;
13          if (z>4)
14                  z = test(z);
15          return (int(z));
16  }
17
18  int main() {
19
20          double a = 3.14;
21
22          int b = hop(int(a));
23          b = hip(a);
24
25          a = 18;
26          do {
27                  a = test(a);
28          } while (a != 0);
29
30          return 0;
31  }
```

Pour cela, remplissez le tableau ci-dessous, en écrivant, si elles existent, les valeurs des variables du programme pour chaque exécution de ligne (selon l'exemple du cours).

Conventions :
– mettre le numéro de la ligne qu'on vient d'exécuter (éventuellement, on peut découper l'exécution d'une même ligne en plusieurs étapes)
– on peut utiliser le symbole " pour répéter la valeur d'une variable à la ligne suivante
– pour une variable qui n'est autre qu'une référence sur une autre variable x, on indiquera $[x]$;

Ligne	a_{main}	b_{main}	ret_{main}	x_{hop}	ret_{hop}	x_{hip}	ret_{hop}	$x_{test(1)}$	$ret_{test(1)}$	$x_{test(2)}$	$ret_{test(2)}$

B.5.2 Grands entiers

Le but de cet exercice est de manipuler des entiers arbitrairement grands. Un entier long sera un entier compris entre 0 et $10^{4n} - 1$, où n est une constante fixée une fois pour toutes, par exemple $n = 30$. On découpera les grands entiers comme si on les considérait écrits en base 10^4, c'est-à-dire sous la forme d'un tableau de n `int`.

1. Créer une classe `entier` comprenant un tableau d'entiers de taille n.
2. Le tableau d'un entier contient la représentation de celui-ci en base 10^4, la première case contenant l'élément de poids le plus faible. Par exemple 15 439 573 458 se codera sous la forme 3458 3957 154. En conséquent, on a besoin de connaître l'indice de l'élément de poids le plus fort afin de savoir où arrêter la lecture du tableau. Ajouter à la classe un champ de type `int` servant à stocker l'indice en question.
3. Ajouter à cette classe un constructeur vide.
4. Définir un constructeur prenant en argument un `int` positif créant un `entier` de même valeur.
5. Ajouter à la classe une méthode affichant à l'écran un `entier`, dans l'ordre standard pour la lecture.
6. Ajouter l'opérateur d'addition (attention aux retenues !).
7. Ajouter l'opérateur de multiplication. On pourra procéder en 3 temps : créer une fonction multipliant un entier par un `int` ($< 10^4$), une autre multipliant un `entier` par 10^4, et utiliser de façon adéquate ces deux fonctions ainsi que l'addition précédemment définie. Ou alors on pourra faire mieux (ce qui n'est pas bien difficile, il suffit d'effectuer une multiplication à la main pour entrevoir différents algorithmes).
8. Calculer 50! et l'afficher.

B.5.3 Constructeurs

Examinez bien le contenu du programme suivant. La question est simple : quelles sont les différentes lignes affichées sur la *sortie standard* (c'est-à-dire en console, via les instructions "`cout << ... ;`") au cours de son exécution ? Justifier vos réponses.

```
1   #include <iostream>
2   using namespace std;
3
4   class foo {
5       int x;
6
7   public:
8       // Constructeurs
9       foo();
10      foo(const foo &f);
11      foo(int i);
12
13      // Destructeur
14      ~foo();
15
16      // Opérateurs
17      foo operator+(const foo &f) const;
18      void operator=(const foo &f);
19
20      // Accesseur
21      int getx() const;
22  };
23
24  foo::foo()
25  {
26      x=0;
27      cout << "constr vide " << endl;
28  }
29
30  foo::foo(int i)
31  {
32      x=i;
33      cout << "constr int " << x << endl;
34  }
35
36  foo::foo(const foo &f)
37  {
38      x=f.x;
39      cout << "constr copie " << x << endl;
40  }
41
42  foo::~foo()
43  {
44      cout << "dest " << x << endl;
45  }
46
47  foo foo::operator+(const foo &f) const
48  {
49      cout << "somme " << x << " " << f.x << endl;
```

```
50        foo somme(x+f.x);
51        return somme;
52    }
53
54    void foo::operator=(const foo &f)
55    {
56        x=f.x;
57        cout << "affecte " << x << endl;
58    }
59
60    int foo::getx() const
61    {
62        return x;
63    }
64
65    void affiche1(foo f)
66    {
67        cout << f.getx() << endl;
68    }
69
70    void affiche2(const foo &f)
71    {
72        cout << f.getx() << endl;
73    }
74
75    int main()
76    {
77        foo a;
78        foo b=1;
79        foo c(2);
80        affiche1(b);
81        affiche2(c);
82        a=b+c;
83        return 0;
84    }
```

B.6 Devoir écrit 2004 : énoncé

B.6.1 Tableau d'exécution

Pour ce premier exercice, il s'agit d'exécuter pas à pas le programme suivant :

```
1   #include <iostream>
2   using namespace std;
3
4   int hop(int x) {
5           x = x+2;
6           return x;
7   }
8
9   int hip(int& y) {
10          y = y*2;
11          return y+1;
12  }
13
14  int f(int& z) {
15          int t = z+3;
16          z=2*t;
17          if (z>20)
18                  return z;
19          return f(t);
20  }
21
22  int main() {
23          int a;
24          a = hop(1);
25          int b;
26          b = hip(a);
27          a = f(a);
28          return 0;
29  }
```

Pour cela, remplissez le tableau ci-dessous, en écrivant, si elles existent, les valeurs des variables du programme pour chaque exécution de ligne (selon l'exemple du cours).

Conventions :
- mettre le numéro de la ligne qu'on vient d'exécuter (éventuellement, on peut découper l'exécution d'une même ligne en plusieurs étapes)
- on peut utiliser le symbole " pour répéter la valeur d'une variable à la ligne suivante
- **une case vide signifie que la variable n'existe plus ou pas encore**
- pour une variable qui n'est autre qu'une référence sur une autre variable x, on indiquera $[x]$;
- s'il y a récursivité, indicer par un entier plusieurs appels imbriqués à une même fonction (f_1, f_2, ...)

Ligne	a_m	b_m	x_{hop}	ret_{hop}	y_{hip}	ret_{hip}	z_{f_1}	t_{f_1}	ret_{f_1}	z_{f_2}	t_{f_2}	ret_{f_2}

B.6.2 Constructeurs

Dans ce deuxième exercice, on vous demande tout simplement de donner et **justifier** l'affichage que produit l'execution du programme suivant sur la sortie standard (c'est-à-dire à l'écran, via la commande `cout<<...`) :

```
1   #include<iostream>
2   using namespace std;
3
4   class Vecteur
5   {
6           int tab[2];
7           void affiche() const;
8   public:
9           Vecteur();
10          Vecteur(int n);
11          Vecteur(const Vecteur &right);
12          ~Vecteur();
13          void operator=(const Vecteur& right);
14          Vecteur operator+(const Vecteur& right) const;
15          int operator*(const Vecteur right) const;
16          int& getX(int i);
17  };
18
19  Vecteur::Vecteur()
20  {
21          for(int i=0;i<2;i++) tab[i]=0;
22          cout<<"Contructeur par defaut: ";
23          affiche();
24  }
25  Vecteur::Vecteur(int x)
```

```
26  {
27          for(int i=0;i<2;i++) tab[i]=x;
28          cout<<"Constructeur de remplissage: ";
29          affiche();
30  }
31  Vecteur::Vecteur(const Vecteur &right)
32  {
33          for(int i=0;i<2;i++) tab[i]=right.tab[i];
34          cout<<"Constructeur par copie: ";
35          affiche();
36  }
37  Vecteur::~Vecteur()
38  {
39          cout<<"Destructeur: ";
40          affiche();
41  }
42  void Vecteur::operator=(const Vecteur& right) {
43          for(int i=0;i<2;i++) tab[i]=right.tab[i];
44          cout<<"Operateur de copie: ";
45          affiche();
46  }
47  Vecteur Vecteur::operator+(const Vecteur& right) const {
48          Vecteur Res;
49          for(int i=0;i<2;i++) Res.tab[i]=tab[i]+right.tab[i];
50          return Res;
51  }
52  int Vecteur::operator*(const Vecteur right) const  {
53          int Res=0;
54          for(int i=0;i<2;i++) Res+=tab[i]*right.tab[i];
55          return Res;
56  }
57  int& Vecteur::getX(int i){
58          return tab[i];
59  }
60  void Vecteur::affiche() const{
61          for (int i=0;i<2;i++)
62                  cout<<tab[i]<<" ";
63          cout<<endl;
64  }
65  void affiche(Vecteur vec){
66          cout<<"Affiche: ";
67          for (int i=0;i<2;i++)
68                  cout<<vec.getX(i)<<" ";
69          cout<<endl;
70  }
71  int main()
72  {
73          Vecteur A;
74          cout<<endl;
```

