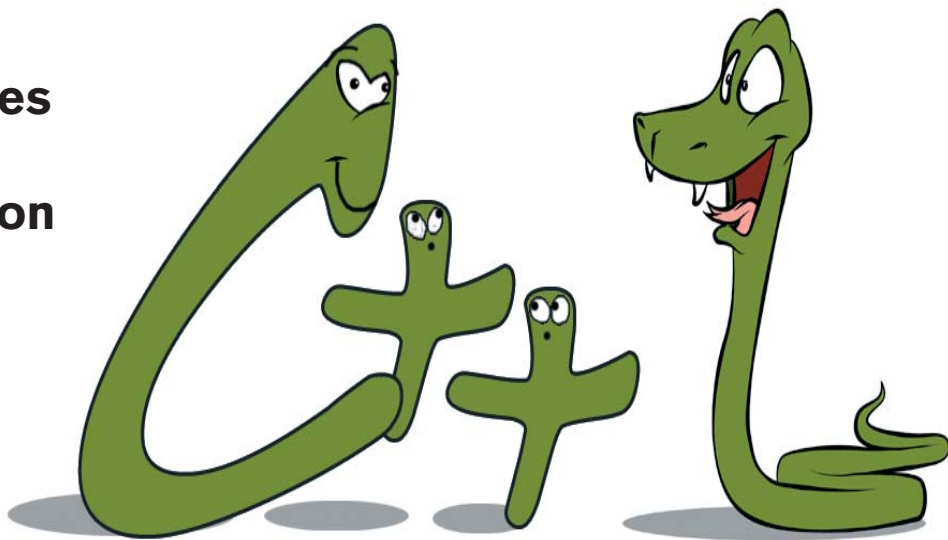


Le Programmeur

Initiation à la programmation avec Python et C++

**Apprenez
simplement les
bases de la
programmation**



Yves Bailly

PEARSON

LE PROGRAMMEUR

Initiation à la programmation avec Python et C++

Yves Bailly

PEARSON

The Pearson logo consists of the word "PEARSON" in a white, sans-serif, uppercase font, positioned above a thin, white, curved line that arches slightly to the right.

Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00

Mise en pages : TyPAO

ISBN : 978-2-7440-4086-3
Copyright© 2009 Pearson Education France
Tous droits réservés

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher.

Sommaire

Introduction	VII	10. Mise en abîme : la récursivité	129
1. Mise en place	3	11. Spécificités propres au C++	139
2. Stockage de l'information : les variables	15	12. Ouverture des fenêtres : programmation graphique	153
3. Des programmes dans un programme : les fonctions	29	13. Stockage de masse : manipuler les fichiers	185
4. L'interactivité : échanger avec l'utilisateur	39	14. Le temps qui passe	193
5. Ronds-points et embranchements : boucles et alternatives	47	15. Rassembler et diffuser : les bibliothèques	207
6. Des variables composées	59	16. Introduction à OpenGL	215
7. Les Tours de Hanoï	79	17. Aperçu de la programmation parallèle	229
8. Le monde modélisé : la programmation orientée objet (POO)	103	18. Aperçu d'autres langages	237
9. Hanoï en objets	115	19. Le mot de la fin	247
		Glossaire	251
		Index	253

Table des matières

Introduction	IX	3.2. Les paramètres	33
Un peu d'histoire	X	3.3. Les blocs d'instructions	35
Le choix des langages	XI	3.4. Retourner une valeur	36
Et maintenant... ..	XII	3.5. Quand créer une fonction ?	38
Convention d'écriture	XIII		
Partie 1 - Découverte	1	CHAPITRE 4. L'interactivité : échanger	
		avec l'utilisateur	39
CHAPITRE 1. Mise en place	3	4.1. Salutations !	40
1.1. Windows	5	4.2. Obtenir un nombre entier	42
1.2. Mac OS X	8	4.3. Obtenir une valeur décimale	44
1.3. Linux	8	4.4. Saisir plusieurs valeurs	44
1.4. Votre premier programme	9	CHAPITRE 5. Ronds-points	
CHAPITRE 2. Stockage de l'information :		et embranchements : boucles	
les variables	15	et alternatives	47
2.1. Des types des données	19	5.1. L'alternative	49
2.2. Les entiers	20	5.2. La boucle	54
2.3. Les décimaux et la notation		5.3. Algèbre de Boole et algorithmique ..	57
scientifique	21	CHAPITRE 6. Des variables composées	59
2.4. Les chaînes de caractères	23	6.1. Plusieurs variables en une : les tableaux	60
2.5. Erreurs typiques liées aux variables .	25	6.2. Plusieurs types en un : structures	67
CHAPITRE 3. Des programmes		6.3. Variables composées et paramètres ..	72
dans un programme : les fonctions	29	CHAPITRE 7. Les Tours de Hanoï	79
3.1. Définir une fonction	31	7.1. Modélisation du problème	82

7.2. Conception du jeu	86
7.3. Implémentation	90

Partie 2 - Développement logiciel 101

CHAPITRE 8. Le monde modélisé :

la programmation orientée objet (POO) ...	103
8.1. Les vertus de l'encapsulation	105
8.2. La réutilisation par l'héritage	109
8.3. Héritage et paramètres : le polymorphisme	112

CHAPITRE 9. Hanoï en objets

9.1. Des tours complètes	116
9.2. Un jeu autonome	119
9.3. L'interface du jeu	122
9.4. Le programme principal	126

CHAPITRE 10. Mise en abîme :

la récursivité	129
10.1. La solution du problème	130
10.2. Une fonction récursive	132
10.3. Intégration au grand programme	135

CHAPITRE 11. Spécificités propres au C++

11.1. Les pointeurs	140
11.2. Mémoire dynamique	142
11.3. Compilation séparée	144
11.4. Fichiers de construction	149

CHAPITRE 12. Ouverture des fenêtres :

programmation graphique	153
12.1. La bibliothèque Qt	154
12.2. Installation des bibliothèques Qt et PyQt	155
12.3. Le retour du bonjour	159
12.4. Un widget personnalisé et boutoné	162
12.5. Hanoï graphique	169

CHAPITRE 13. Stockage de masse :

manipuler les fichiers	185
13.1. Écriture	187
13.2. Lecture	189
13.3. Aparté : la ligne de commande	190

CHAPITRE 14. Le temps qui passe

14.1. Réalisation d'un chronomètre	195
--	-----

Partie 3 - Pour aller plus loin 205

CHAPITRE 15. Rassembler et diffuser :

les bibliothèques	207
15.1. Hiérarchie d'un logiciel	209
15.2. Découpage de Hanoï	210

CHAPITRE 16. Introduction à OpenGL

16.1. Installation	217
16.2. Triangle et cube de couleur	217
16.3. La GLU et autres bibliothèques	226

CHAPITRE 17. Aperçu de la programmation

parallèle	229
17.1. Solution simple pour Hanoï	231
17.2. Solution parallélisée	232
17.3. Précautions et contraintes	236

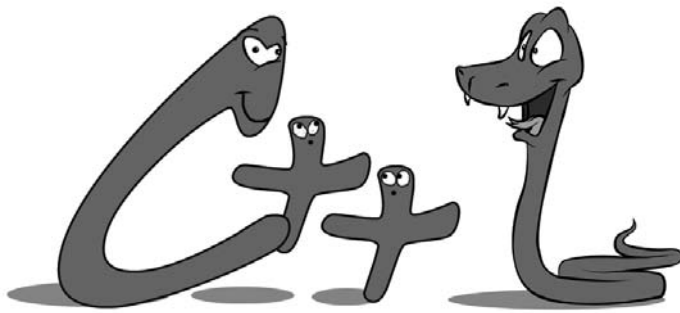
CHAPITRE 18. Aperçu d'autres langages

18.1. Ada	238
18.2. Basic	240
18.3. C#	241
18.4. OCaml	242
18.5. Pascal	243
18.6. Ruby	244

CHAPITRE 19. Le mot de la fin

Glossaire

Index



Introduction

Depuis que vous utilisez un ordinateur, vous avez très certainement entendu parler de cette activité mystérieuse qu'est "programmer", exercée par ces êtres étranges que sont les "programmeurs". Tout ce que vous voyez sur votre écran est l'expression de programmes. Nous vous proposons, dans cet ouvrage, de passer de l'autre côté de l'écran, de découvrir la programmation.

Un ordinateur est un outil, comme un tournevis. Mais à la différence du tournevis, l'ordinateur n'est pas conçu pour exécuter une tâche précise et unique : il l'est, au contraire, pour être fortement modulable et adaptable à pratiquement tous les contextes possibles. Pour cela, il a la capacité d'exécuter un certain nombre d'ordres, plus communément appelés *instructions*, chacune correspondant à une tâche simple et élémentaire. Programmer consiste à définir une succession ordonnée d'instructions élémentaires devant être exécutées par la machine, pour obtenir l'accomplissement d'une tâche complexe. Un aspect essentiel de la programmation est donc la mise en pratique de la méthode cartésienne : décomposer un problème complexe en sous-problèmes simples.

Mais peut-être vous demandez-vous : "Pourquoi programmer ?"

Simplement par jeu ! Écrire un programme et le voir s'exécuter, chercher pourquoi il ne fait pas ce que vous attendiez, est une activité ludique en soi.

Plus sérieusement, aussi bien en milieu professionnel que familial, on éprouve fréquemment le besoin d'automatiser certaines choses répétitives et laborieuses, d'obtenir certains résultats, de traiter certaines données pour lesquelles on ne dispose

pas de logiciel tout prêt – ou alors, ceux qui existent sont mal adaptés ou trop onéreux à acquérir. Dans ce genre de situations, avoir la capacité d'écrire un petit programme peut faire gagner beaucoup de temps et de précision, de limiter les efforts. Voici un autre aspect important de la programmation : elle est bien souvent la meilleure réponse à la paresse !

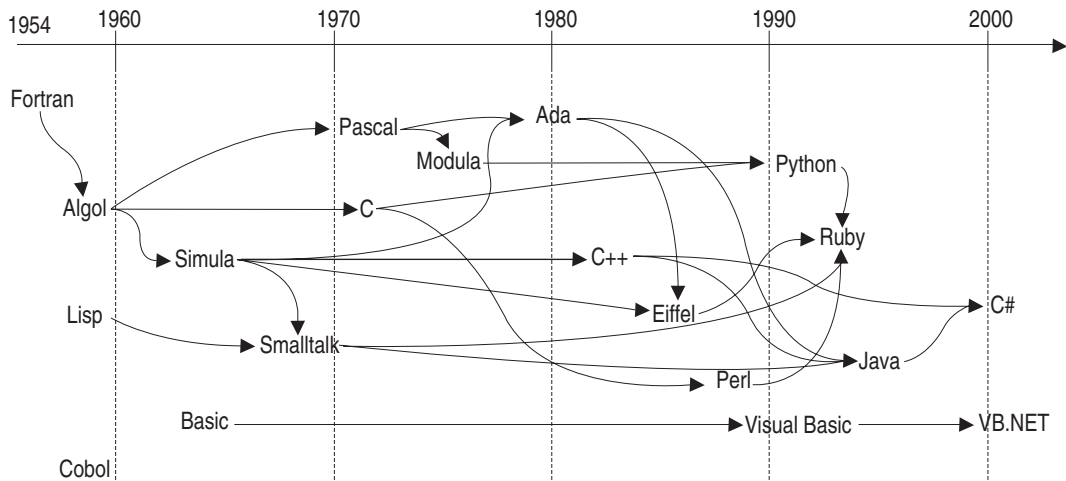
Un peu d'histoire

Les premiers ordinateurs étaient programmés à l'aide de cartes perforées, les instructions étant données sous la forme de longues suites de 0 et de 1 qu'il fallait entrer manuellement. Les programmeurs devaient se confronter directement au *langage machine*, c'est-à-dire au langage brut que comprend la machine – qui n'a rien d'humain. Au milieu des années cinquante, apparurent les premiers langages de programmation, dont l'objectif premier était de détacher le programmeur du langage machine. Dès lors, les programmes sont écrits dans un dialecte relativement compréhensible, respectant une syntaxe et une grammaire strictes et précises. Comme ces langages furent développés par nos amis anglo-saxons, on pourrait les qualifier de langue anglaise très simplifiée.

Tandis que le langage machine est, par nature, très lié à l'électronique de l'ordinateur, le développement des langages de programmation a ouvert la voie de l'*abstraction*, principe fondamental qui cherche à éloigner autant que possible le programmeur des contingences matérielles, pour le rapprocher des concepts, de la "chose" concrète ou abstraite qu'il doit traiter. Lorsqu'on veut modéliser informatiquement un moteur de voiture ou la courbe subtile d'une fonction mathématique, autant ne pas trop s'encombrer l'esprit avec l'état moléculaire des transistors du microprocesseur...

C'est ainsi qu'au fil du temps différents *paradigmes* (ensembles de règles et de principes de conception) de programmation se sont développés. Programmation structurée, fonctionnelle, orientée objet... À chaque évolution, l'objectif est toujours le même : permettre la meilleure représentation informatique possible, la plus intuitive possible, de l'idée sur laquelle opère le programme.

On peut esquisser une sorte d'arbre généalogique des langages de programmation. En donner une liste exhaustive est pratiquement impossible : des milliers sont – ou ont été – utilisés. La Figure I en présente tout de même quelques-uns, les flèches indiquant les sources d'inspiration utilisées pour un langage donné.

**Figure I**

Généalogie simplifiée de quelques langages de programmation.

Il ne s'agit là que d'une vue vraiment très simplifiée, presque caricaturale. Mais vous pouvez constater que les langages s'inspirent fortement les uns des autres. Par ailleurs, la plupart de ceux présentés sur ce schéma sont toujours utilisés et continuent d'évoluer – c'est, par exemple, le cas du vénérable ancêtre Fortran, notamment dans les applications très gourmandes en calculs scientifiques.

Le choix des langages

Il en va de la programmation comme de bien d'autres disciplines : c'est en forgeant que l'on devient forgeron. Cette initiation s'appuiera donc sur deux langages de programmation, appartenant à deux grandes familles différentes : celle des *langages interprétés* et celle des *langages compilés*.

Python

Le langage Python a été conçu en 1991, principalement par le Néerlandais Guido Van Rossum, comme successeur du langage ABC, langage interactif lui-même destiné à remplacer le langage Basic.

C++

Le langage C++ a été conçu en 1983 par Bjarne Stroustrup, comme une extension du langage C, langage proche de l'électronique et encore très largement utilisé, pour lui apporter les principes de la POO (programmation orientée objet).



Python

Le langage Python est un langage orienté objet interprété.

Les programmes écrits dans un langage interprété ne sont pas directement exécutés par l'ordinateur, mais plutôt par un *interpréteur*. Grossièrement, l'interpréteur lit les instructions du programme et les transforme en instructions destinées à la machine. L'interpréteur est donc toujours nécessaire pour exécuter un programme en langage interprété.

Il figure aujourd'hui en bonne place dans de nombreux domaines, de l'analyse scientifique à la programmation de jeux, en passant par la réalisation rapide de petits outils spécialisés ou de sites web dynamiques.

C++

Le langage C++ est un langage orienté objet compilé.

Les programmes écrits dans un langage compilé doivent subir une phase intermédiaire, la *compilation*, à l'issue de laquelle on obtient un fichier directement exécutable par la machine. La compilation est effectuée par un *compilateur*, qui n'est alors plus nécessaire dès que le fichier exécutable a été produit.

Le C++ est aujourd'hui l'un des langages les plus utilisés dans le monde, dans toutes sortes de contextes, notamment ceux où la rapidité d'exécution est primordiale.

Bien qu'appartenant à deux familles différentes, ces deux langages partagent un point commun essentiel : ce sont des langages objet, c'est-à-dire des langages utilisant les techniques de la programmation orientée objet. Nous verrons plus loin ce que ce terme sibyllin représente. Pour l'heure, disons simplement qu'il s'agit, de très loin, du paradigme de programmation le plus répandu dans le monde.

Et maintenant...

Que les impatientes se rassurent, nous allons très bientôt passer à la pratique. Une première partie présentera quelques-uns des concepts fondamentaux de la programmation au moyen de nombreux exemples, à partir desquels nous construirons progressivement un véritable programme permettant de jouer au problème des Tours de Hanoï (aussi appelé problème des Tours de Brahma). Cela après avoir installé sur votre ordinateur les logiciels nécessaires, essentiellement l'interpréteur Python et le compilateur C++.

Une deuxième partie nous emmènera plus loin dans l'exploration des techniques permettant la réalisation d'un logiciel, notamment en décrivant ce qu'est exactement cette mystérieuse programmation objet. Nous aboutirons alors à un programme presque complet, disposant d'une interface graphique attrayante (du moins, espérons-le !).