```
75          Vecteur B(-2);
76          cout<<endl;
77          Vecteur C=B;
78          cout<<endl;
79          B.getX(1)=3;
80          affiche(C);
81          cout<<endl;
82          cout<<"Produit scalaire: "<<B*C<<endl;
83          cout<<endl;
84          A=B+C;
85          cout<<endl;
86          return 0;
87  }
```

B.6.3 Le compte est bon

Le but de cet exercice va être d'écrire un programme capable de résoudre le problème du **Compte est bon** dont voici une description :

- 6 plaques sont tirées au hasard parmi 14 plaques différentes dont chacune peut être choisi de 0 à 2 fois. Les 14 plaques comportent les nombres : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100.
- Un nombre entre 100 et 999 (inclus) est tiré au hasard, on l'appelera par la suite le *total*.
- Une des 4 opérations arithmétiques (addition, soustraction, division entière, multiplication) est utilisée sur deux des 6 plaques pour obtenir un nouveau nombre (attention : la division entière ne peut être utilisée que si le dividande est divisible par le diviseur ; la soustraction ne peut être utilisée que si le premier terme est plus grand que le second). Il nous reste alors 5 nombres (les 4 plaques non choisies et le nouveau nombre) à partir desquels recommencer.
- Le but est d'obtenir finalement le total, en utilisant tout ou partie des 6 plaques ainsi que décrit précédemment.

Par exemple, on tire le total 987 et les plaques suivantes :

1 50 9 1 50 7

Une solution sera par exemple :

$$1 + 1 = 2$$

$$2 * 50 = 100$$

$$100 - 9 = 91$$

$$91 + 50 = 141$$

$$141 * 7 = 987$$

Il n'existe pas toujours de solution. Dans ce cas, le candidat doit trouver une solution approchant au plus près le total. Pour simplifier, le programme que vous écrirez cherchera uniquement des solutions exactes.

La manière dont trouver une solution est très simple : le programme va simplement émunérer toutes les possibilités de combiner les nombres écrits sur les plaques et les

opérateurs arithmétiques, jusqu'à tomber sur une bonne solution ou les avoir explorées toutes.

On utilisera dans le programme partout où cela est possible les constantes globales suivantes :

```
const int total_min=100;
const int total_max=999;

const int nb_plaques=6;
const int nb_plaques_possibles=14;
const int plaques_possibles[nb_plaques_possibles]=
  { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100};

const int nb_operateurs=4;
const char operateurs[nb_operateurs]={'+','-','/','*'};
```

1. Définir une fonction de prototype `int random(int a,int b)` renvoyant un nombre aléatoire entre `a` et `b` (inclus). Préciser les `#include` et `using namespace` nécessaires.
2. Définir une fonction `int genere_total()` renvoyant un nombre aléatoire entre `total_min` et `total_max`
3. Définir une fonction `void affiche_plaques(int tab[nb_plaques])` que l'on utilisera ultérieurement pour afficher à l'écran les plaques tirées au hasard. Que faut-il rajouter comme `#include` ?
4. Définir une fonction `void genere_plaques(int tab[nb_plaques])` effectuant le tirage au hasard des plaques. Afin de se souvenir du nombre de fois où on a tiré chaque plaque et d'imposer d'en tirer au plus 2, on pourra utiliser une variable `int plaque_deja_tiree[nb_plaques_possibles]`.
5. Définir la fonction `main` de votre programme qui initialisera le gestionnaire de nombre aléatoires, effectuera le tirage du total et des plaques, et affichera le tout à l'écran. Que faut-il rajouter comme `#include` ?
6. Nous allons maintenant nous attaquer au cœur du problème, la fonction `bool trouve_combinaison (int plaques[], int n, int total)` qui, étant donnée un tableau de plaques `plaques` de taille `n` et un total `total` :
   – Renvoie `true` et affiche à l'écran une solution si elle existe.
   – Renvoie `false` sinon.
   `trouve_combinaison` sera programmée récursivement suivant le nombre de plaques `n` :
   – Si `n==1`, le problème est trivial
   – Sinon, on effectue l'opération `plaques[i] op plaques[j]` pour tous les `i` et `j` distincts entre `0` et `n` et tous les opérateurs arithmétiques `op`, *si elle est possible*. Ensuite :
     – Si on tombe sur le nombre `total`, on affiche l'opération qui nous a permis d'y accéder et on renvoie `true`.
     – Sinon, on appelle récursivement la fonction `bool trouve_combinaison` sur un nouveau tableau de plaques. Si cet appel récursif renvoie `true`, c'est qu'il est possible d'obtenir une solution à partir de cette étape intermédiaire. On renvoie alors également `true` et on affiche l'opération qui nous a permir d'accéder à cette étape intermédiaire. Sinon, on continue à itérer.

7. Ajouter à votre fonction main l'utilisation de la fonction `trouve_combinaison`. Quand aucune combinaison n'est trouvée, le signaler en affichant un message à l'écran.
8. Le programme que vous avez écrit ne sait pas trouver de solution approchée, ne retient qu'une seule solution (pas forcément la plus courte) et l'affiche en ordre inverse. Que faudrait-il faire (en français, sans essayer de le programmer) pour résoudre ces limitations ?

B.7 Devoir écrit 2006 : énoncé

B.7.1 Enoncé – Tours de Hanoi

Principe et règles

Le jeu est constitué de 3 tiges, A, B et C, sur lesquelles sont empilés des anneaux de taille décroissante, que l'on numérotera de N à 1.

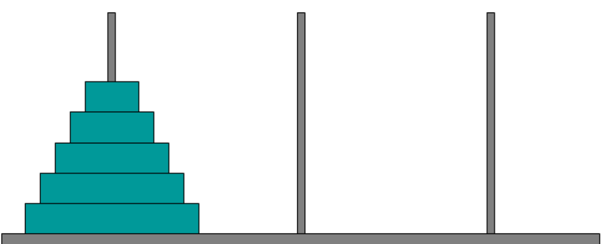


FIG. B.2 – Configuration initiale

Au début du jeu, tous les anneaux sont empilés sur la tige A (voir figure B.2). L'objectif est de transférer tous les anneaux sur la tige C. Il n'est possible de déplacer les anneaux que un par un, et il n'est pas possible de poser un anneau sur un autre de taille inférieure.

On désignera par le terme `configuration` une disposition des anneaux sur les tiges (voir figure B.3).

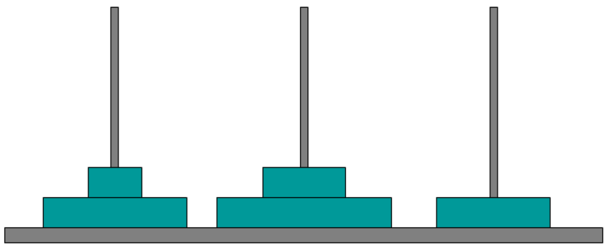


FIG. B.3 – Exemple de configuration

On désignera par le terme `déplacement` de la tige X à la tige Y le fait d'enlever l'anneau situé au sommet de la pile de la tige X pour le positionner au sommet de la pile de la tige Y (voir figure B.4).

Objectif

L'objectif ici est d'écrire un programme qui affiche la suite des déplacements à réaliser pour résoudre le problème avec n anneaux, sous la forme suivante :
$numero_anneau : tige_origine -> tige_destination$
On souhaite de plus afficher l'état du jeu (les anneaux présents sur chaque tige) après chaque déplacement.

Exemple pour 2 anneaux (la figure B.5 en donne une représentation graphique) :

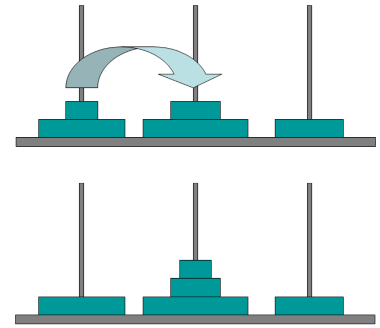


FIG. B.4 – Exemple de déplacement

```
  A> 2 1
  B> . .
  C> . .
deplacement: 1 : A->B
  A> 2 .
  B> 1 .
  C> . .
deplacement: 2 : A->C
  A> . .
  B> 1 .
  C> 2 .
deplacement: 1 : B->C
  A> . .
  B> . .
  C> 2 1
```

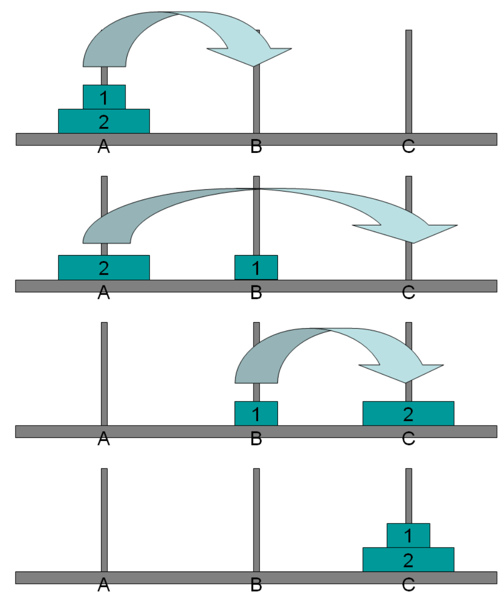


FIG. B.5 – Solution du problème pour N=2

Ce problème apparemment complexe se traite en fait simplement : pour transférer n

anneaux d'une tige X à une tige Y, il "suffit" d'effectuer :
- le transfert des n-1 anneaux supérieurs de la tige X sur la troisième tige Z
- le déplacement de l'anneau n de la tige X à la tige destination Y
- le transfert des n-1 anneaux de la tige Z à la tige Y

Pour transférer n-1 anneaux, on applique la même méthode : transfert de n-2 anneaux de la tige origine à une tige temporaire, déplacement d'un anneau de la tige origine à la tige destination, transfert des n-2 anneaux de la tige temporaire à la tige destination ...

Questions

1. Déclarer une constante N, que l'on fixera à 10, et qui représentera le nombre d'anneaux du problème à résoudre.
2. Créer un objet capable de stocker une configuration. On pourra par exemple représenter chaque tige par un tableau de taille N, et chaque anneau par un entier entre 1 et N. Vous utiliserez au choix une structure ou une classe, selon ce qui vous semble le plus pertinent, ou à défaut ce avec quoi vous êtes le plus à l'aise.
3. Programmer de quoi initialiser une configuration, c'est à dire lui donner sa valeur de départ (tous les anneaux sur la première tige).
4. Programmer une fonction permettant d'afficher une configuration à l'écran, par exemple sous la forme utilisée ci-dessus pour l'exemple pour N=2.
5. Programmer une fonction réalisant un déplacement : pour une tige origine et une tige destination spécifiées, elle prélève le pion supérieur sur la tige origine et le place sur la tige destination.
6. En vous basant sur la méthode proposée, programmer une fonction réalisant un transfert de k anneaux d'une tige à une autre, et affichant au fur et à mesure les déplacements effectués, et l'état du jeu après chaque déplacement.
7. En utilisant les éléments précédents, écrire une fonction main() qui résout le problème des tours de Hanoi de taille N et affiche la solution complète.

Questions complémentaires

1. Si l'on avait souhaité n'afficher que la liste des déplacements à effectuer sans afficher à chaque fois les configurations intermédaires, le programme aurait été beaucoup plus simple. Pourquoi ? Ecrire la fonction correspondante (5 lignes de code environ).
2. Pour résoudre le problème avec N anneaux, combien de déplacements sont nécessaires ?

B.8 Devoir final 2003 : énoncé

B.8.1 Tableau d'exécution

Remplir le tableau d'exécution du programme suivant :

```
1  int suite(int x){
2    if (x%2 == 0)
3      return x/2;
4    else
5      return 3*x + 1;
6  }
7
8  bool test(int y){
9    bool h;
10
11   if (y == 1)
12     h = true;
13   else
14     h = test(suite(y));
15
16   return h;
17 }
18
19 void initialise(int& z){
20   z = 4;
21 }
22
23 int fin(int& w){
24   w += 36;
25   return w+2;
26   w += 1;
27 }
28
29 int main(){
30   int a;
31   initialise(a);
32
33   if (test(a))
34     a = fin(a);
35
36   return 0;
37 }
```

Il s'agit d'écrire, si elles existent, les valeurs des variables du programme pour chaque exécution de ligne. Question bonus : ce code génère un warning à la compilation ; lequel, à quelle ligne ?

Conventions :

- mettre le numéro de la ligne qu'on vient d'exécuter (éventuellement, on peut découper l'exécution d'une même ligne en plusieurs étapes) ;
- laisser en blanc les variables non existantes ;
- mettre un point d'interrogation pour les variables déclarées non initialisées ;
- pour une variable qui n'est autre qu'une référence sur une autre variable x, on indiquera $[x]$;
- on peut utiliser le symbole " pour répéter la valeur d'une variable à la ligne suivante ;
- pour gagner de la place dans le tableau, les noms des variables n'ont été indexés que par la première lettre du nom de la fonction à laquelle elles sont rattachées, suivie le cas échéant du numéro d'appel à la fonction. De même, le retour de la fonction f a pour nom ret_f.

B.8.2 Erreurs

Corriger le code suivant :

```
1   // fichier 'main.cpp'
2
3   class paire {
4           string clef;
5           int valeur;
6   public:
7           paire(string key, int value);
8           void affecte_clef(string key) const;
9           void affecte_valeur(int value) const;
10          string donne_clef() const;
11          int donne_valeur() const;
12  }
13
14  paire::paire(string key, int value) {
15          clef=key;
16          valeur=value;
17  }
18
19  void paire::affecte_clef(string key) {
20          clef=key;
21  }
22
23  void paire::affecte_valeur(int value) {
24          valeur=value;
25  }
26
27  string paire::donne_clef() {
28          return clef;
29  }
30
31  int paire::donne_valeur() {
32          return valeur;
33  }
```

```
34
35  int cherche_valeur(paire repertoire[], int n, string key) {
36          for (i=1, i<=n, i++) {
37                  if repertoire[i].donne_clef = key
38                          return (repertoire[i].donne_valeur);
39          cout << "clef non trouvee" << endl;
40  }
41
42  int main()
43  {
44          paire a={"cheval",4};
45
46          paire b;
47          b.clef="poule";
48          b.valeur=2;
49
50          paire* pattes = new paire[5];
51          pattes = {a, b, paire("araignee",8), paire("baleine",0),
52                    paire("martien",3)};
53
54          cout << "Un martien a "
55                  << cherche_valeur(pattes[5],5,"martien")
56                  << " pattes." << endl;
57
58          return 0;
59  }
```

B.8.3 Qu'affiche ce programme ?

Examinez bien le contenu du programme suivant. La question est simple : quelles sont les différentes lignes affichées sur la *sortie standard* (c'est-à-dire en console, via les instructions "`cout << ... ;`") au cours de son exécution ? **Justifiez vos réponses**.

```
1   #include <iostream>
2   using namespace std;
3
4   class Paire {
5       double x,y;
6   public:
7       Paire();
8       Paire(double a,double b);
9       Paire operator+(const Paire &v) const;
10      Paire operator-(const Paire &v) const;
11      double getx() const;
12      double gety() const;
13      void setxy(double xx,double yy);
14  };
15
16  Paire::Paire() {
```

```
17        cout << "Constructeur sans argument." << endl;
18    }
19
20    Paire::Paire(double a,double b) {
21        x=a;
22        y=b;
23        cout << "Constructeur avec arguments [" << x << "," << y << "]" << endl;
24    }
25
26    Paire Paire::operator+(const Paire& v) const {
27        cout << "Addition [" << x << "," << y << "] [" << v.x << ","
28             << v.y << "]" << endl;
29        Paire w;
30        w.x=x+v.x;
31        w.y=y+v.y;
32        return w;
33    }
34
35    Paire Paire::operator-(const Paire &v) const {
36        cout << "Soustraction [" << x << "," << y << "] [" << v.x << ","
37             << v.y << "]" << endl;
38        return Paire(x-v.x,y-v.y);
39    }
40
41    double Paire::getx() const {
42        return x;
43    }
44
45    double Paire::gety() const {
46        return y;
47    }
48
49    void Paire::setxy(double xx,double yy) {
50        x=xx;
51        y=yy;
52    }
53
54    int main(void) {
55        Paire tab[2];
56        tab[0].setxy(3,4);
57        tab[1]=Paire(4,6);
58
59        Paire a(1,1);
60        Paire b= ( ( a+Paire(1,2) ) + tab[0] ) - tab[1];
61        cout << "Resultat [" << b.getx() << "," << b.gety() << "]" << endl;
62    }
```

B.8.4 Le jeu du Pendu

Tout le monde connaît le jeu de pendu. Rappelons tout de même les règles pour plus de sûreté ... Il s'agit de trouver un mot caché en proposant des lettres une par une. Si la lettre proposée n'est pas présente dans le mot, une erreur est comptabilisée. Si le joueur dépasse 11 erreurs avant d'avoir trouvé le mot, il a perdu. Inversement, si le joueur parvient à découvrir l'intégralité du mot avant de totaliser 11 erreurs, il a gagné.

Commencez par programmer au brouillon !!!

On dispose d'une fonction `string dictionnaire(int n)` qui renvoie un mot de 10 lettres différent pour chaque entier compris entre 0 et 999 passé en argument.

1. Utiliser cette fonction pour programmer une fonction `mot_mystere` qui renvoie un mot mystère à faire découvrir, en faisant en sorte que ce mot change aléatoirement à chaque exécution.

On utilise un tableau de booléens pour stocker le fait que le i^{eme} caractère du mot mystère ait été découvert ou non.
Exemple : pour "maquillage", si le joueur a déjà entré 'i', 'a' et 's', le tableau de booléens est {false, true, false, false, true, false, false, true, false, false}.

2. Programmer une fonction `affiche` qui à partir du mot mystère et du tableau de booléens associé affiche à l'écran le résultat courant (mot mystère dans lequel les caractères non découverts sont remplacés par une étoile '*').
   *Exemple : pour "maquillage", si le joueur a déjà entré 'i', 'a' et 's', le résultat courant est "*a**i**a**". La fonction affiche donc "*a**i**a**".*
3. Programmer une fonction `essai` qui demande à l'utilisateur de rentrer un caractère au clavier et qui met à jour le tableau de booléens en conséquence.
4. Programmer une fonction `reste` qui renvoie le nombre de caractères restant à trouver dans le mot mystère (pour 2 caractères identiques, on comptera 2 pour simplifier).
   *Exemple : pour "maquillage", si le joueur a déjà entré 'i', 'a' et 's', le résultat courant est "*a**i**a**". Il reste 7 caractères à trouver ('m','q','u','l','l','g','e').*

On dispose maintenant de suffisants d'éléments pour construire une version simple du jeu, sans notion de nombre limité d'essais.

5. A partir des fonctions précédentes, programmer une fonction `jeu_mystere` qui demande à l'utilisateur d'entrer un caractère puis affiche le résultat courant jusqu'à ce que le mot mystère soit entièrement découvert, et signale ensuite à l'utilisateur qu'il a gagné.

Il s'agit maintenant de rajouter à cette première version du jeu la notion de nombre limité d'essais.

6. Programmer une nouvelle version de la fonction `essai` pour qu'elle renvoie en plus une variable signalant si le caractère entré par l'utilisateur restait à découvrir (succès) ou non (échec) dans le mot mystère.
   *Exemple : pour "maquillage", si le joueur a déjà entré 'i', 'a' et 's', le résultat courant est "*a**i**a**". Si l'utilisateur rentre 'm', la fonction renvoie 'succès'; si l'utilisateur rentre 'z', la fonction renvoie 'échec'; si l'utilisateur rentre 'a' (déjà rentré), la fonction renvoie 'échec'.*

7. Assembler les fonction précédentes à l'intérieur d'une fonction `jeu_pendu` pour créer un jeu de pendu. Gérer la défaite du joueur si il arrive à 11 échecs. Gérer la victoire du joueur si il découvre l'intégralité du mot mystère avant de totaliser 11 échecs.

B.8.5 Programme mystère

Le programme suivant est inspiré d'un célèbre puzzle. Il s'agit de deviner la logique d'une suite de chaînes de caractères. On donne les quatre premiers éléments de la suite et on demande de trouver les deux suivants. Il y a un piège et certaines personnes mettent plusieurs jours à trouver...

Ici, nous ne vous donnons pas les premiers éléments, mais un programme qui génère la suite complète. Pour quelqu'un qui parle C++, ce programme est donc directement la solution du puzzle. A vous de jouer :

- Que fait exactement ce programme ? Donnez le principe de la suite et détaillez un peu le fonctionnement de la fonction `next()`.
- Qu'affiche t-il ?
- Le programmeur a construit ligne 15 le caractère représentant le chiffre `count` en utilisant `char('0'+count)`. Il a donc supposé que `count` restait plus petit que 10. Pour se rassurer, il a ajouté un `assert()` ligne 14. Etait-ce la peine ? Le `assert()` reste-t-il vrai si on affiche le reste de la suite ? La condition aurait-elle pu être encore plus restrictive ?
- Question subsidiaire (hors examen !) : faites le test autour de vous et voyez combien de personnes trouvent le 5eme élément à partir des 4 premiers...