Enfin, cet ouvrage se terminera par quelques pistes destinées à vous ouvrir de larges horizons, de l’affichage en trois dimensions à la programmation parallèle, sans oublier une petite présentation d’autres langages courants.

Nous n’aurons pas la prétention de vous faire devenir en quelques pages un ingénieur en développement logiciel : c’est un métier à part entière, qui nécessite des années d’études et d’expériences. Ce livre n’est en aucun cas un manuel pour apprendre les langages Python ou C++ : ceux-ci ne sont utilisés que comme support des concepts généraux de la programmation, tels qu’ils se retrouvent dans la plupart des langages de programmation. Les approches qui seront proposées ici ne seront pas toujours les plus efficaces en termes de performances, mais nous recherchons avant tout la compréhension et l’acquisition des concepts. À l’issue de cette lecture, vous serez en mesure de créer vos programmes pour vos propres besoins et d’aller chercher les informations qui vous manqueront inévitablement. Ce sera alors pleinement à vous de jouer.

Convention d’écriture

La flèche ➡ indique la continuité de la ligne de code.

Partie 1

Découverte

CHAPITRE 1. Mise en place

CHAPITRE 2. Stockage de l'information : les variables

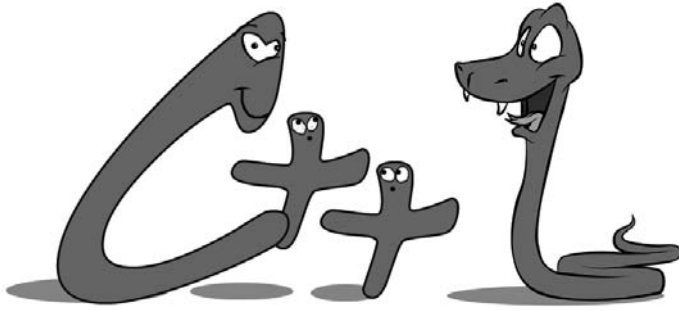
CHAPITRE 3. Des programmes dans un programme : les fonctions

CHAPITRE 4. L'interactivité : échanger avec l'utilisateur

CHAPITRE 5. Ronds-points et embranchements : boucles et alternatives

CHAPITRE 6. Des variables composées

CHAPITRE 7. Les Tours de Hanoï



1

Mise en place

Au sommaire de ce chapitre :

- Windows
- Mac OS X
- Linux
- Votre premier programme

Pour forger, encore faut-il disposer d'une forge et de quelques outils indispensables. Nous allons donc commencer par mettre en place un *environnement de développement*, c'est-à-dire installer sur votre ordinateur les outils nécessaires à la pratique de la programmation.

Si vous cherchez un peu, vous trouverez sans doute de nombreux environnements sophistiqués, chatoyants, parfois onéreux. Mais n'oubliez pas que même les plus grands pilotes de Formule 1 ont appris à conduire, probablement sur de petites voitures... Aussi allons-nous nous contenter d'un environnement minimal, simple à maîtriser. Au moins dans un premier temps, nous n'aurons besoin que de trois outils :

- un interpréteur du langage Python, pour exécuter nos programmes écrits en Python ;
- un compilateur du langage C++, pour compiler nos programmes écrits en C++ afin d'obtenir un fichier exécutable ;
- un éditeur de texte adapté à la programmation.

Détaillons ce dernier point. Un *éditeur de texte*, raccourci pour dire en réalité "programme permettant de modifier un fichier ne contenant que du texte brut", n'est en aucun cas un logiciel de traitement de texte. En effet, un fichier issu d'un traitement de texte ne contient pas que des caractères : il contient également de nombreuses informations de mise en forme, comme le gras, l'italique, la taille de certaines lettres... Toutes ces informations sont parfaitement indigestes pour un interpréteur ou un compilateur.

Ce qu'il nous faut, c'est un logiciel ne stockant réellement *que* les lettres que nous allons taper au clavier pour écrire nos programmes. Sous Windows, un exemple typique est Notepad ; sous GNU/Linux, KEdit ou GEdit ; sous Mac OS X, TextEdit. S'ils pourraient convenir, ces (petits) logiciels sont pourtant trop simples pour notre tâche de programmation : il faut des outils possédant certaines fonctionnalités adaptées. On pourrait citer les célèbres Vim ou Emacs, sur tous les systèmes, mais leur prise en main peut être pour le moins délicate. Les sections suivantes proposeront des outils plus simples à manipuler.

1.1. Windows

Pour commencer, nous vous recommandons d'installer l'éditeur de texte Notepad++, que vous trouverez sur le site <http://notepad-plus.sourceforge.net/fr/site.htm>. Il sera plus que suffisant pour nos besoins. À partir de la page indiquée, suivez le lien Télécharger, puis Télécharger les binaires de Notepad++, enfin choisissez le fichier `npp.4.6.Installer.exe`. Exécutez ce programme une fois téléchargé afin d'installer Notepad++.

Ensuite, l'interpréteur Python. Récupérez le fichier d'installation correspondant à votre système depuis le site <http://www.python.org/download/releases/2.5.1/>, le plus souvent sous la forme d'un fichier d'installation MSI. Par exemple, si vous êtes sur un système 64 bits basé sur un processeur AMD, choisissez le fichier `python-2.5.1.amd64.msi`. Si vous êtes sur un système 32 bits, probablement le cas le plus répandu, choisissez le fichier `python-2.5.1.msi`. Ouvrez ce fichier, acceptez les options par défaut proposées. Et voilà, l'interpréteur Python est installé !

Enfin, nous allons installer un compilateur C++, nommément celui faisant partie de la suite GCC. Celle-ci est un ensemble de compilateurs très largement utilisés dans la vaste communauté du logiciel libre. Pour en obtenir une version sous Windows, nous allons installer l'environnement MinGW, dont vous pouvez télécharger le programme d'installation à partir du site http://sourceforge.net/project/showfiles.php?group_id=2435&package_id=240780. Choisissez la version la plus récente, celle contenue dans le fichier `MinGW-5.1.3.exe` au moment où ces lignes sont écrites. Une fois ce fichier téléchargé, exécutez-le. L'installation étant légèrement plus compliquée que celle de l'interpréteur Python, nous allons la détailler.

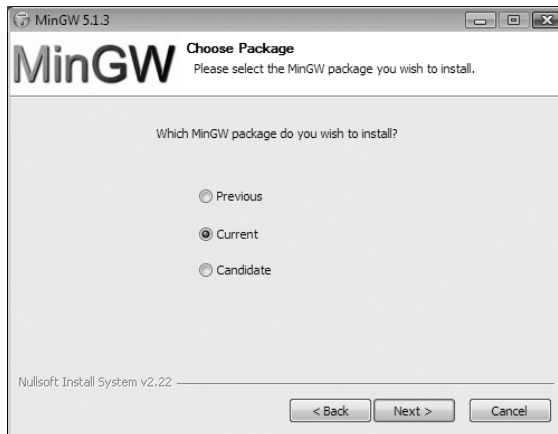
Le téléchargement des composants puis l'installation devraient se dérouler sans problème. Terminez tout cela en choisissant Next autant de fois que nécessaire.

Nous en avons presque terminé. Pour finir, créez un répertoire Programmes dans lequel seront stockés les programmes que vous allez écrire (par exemple à la racine du disque C:), puis créez un petit fichier de commande pour obtenir une ligne de commande dans laquelle nous pourrions effectuer les compilations et exécutions. Pour cela, créez sur le bureau un fichier nommé `prog.bat` et contenant les lignes suivantes :

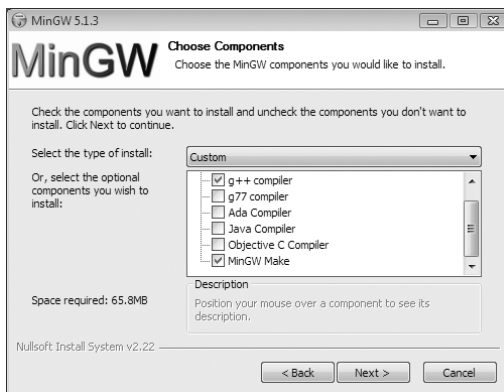
```
set PATH=C:\Python25;C:\MinGW\bin;C:\MinGW\libexec\gcc\mingw32\3.4.5;%PATH%
C:
cd \Programmes
cmd.exe
```



Cette installation nécessite une connexion à Internet, pour télécharger les composants nécessaires. Dans le premier écran, choisissez Download and Install puis cliquez sur Next. Acceptez ensuite l'écran de licence en cliquant sur I Agree.



Ensuite vient le choix du type de paquetage à installer, parmi une ancienne version, la version stable actuelle ou une version expérimentale. Choisissez Current, pour installer la version stable actuelle.



Vous devez maintenant choisir les éléments à installer. En plus du composant de base (le premier de la liste), cochez les cases devant g++ compiler et MinGW Make. Si vous le souhaitez, rien ne vous interdit d'installer également les autres compilateurs (pour les langages Fortran, Ada, Java et Objective-C), mais nous ne les aborderons pas ici !



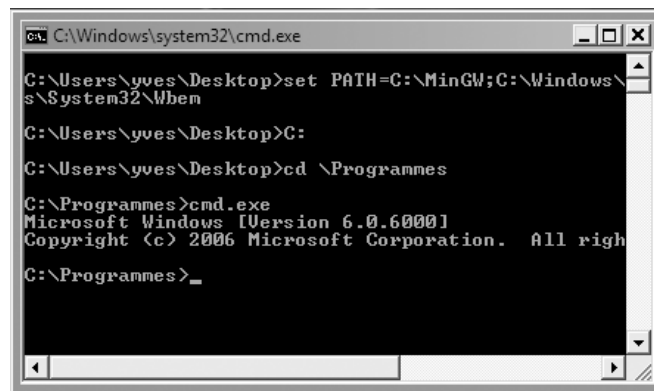
Nous vous recommandons vivement d'accepter le choix par défaut du répertoire de destination, à savoir `C:\MinGW`. Vous pouvez naturellement choisir un autre répertoire, mais alors il vous faudra adapter les indications qui vous seront données par la suite.

Si vous avez installé Python ou MinGW à des endroits autres que ceux proposés par défaut, modifiez la ligne commençant par `set PATH` en conséquence : elle est destinée à faciliter l'accès à l'interpréteur et au compilateur. De même, si vous avez créé le répertoire Programmes à un autre endroit que la racine du disque `C:`, adaptez les deux lignes suivantes.

Lorsque vous lancez ce fichier de commande, vous devriez obtenir quelque chose comme ce qui est illustré par la Figure 1.1 (exemple sous Windows Vista).

Figure 1.1

Ligne de commande pour la programmation sous Windows.



Vous êtes maintenant prêt à commencer à programmer !

1.2. Mac OS X

Un excellent éditeur de texte adapté à la programmation est Smultron, disponible à partir de la page <http://smultron.sourceforge.net>. Sélectionnez la version adaptée à votre version de Mac OS.

Ensuite, le plus simple pour disposer d'un compilateur C++ est d'installer le logiciel Xcode, distribué gratuitement par Apple. Il s'agit en fait d'un environnement de développement complet, le compilateur fourni étant celui issu de la suite GCC, le même que nous allons utiliser sous Windows et GNU/Linux. Vous trouverez l'environnement Xcode à partir de la page <http://developer.apple.com/tools/xcode/>, le téléchargement nécessitant toutefois de s'enregistrer auprès d'Apple. Cela représente environ 1 Go à télécharger, l'installation occupant environ 2,5 Go : prenez donc garde à ne pas remplir votre disque !

Pour l'interpréteur Python, récupérez le fichier d'installation `python-2.5.1-macosx.dmg` depuis le site <http://www.python.org/download/releases/2.5.1/>. Python nécessite au minimum une version 10.3.9 de Mac OS X et fonctionne aussi bien sur les systèmes à base de processeurs PowerPC ou Intel.

Enfin, à partir de Finder, créez un répertoire Programmes dans votre dossier personnel : c'est là que vous stockerez les différents fichiers que vous allez créer par la suite.

1.3. Linux

Normalement, tout est déjà installé si vous utilisez une distribution généraliste, comme Mandriva, Debian, Ubuntu, RedHat, SuSE... cette liste n'étant pas exhaustive. Si tel n'était pas le cas, il suffit d'installer quelques paquetages supplémentaires. Leurs noms peuvent varier d'une distribution à l'autre, mais ils devraient se ressembler.

Pour le compilateur C++, recherchez les paquetages nommés `g++` et `libstdc++`.

Pour l'interpréteur Python, recherchez les paquetages nommés `python2.5` et `python2.5-dev`.

Concernant l'éditeur de texte, vous disposez d'un choix assez considérable, pour ne pas dire déroutant. Si vous utilisez principalement l'environnement Gnome, l'éditeur GEdit convient parfaitement. Si vous utilisez plutôt l'environnement KDE, l'éditeur Kate est probablement le mieux adapté. Sachez que, quel que soit votre environnement de bureau, vous pouvez utiliser n'importe lequel de ces deux éditeurs. D'autres sont naturellement disponibles, comme les célèbres Emacs ou Vim, mais s'ils sont extrêmement puissants, ils peuvent être délicats à prendre en main.

Enfin, créez un répertoire Programmes dans votre répertoire personnel pour contenir vos programmes.

1.4. Votre premier programme

Nous allons sacrifier à une vieille tradition : lorsqu'il s'agit de présenter un langage ou une technique de programmation, le premier programme consiste habituellement à tout simplement afficher le message "Bonjour, Monde !" Cette tradition remonte (au moins) à l'ouvrage de Brian Kernighan, *The C Programming Language*, présentant le langage C en 1974, qui a eu une grande influence sur plus d'une génération de programmeurs. Voici donc les programmes, en Python et en C++ :

Python	C++
<p>Fichier <code>bonjour.py</code> :</p> <pre>1. # -*- coding: utf8 -*- 2. print "Bonjour, Monde !"</pre>	<p>Fichier <code>bonjour.cpp</code> :</p> <pre>1. #include <iostream> 2. int main(int argc, char* argv[]) 3. { 4. std::cout << "Bonjour, Monde !" 5. << std::endl; 6. return 0; 7. }</pre>

Si vous êtes surpris de voir que le programme C++ est beaucoup plus long que le programme Python, sachez que c'est une manifestation du fait que le langage Python est un langage dit de *haut niveau*, tandis que le langage C++ est de plus *bas niveau*. Dit autrement, en Python, l'interpréteur masque beaucoup d'aspects que le compilateur C++ laisse au grand jour. Chacune des deux approches présente des avantages et des inconvénients : tout dépend du contexte et de l'objectif de votre programme.

Généralement, on peut dire que masquer les détails permet de programmer plus rapidement, tandis qu'avoir la possibilité d'agir sur ces détails permet d'améliorer les performances au moment d'exécuter le programme.

Mais revenons justement à nos programmes. Les deux textes que vous voyez, et que vous êtes invité à recopier dans votre éditeur de texte, constituent ce que l'on appelle le *code source* d'un programme. Le code source est ainsi le texte que le programmeur tape dans un éditeur. C'est ce texte que nous allons bientôt soumettre, pour l'un, à l'interpréteur Python, pour l'autre, au compilateur C++.

Une précision avant cela : vous aurez remarqué que les lignes des programmes sont numérotées, de 1 à 2 en Python, de 1 à 6 en C++. Cette numérotation n'a pas d'autre objectif que de nous permettre de faire référence aisément à telle ou telle partie du programme, dans le cadre de cet ouvrage. En aucun cas, vous ne devez recopier ces numéros dans votre éditeur de texte. S'il existe bien des langages de programmation qui exigeaient que les lignes soient numérotées, comme les premiers Basic ou Cobol, ce procédé a été abandonné depuis fort longtemps car inefficace. Les Figures 1.2 à 1.4 montrent à quoi pourrait ressembler votre éditeur sur écran.