```
1   // Programme mystère...
2
3   #include <iostream>
4   #include <string>
5   #include <cassert>
6   using namespace std;
7
8   string next(string s) {
9       string t;
10      int count=1;
11      for (unsigned int i=0;i<s.size();i++) {
12          if (i==s.size()-1 || s[i]!=s[i+1]) {
13              string u="xx";
14              assert(count<10);
15              u[0]=char('0'+count); // caractère représentant le chiffre count
16              u[1]=s[i];
17              t+=u;
18              count=1;
19          } else
20              count++;
21      }
22      return t;
23  }
24
25  int main()
```

```
26  {
27      string s="1";
28      cout << s << endl;
29      for (int i=0;i<8;i++) {
30          s=next(s);
31          cout << s << endl;
32      }
33      return 0;
34  }
```

...

B.9 Devoir final 2004 : énoncé

B.9.1 Erreurs

Corriger le code suivant : (**effectuer les corrections directement sur le script en annexe**)

```
1   // fichier 'main.cpp'
2   #include <win>
3
4   const int w;
5   const int h;
6
7   struct point
8   {
9           int x,y;
10  }
11
12  point operator+(point p1, point p2)
13  {
14          point p
15          p.x = p1.x+p2.x
16          p.y = p1.y+p2.y
17  }
18
19  class quadrilatere
20  {
21          point t[4];
22          quadrilatere(point pts[4]);
23          affiche(Color col=Black) const;
24          void translate(point v) const;
25          bool dans_la_fenetre();
26  }
27
28  quadrilatere::quadrilatere(point pts[4])
29  {
30          t=pts;
31  }
32
33  quadrilatere::affiche(Color col) const
34  {
35          for(int i=0;i<=4;i++)
36          {
37                  DrawLine(t[i].x,t[i].y,t[(i+1)%4].x,t[(i+1)%4].y,col);
38          }
39  }
40
41  void quadrilatere::translate(point v) const
42  {
```

```
43          for(int i=0;i<4;i=i+1)
44                  t[i]=t[i]+v;
45  }
46
47  bool quadrilatere::dans_la_fenetre()
48  {
49          bool in=false;
50          for(int i=0;(i<4) && in;i++)
51          {
52                  if ((t[i].x<0) or (t[i].x>=w) or (t[i].y<0) or (t[i].y>=h))
53                  then in=true;
54          }
55          return in;
56  }
57
58  int main()
59  {
60          OpenWindow(w,h);
61
62          quadrilatere Q;
63          Q(pts);
64
65          point pts[4];
66          pts={{10,10},{10,100},{100,100},{100,10}};
67
68          point v={1,2};
69
70          while(Q.dans_la_fenetre())
71          {
72                  Q.affiche();
73                  MilliSleep(10);
74                  Q.affiche(White);
75                  Q.translate(v);
76          }
77
78          delete [] Q.t;
79          Terminate();
80          return 0;
81  }
```

B.9.2 Qu'affiche ce programme ?

Examinez bien le contenu du programme suivant. La question est simple : quelles sont les différentes lignes affichées sur la *sortie standard* (c'est-à-dire en console, via les instructions "`cout << ... ;`") au cours de son exécution ? **Justifiez vos réponses**.

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

```
4   int NbPt;
5
6   void check() {
7           cout<<endl;
8           cout<<"n_points= "<<NbPt<<endl;
9           cout<<endl;
10  }
11
12
13  struct Point {
14          double x,y;
15
16          Point()
17          {
18                  NbPt++;
19                  x=y=0;
20                  cout<<"Point: Void Cons"<<endl;
21          };
22
23          Point( const Point &model)
24          {
25                  NbPt++;
26                  x=model.x;
27                  y=model.y;
28                  cout<<"Point: Copy Cons"<<endl;
29          }
30
31          Point( const double &_x, const double &_y)
32          {
33                  NbPt++;
34                  x=_x; y=_y;
35                  cout<<"Point: (x,y) Cons"<<endl;
36          }
37
38          ~Point()
39          {
40                  NbPt--;
41                  cout<<"Point: Dest"<<endl;
42          }
43
44          void display() const
45          {
46                  cout<<"x= "<<x<<" y= "<<y<<endl;
47          }
48  };
49
50  void displayPoint ( Point T)
51  {
52          T.display();
```

```
53  }
54
55  void displayTriangle2 ( Point *T) {
56          cout<<"A"<<endl; T[0].display();
57          cout<<"B"<<endl; T[1].display();
58          cout<<"C"<<endl; T[2].display();
59  }
60
61  void displayTriangle1 ( Point *&T) {
62          cout<<"A"<<endl; displayPoint(T[0]);
63          cout<<"B"<<endl; displayPoint(T[1]);
64          cout<<"C"<<endl; displayPoint(T[2]);
65  }
66
67  void buildTriangleStatic ( Point *T, const double *array) {
68          for(int i=0;i<3;i++) {
69                  T[i]=Point(array[2*i],array[2*i+1]);
70          }
71  }
72
73  Point* buildTriangleDynamic( const double *array) {
74          Point *T=new Point[3];
75          for(int i=0;i<3;i++) {
76                  T[i]=Point(array[2*i],array[2*i+1]);
77          }
78          return T;
79  }
80
81  void destroyTriangleDynamic (Point *T) {
82          delete[] T;
83  }
84
85  int main()
86  {
87          NbPt=0;
88          const double d6_1[6]={1.1,2.2,3.3,-0.1,-0.2,-0.3};
89          const double d6_2[6]={1,1,2,2,3,3};
90
91          check();
92          Point* t1=buildTriangleDynamic(d6_1);
93          check();
94          Point t2[3];
95          check();
96          buildTriangleStatic(t2,d6_2);
97          check();
98
99          displayTriangle1(t1);
100         check();
101         displayTriangle2(t2);
```

```
102           check();
103
104           destroyTriangleDynamic(t1);
105           check();
106           t1=t2;
107           check();
108           displayTriangle1(t1);
109           check();
110
111           return 0;
112   }
```

B.9.3 Chemins dans un graphe

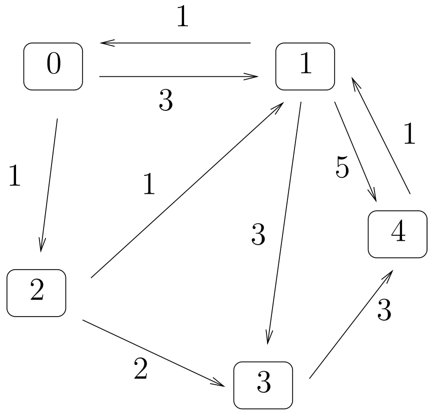


FIG. B.6 – Chemins minimaux

On se propose de calculer les plus courts chemins dans un graphe par l'algorithme de Floyd. Soient n villes et, pour $0 \leqslant i < n$ et $0 \leqslant j < n$, C_{ij} le coût du trajet de la ville i vers la ville j (avec $C_{ii} = 0$, C_{ij} non nécessairement égal à C_{ji}, et éventuellement $C_{ij} = \infty$ si ce trajet n'existe pas). Le coût d'un chemin $i_1, i_2, ..., i_p$ est la somme $\Sigma_{k=1}^{p-1} C_{i_k i_{k+1}}$. Pour trouver les coûts de tous les chemins minimaux entre toutes les villes, il suffit de construire, pour $0 \leqslant k < n$, les matrices $D^{(k)}$ définies par :

$$\begin{cases} D^{(0)} = C \\ D_{ij}^{(k)} = \min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}) \end{cases}$$

Le coût du chemin minimum entre i et j est alors $D_{ij}^{(n-1)}$ (éventuellement ∞ si on ne peut relier i à j).

Questions

1. Définir une matrice `int C[n][n]` représentant le graphe figure B.6 dans laquelle chaque flèche de i vers j représente un C_{ij}
2. Ecrire une fonction `void floyd(const int C[n][n],int D[n][n])` calculant $D^{(n-1)}$ à partir de C
3. Appeler cette fonction et afficher les coûts des chemins minimaux pour tous les (i, j).

Chemins

Pour mémoriser les chemins et non plus seulement leur coût, il suffit de rajouter une matrice P correspondant aux prédécesseurs dans les chemins minimaux. Cette matrice est définie comme suit :

- Initialement, $P_{ij} = i$ si $i \neq j$ et $C_{ij} < \infty$, $P_{ij} = \infty$ sinon.
- Lorsque $D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$ est retenu comme valeur de $D_{ij}^{(k)}$, alors faire $P_{ij} = P_{kj}$

Après quoi, le chemin minimal de i vers j est celui de i vers P_{ij}, suivi du trajet $P_{ij} \to j$.

1. Modifier la fonction `floyd` pour qu'elle remplisse cette matrice P.
2. Ecrire une fonction récursive `void chemin(const int P[n][n],int i, int j)` affichant le chemin minimal de i vers j pour P donnée.
3. Utiliser les deux fonctions précédentes pour calculer et afficher tous les chemins minimaux dans le graphe précédent.

B.9.4 Tours de Hanoï

Principe et règles

Le jeu est constitué de 3 tiges, A, B et C, sur lesquelles sont empilés des anneaux de taille décroissante, que l'on numérotera de N à 1.

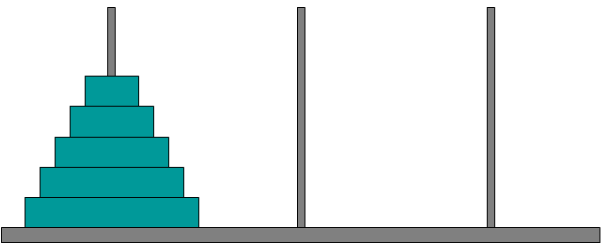


Fig. B.7 – Configuration initiale

Au début du jeu, tous les anneaux sont empilés sur la tige A (voir figure B.7). L'objectif est de transférer tous les anneaux sur la tige C. Il n'est possible de déplacer les anneaux que un par un, et il n'est pas possible de poser un anneau sur un autre de taille inférieure.

On désignera par le terme `configuration` une disposition des anneaux sur les tiges (voir figure B.8).

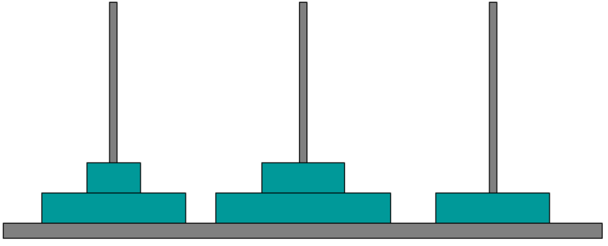


Fig. B.8 – Exemple de configuration

On désignera par le terme `déplacement` de la tige X à la tige Y le fait d'enlever l'anneau situé au sommet de la pile de la tige X pour le positionner au sommet de la pile de la tige Y (voir figure B.9).

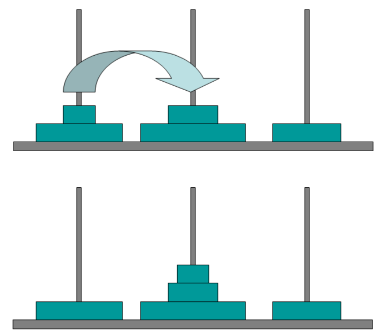


FIG. B.9 – Exemple de déplacement

Objectif

L'objectif ici est d'écrire un programme qui affiche la suite des déplacements à réaliser pour résoudre le problème avec n anneaux, sous la forme suivante :
$numero_anneau : tige_origine- > tige_destination$
On souhaite de plus afficher l'état du jeu (les anneaux présents sur chaque tige) après chaque déplacement.

Exemple pour 2 anneaux (la figure B.10 en donne une représentation graphique) :

```
  A> 2 1
  B> . .
  C> . .
deplacement: 1 : A->B
  A> 2 .
  B> 1 .
  C> . .
deplacement: 2 : A->C
  A> . .
  B> 1 .
  C> 2 .
deplacement: 1 : B->C
  A> . .
  B> . .
  C> 2 1
```

Ce problème apparemment complexe se traite en fait simplement : pour transférer n anneaux d'une tige X à une tige Y, il "suffit" d'effectuer :
- le transfert des n-1 anneaux supérieurs de la tige X sur la troisième tige Z
- le déplacement de l'anneau n de la tige X à la tige destination Y
- le transfert des n-1 anneaux de la tige Z à la tige Y

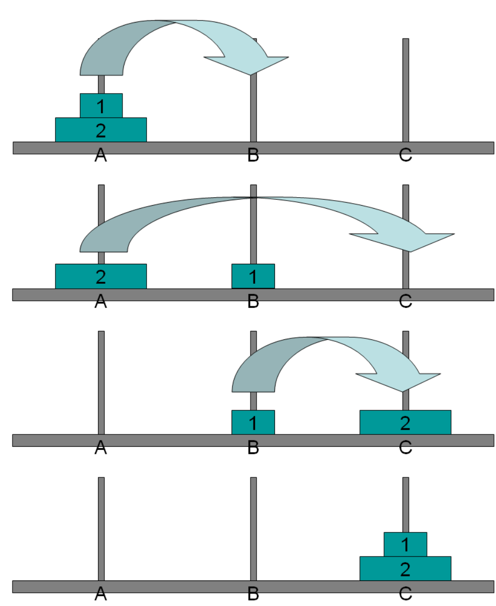


Fig. B.10 – Solution du problème pour N=2

Pour transférer n-1 anneaux, on applique la même méthode : transfert de n-2 anneaux de la tige origine à une tige temporaire, déplacement d'un anneau de la tige origine à la tige destination, transfert des n-2 anneaux de la tige temporaire à la tige destination ...

Questions

1. Déclarer une constante N, que l'on fixera à 10, et qui représentera le nombre d'anneaux du problème à résoudre.
2. Créer un objet capable de stocker une configuration. On pourra par exemple représenter chaque tige par un tableau de taille N, et chaque anneau par un entier entre 1 et N. Vous utiliserez au choix une structure ou une classe, selon ce qui vous semble le plus pertinent, ou à défaut ce avec quoi vous êtes le plus à l'aise.
3. Programmer de quoi initialiser une configuration, c'est à dire lui donner sa valeur de départ (tous les anneaux sur la première tige).
4. Programmer une fonction permettant d'afficher une configuration à l'écran, par exemple sous la forme utilisée ci-dessus pour l'exemple pour N=2.
5. Programmer une fonction réalisant un déplacement : pour une tige origine et une tige destination spécifiées, elle prélève le pion supérieur sur la tige origine et le place sur la tige destination.
6. En vous basant sur la méthode proposée, programmer une fonction réalisant un transfert de k anneaux d'une tige à une autre, et affichant au fur et à mesure les déplacements effectués, et l'état du jeu après chaque déplacement.
7. En utilisant les éléments précédents, écrire une fonction main() qui résoud le problème des tours de Hanoi de taille N et affiche la solution complète.

Questions complémentaires

1. Si l'on avait souhaité n'afficher que la liste des déplacements à effectuer sans afficher à chaque fois les configurations intermédaires, le programme aurait été beaucoup

plus simple. Pourquoi ? Ecrire la fonction correspondante (5 lignes de code environ).

2. Pour résoudre le problème avec N anneaux, combien de déplacements sont nécessaires ?

B.9.5 Table de hachage

On a souvent besoin, en informatique, de manipuler des données sous forme de *tableaux associatifs*, dans lesquels on associe une *valeur* à chaque *clef* d'un ensemble. C'est le cas quand on veut représenter un annuaire (à un nom, on associe un numéro de téléphone) ou un dictionnaire monolingue (à un mot, on associe une définition). On se place dans le cas où clefs et valeurs sont représentées par des chaînes de caractères et où une unique valeur est associée à chaque clef.

Une manière efficace de représenter cette notion abstraite de tableau associatif est d'utiliser une *table de hachage*. On se fixe un entier naturel N et on alloue en mémoire un tableau de N cases. On suppose que l'on dispose d'une *fonction de hachage* h qui à une clef (une chaîne de caractères, donc), associe un entier entre 0 et $N-1$. On stockera le couple clef/valeur (c, v) dans la case $h(c)$ du tableau. Ainsi, pour rechercher par la suite quelle valeur est associée à une clef c, il suffira de regarder dans la case $h(c)$, sans avoir besoin de parcourir l'intégralité du tableau.

Un problème qui se pose est que h n'est pas une fonction injective. Non seulement il y a potentiellement un nombre infini de clefs, mais on souhaite également garder N relativement petit pour ne pas avoir de trop gros besoins en mémoire. Il faut donc prévoir les *collisions*, c'est-à-dire les cas de deux clefs distinctes c et c' telles que $h(c) = h(c')$. Plusieurs stratégies de gestion des collisions existent, on en choisira une très simple : si, au moment d'insérer le couple (c', v'), la case $h(c')$ est déjà occupée par le couple (c, v), on retente l'insertion dans la case $h(c')+1$. Si celle-ci est à nouveau occupée, on essaye la case suivante, etc., jusqu'à trouver une case vide ou avoir parcouru l'intégralité du tableau (on considère que la case 0 succède à la case $N-1$), auquel cas l'insertion échoue (le tableau associatif comportera donc au plus N éléments). La recherche d'une clef dans la table de hachage se passe de façon similaire.

Une table de hachage est donc caractérisée par :
- sa taille N (on prendra N=1021)
- la fonction de hachage h. Pour une clef $c = (x_0 \dots x_{l-1})$ de longueur l, on prendra :

$$h(c) = \left(\sum_{i=0}^{l-1} B^{l-1-i} x_i \right) mod\, N$$

  où B est une constante, que l'on fixera à 256 et $x\, mod\, y$ désigne le reste de la division euclidienne de x par y
- la manière de gérer les collisions, décrite plus haut

1. Définir les constantes globales `N` et `B`.
2. Définir la fonction de hachage `int hachage(const string &clef)`.
3. Définir une structure `Entree`, qui correspondra à une case de la table de hachage. Cette structure contient trois champs : la clef, la valeur et un booléen indiquant si la case est occupée.
4. Définir une classe `TableHachage` comprenant :
   - un tableau de `N` `Entree`

– une fonction membre `void inserer(const string &clef, const string &valeur);`
– une fonction membre `string rechercher(const string &clef) const;`

Un constructeur est-il nécessaire ? Pourquoi ? Le définir le cas échéant.

5. Définir la fonction `TableHachage::inserer`. Dans le cas où la table est pleine, on pourra simplement afficher un message à l'écran et ne rien faire d'autre.
6. Définir la fonction `TableHachage::rechercher`. Dans le cas où la clef n'est pas présente dans la table, on pourra renvoyer la chaîne vide.
7. Définir une fonction `main` utilisant un objet de type `TableHachage`. On pourra stocker le mini-annuaire suivant, puis rechercher et afficher le numéro de téléphone du SAMU.

SAMU	15
Police Secours	17
Pompiers	18

8. Sans rien programmer, expliquer ce qu'il faudrait faire pour rajouter la possibilité d'enlever un élément à la table de hachage.

B.10 Devoir final 2005 : énoncé

B.10.1 Erreurs à corriger

Corriger le code fourni en annexe : **effectuer les corrections directement sur le script**. *Conseil : il y a une vingtaine d'erreurs.*

B.10.2 Qu'affiche ce programme ?

Examinez bien le contenu du programme suivant. La question est simple : quelles sont les différentes lignes affichées sur la *sortie standard* (c'est-à-dire en console, via les instructions "`cout << ... ;`") au cours de son exécution ? Procédez avec méthode en faisant attention aux retours de fonction *par valeurs*, et à l'équilibre nécessaire entre appels aux constructeurs/destructeurs. **Justifiez vos réponses**.