Windows

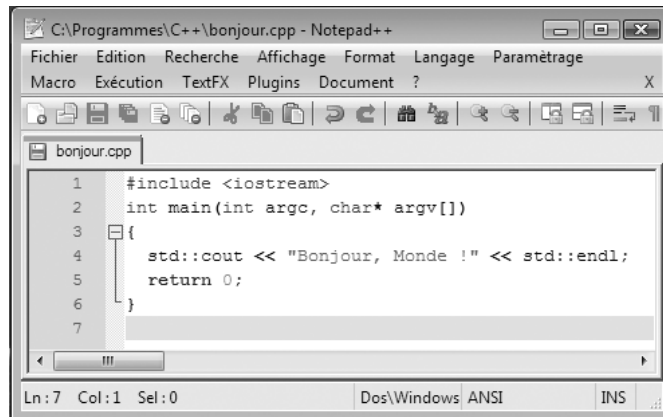
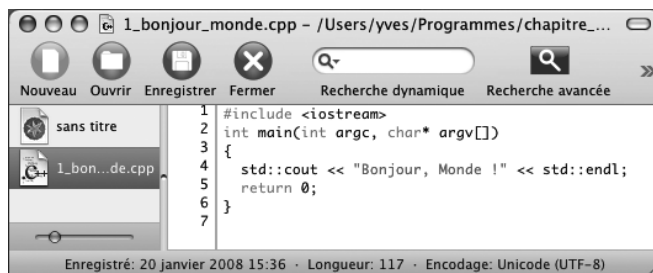
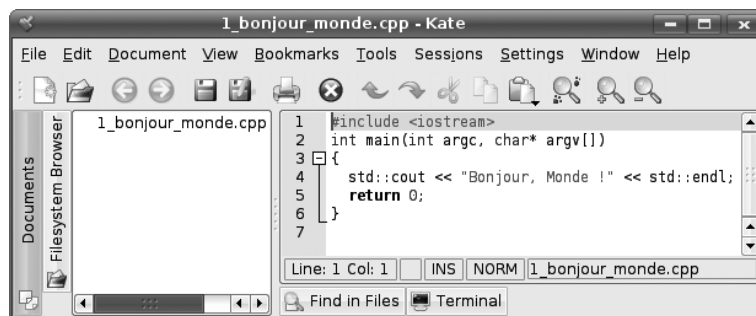


Figure 1.2

Éditeur de texte sous Windows (Notepad++).

Mac OS X**Figure 1.3**

Éditeur de texte sous Mac OS X (Smultron).

Linux**Figure 1.4**

Éditeur de texte sous Linux (Kate).

Remarquez que les lignes sont automatiquement numérotées par l'éditeur lui-même : vous n'avez donc pas à vous en soucier. Vous voyez également que, dès que vous avez sauvegardé votre fichier, son aspect change : certains mots se voient attribuer une couleur. C'est la *mise en évidence syntaxique* (ou *syntax highlight* en anglais), une aide visuelle précieuse lorsqu'on programme. À chaque couleur correspond une certaine nature de mot dans le programme, ce que nous allons découvrir au fil des pages qui suivent.

Enfin, la norme du langage C++ exige que tout programme se termine par une ligne vide : c'est pourquoi vous voyez sept lignes sur les captures d'écran au lieu de six. Cette ligne vide supplémentaire sera implicite dans tous les exemples qui seront donnés par la suite, donc ne l'oubliez pas.

1.4.1. Exécution pour Python

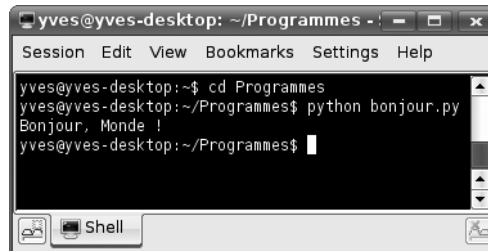
Étant un langage interprété, le programme Python peut être exécuté directement par l'interpréteur, comme ceci :

```
python bonjour.py
```

Par exemple, la Figure 1.5 vous montre le résultat dans une ligne de commande Linux.

Figure 1.5

Exécution du programme sous Linux.



C'est un des principaux avantages des langages interprétés : leur mise en œuvre est très simple.

Si on regarde ce très court programme, la première ligne indique selon quel encodage le texte du programme a été enregistré. Cela est particulièrement important pour les messages que vous voudrez afficher, notamment s'ils contiennent des lettres accentuées. L'affichage proprement dit du message a lieu la ligne suivante. À la fin de cet affichage, un saut de ligne est automatiquement effectué.

Remarquez l'absence de toute ponctuation à la fin de la ligne 2, celle contenant l'instruction d'affichage : c'est là une différence fondamentale avec le programme en langage C++, sur lequel nous allons nous pencher immédiatement.

1.4.2. Compilation et exécution pour le C++

Nous l'avons dit, le langage C++ est un langage compilé, c'est-à-dire que le code source doit être traité par le compilateur pour obtenir un fichier exécutable.

Allez à la fenêtre de ligne de commande, dans le répertoire dans lequel vous avez enregistré votre fichier `bonjour.cpp`. Exécutez l'instruction :

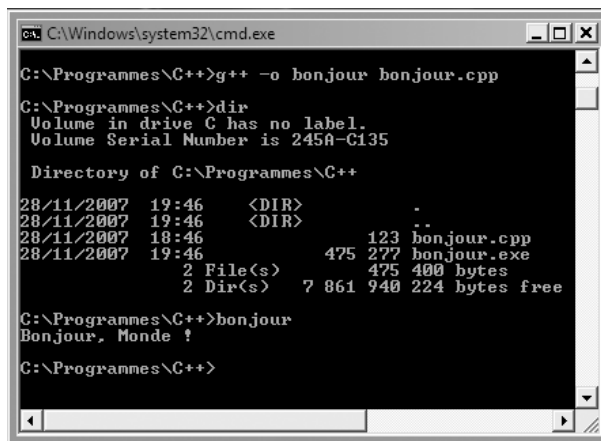
```
g++ -o bonjour bonjour.cpp
```

Cela va créer un fichier exécutable nommé `bonjour` (nanti de l'extension `.exe` sous Windows) dans le répertoire courant. Vous pouvez exécuter ce programme immédiatement

en appelant simplement `bonjour`, ou `./bonjour` si vous êtes sous GNU/Linux. La Figure 1.6 montre un exemple sous Windows.

Figure 1.6

Compilation et exécution sous Windows.



```
C:\Windows\system32\cmd.exe

C:\Programmes\C++>g++ -o bonjour bonjour.cpp

C:\Programmes\C++>dir
Volume in drive C has no label.
Volume Serial Number is 245A-C135

Directory of C:\Programmes\C++

28/11/2007  19:46    <DIR>          .
28/11/2007  19:46    <DIR>          ..
28/11/2007  18:46                123  bonjour.cpp
28/11/2007  19:46            475 277  bonjour.exe
                2 File(s)          475 400 bytes
                2 Dir(s)  7 861 940 224 bytes free

C:\Programmes\C++>bonjour
Bonjour, Monde !

C:\Programmes\C++>
```

Félicitations, vous venez de saisir, de compiler et d'exécuter votre premier programme en langage C++ ! Ce n'était pas si difficile, n'est-ce pas ?

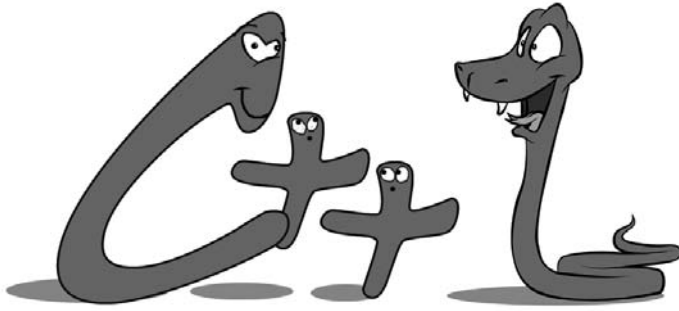
Mais examinons un peu ce code. La première ligne, dont le sens profond s'éclaircira plus tard, est nécessaire pour nous permettre d'utiliser les capacités d'affichage du C++. L'affichage en lui-même est effectué ligne 4, en "envoyant" un texte délimité par des guillemets dans un "objet d'affichage". Cet objet est `std::cout`, l'envoi en lui-même est symbolisé par `<<`. Le préfixe `std::` signale qu'il s'agit d'un objet de la *bibliothèque standard*, ce qui signifie qu'il est normalement disponible sur n'importe quel autre compilateur C++. Nous verrons plus loin la signification exacte de ce préfixe. Le morceau `cout` pourrait se traduire par *console output*, c'est-à-dire *sortie vers l'écran*.

À la suite du texte, on envoie (toujours avec `<<`) l'objet `std::endl`, qui est une façon de représenter un saut de ligne. Encore une fois, cet objet fait partie de la bibliothèque standard, `endl` signifiant *end of line* (*fin de ligne* en français). Sans cela, nous n'aurions pas de saut de ligne après l'affichage du texte, ce qui pourrait être disgracieux.

Les autres lignes font partie de ce qu'on pourrait appeler "l'emballage" minimal d'un programme C++. Inutile de rentrer pour l'instant dans les détails (mais cela viendra). Pour l'heure, il suffit de savoir que, sans elles, le programme serait incorrect et donc ne pourrait pas être transformé en fichier exécutable par le compilateur.

Peut-être encore plus que l'interpréteur Python, un compilateur C++ peut se révéler particulièrement tatillon sur l'exactitude de ce qu'on écrit.

Enfin, dernière remarque, la présence du point-virgule est obligatoire pour séparer deux instructions du programme. Si vous oubliez un seul point-virgule dans votre programme, le compilateur vous couvrira d'insultes diverses et refusera de créer un fichier exécutable.



Stockage de l'information : les variables

Au sommaire de ce chapitre :

- Des types des données
- Les entiers
- Les décimaux et la notation scientifique
- Les chaînes de caractères
- Erreurs typiques liées aux variables

Nous n'avons manipulé dans les exemples précédents que des *valeurs littérales*, c'est-à-dire des données qu'on écrit littéralement, en toutes lettres. Mais l'essence même d'un programme est de travailler sur des données (plus ou moins) quelconques, pour les combiner, les modifier, afin de produire un certain résultat. Il est donc nécessaire de les stocker afin de pouvoir les réutiliser plus tard. Ce stockage s'effectue dans la mémoire de l'ordinateur et chaque donnée, pour être ultérieurement identifiée, reçoit un nom : c'est ce qu'on désigne usuellement par le terme de *variable*. Au sens strict, une variable est une zone mémoire contenant une donnée susceptible d'être modifiée, le nom n'étant qu'un moyen d'y accéder. Dans la pratique, on identifie le nom à la variable, par abus de langage.

Considérez les deux programmes suivants :

Python

```
1. # -*- coding: utf8 -*-  
2. x = 5  
3. print 10*x
```

La ligne 2 a deux effets : d'abord, elle *déclare* la variable x, ensuite la valeur 5 y est stockée. Tout cela au moyen du simple signe =.

En Python, la déclaration d'une variable se fait simplement en lui affectant une valeur.

C++

```
1. #include <iostream>  
2. int main(int argc, char* argv[])  
3. {  
4.     int x;  
5.     x = 5;  
6.     std::cout << 10*x << std::endl;  
7.     return 0;  
8. }
```

La ligne 4 a pour effet de *déclarer* la variable x. La valeur 5 y est stockée en ligne 5, par le signe =.

En C++, les variables doivent être déclarées avant toute utilisation. Mais une déclaration peut combiner une affectation.

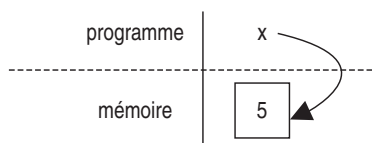
Exécutez ces deux programmes, comme nous l'avons vu précédemment. Les deux devraient simplement afficher la valeur 50, alors que celle-ci n'apparaît nulle part... Elle est en fait *calculée* à partir de la valeur contenue dans une variable, ici nommée x.

Dans pratiquement tous les langages de programmation, une variable doit être *déclarée* avant d'être utilisée. La déclaration d'une variable consiste (au minimum) à lui donner un nom, afin que l'interpréteur ou le compilateur reconnaisse ce nom par la suite et "sache" qu'il correspond à une variable dont on veut utiliser le contenu.

Techniquement, la déclaration d'une variable consiste à réserver une zone d'une certaine taille dans la mémoire de l'ordinateur et à identifier cette zone avec une étiquette (voir Figure 2.1). L'opération consistant à placer une certaine valeur dans cette zone est l'*affectation*. L'étiquette, c'est-à-dire le nom de la variable, est un moyen d'accéder à la valeur contenue dans cette zone.

Figure 2.1

Une variable permet de stocker une valeur dans la mémoire de l'ordinateur et d'accéder à celle-ci.



Regardez les instructions d'affichage des deux programmes précédents, ligne 3 en Python et ligne 6 en C++. Le symbole étoile (ou astérisque) * est celui de la multiplication. L'écriture $10 \times x$ peut donc se traduire par :

multiplier par 10 le contenu de la zone mémoire identifiée par x

Ou, plus simplement et par un abus très commun de langage :

multiplier par 10 la valeur de x

On demande ensuite simplement l'affichage du résultat de cette opération.

Il est naturellement possible d'affecter une autre valeur à la variable x, par exemple le résultat d'une opération l'impliquant elle-même :

Python	C++
<pre>1. x = 5 2. x = 10*x 3. print x</pre>	<pre>1. x = 5; 2. x = 10*x; 3. std::cout << x << std::endl;</pre>

La deuxième ligne des extraits précédents peut surprendre : quel sens cela a-t-il de dire que x est égal à dix fois sa propre valeur ? Il faut bien comprendre que, dans ces contextes, les symboles = ne représentent pas une situation d'égalité, mais qu'ils sont une opération dans laquelle on change le contenu dans la variable figurant à gauche de l'opérateur pour y placer la valeur figurant à droite.

Au cours de vos expérimentations, vous constaterez probablement une certaine difficulté à trouver un nom à vos variables. La tentation est alors grande de les nommer simplement par a, b, c... Surtout ne cédez pas à cette tentation. Rappelez-vous qu'un

programme n'est écrit qu'une seule fois mais généralement relu de nombreuses fois. Comparez ces trois extraits de code, qui font exactement la même chose :

<pre>d = 100; t = 2; v = d/t;</pre>	<pre>distance = 100; temps = 2; vitesse = distance/temps;</pre>	<pre>distance_km = 100; temps_h = 2; vitesse_km_h = distance_km/temps_h;</pre>
---	---	--

Vous l'aurez deviné, le symbole barre inclinée (ou *slash*) / est celui de la division. L'extrait de gauche ne présente aucune signification. L'extrait au milieu est, par contre, assez limpide dans ses intentions, bien que subsiste une incertitude sur les unités utilisées. L'extrait de droite est parfaitement clair, les noms donnés aux variables indiquent exactement l'opération effectuée.

Pour nommer vos variables, vous disposez des 26 lettres de l'alphabet (sans accent), des 10 chiffres et du caractère de soulignement (_). De plus, les langages C++ et Python sont sensibles à la casse, c'est-à-dire qu'ils différencient minuscules et majuscules : var, VAR et Var pourraient être les noms de trois variables différentes¹. Cela vous laisse donc au total 63 caractères disponibles à combiner entre eux, la seule contrainte est que le nom doit commencer par une lettre (ou le caractère de soulignement, mais ce n'est pas conseillé, car cela peut avoir une signification particulière dans certains contextes). Usez et abusez des possibilités offertes. Si une paresse légitime fait que les noms des variables vous paraissent bien longs à taper, pensez au temps considérable que vous gagnerez quand vous voudrez reprendre et modifier votre programme dans plusieurs mois, voire plusieurs années. Une paresse efficace doit avoir une vision à long terme !

1. D'autres langages, comme Basic, Pascal ou Ada, ne font pas cette distinction entre minuscules et majuscules. En revanche, certains comme Ada autorisent l'utilisation de pratiquement n'importe quel caractère, même ceux issus d'alphabets non latins.

2.1. Des types des données

Nous n'avons manipulé dans les exemples précédents que des nombres, plus précisément des nombres entiers. Heureusement, les possibilités ne se limitent pas à cela. Voyez par exemple :

Python	C++
<pre>1. # -*- coding: utf8 -*- 2. pi = 3.14159 3. texte = "Pi vaut " 4. print texte, pi</pre>	<pre>1. #include <iostream> 2. #include <string> 3. int main(int argc, char* argv[]) 4. { 5. double pi = 3.14159; 6. std::string texte = "Pi vaut "; 7. std::cout << texte << pi 8. ➡ << std::endl; 9. return 0; 10. }</pre>

Remarquez dans le programme C++, en lignes 5 et 6, la combinaison d'une déclaration d'une variable et d'une affectation.

Dans ces programmes, la variable `pi` est une valeur décimale, un nombre en *virgule flottante*. La variable `texte` contient du texte, ce qu'on désigne par *chaîne de caractères* (imaginez des caractères individuels alignés le long d'une chaîne, afin de former un texte). À l'évidence, ces variables ne sont pas de la même nature que la variable `x` utilisée précédemment.

On ne parle pas ici de nature d'une variable, mais plutôt de son *type*. Le type d'une variable définit la nature de ce qu'on peut y stocker, ce qui détermine directement la façon dont cette variable est codée à l'intérieur de l'ordinateur (qui, rappelons-le, ne sait manipuler que des groupes de 0 et de 1) ainsi que l'espace dont elle a besoin.

Formellement, un type est défini par l'ensemble des valeurs qu'on peut stocker dans les variables de ce type, ainsi que les opérations disponibles entre ces valeurs. Plus simplement, disons que le type d'une variable permet d'attacher à celle-ci une sémantique rudimentaire.

La plupart des langages interprétés, comme Python, sont capables de déterminer par eux-mêmes le type d'une variable selon la valeur qui lui est affectée et selon le contexte, ce type pouvant éventuellement changer par la suite. Par contre, les langages compilés exigent en général que le type soit donné explicitement et définitivement dès la déclaration. Dans les programmes C++ que nous avons vus jusqu'ici, les

éléments `int`, `double` et `std::string` sont des noms de types, désignant respectivement des nombres entiers relatifs, des nombres à virgule flottante et des chaînes de caractères. `int` et `double` sont normalement toujours disponibles, tandis que, pour pouvoir utiliser `std::string`, il est nécessaire d'ajouter une ligne `#include <string>` au début du programme. Mais nous reviendrons plus tard sur cet aspect.

À mesure que nous avancerons dans nos expérimentations, nous découvrirons de nouveaux types et diverses opérations disponibles sur chacun d'eux. Nous verrons également comment créer nos propres types. Mais détaillons un peu les trois grands types précédents.

2.2. Les entiers

Les types entiers permettent de représenter de manière exacte des valeurs entières, comme les dénombrements. Par exemple, vous pouvez calculer ainsi le nombre de combinaisons possibles au Loto national :

Python	C++
<pre>1. # -*- coding: utf8 -*- 2. comb_loto = (49*48*47*46*45*44) / \ 3. (6*5*4*3*2) 4. print comb_loto</pre>	<pre>1. #include <iostream> 2. int main(int argc, char* argv[]) 3. { 4. int comb_loto = (49*48*47*46*45*44) / 5. (6*5*4*3*2); 6. std::cout << comb_loto << std::endl; 7. return 0; 8. }</pre>

Le programme en Python affiche le résultat 13 983 816, tandis que le programme C++ affiche 2 053 351. Lequel a raison, pourquoi l'autre a-t-il tort ?

Cet exemple a deux objectifs. D'abord montrer comment vous pouvez écrire des opérations relativement complexes, éventuellement en utilisant des parenthèses comme vous le feriez sur une feuille de papier. Remarquez, de plus, que l'opération est écrite sur deux lignes, afin de lui donner un aspect plus harmonieux. La présentation du code n'est pas qu'un souci esthétique, cela facilite également la relecture. En Python, il est nécessaire d'ajouter un caractère barre inverse `\` (ou *backslash*) afin d'indiquer que l'instruction se poursuit sur la ligne suivante. Ce n'est pas nécessaire en C++.

Ensuite, ayez toujours un regard critique sur les résultats que vous obtenez de vos programmes : jugez-en la cohérence et la crédibilité. Dans cet exemple, sans pour

autant douter un instant de l'altruisme de la Française des Jeux, on peut considérer que la valeur la plus probable est la plus élevée, c'est-à-dire celle donnée par le programme Python. Mais alors, pourquoi une valeur erronée en C++ ?

Parce que les ordinateurs en général, et les programmes en particulier, présentent une limite à la quantité de chiffres qu'ils peuvent manipuler. Le langage Python, qui est un langage de haut niveau¹, est organisé de telle sorte que cette limite est très élevée, en fait pratiquement celle autorisée par la mémoire dont vous disposez – cela au prix de performances amoindries. En C++, par contre, cette limite est définie par la capacité du processeur, la taille de l'unité de donnée qu'il peut accepter. Elle est donc bien inférieure, mais, en contrepartie, les calculs se font beaucoup plus rapidement.

Le produit intermédiaire $49 \times 48 \times 47 \times 46 \times 45 \times 44$, qui vaut 10 068 347 520, dépasse de loin la limite d'un processeur ou d'un système 32 bits (encore les plus courants aujourd'hui). Ce calcul provoque un *débordement*, c'est-à-dire qu'à une certaine étape le processeur ne peut plus contenir la valeur et "oublie" tout simplement une partie du nombre. Il en résulte une erreur de calcul.

2.3. Les décimaux et la notation scientifique

Les nombres entiers ne suffisent naturellement pas pour représenter le monde qui nous entoure. Par exemple, la surface d'un disque n'a que peu de chances d'être un entier. Aussi utilise-t-on fréquemment des nombres décimaux pour représenter des dimensions physiques :

Python	C++
<pre>1. # -*- coding: utf8 -*- 2. aire_disque = 3.141592654 * \ 3. 2.0 * 2.0 4. print aire_disque</pre>	<pre>1. #include <iostream> 2. int main(int argc, char* argv[]) 3. { 4. double aire_disque = 3.141592654 * 5. 2.0 * 2.0; 6. std::cout << aire_disque 7. << std::endl; 8. return 0; 9. }</pre>

1. Un langage est dit de "haut niveau" s'il permet (ou impose) une certaine distance vis-à-vis du matériel par l'utilisation de concepts plus ou moins sophistiqués. Au contraire, un langage est dit de "bas niveau" s'il est en prise assez directe avec l'électronique de l'ordinateur. Le langage C++ est dans une position intermédiaire, autorisant les deux approches.

Remarquez l'utilisation du point anglo-saxon en lieu et place de notre virgule.

Le programme Python affiche 12.5663706144, tandis que le programme C++ affiche 12.5664. Ces deux valeurs sont proches, mais elles ne sont pas égales... Encore un problème de précision ?

Tel est bien le cas, mais pas uniquement. Pour la même raison que les entiers, l'ordinateur ne peut manipuler qu'un nombre limité de décimales (nombre total de chiffres), environ une douzaine. Les calculs sont donc nécessairement entachés d'une certaine erreur, même minime. À cette approximation "interne" s'ajoute l'approximation de l'affichage : si Python affiche normalement toutes les décimales disponibles, C++ se contente en général de 6 – même si davantage sont disponibles dans la mémoire. Une fois de plus, ne croyez pas aveuglément ce que vous voyez !

Les nombres décimaux permettent en outre de représenter des valeurs plus grandes que les nombres entiers. Essayez, par exemple, de refaire ainsi le calcul du nombre de possibilités au Loto en C++ :

```
double comb_loto = (49.0*48.0*47.0*46.0*45.0*44.0) /  
                    (6.0*5.0*4.0*3.0*2.0);
```

Ce qui donne à l'affichage 1.39838E+07 (ou 1.39838e+07, avec un "e" minuscule, ce qui est identique).

Cette forme d'écriture, un nombre décimal suivi d'un E, d'un signe puis d'un entier, est appelée *notation scientifique*. Elle permet de représenter de grandes valeurs avec peu de chiffres. L'entier relatif qui suit le E est la puissance de 10 par laquelle il faut multiplier le premier nombre pour avoir le résultat. Par exemple, l'écriture 1.234E+24 signifie littéralement "1.234 multiplié par 10²⁴", soit 1 234 000 000 000 000 000 000 000. Mais encore une fois, tous ces chiffres ne sont pas réellement stockés dans l'ordinateur... Essayez d'afficher le résultat du calcul suivant, en Python comme en C++ :

```
deux = 1.0E+50 + 2.0 - 1.0E+50
```

Il suffit de regarder cette instruction une seconde pour s'apercevoir que les deux grands nombres s'annulent : mathématiquement, ce calcul vaut donc exactement 2.

Mais l'ordinateur effectue les calculs opérateur par opérateur : il va donc commencer par calculer $1.0E+50 + 2$. Du fait des limites de précision, la valeur 2 est négligeable par rapport à la valeur 10^{50} : cette somme n'est en réalité même pas effectuée. Ce qui donne 10^{50} , auquel on soustrait 10^{50} ... ce qui donne 0. Mathématiquement faux, mais informatiquement inévitable.

Ces virgules qui flottent

Peut-être vous demandez-vous pourquoi on désigne les nombres décimaux par le terme de *nombres à virgule flottante*. Cela est dû au fait que, dans la représentation de ces nombres, la virgule (qui est représentée par un point, selon la convention anglo-saxonne) n'a pas une position fixe, c'est-à-dire que le nombre de chiffres à gauche et à droite de la virgule peut varier selon la situation.

Il est important de comprendre que le nombre que nous voyons à l'écran n'est qu'une représentation du contenu d'une zone mémoire de l'ordinateur, qui finalement ne sait manipuler que des suites de 0 et de 1, le fameux codage binaire. Nous avons vu que dans le cas du type décimal de Python, environ 16 chiffres sont disponibles. La virgule flottante nous permet donc de contenir en mémoire les nombres 1.123456789012345 et 123456789012345.1 avec la même précision, sans arrondi (ou presque).

Cette représentation s'oppose à celle dite à virgule fixe. Celle-ci impose un nombre fixe de chiffres de part et d'autre de la virgule. Selon ce principe, pour pouvoir représenter avec la même précision les deux nombres précédents, il faudrait pouvoir disposer de 32 chiffres, ce qui impliquerait que la zone mémoire soit deux fois plus vaste.

De nos jours, les processeurs des ordinateurs utilisent (presque) tous la représentation en virgule flottante, car c'est celle qui offre le meilleur compromis entre précision, complexité de traitement et occupation mémoire.

2.4. Les chaînes de caractères

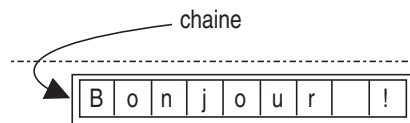
Aussi étonnant que cela puisse paraître, notre électronique actuelle moderne et surpuissante n'a toujours aucune idée de ce qu'est une simple lettre. Encore moins la notion de phrase ou de texte ! Du point de vue de l'ordinateur, ce que nous percevons comme *lettres* n'est en fait que codes numériques et nombres entiers. La correspondance entre une valeur numérique et une lettre précise se fait au travers d'un processus relativement complexe, que nous passerons sous silence pour l'instant.

Une chaîne de caractères n'est ainsi qu'une suite ordonnée de codes numériques, que nous décidons de considérer comme un tout ayant un sens. Considérons les programmes suivants :

Python	C++
<pre>1. # -*- coding: utf8 -*- 2. chaine = "Bonjour !" 3. print chaine</pre>	<pre>1. #include <iostream> 2. #include <string> 3. int main(int argc, char* argv[]) 4. { 5. std::string chaine; 6. chaine = "Bonjour !"; 7. std::cout << chaine << std::endl; 8. return 0; 9. }</pre>

La variable `chaine` (sans accent circonflexe sur le "i", car les noms de variables ne doivent pas contenir de lettre accentuée) est alors un point d'accès à une suite de caractères (représentés par des codes numériques), un peu comme si on avait déclaré autant de variables de type caractère qu'il y en a dans la chaîne. Seulement, cette suite est vue comme un tout, ce que l'on pourrait représenter selon la Figure 2.2.

Figure 2.2
Représentation d'une chaîne de caractères.



Toutefois, cette représentation est approximative : le programme a besoin de "connaître" la longueur de cette chaîne, le nombre de caractères qu'elle contient. Chaque langage de programmation applique sa propre technique pour cela, qui ne présente pas beaucoup d'intérêt pratique pour nous. Disons simplement que dans le schéma précédent, il manque un petit bloc d'informations utilisé pour les besoins propres du programme, afin qu'il puisse manipuler cette chaîne.

Comme pour les types numériques, de nombreuses opérations sont disponibles sur les chaînes de caractères.

Par exemple :

Python

```
# -*- coding: utf8 -*-
bonjour = "Bonjour, "
monde = "Monde !"
print bonjour + monde
tiret = "-"
print 16*tiret
```

C++

```
#include <iostream>
#include <string>
int main(int argc, char* argv[])
{
    std::string bonjour = "Bonjour, ";
    std::string monde = "Monde !";
    std::cout << bonjour+monde << std::endl;
    return 0;
}
```

L'opérateur d'addition + permet de "coller" deux chaînes, l'une au bout de l'autre, pour en créer une nouvelle plus longue. Dans le jargon informatique, on appelle cela la *concaténation* de deux chaînes de caractères.

Plus curieux, l'opérateur de multiplication * permet de dupliquer une chaîne pour en produire une plus longue. Cet opérateur n'est toutefois disponible qu'en Python : c'est pourquoi, ici, le programme C++ n'est pas équivalent. Nous verrons plus loin comment obtenir un résultat similaire.

2.5. Erreurs typiques liées aux variables

Inévitablement, vous commettrez des erreurs lors de la saisie de vos programmes. Ce n'est pas grave, c'est parfaitement naturel, surtout au début. Une erreur commune est d'utiliser une variable avant de l'avoir déclarée :

Python

```
1. # -*- coding: utf8 -*-
2. print une_variable
```

C++

```
1. #include <iostream>
2. int main(int argc, char* argv[])
3. {
4.     std::cout << une_variable;
5. }
```

Le retour à la ligne par `std::endl` a ici été omis, car ne présentant aucun intérêt pour la démonstration.

Cela sera impitoyablement sanctionné par l'interpréteur et par le compilateur par une bordée d'insultes diverses, dans ce style :

Python	C++
Traceback (most recent call last): File "test.py", line 2, in <module> print une_variable NameError: name 'une_variable' is not defined	test.cpp: In function 'int main(int, char**)': test.cpp:4: error: 'une_variable' was ➡ not declared in this scope

Python nous dit "'une_variable' is not defined", C++ plutôt "'une_variable' was not declared". Mais la signification est la même : vous avez tenté d'utiliser une variable sans l'avoir déclarée auparavant. Le message d'erreur contient d'autres informations utiles, comme le fichier dans lequel elle apparaît et la ligne concernée.

Il peut arriver également que l'on tente de combiner deux variables de façon incorrecte, le plus souvent quand elles sont de types différents. Par exemple :

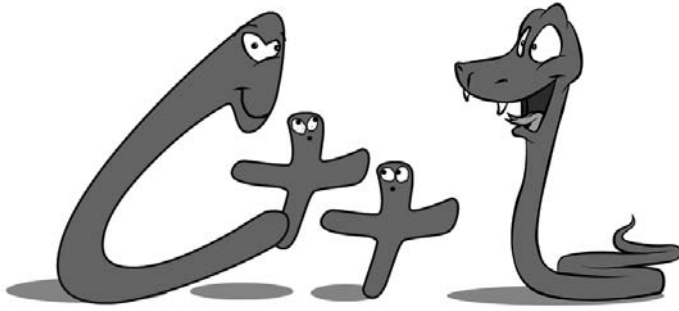
Python	C++
1. # -*- coding: utf8 -*- 2. var_1 = "chaîne" 3. var_2 = 5 4. print var_1 + var_2	1. #include <iostream> 2. int main(int argc, char* argv[]) 3. { 4. std::string var_1 = "chaîne"; 5. int var_2 = 5; 6. std::cout << var_1 + var_2; 7. }

Ces programmes tentent une étrange opération : additionner une chaîne et un entier... À moins qu'il ne s'agisse de concaténation ? Autant additionner des chaussures avec des tomates. Cette opération n'a pas de sens, elle est donc immédiatement sanctionnée :

Python	C++
Traceback (most recent call last): File "test.py", line 4, in <module> print var_1 + var_2 TypeError: cannot concatenate 'str' ➡ and 'int' objects	test.cpp: In function 'int main(int, char**)': test.cpp:6: error: no match for ➡ 'operator+' in 'var_1 + var_2'

À nouveau, interpréteur et compilateur nous donnent la localisation précise de l'erreur. Python nous dit qu'il est impossible de concaténer (*cannot concatenate*) des objets de types chaîne et entier, tandis que C++ nous informe qu'il n'existe pas d'opérateur + valide (*no match for 'operator+'*) pour l'opération demandée.

Aussi agaçants qu'ils puissent être, ces messages d'erreurs ne sont pas là pour vous ennuyer mais pour vous aider à construire un programme correct. Ils ne font que signaler une incohérence, manifestation probable d'un problème dans la conception du programme. Corriger ces erreurs permet non seulement d'obtenir un programme qui fonctionne mais, de plus, augmente les chances que ce programme fonctionne correctement.



Des programmes dans un programme : les fonctions

Au sommaire de ce chapitre :

- Définir une fonction
- Les paramètres
- Les blocs d'instructions
- Retourner une valeur
- Quand créer une fonction ?