```
1   #include <iostream>
2   #include <cmath>
3   using namespace std;
4
5   class Complexe{
6           double m_Re, m_Im;
7   public:
8           Complexe();
9           Complexe(double re, double im);
10          Complexe(const Complexe & c);
11          ~Complexe();
12          void operator= (const Complexe & c);
13          const Complexe operator* (const Complexe & c) const;
14          void conjugue();
15          double norme() const;
16          void print() const;
17  };
18
19  Complexe::Complexe(){
20          cout << "Constructeur : vide" << endl;
21  }
22
23  Complexe::Complexe(double re, double im){
24          m_Re = re;      m_Im = im;
25          cout << "Constructeur : ";
26          print();
27  }
28
29  Complexe::Complexe(const Complexe & c){
30          m_Re = c.m_Re;  m_Im = c.m_Im;
31          cout << "Constructeur Copie : ";
32          print();
33  }
34
```

```
35  Complexe::~Complexe(){
36          cout << "Destructeur : ";
37          print();
38  }
39
40  void Complexe::operator= (const Complexe & c){
41          cout << "dans op= : " ;
42          m_Re = c.m_Re;  m_Im = c.m_Im;
43          print();
44  }
45
46  const Complexe Complexe::operator* (const Complexe & c) const{
47          cout << "dans op* : " << endl;
48          Complexe result(c.m_Re*m_Re-c.m_Im*m_Im,c.m_Re*m_Im+c.m_Im*m_Re);
49          cout << "fin op*." << endl;
50          return( result );
51  }
52
53  void Complexe::conjugue (){
54          cout << "dans conjugue : ";
55          m_Im = -m_Im;
56          print();
57  }
58
59  double Complexe::norme() const{
60          cout << "dans norme : " << endl;
61          Complexe c(m_Re,m_Im);
62          c.conjugue();
63          c = operator*(c);
64          cout << "fin norme. " << endl;
65          return( sqrt(c.m_Re) );
66  }
67
68  void Complexe::print() const{
69          cout << m_Re << " + I * " << m_Im << endl;
70  }
71
72  int main(){
73          cout << "1" << endl;
74          Complexe c1(3,4);
75          Complexe c2(0,1);
76          cout << "2" << endl;
77          Complexe c3 = c1;
78          cout << "3" << endl;
79          c2 = (c1*c2);
80          cout << "4" << endl;
81          double d = c1.norme();
82          cout << "norme de c1 : " << d << endl;
83          cout << "5" << endl;
```

```
84          return 0;
85    }
```

B.10.3 Tableau d'exécution

Remplir le tableau d'exécution du programme suivant : **le tableau d'exécution est fourni en annexe**.

```
1     int U(int &n)
2     {
3             n += 1;
4             return V(n)-5;
5     }
6
7     int V(int n)
8     {
9             if (n == 4 || n == 16)
10                    return -1;
11
12            int tmp;
13            n = n+2;
14            tmp = U(n);
15            return tmp + n;
16    }
17
18    int setValueToTwelve(int a)
19    {
20            a = 12;
21            return a;
22    }
23
24    int main()
25    {
26            int a = 0 ;
27            int fin;
28
29            fin = setValueToTwelve(a);
30            fin = U(a);
31
32            if(fin-7)
33                    a = 12;
34            else if(fin+7)
35                    a = 0;
36            else
37                    a = 4;
38
39            return 0;
40    }
```

Il s'agit d'écrire, si elles existent, les valeurs des variables du programme pour chaque exécution de ligne.

Conventions :
- mettre le numéro de la ligne qu'on vient d'exécuter (éventuellement, on peut découper l'exécution d'une même ligne en plusieurs étapes) ;
- laisser en blanc les variables non existantes ;
- mettre un point d'interrogation pour les variables déclarées non initialisées ;
- pour une variable qui n'est autre qu'une référence sur une autre variable x, on indiquera $[x]$;
- on peut utiliser le symbole " pour répéter la valeur d'une variable à la ligne suivante ;
- pour gagner de la place dans le tableau, les noms des variables n'ont été indexés que par la première lettre du nom de la fonction à laquelle elles sont rattachées, suivie le cas échéant du numéro d'appel à la fonction. De même, le retour de la fonction f a pour nom ret_f.

B.10.4 Résolveur de Sudoku

Le but de cet exercice est d'écrire un programme capable de résoudre le problème du **Sudoku** dont on rappelle les règles :
- une grille 9×9 partiellement remplie est donnée, chaque case de cette grille doit contenir un entier de 1 à 9
- il faut compléter la grille en respectant les trois règles suivantes :
  - dans une même ligne, ne peuvent apparaître qu'une seule fois les entiers de 1 à 9,
  - dans une même colonne, ne peuvent apparaître qu'une seule fois les entiers de 1 à 9,
  - dans une même région, ne peuvent apparaître qu'une seule fois les entiers de 1 à 9 : une « région » est l'un des neuf blocs 3×3 qui constituent la grille.

Dans la suite du problème, on considère la structure suivante :

```
1  struct Grille {
2      int valeur[9][9];
3      bool indice[9][9];
4  };
```

qui indique, pour chacune des 81 cases de la grille, sa valeur (un entier entre 1 et 9 ou bien 0 si la case n'a pas de valeur) et si la case en question fait partie des indices (sa valeur est fixée).

1. Définir une fonction de prototype `void affiche_grille(const Grille& g)` qui affiche une grille (avec `cout` comme à droite de FIG. B.11) : si, en une case de la grille, il n'y a pas encore de valeur affectée, un caractère `'X'` est affiché à la place.
2. Écrire une fonction `Grille lit_grille(string l[9])` qui lit une grille en entrée (un caractère après l'autre, ligne par ligne). Chaque `string` du tableau passé en paramètre représente une ligne de la grille. Si `s` est l'une de ces lignes, alors :
   - `s[i]` est le i-ème caractère de la ligne,
   - si `s[i]` est égal à `'X'` alors la case est à compléter,
   - sinon, il s'agit d'un indice de valeur `s[i] - '0'`.

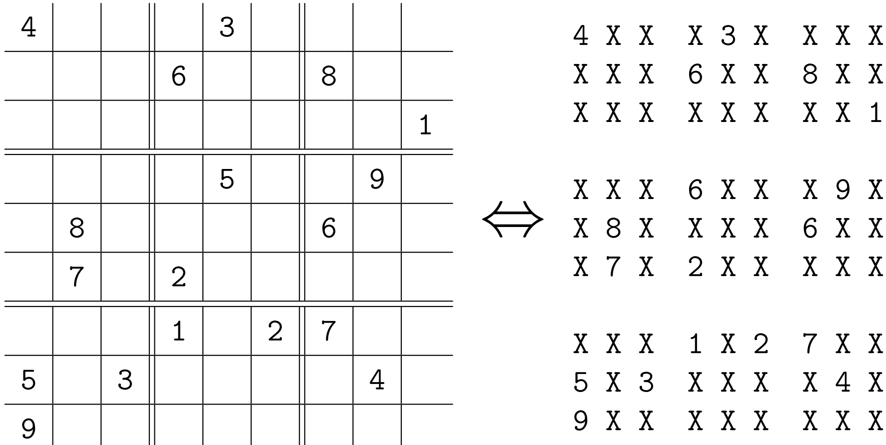


Fig. B.11 – Exemple de grille

4	6	8	9	3	1	5	2	7
7	5	1	6	2	4	8	3	9
3	9	2	5	7	8	4	6	1
1	3	4	7	5	6	2	9	8
2	8	9	4	1	3	6	7	5
6	7	5	2	8	9	3	1	4
8	4	6	1	9	2	7	5	3
5	1	3	8	6	7	9	4	2
9	2	7	3	4	5	1	8	6

Fig. B.12 – Solution correspondante

3. Programmer une fonction `bool verifie_case(const Grille& g, int i, int j)` qui indique si la valeur actuellement attribuée à la case (i, j) entre en conflit avec les autres cases valides de la ligne i, de la colonne j et de la région dans laquelle se trouve la case.
4. Définir une fonction de prototype `bool case_suivante(const Grille& g, int& i, int& j)` qui passe d'une case (i, j) dans la grille à la suivante dans l'ordre lexigraphique (la nouvelle case sera renvoyée par référence) : les cases d'indice sont sautées, et, si il n'y a pas de case suivante ((i, j) est la dernière case de la grille), alors `false` est renvoyé.
5. Programmer une fonction récursive `bool resout_grille(Grille& g, int i, int j)`, qui va :
   – donner successivement à la case (i, j) toutes les valeurs possibles,
   – tester à chaque fois si la valeur entre en conflit ou non avec le reste de la grille déjà remplie,
   – s'il n'y a pas de conflit, s'appeler récursivement sur la case suivante (uniquement si la fin de la grille n'a pas déjà été atteinte).

   Cette fonction renvoie `true` si la grille est soluble à partir de la configuration donnée, et `false` sinon. Quel est l'intérêt de passer la structure Grille par référence ?
6. Définir une fonction de prototype `bool resout_grille(Grille& g)` qui appelle la fonction précédente avec la bonne position de départ pour la résolution.
7. Programmer la fonction `int main()` qui lit une grille (à définir dans le code), l'affiche, puis essaie de la résoudre en affichant la solution s'il y en a une ou bien un message.
8. L'algorithme proposé réalise beaucoup de calculs inutiles et n'exploite pas suffisamment les indices. Suggérer des améliorations (ne pas les programmer !).

B.11 Devoir final 2006 : énoncé

B.11.1 Erreurs à corriger

Corrigez les erreurs dans le programme suivant. Cherchez bien : il y en a 23

```
1   int dmax=10;
2   class polynom {
3       double coef[dmax];
4       int deg;
5       void updateDegree() const;
6   public:
7       polynom() {
8           deg=0;
9           coef[0]=0;
10      }
11      void set(int d,double *c);
12      void print() const;
13      polynom derivate() const;
14      polynom integrate() const;
15      void op-(const polynom& Q) const;
16  }
17
18  void polynom::set(int d,double *c) {
19      deg=d;
20      for (int i=1;i<=d+1;i++)
21          coef[i]=c[i];
22  }
23
24  void polynom::print() const {
25      for (int i=deg;i>=0;i++) {
26          if (i<deg & coef[i]>=0)
27              cout<<'+';
28          cout << coef[i] << "*x^"<<i;
29      }
30  }
31
32  polynom derivate(){
33      polynom D;
34      if (deg>0) {
35          D.deg=deg-1;
36          for (int i=0,i<deg,i++)
37              D.coef[i]=(i+1)*coef[i+1];
38      }
39  }
40
41  polynom polynom::integrate(){
42      polynom I;
43      I.deg=deg+1;
44      for (int i=1;i<=deg+1;i++)
```

```
45            I.coef[i]=coef[i-1]/i;
46        return I;
47    }
48
49    int polynom::updateDegree() {
50        for (int i=deg;i>=1;i--)
51            if (coef[i]==0)
52                deg=i-1;
53    }
54
55    polynom polynom::op-(const polynom& Q) const {
56        polynom P;
57        P.deg=max(deg,Q.deg);
58        for (int i=0;i<=P.deg;i++) {
59            double c1,c2;
60            if (i>deg) c1=0 else c1=coef[i];
61            if (i>Q.deg) c2=0 else c2=Q.coef[i];
62            P.coef[i]=c1-c2;
63        }
64        P.updateDegree;
65        return P;
66    }
67
68    int main() {
69        polynom P;
70        int t[4]={5,2,3,4};
71        P.set(3,t);
72        polynom Q;
73        Q=P.derivate();
74        polynom R;
75        R=Q.integrate();
76        S=P-R;
77        cout << "P=" ; P.print; cout << endl;
78        cout << "Q=" ; Q.print; cout << endl;
79        cout << "R=" ; R.print; cout << endl;
80        cout << "S=" ; S.print; cout << endl;
81    }
```

B.11.2 Qu'affiche ce programme ?

Examinez bien le contenu du programme suivant. La question est simple : quelles sont les différentes lignes affichées sur la *sortie standard* (c'est-à-dire en console, via les instructions "`cout << ... ;`") au cours de son exécution ? Procédez avec méthode en faisant attention aux retours de fonction *par valeurs*, et à l'équilibre nécessaire entre appels aux constructeurs/destructeurs. **Justifiez vos réponses**.

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

```
4
5  int number_of_points;
6
7  class Point3{
8          int which_point;
9          int X,Y,Z;
10
11         void init(int _x=0, int _y=0, int _z=0){
12                 which_point = number_of_points++;
13                 X=_x;   Y=_y;   Z=_z;
14         }
15         void print(){
16                 cout << " Point #" << which_point ;
17                 cout << " [ " << X << ", " << Y << ", " << Z << "]" << endl;
18         }
19
20 public:
21
22         Point3(){
23                 cout << "Constructor #0: " ;
24                 init();
25                 print();
26         }
27         Point3(int _x, int _y, int _z){
28                 cout << "Constructor #1: ";
29                 init(_x,_y,_z);
30                 print();
31         }
32         Point3(const Point3 &pt){
33                 cout << "Constructor #2: ";
34                 init(pt.X,pt.Y,pt.Z);
35                 print();
36         }
37         ~Point3(){
38                 cout << "Destructor:     " ;
39                 print();
40                 number_of_points--;
41         }
42
43         Point3 operator+(Point3 pt){
44                 cout << "Dans operator+ :" ;    print();
45                 Point3 tmp(X + pt.X , Y + pt.Y , Z + pt.Z);
46                 cout << "Fin operator+  " <<endl;
47                 return tmp;
48         }
49         const Point3& operator=(const Point3 &pt){
50                 cout << "Dans operator= :" ;    print();
51                 X=pt.X;         Y=pt.Y;         Z=pt.Z;
52                 cout << "Fin operator=  :" ;    print();
```

```
53                  return pt;
54          }
55  };
56
57  int main()
58  {
59          number_of_points = 0 ;
60          cout << number_of_points << endl;
61          Point3 PTS[2];
62          cout << number_of_points << endl;
63          PTS[0] = Point3(1,2,3);
64          cout << number_of_points << endl;
65          Point3 PT(3,2,1);
66          PTS[1] = PTS[0] + PT;
67          cout << number_of_points << endl;
68          return 0;
69  }
```

B.11.3 Tableau d'exécution

Remplir le tableau d'exécution du programme suivant : **le tableau d'exécution est fourni en annexe**.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  void initClasse(int * classe , bool * bienHabilles, int nbEleves)
 5  {
 6          int identifiant_eleve;
 7
 8          for ( int i = 0 ; i < nbEleves ; i++ )
 9          {
10                  identifiant_eleve = i*2;
11                  classe[i] = identifiant_eleve;
12                  if( i%2 == 0 )
13                          bienHabilles[i] = false;
14                  else
15                          bienHabilles[i] = true;
16          }
17  }
18
19  bool tousBienHabilles(bool * bienHabilles, int nbEleves)
20  {
21          bool ok = true;
22          for ( int i = 0 ; i < nbEleves ; i++ )
23          {
24                  if( !bienHabilles[i] )
25                          return false;
26          }
```

```
27
28          nbEleves = 2;
29          return ok;
30          nbEleves = 1;
31  }
32
33  void PrendreLaPhoto(int & nbEleves)
34  {
35          nbEleves--;
36
37          int &n = nbEleves;
38          for (int i = 0 ; i < nbEleves ; i++ )
39                  cout << "Eleve " << i << " dit : cheese" <<  endl;
40          cout << "Photographe dit : souriez ... Click Clack, merci ... " << endl;
41
42          n++ ;
43  }
44
45
46  int main()
47  {
48
49          int nbEleves = 2;
50          int *classe = new int[nbEleves];
51          int *copieDeClasse = classe;
52          bool * bienHabilles = new bool [nbEleves];
53
54          initClasse(classe,bienHabilles,nbEleves);
55
56          for (int i = 0 ; i < nbEleves ; i++ )
57                  copieDeClasse[i] =classe[i] ;
58
59          bienHabilles[0] = true;
60
61          if( tousBienHabilles(bienHabilles,nbEleves ) )
62                  PrendreLaPhoto(nbEleves);
63
64          delete[] bienHabilles;
65          delete[] classe;
66
67          return 0;
68  }
```

Il s'agit d'écrire, si elles existent, les valeurs des variables du programme pour chaque exécution de ligne.