Jusqu'à présent, nous avons essentiellement exécuté des instructions à la suite les unes des autres. Mais imaginez une suite d'instructions qui revient sans cesse dans un programme : la retaper chaque fois peut rapidement devenir fastidieux. Prenons un exemple : vous voulez afficher une suite de nombres, un par ligne, chacun étant séparé des autres par une ligne de caractères de soulignement (*underscore* en anglais) et une ligne de tirets. Essayez d'écrire vous-même le programme, par exemple pour afficher les nombres 10, 20, 30 et 40. Le programme pourrait ressembler à ceci :

Python

```
1. # -*- coding: utf8 -*-
2. print 10
3. print "_"*10
4. print "-"*10
5. print 20
6. print "_"*10
7. print "-"*10
8. print 30
9. print "_"*10
10. print "-"*10
11. print 40
```

C++

```
1. #include <iostream>
2. #include <string>
3. int main(int argc, char* argv[])
4. {
5.     std::string soulign = "_____";
6.     std::string tirets = "-----";
7.     std::cout << 10 << std::endl;
8.     std::cout << soulign << std::endl;
9.     std::cout << tirets << std::endl;
10.    std::cout << 20 << std::endl;
11.    std::cout << soulign << std::endl;
12.    std::cout << tirets << std::endl;
13.    std::cout << 30 << std::endl;
14.    std::cout << soulign << std::endl;
15.    std::cout << tirets << std::endl;
16.    std::cout << 40 << std::endl;
17.    return 0;
18. }
```

Pour commencer, essayez d'imaginer ce que fait ce programme. Ensuite, saisissez-le dans votre éditeur de texte et exécutez-le. Vous voyez qu'une paire de lignes revient trois fois. Ce ne sont que deux lignes et que trois fois, mais imaginez si cela représentait cinquante lignes devant être répétées vingt fois !

Répetons-le, les informaticiens sont paresseux, donc une solution a été cherchée pour éviter de répéter ainsi une séquence d'instructions. Dans les "temps anciens", on parlait de *sous-programmes*, c'est-à-dire qu'une suite d'instructions étaient regroupées dans une sorte de miniprogramme, lequel était exécuté à la demande par ce qui devenait alors le *programme principal*. Ceux d'entre vous qui ont pratiqué des langages comme le Basic se rappellent peut-être des instructions comme GOTO ou GOSUB, qui étaient les prémices de la programmation structurée.

Aujourd'hui, on parle plutôt de *fonctions*, parfois de *procédures*. Une fonction est donc une suite d'instructions, qui peut être exécutée à la demande par un programme : tout se passe alors comme si on avait inséré les instructions de la fonction à l'endroit où on l'appelle.

3.1. Définir une fonction

Créer une fonction est très simple, autant en Python qu'en C++. Voyez les deux programmes précédents modifiés :

Python

```
1. # -*- coding: utf8 -*-
2. def Separateur():
3.     print "-"*10
4.     print "-"*10
5.
6. print 10
7. Separateur()
8. print 20
9. Separateur()
10. print 30
11. Separateur()
12. print 40
```

La fonction est ici définie aux lignes 2 à 4. Cette définition est introduite par le mot *def* (pour *define*, définir en anglais) suivi du nom de la fonction, suivi d'une paire de parenthèses et enfin du caractère deux-points. Les deux lignes suivantes constituent le *corps* de la fonction.

C++

```
1. #include <iostream>
2. #include <string>
3. void Separateur()
4. {
5.     std::cout << "          " << std::endl;
6.     std::cout << "-----" << std::endl;
7. }
8.
9. int main(int argc, char* argv[])
10. {
11.     std::cout << 10 << std::endl;
12.     Separateur();
13.     std::cout << 20 << std::endl;
14.     Separateur();
15.     std::cout << 30 << std::endl;
16.     Separateur();
17.     std::cout << 40 << std::endl;
18.     return 0;
19. }
```

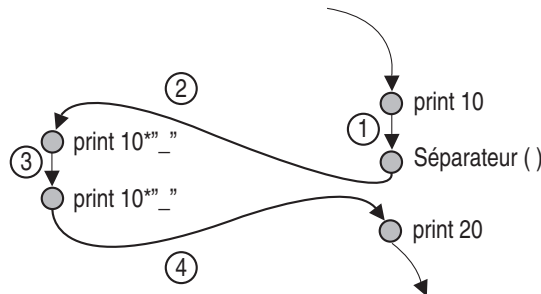
La fonction est ici définie aux lignes 3 à 7. Cette définition commence par donner le type des valeurs éventuellement fournies par la fonction (ici, *void* signifie "rien"), suivi de son nom, suivi d'une paire de parenthèses. Le *corps* de la fonction est encadré d'une paire d'accolades.

Nous définissons ici une fonction nommée `Séparateur()`. Cette fonction a pour rôle d'afficher la double ligne de séparation voulue. On désigne par *corps de la fonction* la suite d'instructions qu'elle doit exécuter. Remarquez la présentation de ces instructions : elles sont décalées vers la droite, simplement par l'ajout d'un certain nombre d'espaces en début de chaque ligne. Ce décalage vers la droite est appelé *indentation*. Cette indentation est purement décorative en C++ ; elle ne vise qu'à améliorer la présentation du code et donc à faciliter sa relecture. Par contre, elle est absolument essentielle en Python : c'est précisément cette indentation des lignes qui suivent celle commençant par `def` qui indique quelles sont les instructions "appartenant" à la fonction.

Dans les deux cas, la ligne vide qui suit la fonction (ligne 5 en Python et ligne 8 en C++) n'a qu'un but esthétique, afin de bien séparer visuellement la fonction du reste du programme.

La suite, justement, vous montre comment *utiliser* cette fonction : lignes 7, 9 et 11 en Python, lignes 12, 14 et 16 en C++. Il suffit de donner son nom, suivi d'une paire de parenthèses. Tout se passe alors comme si les instructions contenues dans la fonction étaient placées à l'endroit où son nom apparaît. On peut schématiser le déroulement des opérations (en se basant sur le code Python, mais c'est exactement la même chose en C++) comme illustré à la Figure 3.1.

Figure 3.1
Chemin des instructions
lors de l'appel
d'une fonction.



Les opérations se déroulent ainsi :

1. L'instruction `print 10` est exécutée normalement.
2. On rencontre l'instruction `Séparateur()` : Python (ou C++) reconnaît qu'il s'agit d'une fonction précédemment définie ; le programme est alors *dérouté* vers les instructions de la fonction.

3. Les instructions contenues dans la fonction sont exécutées.
4. À l'issue de l'exécution de la fonction, on retourne "dans" le programme qui reprend son cours normal.

La deuxième étape est désignée par le terme d'*appel de fonction*. L'ensemble de ce mécanisme, de la définition à l'utilisation de fonctions, est à la base de ce que l'on appelle la *programmation structurée*.

Mais regardez d'un peu plus près le programme en C++, en particulier la ligne 9 :

```
int main(int argc, char* argv[])
```

On retrouve les mêmes éléments que pour la déclaration de la fonction `Separateur()` : d'abord un type (ici le type entier `int`), puis un nom (ici `main`, que l'on peut traduire par *principale*), puis une paire de parenthèses – sauf qu'ici des choses figurent à l'intérieur. Il s'agit bel et bien d'une fonction : tout programme en C++ (ou en C) doit ainsi déclarer une fonction nommée `main()`, qui est en fait la fonction exécutée au moment où le programme est lancé. Mais que sont ces éléments entre les parenthèses ?

3.2. Les paramètres

On ne le répétera jamais assez, un programme évolue au cours du temps, au fur et à mesure que les besoins changent. Par exemple, on pourrait estimer qu'une largeur de 10 caractères pour les lignes séparatrices est insuffisante ; 20 serait préférable. Dans le premier programme, cela nécessite de changer six valeurs, alors qu'il suffit d'en changer deux dans le second. Nous avons donc gagné en facilité de maintenance et d'évolution. Toutefois, ce n'est pas encore satisfaisant.

Il est prévisible que, plus tard, nous voudrions encore changer la largeur de nos lignes de tirets. Mieux : sans doute voudrions-nous un jour pouvoir afficher des lignes de différentes longueurs ! Par exemple, une ligne de 10 tirets, puis une ligne de 20, puis une ligne de 30... On pourrait imaginer créer une fonction pour chaque largeur de ligne, mais on retomberait alors dans les mêmes problèmes que ceux du premier programme. Ce qu'il nous faut, c'est une fonction plus *générique*, c'est-à-dire qui puisse adapter son action selon la situation. Pour réaliser cela, nous allons utiliser des *paramètres*.

Modifions légèrement les programmes précédents :

Python	C++
<pre> 1. # -*- coding: utf8 -*- 2. def Separateur(largeur): 3. print largeur*'_' 4. print largeur*'-' 5. 6. print 10 7. Separateur(10) 8. print 20 9. Separateur(20) 10. print 30 11. Separateur(30) 12. print 40 </pre>	<pre> 1. #include <iostream> 2. #include <string> 3. void Separateur(int largeur) 4. { 5. std::cout << std::string ↳(largeur, '_') 6. << std::endl; 7. std::cout << std::string ↳(largeur, '-') 8. << std::endl; 9. } 10. int main(int argc, char* argv[]) 11. { 12. std::cout << 10 << std::endl; 13. Separateur(10); 14. std::cout << 20 << std::endl; 15. Separateur(20); 16. std::cout << 30 << std::endl; 17. Separateur(30); 18. std::cout << 40 << std::endl; 19. return 0; 20. } </pre>

Nous avons ajouté un élément, nommé `largeur`, entre les parenthèses de la définition de la fonction `Separateur()`. Il s'agit d'un paramètre donné à la fonction, au sein de laquelle il se comporte comme une variable, dont la valeur n'est pas connue *a priori*. Ce paramètre détermine le nombre de caractères qui seront affichés sur chaque ligne. À ce sujet, notez qu'on place habituellement un caractère unique entre apostrophes plutôt qu'entre guillemets. Notez également que l'on construit une chaîne de caractères en C++ comme la répétition d'un unique caractère (lignes 5 et 7) : on retrouve le nom du type `std::string`, puis on donne entre parenthèses le nombre de répétitions et le caractère à répéter.

Lorsque la fonction est utilisée (lignes 7, 9 et 11 en Python, lignes 13, 15 et 17 en C++), on place désormais une valeur entre les parenthèses de l'appel de fonction. Cette valeur est transmise à la fonction par l'intermédiaire du paramètre `largeur` : tout se passe alors comme si `largeur` était une variable contenant la valeur indiquée au moment de l'appel de fonction. Au premier appel, `largeur` vaut 10, au deuxième, elle vaut 20, et 30 au troisième appel.

3.3. Les blocs d'instructions

Les instructions contenues dans une fonction forment un tout : elles vont ensemble. On dit qu'elles constituent un *bloc d'instructions*. Plus généralement, un bloc est un ensemble d'instructions qui se distinguent du programme. Cette notion est fondamentale car en découle (en partie) la notion de *portée* d'une variable : toute variable n'existe qu'à l'intérieur du bloc dans lequel elle a été déclarée.

Python

Un bloc est défini par l'indentation des lignes qui le composent. Une suite de lignes indentées d'un même nombre de caractères constitue donc un bloc.

Par exemple :

```
1. # -*- coding: utf8 -*-
2. print "Début"
3.     entier = 1
4.     print entier
5. print entier
```

Les lignes 3 et 4 constituent un bloc distinct du reste du programme.

C++

Un bloc est délimité par une paire d'accolades.

Par exemple :

```
1. #include <iostream>
2. int main(int argc, char* argv[])
3. {
4.     std::cout << "Début";
5.     {
6.         int entier = 1;
7.         std::cout << entier;
8.     }
9.     std::cout << entier;
10. }
```

Les lignes 6 et 7 constituent un bloc distinct du reste du programme.

Dans les deux programmes précédents, une variable nommée `entier` est déclarée à l'intérieur d'un bloc. Cette variable n'a alors d'existence qu'à l'intérieur de ce bloc : en dehors, elle n'existe pas. Aussi, la ligne 5 du programme Python et la ligne 9 du programme C++ provoqueront-elles une erreur, car à cet endroit on se situe en dehors du bloc d'instructions précédent : la variable `entier` est donc inconnue à ce point précis.

Les blocs sont essentiellement utilisés pour rassembler des instructions au sein d'une fonction. Une utilisation comme celle montrée précédemment est plus rare, mais présente parfois un intérêt quand on souhaite faire "disparaître" des variables dont on n'a plus besoin. Incidemment, cela montre que vous pouvez parfaitement déclarer des variables à l'intérieur d'une fonction.

3.4. Retourner une valeur

Il nous reste un dernier pas à faire sur le chemin de la généralité (enfin, pour l'instant). Notre fonction effectue certains calculs, dans la mesure où on considère que la construction d'une chaîne de caractères est un calcul, puis affiche le résultat de ces calculs. C'est justement là le dernier problème qui reste à résoudre : cette fonction *ne devrait pas* afficher quoi que ce soit.

La possibilité de définir des fonctions, ou d'autres types d'entités que nous rencontrerons plus tard, participe du grand principe de la *programmation structurée*. C'est-à-dire qu'on s'efforce d'organiser les programmes selon une méthode cartésienne, consistant à diviser les gros problèmes en plus petits. On constitue ainsi des éléments de programme, qui communiquent et travaillent ensemble, tout en évitant dans la mesure du possible que ces éléments dépendent trop les uns des autres. Tous les langages de programmation modernes offrent des techniques pour atteindre cet objectif, les fonctions en étant une parmi d'autres. Le but est d'éviter le fouillis que présentaient d'anciens programmes écrits, par exemple, en langage Basic.

De l'expérience accumulée depuis les origines de la programmation, deux grands ensembles se sont dégagés, chacun étant présent dans chaque programme :

- **L'interface.** Partie du programme responsable de l'interaction avec l'utilisateur, c'est-à-dire de ce que doit afficher le programme et les moyens par lesquels l'utilisateur peut donner des informations au programme ; on utilise parfois le terme savant d'IHM (*interface homme-machine*) ou l'anglicisme *look and feel* (apparence et manière d'être).
- **Le noyau.** Cœur du programme, contenant ce qu'on pourrait appeler (avec les précautions d'usage) son intelligence : la partie du programme qui effectue des calculs pour produire des résultats – certains devant être affichés, d'autres pas.

L'expérience a montré qu'il était vivement recommandé de séparer nettement ces deux grandes parties. La structure du programme n'en est que plus claire et aisée à appréhender ; ainsi, la modification d'une partie risque moins d'entraîner des modifications importantes dans l'autre partie. Ne pas respecter ce principe élémentaire débouche inévitablement, tôt ou tard, sur un effort considérable si le programme doit évoluer : si tout est trop mélangé, un changement d'un côté nécessitera un changement de l'autre, qui à son tour risquera d'entraîner un changement dans la première partie... Cela devient rapidement infernal.

Notre exemple, si trivial soit-il, n'échappe pas à cette règle. Nous avons une partie de programme qui effectue un calcul, à savoir construire une chaîne de caractères.

À strictement parler, la fonction qui effectue ce calcul n'a pas à "savoir" ce que l'on va faire de cette chaîne de caractères. Pour l'instant, la fonction provoque un affichage. Mais comment faire si, demain, au lieu d'afficher cette chaîne nous voulons l'écrire dans un fichier ou la combiner de manière complexe avec d'autres chaînes ? En l'état actuel, une telle évolution du programme nécessiterait des modifications profondes s'étendant sur tout le code.

L'idée consiste donc à définir des fonctions qui font ce pour quoi elles ont été créées, mais pas plus. Notre fonction calcule une chaîne de caractères, d'accord. Mais ce n'est pas son rôle de l'afficher. Ce rôle est dévolu au programme principal. Il faut donc un moyen pour "récupérer" ce que la fonction aura produit. On dit alors que la fonction va *retourner un résultat*, ce résultat étant la *valeur de retour* de la fonction. Ce qui sera fait de ce résultat est de la responsabilité de celui qui appelle la fonction, ici, le programme principal. Voyez la nouvelle – et dernière – version de notre programme :

Python

```
1. # -*- coding: utf8 -*-
2. def Separateur(largeur):
3.     chaîne = largeur*'_' + "\n" + \
4.         largeur*'- ' + "\n"
5.     return chaîne
6.
7. print 10
8. print Separateur(10),
9. print 20
10. print Separateur(20),
11. print 30
12. print Separateur(30),
13. print 40
```

La virgule à la fin des instructions print des lignes 8, 10 et 12 demande à Python de ne pas sauter de ligne après l'affichage. En effet, le saut de ligne est déjà contenu dans la chaîne produite par `Separateur()`.