Conventions :

– mettre le numéro de la ligne qu'on vient d'exécuter (éventuellement, on peut découper l'exécution d'une même ligne en plusieurs étapes) ;

- laisser en blanc les variables non existantes ;
- mettre un point d'interrogation pour les variables déclarées non initialisées ;
- pour une variable qui n'est autre qu'une référence sur une autre variable x, on indiquera $[x]$; même chose si deux pointeurs adressent le même espace mémoire
- on peut utiliser le symbole " pour répéter la valeur d'une variable à la ligne suivante ;
- pour gagner de la place dans le tableau, les noms des variables n'ont été indexés que par la première lettre du nom de la fonction à laquelle elles sont rattachées, suivie le cas échéant du numéro d'appel à la fonction. De même, le retour d'une fonction f a pour nom ret_f.
- dans le cas d'un tableau, on indiquera les valeurs prises dans tout le tableau dans **une seule cellule** du tableau d'exécution. Par exemple, si on a **int tab[2] = {4,6} ;**, on indiquera le contenu de **tab** par : **4,6** dans la cellule correspondante du tableau d'exécution.
- dans le cas des boucles **for**, il faudra mettre en évidence chaque passage. Ne pas oublier les variables *compteurs*

B.11.4 Huit dames

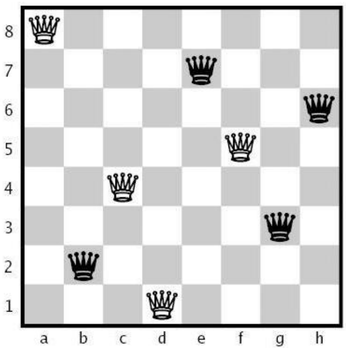


Fig. B.13 – Une solution du problème des huit dames

Le but de cet exercice est d'écrire un programme capable de générer toutes les solutions du problème des 8 dames. Il s'agit de poser huit dames d'un jeu d'échecs sur un échiquier de 8×8 cases sans que les dames ne puissent se menacer mutuellement, conformément aux règles du jeu d'échecs. Par conséquent, deux dames ne doivent jamais partager la même rangée, colonne, ou diagonale. Une solution possible est représentée figure B.13. On demande de fournir les programmes correspondant à trois solutions successives. Programmez bien de façon progressive, en découpant en fonctions, etc. Même s'il vaut habituellement 8, mettez le nombre N de dames (et donc la taille de l'échiquier en paramètre constant au début du programme).

1. Solution 1 :
   - Une configuration, c'est-à-dire les positions des huit dames, est mémorisée sous forme d'un tableau de coordonnées.
   - Écrire une fonction qui affiche une configuration.
   - Écrire une fonction récursive qui génère toutes les configurations possibles (sans tenir compte des menaces !)
   - Écrire une fonction qui vérifie pour une configuration donnée qu'il s'agit d'une solution, c'est-à-dire que les dames ne se menacent pas.

- Compléter la fonction de génération pour afficher les solutions.
- Combien explore t'on de configurations ? Est-ce beaucoup pour un ordinateur quand $N = 8$?

2. Solution 2 :

   - Afin de limiter le nombre de configurations explorées, modifier la solution précédente pour vérifier, au moment de placer un dame, qu'elle n'est pas en prise avec celles qui sont déjà posées.

3. Solution 3 :

   - Profitant du fait qu'il y a exactement une dame par colonne, changer la façon de représenter une solution.
   - Changer en conséquence la façon de générer les configurations. Combien y-a-t'il maintenant de configurations possibles ?
   - Pour accélérer les tests, il est astucieux de mémoriser pour chaque ligne et chaque diagonale si celle-ci est déjà "occupée" par une dame précédemment posée. Mettre en oeuvre cette solution.

4. NB : les meilleures solutions algorithmiques actuelles arrivent à résoudre le problème avec $N = 1000000$ en très peu d'étapes !

Annexe C

La CLGraphics

Voir supplément.

Annexe D

Fiche de référence finale

Fiche de référence (1/5)

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=254;
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);
int i,j;
double x=double(i)/j;
```
- Pile/Tas
- Type énuméré :
```
enum Dir{nord,est,
         sud,ouest};
void avance(Dir d);
```
- Variables statiques :
```
int f() {
 static bool first=true;
 if (first) {
  first=false;
  ...
 }
 ...
}
```

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:
    ...;
}
```
```
mx=(x>y)?x:y;
```

Boucles

```
do {
  ...
} while (!ok);
```
```
int i=1;
while (i<=100) {
  ...
  i=i+1;
}
```
```
for (int i=1;i<=100;i++)
  ...
```
```
for (int i=1,j=100;j>i;
     i=i+2,j=j-3)
  ...
```
```
for (int i=...)
  for (int j=...) {
    // saute le cas i==j
    if (i==j)
      continue;
    ...
  }
```
```
for (int i=...) {
  ...
  if (t[i]==s){
    ...
    // quitte la boucle
    break;
  }
  ...
}
```

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/Récursif

– 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[10],y[10];
  for (int i=0;i<10;i++)
       y[i]=2*x[i];
  ```
  –
  ```
  const int n=5;
  int i[n],j[2*n]; // OK
  ```

– Initialisation :
```
int t[4]={1,2,3,4};
string s[2]={"ab","cd"};
```
– Affectation :
```
int s[4]={1,2,3,4},t[4];
for (int i=0;i<4;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 une_autre(obj A);
  };
  void obj::a_moi() {
    x=..;    // OK
    ..=y;    // OK
    z=..;    // OK
  }
  void obj::pour_tous() {
    x=..;     // OK
    a_moi(); // OK
  }
  void 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
  ```
- ```
  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 :
  ```
  point point::operator+(
                  point b) {
    return point(x+b.x,
                 y+b.y);
  }
  ...
    c=point(1,2)
      +f(point(2,3));
  ```
- Destructeur :
  ```
  obj::~obj() {
    ...
  }
  ```
- Constructeur de copie :
  ```
  obj::obj(const obj& o) {
    ...
  }
  ```
  Utilisé par :
  - `obj b(a);`
  - `obj b=a;`
  `//Différent de obj b;b=a;`
  - paramètres des fonctions
  - valeur de retour
- Affectation :
  ```
  const obj& obj::operator=
             (const obj&o) {
    ...
    return o;
  }
  ```
- Objets avec allocation dynamique automatique : cf section 10.11
- Accesseurs :
  ```
  class mat {
     double *x;
  public:
   inline double& operator()
            (int i,int j) {
       assert(i>=0 ...);
       return x[i+M*j];
    }
    inline double operator()
        (int i,int j)const {
       assert(i>=0 ...);
       return x[i+M*j];
    }
    ...
  ```

Compilation séparée

- `#include "vect.h"`, y compris 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 nom;
  ifstream f(nom.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 l'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 somme()const;
  };
  ...
  template <typename T>
  T paire<T>::somme()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;
    hop<5> B;
    ...
  ```

Conseils

- Travailler en local
- `CertisLibs Project`
- Nettoyer en quittant.
- Erreurs et warnings : cliquer.
- Indenter.
- Ne pas laisser de warning.
- Utiliser le debuggeur.
- Faire des fonctions.
- Tableaux : quand c'est utile ! (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.
- Debug/Release : nettoyer les deux.
- Faire des objets.
- Ne pas toujours faire des objets !
- Penser interface / implémentation / utilisation.

Clavier

- Build : F7
- Start : Ctrl+F5
- Compile : Ctrl+F7
- Debug : F5
- Stop : Maj+F5
- Step over : F10
- Step inside : F11
- Indent : Ctrl+K,Ctrl+F
- Add New It. : Ctrl+Maj+A
- Add Exist. It. : Alt+Maj+A
- Step out : Maj+F11
- Run to curs. : Click droit
- Complétion : Alt+→
- 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>
  ...
  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');
  size_t j=s.find('h',3);
  size_t i=s.find("hop");
  size_t j=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 <cassert>
  ...
   assert(x!=0);
   y=1/x;
– #include <ctime>
  s=double(clock())
          /CLOCKS_PER_SEC;
– #define _USE_MATH_DEFINES
  #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;
  double x;
  x=i/j; // NON!
  x=double(i/j); //NON!
– double x[10],y[10];
  for (int i=1;i<=10;i++)
       // NON!
       y[i]=2*x[i];
– int n=5;
  int t[n]; // NON
– int f()[4] { // NON!
      int t[4];
      ...
      return t; // NON!
  }
  ...
  int t[4]
  t=f();
– int s[4]={1,2,3,4},t[4];
  t=s; // NON!
– int t[2];
  t={1,2}; // NON!
– struct Point {
     double x,y;
  } // NON!
– Point a;
  a={1,2}; // NON!
– #include "vect.cpp"// NON!
– void 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; // Déja fait!
– int *t,s;// s est int
         // et non int* !
  t=new int[n];
  s=new int[n];// NON!
– class point {
    int x,y;
  public:
    ...
  };
  ...
    point a={2,3}; // NON!
```

– Oublier de redéfinir le constructeur vide.

```
– point p=point(1,2);// NON!
  point p(1,2);      // OUI!
– obj* t=new obj[n];
  ...
  delete t; // oubli de []
– //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! Pour les
      // boucles seulement!
  }
– for (i ...)
   for (j ...) {
    ...
    if (...)
     break;//NON! Ne quitte
     // Que la boucle en j!
– int i;
  double x;
  ...
  j=max(i,0);//OK
  y=max(x,0);//NON! Utiliser
  // 0.0 et non 0 (max est
  // un template (STL)...)
```

CertisLibs

– Voir documentation...