C++

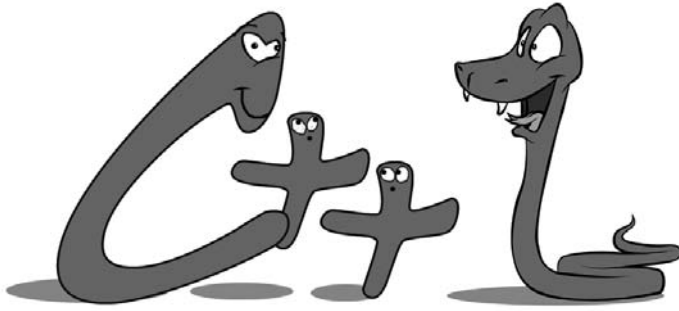
```
1. #include <iostream>
2. #include <string>
3. std::string Separateur(int largeur)
4. {
5.     std::string chaîne =
6.         std::string(largeur, '_') +
7.         "\n" +
8.         std::string(largeur, '- ') +
9.         "\n";
10.    return chaîne;
11. }
12. int main(int argc, char* argv[])
13. {
14.     std::cout << 10 << std::endl
15.               << Separateur(10)
16.               << 20 << std::endl
17.               << Separateur(20)
18.               << 30 << std::endl
19.               << Separateur(30)
20.               << 40 << std::endl;
21.    return 0;
22. }
```

Désormais, la fonction `Separateur()` construit une chaîne de caractères et la "retourne". Dans les deux langages, le retour d'une valeur est effectué par l'instruction `return` (ligne 5 en Python, ligne 10 en C++). Invoquer `return` provoque la sortie immédiate de la fonction, même si des instructions se trouvent après. Par ailleurs, notez comment la déclaration de la fonction a dû être changée dans le programme C++ : `void` est devenu `std::string`, signalant ainsi littéralement que la fonction ne retourne pas *rien*, mais une valeur de type `std::string`. Une fonction définie ainsi peut être rapprochée de la notion mathématique éponyme.

Enfin, le lecteur attentif aura probablement remarqué au sein de la fonction `Separateur()` la présence de la chaîne `"\n"`, concaténée aux suites de tirets. Il s'agit en réalité d'une chaîne d'un unique caractère, représenté par le symbole `\n` : celui-ci correspond tout simplement au saut de ligne. Lorsque ce caractère est affiché, les affichages suivants commencent au début de la ligne suivante.

3.5. Quand créer une fonction ?

C'est sans doute l'une des premières questions que vous vous poserez lors de vos expérimentations personnelles. Il n'y a pas vraiment de réponse absolue. À mesure que vous gagnerez en expérience, l'intérêt des fonctions vous apparaîtra plus clairement. Pour vous donner une idée, quelqu'un a dit un jour : *"Si ça fait plus de trois lignes et que je m'en serve plus de trois fois, alors je fais une fonction."* C'est un peu simpliste, mais vous voyez l'idée. D'une manière générale, on fait une fonction lorsqu'une séquence d'instructions a toutes les chances d'être utilisée de nombreuses fois, quitte à remplacer quelques valeurs par des paramètres.



L'interactivité : échanger avec l'utilisateur

Au sommaire de ce chapitre :

- Salutations !
- Obtenir un nombre entier
- Obtenir une valeur décimale
- Saisir plusieurs valeurs

Afficher des résultats est une fonction importante de l'immense majorité des programmes. Encore faut-il les présenter correctement, d'une manière aisément compréhensible par l'utilisateur... Lequel utilisateur doit généralement entrer des informations pour que le programme puisse calculer ces résultats. Cet échange d'informations entre un logiciel et celui qui l'utilise est généralement désigné par le terme d'*interactivité* : le programme et l'utilisateur agissent l'un sur l'autre, par l'intermédiaire de l'écran (ou tout autre dispositif d'affichage, comme une imprimante) et le clavier (ou tout autre dispositif d'entrée, comme la souris). Voyons ensemble comment mettre en œuvre cette interactivité avec quelques exemples simples.

4.1. Salutations !

Notre premier programme saluait le monde. Voici comment écrire un programme qui demande son nom à l'utilisateur, avant de le saluer :

Python

```
1. # -*- coding: utf8 -*-
2. print "Quel est votre nom ?",
3. nom = raw_input()
4. print "Bonjour,", nom, "!"
```

La virgule dans l'instruction `print` insère un espace à l'affichage et empêche le retour à la ligne : c'est à cette fin qu'elle est utilisée ligne 2.

C++

```
1. #include <iostream>
2. #include <string>
3. int main(int argc, char* argv[])
4. {
5.     std::cout << "Quel est votre nom ? ";
6.     std::string nom;
7.     std::cin >> nom;
8.     std::cout << "Bonjour, " << nom
9.         << " !" << std::endl;
10.    return 0;
11. }
```

Essayez de deviner par vous-même ce qui se passe dans ces programmes, en usant de votre imagination pour déterminer ce que font les éléments encore inconnus.

Le principe est extrêmement simple : on crée une variable, nom, destinée à recevoir une chaîne de caractères contenant justement le nom de l'utilisateur. Ce nom est demandé, récupéré à partir du clavier, puis stocké dans la variable. Laquelle est enfin utilisée pour saluer l'utilisateur.

La partie "récupération à partir du clavier" est effectuée ligne 3 en Python, ligne 7 en C++.

Python

On utilise l'instruction `raw_input()`, qui est, en fait, une fonction retournant une chaîne de caractères. Lorsque cette fonction est invoquée, le programme est comme "bloqué" : il attend que l'utilisateur appuie sur la touche Entrée. Dès que cette action survient, la fonction retourne les caractères qui ont été saisis auparavant.

C++

On utilise l'objet `std::cin`, qui est le miroir de l'objet `std::cout` que vous connaissez déjà. Remarquez d'ailleurs que les chevrons `>>` sont en sens inverse, comme si des caractères "sortaient" de `std::cin` pour aller dans la variable. À cet endroit, le programme est en attente de l'appui de la touche Entrée, après quoi les caractères saisis auparavant sont transmis par `std::cin`.

Comme vous le voyez, il n'y a là rien de bien compliqué (pour l'instant...). Comme toujours, la différence essentielle entre les deux programmes est qu'en C++, la variable nom doit être déclarée avant de pouvoir y placer quoi que ce soit.

Sans doute serez-vous amené fréquemment à parler d'*entrées-sorties* d'un programme. Par convention, le point de référence est le programme lui-même : des informations *entrent* dans le programme ou *sortent* du programme. La distinction est particulièrement visible en C++ : si vous considérez les objets `std::cin` et `std::cout` comme des canaux de communication avec le monde extérieur au programme, les chevrons indiquent dans quel sens les informations transitent.

4.2. Obtenir un nombre entier

Nous venons de voir comment obtenir une chaîne de caractères de l'utilisateur. Mais qu'en est-il lorsque nous voulons obtenir un nombre entier ? Par exemple, étant donné un jeu de cartes et un nombre de joueurs, comment savoir combien de cartes aura chaque joueur, et éventuellement combien il en restera ? Une solution possible est la suivante :

Python

```
1. # -*- coding: utf8 -*-
2. print "Combien de cartes ?",
3. nb_cartes = int(raw_input())
4. print "Combien de joueurs ?",
5. nb_joueurs = int(raw_input())
6. print "Chaque joueur aura", \
7.     nb_cartes / nb_joueurs, \
8.     "cartes, il en restera", \
9.     nb_cartes % nb_joueurs
```

C++

```
1. #include <iostream>
2. int main(int argc, char* argv[])
3. {
4.     int nb_cartes = 0;
5.     int nb_joueurs = 1;
6.     std::cout << "Combien de cartes ? ";
7.     std::cin >> nb_cartes;
8.     std::cout << "Combien de joueurs ? ";
9.     std::cin >> nb_joueurs;
10.    std::cout << "Chaque joueur aura "
11.              << nb_cartes / nb_joueurs
12.              << " cartes, il en restera "
13.              << nb_cartes % nb_joueurs
14.              << std::endl;
15.    return 0;
16. }
```

Cet exemple met en lumière une nouvelle différence entre Python et C++. Comparons deux extraits de code faisant exactement la même chose, à savoir obtenir un entier de l'utilisateur :

Python

```
nb_cartes = int(raw_input())
```

C++

```
int nb_cartes;
std::cin >> nb_cartes;
```

Le code en C++ est très proche de celui que nous avons vu pour la chaîne de caractères. Par contre, le code Python "enferme" la fonction `raw_input()` dans ce qui pourrait être une fonction nommée `int()`.

Rappelez-vous, nous avons dit que, d'une part, Python détermine le type d'une variable à partir du contexte et de sa valeur, et d'autre part, que `raw_input()` retourne une chaîne de caractères. Or, ici, nous voulons des nombres entiers afin d'effectuer certains calculs. Contrairement au C++, où toute variable doit être déclarée avec un type clairement spécifié, Python n'a aucun moyen de "deviner" que nous voulons un nombre entier. Le rôle de cette pseudo-fonction `int()` est précisément de préciser le type recherché : à partir de la chaîne de caractères renvoyée par `raw_input()`, on "fabrique" une valeur numérique, laquelle sera stockée dans la variable – qui sera donc, par conséquent, de type numérique et non pas de type chaîne de caractères.

Du côté du C++, la fabrication de la valeur numérique à partir des caractères donnés par l'utilisateur est effectuée automatiquement par l'objet `std::cin` lui-même, celui-ci ayant reconnu que la variable dans laquelle il doit placer ses informations est de type numérique.

Nous sommes donc en présence d'une *conversion de type* : à partir d'une valeur d'un type donné (une chaîne de caractères), on fabrique une valeur d'un autre type (une valeur numérique entière). Dans le verbiage de la programmation, on parle de *transy-page* (type *cast*, ou simplement *cast*, en anglais).

Dernier mot, remarquez l'utilisation de l'opérateur `%` : celui-ci donne le reste d'une division entre nombres entiers, en Python comme en C++.

4.3. Obtenir une valeur décimale

Pour être complet, voyons maintenant un exemple utilisant des nombres à virgule flottante, par exemple pour calculer la vitesse moyenne d'un trajet :

Python

```
1. # -*- coding: utf8 -*-
2. print "Distance (en km) ?",
3. distance = float(raw_input())
4. print "Durée (en h) ?",
5. duree = float(raw_input())
6. print "Vitesse :", distance/duree
```

C++

```
1. #include <iostream>
2. int main(int argc, char* argv[])
3. {
4.     double distance = 0.0;
5.     double duree = 1.0;
6.     std::cout << "Distance (en km) ? ";
7.     std::cin >> distance;
8.     std::cout << "Durée (en h) ? ";
9.     std::cin >> duree;
10.    std::cout << "Vitesse : "
11.               << distance / duree
12.               << std::endl;
13.    return 0;
14. }
```

Nous retrouvons un transtypage en Python, cette fois-ci en utilisant la pseudo-fonction `float()`. En dehors de cela, ces exemples sont parfaitement équivalents aux précédents !

4.4. Saisir plusieurs valeurs

Si vous utilisez la version C++ du programme de salutation vu précédemment en lui donnant un nom complet (prénom suivi du nom), vous remarquerez sans doute un problème qui n'apparaît pas en Python. Comparez les Figures 4.1 et 4.2.

Python



Figure 4.1

Salutation en Python, avec nom et prénom.

C++

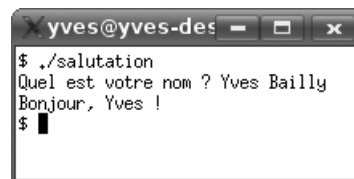


Figure 4.2

Salutation en C++, avec nom et prénom.

Le programme C++ ne "récupère" qu'une partie de ce qui a été saisi, en fait jusqu'à l'espace séparant les deux mots. Dans certaines situations, cela peut être ennuyeux. Au contraire, le programme Python récupère bien les deux mots en une seule chaîne de caractères... Mais alors, comment faire si vous voulez effectivement les séparer pour les stocker dans deux variables ?

4.4.1. Le cas du C++

Pour récupérer une ligne complète sous la forme d'une chaîne de caractères, il est nécessaire d'utiliser une fonction spécialement dédiée à cela. Il suffit de modifier ainsi la ligne 7 du programme :

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

La fonction standard `std::getline()` reçoit deux paramètres :

- d'abord un flux de caractères, c'est-à-dire un objet à partir duquel obtenir des caractères, ce qu'est précisément `std::cin` ;
- ensuite, une variable de type chaîne de caractères, dans laquelle placer les caractères obtenus à partir du flux précédent.

Tous les caractères du flux sont placés dans la chaîne, jusqu'à ce que `std::getline()` rencontre un saut de ligne : il s'agit d'un caractère spécial "créé" par l'appui de la touche Entrée. Autrement dit, `std::getline()` place dans une chaîne tous les caractères se trouvant sur une ligne. Ainsi modifié, le programme C++ se comporte comme le programme Python.

Si, au contraire, vous voulez effectivement obtenir les mots de la ligne dans différentes variables, vous devez déclarer autant de variables et les donner comme "cible" à `std::cin`. Par exemple, pour lire en une seule fois un prénom, un nom et un âge (donc une valeur numérique) :

```
std::string prenom;  
std::string nom;  
int age;  
std::cin >> prenom >> nom >> age;
```

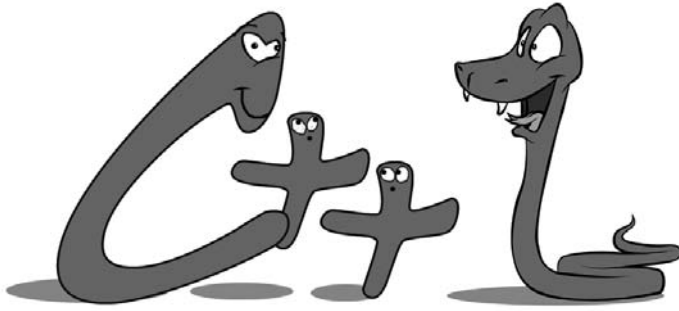
L'utilisateur peut alors saisir un texte comme "Yves Bailly 34" (sans les guillemets !), les trois éléments seront bien répartis dans les trois variables.

4.4.2. Le cas de Python

La fonction Python `raw_input()` est l'équivalent de la fonction C++ `std::getline()` que nous venons de voir. Pour obtenir les différents éléments d'une chaîne entrée par l'utilisateur, il est nécessaire de *découper* (*split* en anglais) celle-ci, puis d'affecter chaque morceau aux variables concernées. Le code qui suit anticipe un peu sur l'avenir, nommément les Chapitres 6 et 9, mais vous devriez pouvoir l'appréhender sans difficulté :

```
chaîne = raw_input()
morceaux = chaîne.split()
prenom = morceaux[0]
nom = morceaux[1]
age = int(morceaux[2])
```

Le découpage de la chaîne est réalisé en lui appliquant une *opération* nommée `split()`. Sans entrer pour l'instant dans les détails, une opération est un peu comme une fonction qui s'applique sur un objet dont le nom la précède, séparé par un point (cela sera expliqué au Chapitre 9). Ici, on applique donc l'opération `split()` à l'objet `chaîne`. Le résultat est le morcellement de la chaîne, les morceaux étant stockés dans une variable nommée `morceaux` (qui est, en fait, un tableau, mais nous reverrons cela au Chapitre 6). Chaque morceau de la chaîne, lui-même une chaîne de caractères, est finalement obtenu en donnant le nom du morcellement (ici `morceaux`) suivi du numéro du morceau entre crochets, le premier ayant le numéro 0 (zéro), le deuxième le numéro 1, et ainsi de suite. On retrouve d'ailleurs le transtypage nécessaire pour obtenir une valeur de type numérique. Cet extrait de code Python est ainsi équivalent à l'extrait de code C++ que nous venons de voir.



5

Ronds-points et embranchements : boucles et alternatives

Au sommaire de ce chapitre :

- L'alternative
- La boucle
- Algèbre de Boole et algorithmique

Les programmes que nous avons vus jusqu'ici sont essentiellement linéaires, c'est-à-dire que les instructions sont exécutées l'une après l'autre, dans l'ordre dans lequel elles apparaissent. Cela présente le mérite de la simplicité mais, également, l'inconvénient d'être trop sommaire.

Dans votre vie quotidienne, vous êtes sans cesse amené à faire des choix guidés par votre environnement. Par exemple, avant de partir en balade, sans doute vous posez-vous la question en regardant le ciel : "Pleut-il ?" Si oui, vous mettez des bottes, si non, vous mettez des chaussures. De même, il arrive que l'on doive effectuer des tâches caractérisées par la répétition d'une ou de plusieurs actions, de la laborieuse vérification d'un relevé de compte bancaire à l'épluchage d'un kilo de pommes de terre. Pour triviales qu'elles puissent paraître, ces situations se retrouvent également en programmation. Afin d'illustrer cela, nous allons réaliser le petit jeu suivant : le programme va choisir un nombre au hasard entre 1 et 100, que l'utilisateur devra deviner en faisant des propositions. À chaque proposition, le programme affichera "trop grand" ou "trop petit", selon le cas.

Le début de ce programme pourrait ressembler à ceci :

Python

```
1. # -*- coding: utf8 -*-
2. from random import randint
3. nombre_a_trouver = randint(1, 100)
4. print "Donner un nombre :"
5. chaine = raw_input()
6. nombre_joueur = int(chaine)
```

La fonction `randint()` (ligne 3) donne un nombre entier au hasard compris entre les deux valeurs qu'elle reçoit en paramètre. Cette fonction est rendue disponible par la ligne 2, dont la signification sera explicitée au chapitre consacré aux bibliothèques.

C++

```
1. #include <iostream>
2. #include <cstdlib>
3. int main(int argc, char* argv[])
4. {
5.     srand((long)argv);
6.     int nombre_a_trouver =
7.         (random()%100) + 1;
8.     int nombre_joueur;
9.     std::cout << "Donnez un nombre : ";
10.    std::cin >> nombre_joueur;
```

La fonction `random()` (ligne 7) donne un nombre entier au hasard compris entre 0 et une valeur maximale nommée `RAND_MAX`. Cette fonction est rendue disponible par la ligne 2.

Nous anticipons un peu sur l'avenir pour bénéficier de la fonction `randint()` en Python et de la fonction `random()` en C++. Remarquez que ces fonctions ne sont pas

exactement équivalentes : si la première permet de spécifier l'intervalle dans lequel on souhaite obtenir un nombre au hasard, la seconde donne toujours un nombre entre 0 et `RAND_MAX` – dont la valeur dépend du compilateur utilisé et du système. On doit donc ajuster ce que nous donne `random()`.

Pour cela, on commence par demander le reste de la division du nombre donné par `random()` par le nombre le plus grand qu'on souhaite obtenir, ici 100. Le reste d'une division entre nombres entiers est donné par l'opérateur `%` (en C++ comme en Python). Ce reste est forcément compris entre 0 et 99. On ajoute donc 1 pour obtenir effectivement un nombre entre 1 et 100. Ainsi s'explique la formule ligne 7.

Il convient de préciser ici que le hasard n'existe pas en programmation. Si vous appelez plusieurs fois `random()`, vous obtiendrez une suite de nombres qui correspond à l'application d'une formule complexe, conçue de façon à *simuler* le hasard. Cette formule utilise ce que l'on appelle une *graine*, à partir de laquelle la suite de nombres est calculée. Sans autre précision, cette graine est toujours 1, ce qui signifie que la suite de nombres générée par `random()` est toujours la même. La ligne 5 du programme C++ a pour effet de modifier cette graine, au moyen de la fonction `srandom()`. Sans entrer dans les détails, le procédé utilisé ici exploite la localisation en mémoire du programme lors de son exécution, laquelle localisation est très certainement différente à chaque exécution. On obtient ainsi une nouvelle suite de valeurs au hasard à chaque exécution. Remarquez que ce n'est pas nécessaire en Python, la graine étant automatiquement modifiée à chaque exécution.

Le hasard en programmation, que l'on peut rapprocher de la notion de *variable aléatoire* en probabilités, est un sujet bien plus complexe qu'il n'y paraît. Ce que nous venons de voir devrait toutefois répondre à la plupart de vos besoins. Mais revenons au sujet principal de ce chapitre.

5.1. L'alternative

Une alternative est une situation où l'on a le choix entre deux possibilités, deux chemins. Chacune des possibilités est parfois désignée par le terme de *branche* de l'alternative. Plaçons-nous dans la situation où l'utilisateur a saisi un nombre : il nous faut déterminer le message à afficher. Suivez bien, le procédé évoqué maintenant est l'un des plus fondamentaux pour concevoir une partie de programme, voire un programme complet.

Mettez-vous à la place du programme, là où nous en sommes, et agissez mentalement à sa place. Vous disposez d'un nombre de référence, celui tiré au hasard, ainsi que d'un nombre donné par l'utilisateur : ce sont les *données d'entrée*, celles que vous devez traiter. Par ce traitement, vous devez produire un résultat qui est un message adapté à la comparaison de ces deux nombres. Décomposez autant que possible votre processus de pensée, en oubliant pour l'instant le cas particulier où les deux nombres sont égaux. Cela devrait ressembler à ceci :

si le nombre donné est plus grand que le nombre à trouver, alors le message est "Trop grand", sinon le message est "Trop petit".

Cette simple phrase, parfois désignée par le terme de *principe*, est une description synthétique et concise de ce que l'on veut faire. Maintenant, rapprochons-nous de la programmation, en remplaçant certains termes par les variables qui les représentent :

si nombre_joueur est plus grand que nombre_a_trouver, alors etc.

La notion de message, dans notre cas, correspond à un affichage à l'écran. Par ailleurs, la comparaison de deux nombres se fait en utilisant les opérateurs mathématiques usuels que sont les symboles < et >. Poursuivons le "codage" de notre principe, par exemple en utilisant le langage Python :

si nombre_joueur > nombre_a_trouver, alors print "Trop grand", sinon print "Trop petit".

Voilà qui commence à ressembler à un programme ! Il nous reste à traduire les mots "si", "alors" et "sinon". Le mot "si" se traduit en anglais, ainsi que dans la plupart des langages de programmation, par `if`. Le mot "alors" se traduit par `then`, mais de nombreux langages, dont Python et C++, omettent simplement ce mot. Enfin, "sinon" se traduit par `else`. Ce qui nous donne les instructions suivantes :

Python

```
1. if ( nombre_joueur > \
2.     nombre_a_trouver ) :
3.     print "Trop grand !"
4. else:
5.     print "Trop petit !"
```

C++

```
1.  if ( nombre_joueur >
2.      nombre_a_trouver )
3.  {
4.      std::cout << "Trop grand !"
5.                << std::endl;
6.  }
```



Python

Remarquez la fin des lignes 2 et 4, vous retrouvez les deux-points que nous avons rencontrés pour la définition d'une fonction. Les lignes 3 et 5 sont indentées : de même que les instructions d'une fonction sont rassemblées dans un bloc d'instructions, les instructions correspondant à chacune des branches de l'alternative sont rassemblées dans un bloc. Ici chacun des deux blocs n'est constitué que d'une seule ligne, mais naturellement vous êtes libre d'en rajouter.

Rappelez-vous qu'en Python l'indentation est très importante : c'est ce décalage vers la droite, obtenu en insérant des espaces en début de ligne, qui détermine ce qui fait partie de tel ou tel bloc.

C++

```
7.  else
8.  {
9.      std::cout << "Trop petit !"
10.         << std::endl;
11. }
```

On retrouve ici les accolades que nous avons rencontrées lors de la définition d'une fonction. En effet, les instructions qui doivent être exécutées dans chacune des branches de l'alternative sont rassemblées dans un bloc, délimité par ces accolades. Ici, nous n'avons qu'une seule instruction pour chacun de ces blocs : dans ce cas particulier, les accolades sont facultatives. Mais il est vivement recommandé de toujours les utiliser, afin d'éviter toute ambiguïté.

L'alternative est introduite par la première ligne de ces deux extraits :

```
if ( nombre_joueur > nombre_a_trouver ) ...
```

Ce qui se trouve entre les parenthèses constitue la *condition*, ou le *test*, qui permet de déterminer quelle branche de l'alternative sera exécutée. Si le test réussit, c'est-à-dire si la condition est *vraie*, alors on exécute la première branche, celle qui se trouve juste après le `if`. Sinon, on exécute l'autre branche. Une fois retraduit en français, tout cela est en fait très naturel ! Ne négligez pas l'importance de la formulation et de la sémantique en langage naturel, elle est bien souvent un guide précieux pour l'écriture du code informatique.

Dans certaines situations, vous n'aurez pas besoin de deuxième branche à l'alternative. Vous voudrez simplement exécuter quelque chose si une condition est remplie, et ne rien faire sinon. Dans ce cas, vous avez essentiellement deux possibilités.

La première consiste à explicitement indiquer que la seconde branche ne fait rien, ce qui s'écrit ainsi :

<i>Python</i>	<i>C++</i>
<pre>if (une_condition): des_instructions else: pass autres_instructions</pre> <p>L'instruction <code>pass</code> signifie exactement "<i>passer à la suite sans rien faire</i>".</p>	<pre>if (une_condition) { des_instructions; } else { } autres_instructions;</pre>

Si vous vous demandez quel est l'intérêt d'utiliser une instruction comme `pass` en Python ou de créer un bloc vide en C++, disons simplement que de nombreux programmeurs estiment que, d'une part, cela améliore la lisibilité du programme, d'autre part, cela prépare en quelque sorte la place pour de futures évolutions.

Comparez avec l'autre possibilité, qui consiste à tout simplement "oublier" la partie `else` :

<i>Python</i>	<i>C++</i>
<pre>if (une_condition): des_instructions autres_instructions</pre>	<pre>if (une_condition) { des_instructions; } autres_instructions;</pre>

Cette forme plus concise est probablement la plus répandue. À vous de choisir celle qui vous convient le mieux, bien qu'il soit recommandé d'utiliser la forme longue (avec blocs vides), au moins dans un premier temps.

Comme exercice, essayez de modifier un peu le principe précédent pour traiter le cas où les deux nombres sont égaux et de modifier le code en conséquence, sachant que, pour tester l'égalité entre deux nombres, il faut utiliser l'opérateur `==` (et non pas le simple signe `=`, c'est une source très courante d'erreurs).

Voici une possibilité :

Python

```
1. if ( nombre_joueur == \
2.     nombre_a_trouver ) :
3.     print "Vous avez gagné !"
4. else:
5.     if ( nombre_joueur > \
6.         nombre_a_trouver ) :
7.         print "Trop grand !"
8.     else:
9.         print "Trop petit !"
```

Respectez bien l'indentation, elle est très importante en Python, surtout dans des situations comme celle-ci.

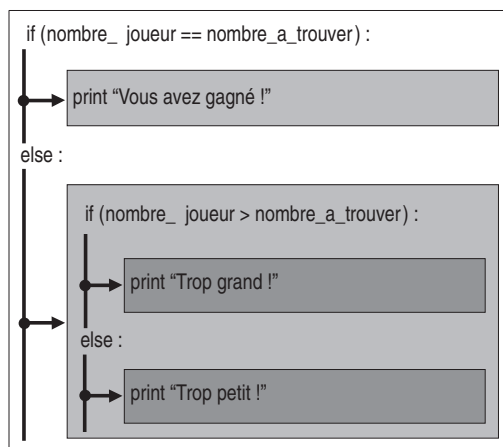
C++

```
1.  if ( nombre_joueur ==
2.      nombre_a_trouver )
3.  {
4.      std::cout << "Vous avez gagné !";
5.  }
6.  else
7.  {
8.      if ( nombre_joueur >
9.          nombre_a_trouver )
10.     {
11.         std::cout << "Trop grand !"
12.                 << std::endl;
13.     }
14.     else
15.     {
16.         std::cout << "Trop petit !"
17.                 << std::endl;
18.     }
19. }
```

Cette solution est un exemple typique d'une *imbrication de blocs*, que l'on peut schématiser comme à la Figure 5.1, où un cadre plus foncé représente un bloc plus "interne" :

Figure 5.1

*Blocs d'instructions
imbriqués.*



Les blocs d'instructions peuvent ainsi être imbriqués les uns dans les autres. En fait, vous constaterez qu'il s'agit d'une pratique très courante, à laquelle il est pratiquement impossible d'échapper !

Votre attention est attirée sur la matérialisation de ces blocs dans le code source : s'ils sont relativement aisément repérables en C++, grâce aux accolades, la seule façon de les reconnaître en Python se fonde, par contre, sur l'indentation. Répétons-le, cette indentation n'est pas seulement esthétique en Python : c'est elle qui définit réellement les blocs d'instructions, l'interpréteur va s'appuyer dessus pour identifier les blocs. Alors qu'en C++, elle n'est rien d'autre qu'une aide visuelle, le compilateur n'en faisant aucun cas.

5.2. La boucle

Nous avons progressé, mais notre programme n'est pas terminé : actuellement, l'utilisateur ne peut proposer qu'un seul nombre, puis le programme se termine. Comment faire pour que l'utilisateur puisse proposer plusieurs nombres ? Une solution naturelle consisterait à dupliquer les lignes de code. Mais cela pose au moins deux problèmes majeurs.

Si les lignes à dupliquer sont très nombreuses, on augmenterait ainsi considérablement la *masse totale* du code source, c'est-à-dire le nombre d'instructions. Or il a été démontré que les risques d'erreurs et la difficulté de maintenance d'un programme augmentent mécaniquement et rapidement avec sa taille : cette solution conduirait donc à un programme qu'il serait extrêmement pénible de faire évoluer et de corriger en cas de problème.

Dans bien des situations, dont la nôtre, on ne sait pas à l'avance combien de fois il sera nécessaire de répéter ces mêmes instructions. Ici, si on peut estimer qu'une centaine de répétitions serait largement suffisante et supportable, cela ne serait vraiment pas raisonnable dans les cas où les instructions doivent être répétées des milliers ou des millions de fois.

Le premier point pourrait être réglé par l'utilisation d'une fonction. Mais cela ne changerait rien au second. Nous avons besoin de quelque chose de nouveau.

Encore une fois, cherchons un principe, formulé en langage clair. Considérons le suivant :

tant que les nombres ne sont pas égaux, afficher un message adapté puis demander un nouveau nombre.

La partie "afficher un message" a déjà été traitée, nous pouvons donc remplacer ce membre par notre principe précédent (que nous savons déjà traduire) :

tant que les nombres ne sont pas égaux, si le nombre donné est plus grand que le nombre de référence, alors afficher "Trop grand", sinon afficher "Trop petit", puis demander un nouveau nombre.

Ce qui nous intéresse ici est le premier membre de cette phrase. Les mots "tant que" sous-entendent que nous allons effectuer les opérations qui suivent jusqu'à ce qu'une certaine condition soit vérifiée ou, plus précisément ici, tant qu'une condition n'est *pas* vérifiée. Premier problème, comment traduire l'expression "ne sont pas égaux" ? Il suffit d'exprimer la négation de la condition "sont égaux". En langage Python comme en C++, la négation d'une condition s'écrit simplement `not`. En effectuant les remplacements par les variables que nous avons, et en utilisant les opérateurs de comparaison, nous pouvons réécrire la phrase ainsi :

tant que not (nombre_joueur == nombre_a_trouver), si (nombre_joueur > nombre_a_trouver) alors print "Trop grand", sinon print "Trop petit", puis demander un nouveau nombre.

Maintenant, si vous saviez que les mots "tant que" se traduisent en anglais par `while`, avec un peu d'imagination vous devriez pouvoir écrire le code correspondant à cette phrase. Précisons simplement que `while` est également un mot clé en Python et en C++ pour représenter une boucle... Voici ce que cela devrait donner finalement comme programmes complets :

Python

```
1. # -*- coding: utf8 -*-
2. from random import randint
3. nombre_a_trouver = randint(1, 100)
4. print "Donnez un nombre :",
5. chaine = raw_input()
6. nombre_joueur = int(chaine)
7. while ( not ( nombre_joueur == \
8.             nombre_a_trouver ) ):
9.     if ( nombre_joueur > \
10.         nombre_a_trouver ):
11.         print "Trop grand !"
12.     else:
13.         print "Trop petit !"
```

C++

```
1. #include <iostream>
2. #include <cstdlib>
3. int main(int argc, char* argv[])
4. {
5.     srandom((long)argv);
6.     int nombre_a_trouver =
7.         (random()%100) + 1;
8.     int nombre_joueur;
9.     std::cout << "Donnez un nombre : ";
10.    std::cin >> nombre_joueur;
11.    while ( not ( nombre_joueur ==
12.                 nombre_a_trouver ) )
```



Python

```
14. print "Donnez un nombre :",
15. chaine = raw_input()
16. nombre_joueur = int(chaine)
17. print "Vous avez trouvé !"
```

Nous ne le répéterons jamais suffisamment : prenez garde à l'indentation. Si vous constatez que ce programme ne fonctionne pas correctement, vérifiez votre indentation. Les instructions d'un même bloc doivent être décalées vers la droite d'un même nombre d'espaces !

Cela n'est valable qu'en langage Python, l'indentation n'a qu'un but esthétique dans l'immense majorité des autres langages.

C++

```
13. {
14.     if ( nombre_joueur >
15.         nombre_a_trouver )
16.     {
17.         std::cout << "Trop grand !\n";
18.     }
19.     else
20.     {
21.         std::cout << "Trop petit !\n";
22.     }
23.     std::cout << "Donnez un nombre : ";
24.     std::cin >> nombre_joueur;
25. }
26. std::cout << "Vous avez trouvé !\n";
27. return 0;
28. }
```

Remarquez que les sauts de ligne répondent au caractère `\n` plutôt qu'à `std::endl`.

Si vous avez essayé de réaliser vous-même ce programme et que vous aboutissiez à une solution un peu différente, cela ne signifie pas forcément que vous êtes dans l'erreur : pour une même tâche, dès que la quantité de code augmente, différents programmeurs "produiront" différentes solutions, chacun s'appuyant sur son expérience, ses connaissances et même sa sensibilité. C'est ainsi que le *style* fait son apparition dans une activité pourtant apparemment si stricte.

Ce que nous venons de réaliser est parfois désigné par le terme de *répétitive*, et plus communément par le terme de *boucle*. Une boucle consiste à répéter une suite d'instructions (rassemblées dans un bloc, comme vous pouvez le constater ici), chaque répétition (ou *tour de boucle*) étant précédée d'une condition. Si cette condition est

vérifiée, le bloc de la boucle est exécuté, sinon il est ignoré et le programme se poursuit à l'instruction suivant immédiatement la boucle.

Dans notre exemple, la condition est la négation (not) de l'égalité de deux nombres. Cela signifie que la boucle s'arrêtera dès que les deux nombres seront égaux. Formulé différemment et plus précisément, la boucle s'arrêtera dès qu'il sera faux de dire que les deux nombres ne sont pas égaux : comme en langage naturel, une double négation équivaut à une affirmation. Ou encore : la boucle se poursuivra tant qu'il sera vrai de constater qu'il est faux de dire que les deux nombres sont égaux...

5.3. Algèbre de Boole et algorithmique

Nous venons de voir trois notions très importantes en programmation, dont une de façon implicite.

D'abord la notion d'alternative, qui permet d'orienter le flux d'un programme, à la manière d'un aiguillage sur une voie ferrée. Grâce à cela, certaines parties d'un programme ne seront exécutées que selon certaines conditions données.

Puis la notion de boucle, qui permet d'exécuter plusieurs fois une même suite d'instructions, comme si la voie ferrée formait un anneau, revenant sur elle-même. La sortie de l'anneau, c'est-à-dire la fin de la boucle, dépend d'une condition donnée (d'un aiguillage).

La troisième notion vient juste d'être mentionnée deux fois, car elle est implicite dans les deux précédentes : la notion de *condition*. La valeur d'une condition est donnée par le résultat d'un calcul particulier, qui est le test correspondant à la condition. À strictement parler, le type de cette valeur n'est ni un entier, ni une chaîne de caractères. Il s'agit d'un type dit *booléen* (prononcez "bou-lé-un"), par référence au modèle développé par le mathématicien et logicien britannique George Boole (1815-1864, dates à mettre en regard du terme si commun "nouvelles technologies"...).

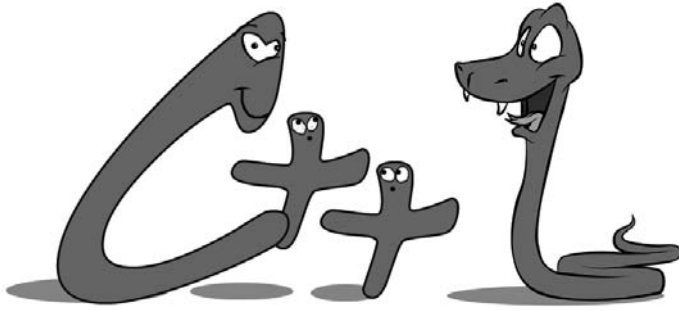
Une variable entière (de type entier) peut prendre les valeurs 1, 2, 3, 197635, et bien d'autres. Une variable booléenne (de type booléen) ne peut prendre que deux valeurs : soit *vrai* (*true*), soit *faux* (*false*). Par commodité, on note souvent chacune de ces valeurs respectivement 1 et 0, mais ce n'est qu'une représentation. Les règles qui régissent les calculs qu'on peut effectuer sur de telles variables (et bien d'autres choses encore) sont rassemblées dans ce qu'on appelle l'*algèbre de Boole*, conçu justement par M. Boole. Nous avons rencontré l'une de ces opérations : la négation.

Cette opération donne une valeur opposée à la valeur de la variable sur laquelle elle est appliquée : si une variable a la valeur *vrai*, sa négation est la valeur *faux*. Si une variable a la valeur *faux*, sa négation a la valeur *vrai*. Cela a été utilisé dans l'écriture de la condition de la boucle dans les programmes précédents.

Ces trois notions font aujourd'hui partie des fondements de ce qu'on appelle l'*algorithmique*, la science qui étudie les *algorithmes*. L'étymologie de ce mot remonte au Moyen Âge, au moment de la traduction d'un ouvrage publié en l'an 825 par l'astronome et mathématicien perse Muhammad ibn Mūsā al-Khūwārizmī, ouvrage dont le titre (*Al-jabr wa'l-muqābala*) a donné le mot *algèbre*. Un autre ouvrage du même auteur a été le premier à traiter du système des nombres indiens que nous utilisons aujourd'hui, premier système de l'histoire à inclure le nombre zéro.

Un algorithme est la description d'une suite d'actions ou d'opérations à effectuer dans un certain ordre. Au cours de cette suite, peuvent intervenir des boucles ou des alternatives. La notion même d'algorithme est loin d'être nouvelle. On doit à Euclide un procédé systématique pour trouver le plus grand diviseur commun (PGDC) de deux nombres, vers le III^e siècle avant l'an 1. À peine plus jeune, Ératosthène a imaginé une méthode, appelée crible d'Ératosthène, qui permet d'obtenir les nombres premiers. La notion d'algorithme a été formalisée par lady Ada Lovelace Byron, fille du poète éponyme au XIX^e siècle, en hommage de laquelle fut nommé le rigoureux langage de programmation Ada.

Nous venons de pratiquer un peu l'algorithmique, en imaginant les successions d'actions nécessaires pour obtenir le résultat désiré. L'action de traduire cet algorithme en langage Python ou C++ est désignée par le terme d'*implémentation de l'algorithme*. Tout travail de programmation est, en fait, composé au minimum de ces deux étapes : conception puis implémentation de l'algorithme. Avec l'expérience, dans bien des situations la première phase devient presque inconsciente et on a tendance à se jeter sur le clavier. Il faut se méfier de cette tendance, qui conduit parfois à omettre des cas particuliers importants.



6

Des variables composées

Au sommaire de ce chapitre :

- Plusieurs variables en une : les tableaux
- Plusieurs types en un : structures
- Variables composées et paramètres

Plaçons-nous un instant dans la peau d'un enseignant distribuant des notes à ses élèves et voulant obtenir quelques informations, comme la note moyenne à l'issue d'une épreuve, ceux qui sont au-dessus, ceux qui sont en dessous... Bref, un certain nombre de statistiques simples mais laborieuses à obtenir "manuellement". Ajoutons que notre enseignant ignore que son établissement scolaire dispose de très bons logiciels pour faire cela.

Avec ce que nous venons de voir, vous êtes déjà à même de concevoir un programme capable de calculer une moyenne : il suffit de demander des notes les unes après les autres, de les additionner au fur et à mesure, pour enfin afficher le total divisé par le nombre de notes. Seulement, cela ne nous permet pas de déterminer quelles sont les notes au-dessus ou en dessous de cette moyenne : il faudrait pour cela se "souvenir" de chaque note, pour pouvoir la comparer à la moyenne après que celle-ci a été calculée. Le problème essentiel étant qu'on ne sait pas forcément à l'avance *combien* il y aura de notes dont il faudra se souvenir...

Peut-être pensez-vous créer un grand nombre de variables, comme `note_01`, `note_02`, etc., pour mémoriser toutes ces valeurs. Mais jusqu'où devrez-vous aller ? Si l'enseignant est dans un collège, une quarantaine de variables devraient suffire, ce qui est déjà assez pénible à saisir dans un programme. Mais s'il est dans une université, alors il en faudra peut-être des centaines... Mais il est hors de question de créer "à la main" des centaines de variables !

Heureusement, des outils existent pour ce type de situations.

6.1. Plusieurs variables en une : les tableaux

Plutôt que de déclarer des centaines de variables, nous allons en déclarer une unique, ayant la capacité de contenir des centaines de valeurs. Ce sera par la suite possible de manipuler chacune de ces valeurs, comme s'il s'agissait d'une variable individuelle.

Selon les situations et les besoins, il existe différentes techniques pour réaliser un tel stockage. Nous ne verrons ici que la plus simple : l'utilisation de *tableaux* (*array* en anglais). Le principe est extrêmement simple. Imaginez qu'une valeur que vous voulez mémoriser soit un œuf. Si vous n'avez besoin que d'un œuf (que d'une seule valeur), vous pouvez le saisir directement. Mais si vous avez besoin d'une douzaine d'œufs, vous aurez naturellement recours à une boîte à œufs : vous ne manipulez plus les œufs un par un, mais plutôt la boîte dans son ensemble. Un tableau n'est rien d'autre qu'une

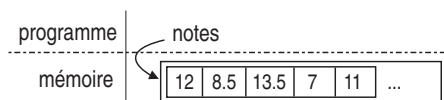
boîte à œufs, où les valeurs sont rangées les unes à côté des autres. De même que vous pouvez désigner chaque œuf dans la boîte en les dénombrant (le premier œuf, le deuxième, etc.), les valeurs dans un tableau sont numérotées : elles sont donc implicitement *ordonnées*, telle valeur se trouvant avant ou après telle autre valeur.

Un tableau est un exemple de *conteneur*, c'est-à-dire d'un type de variable destiné à contenir plusieurs valeurs. Pour poursuivre votre omelette, vous pouvez utiliser un panier plutôt qu'une boîte : dans ce cas, les œufs sont désordonnés. Il existe des types de conteneurs dans lesquels les valeurs sont mémorisées "en vrac".

Revenons à notre enseignant. Pour mémoriser ses notes, il va donc utiliser un tableau, par exemple nommé `notes`. La Figure 6.1 montre comment on peut représenter la variable `notes`.

Figure 6.1

Un tableau de notes.



Peut-être cette représentation vous rappelle-t-elle quelque chose : celle d'une chaîne de caractères, que nous avons vue au Chapitre 2. Ce n'est pas un hasard : fondamentalement, une chaîne de caractères est un tableau de caractères. Vous manipulez donc des tableaux depuis le début !

Voyons enfin comment déclarer et utiliser un tableau dans un programme. Voici le début du programme `moyenne` que pourrait créer l'enseignant :

Python	C++
<pre># -*- coding: utf8 -*- print "Combien de notes ?", nb_notes = int(raw_input()) notes = nb_notes * [0.0]</pre>	<pre>#include <iostream> #include <vector> int main(int argc, char* argv[]) { int nb_notes = 0; std::cout << "Combien de notes ? "; std::cin >> nb_notes; std::vector<double> notes(nb_notes, 0.0);</pre>

Le programme commence par demander combien il devra mémoriser de notes (nous verrons un peu plus loin comment faire si cette information n'est pas connue), ce

nombre étant stocké dans la variable `nb_notes`. Puis le tableau `notes` lui-même est créé.

La syntaxe en Python n'est pas sans rappeler celle permettant de dupliquer une chaîne de caractères : on utilise l'opérateur de multiplication `*`, en donnant d'un côté le nombre d'éléments dans le tableau, de l'autre la valeur initiale de chacun de ces éléments entre crochets. Ce sont les crochets qui indiquent que nous construisons un tableau, plutôt que de simplement multiplier deux nombres.

Comme on pouvait s'y attendre, les opérations sont un peu plus complexes en C++. Notez d'abord la deuxième ligne de l'extrait de code précédent : `#include <vector>`. Elle est nécessaire pour pouvoir utiliser les tableaux, du moins ceux représentés par le type `std::vector<>`. Remarquez bien les chevrons `<` et `>` à la fin du nom du type. Entre eux, vous devez inscrire le type des valeurs qui seront stockées dans le tableau, ici le type `double`. Cela pourrait être (presque) n'importe quel autre type. Dans notre cas, `std::vector<double>` est donc un type "*tableau de valeurs de type double*". Si vous écrivez `std::vector<std::string>`, il s'agit d'un type "*tableau de valeurs de type chaîne de caractères*". Vous pouvez même déclarer une variable contenant un tableau de tableaux, par exemple :

```
std::vector<std::vector<double> > variable;
```

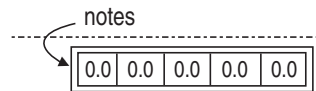
On dit alors que `variable` est de type "*tableau de valeurs de type tableau de valeurs de type double*", ou plus simplement, "*tableau de tableaux de doubles*". Il s'agit d'un tableau à deux dimensions. Un tableau "simple", dont les valeurs ne sont pas elles-mêmes des tableaux, est à une dimension. Vous pouvez créer ainsi des tableaux aussi complexes que vous le souhaitez, mais prenez garde à ne pas vous perdre dans des méandres spatio-temporels...

À la suite du type `std::vector<double>` se trouve, comme toujours, le nom de la variable, ici accompagnée du nombre d'éléments dans le tableau (`nb_notes`) et la valeur de chacun de ces éléments (`0.0`). Cette écriture est parfaitement analogue à celle que nous avons rencontrée précédemment pour créer des lignes de tirets, à la section 3.2. Une preuve supplémentaire qu'une chaîne de caractères est un type particulier de tableau dont les valeurs sont des caractères.

En Python comme en C++, à la suite de ces instructions nous disposons d'une variable `notes` permettant d'accéder à un tableau de valeurs de type `double`, pour l'instant toutes initialisées à `0.0`. Si l'utilisateur a demandé cinq notes, la situation en mémoire est telle que montrée par la Figure 6.2.

Figure 6.2

Tableau de notes
initialisées à 0.



6.1.1. La boucle bornée

Maintenant que nous avons notre tableau, il est temps d'y placer les notes obtenues par les élèves. Nous allons avoir besoin d'une boucle : pour chaque élève, demander la note obtenue et la placer dans le tableau. Vous pourriez construire une boucle en utilisant le mot clé `while` vu au chapitre précédent, mais nous nous trouvons ici dans une situation un peu particulière : nous savons exactement combien de notes il y a à demander. Dans ce genre de cas, assez typique lorsqu'on utilise des tableaux, on se sert plutôt d'un autre type de boucle, une boucle bornée. Celle-ci s'appuie sur une variable utilisée comme compteur pour dénombrer le nombre de fois que la boucle doit être exécutée. Ce type de boucle est introduit par le mot clé `for`, dont voici une application :

Python

```
for indice_note in range(nb_notes):  
    print "Note", indice_note+1, "?",  
    notes[indice_note] = \  
        float(raw_input())
```

La fonction `range()` permet de déclarer un intervalle de valeurs entières comprises entre 0 et la valeur passée en paramètre, moins 1. Par exemple, `range(5)` représente un intervalle compris entre 0 et 4, soit les valeurs 0, 1, 2, 3 et 4.

C++

```
for(int indice_note = 0;  
    indice_note < nb_notes;  
    ++indice_note)  
{  
    std::cout << "Note "  
               << indice_note+1  
               << " ? ";  
    std::cin >> notes[indice_note];  
}
```

Pour une variable `v` de type entier, `++v` augmente la valeur de `v` de 1 ; cette écriture est donc équivalente à `v = v+1`.

La ligne qui commence par `for` (ou les trois lignes en C++, remarquez les parenthèses) pourrait se traduire par :

pour indice_note allant de 0 à nb_notes-1, faire...

Cette forme de boucle est équivalente à une boucle introduite par `while`. On pourrait écrire la même chose de la façon suivante :

Python

```
indice_note = 0
while ( indice_note < nb_notes ):
    print "Note", indice_note+1, "?",
    notes[indice_note] = \
        float(raw_input())
    indice_note += 1
```

C++

```
int indice_note = 0;
while ( indice_note < nb_notes )
{
    std::cout << "Note "
               << indice_note+1
               << " ? ";
    std::cin >> notes[indice_note];
    ++indice_note;
}
```

L'écriture avec `for()` et celle avec `while()` se comportent de la même façon¹, mais la première est généralement considérée comme plus "compacte" car elle rassemble au même endroit la déclaration du compteur de boucle (ici, `indice_note`), la vérification que ce compteur est dans l'intervalle voulu et l'incrément de la valeur de ce compteur. À l'intérieur de la boucle, le compteur se comporte comme une variable. Toutefois, **ne modifiez jamais la valeur du compteur à l'intérieur de la boucle !** Bien que cela soit techniquement possible, l'expérience montre qu'une modification du compteur d'une boucle `for()` à l'intérieur de la boucle (c'est-à-dire dans les instructions exécutées par la boucle) conduit généralement à des catastrophes, pouvant aller jusqu'au plantage du programme.

Mais sans doute vous demandez-vous pourquoi le compteur de boucle va de 0 à `nb_notes-1`, plutôt que de 1 à `nb_notes`. La raison est simple : en Python comme en C++, le premier élément d'un tableau porte le numéro 0 (zéro). Plutôt que de numéro, on parle d'*indice*, d'où le nom donné au compteur. Vous pouvez voir cet indice comme le nombre de cases qu'il faut sauter dans le tableau pour arriver à l'élément : pour le premier élément, il faut sauter zéro (aucune) case, pour le deuxième, il faut en sauter une, et ainsi de suite.

1. En réalité, ces deux formes ne sont pas exactement équivalentes et interchangeables. En Python, `range()` crée en fait un tableau temporaire qui sera parcouru, ce qui n'arrive pas avec la boucle `while()`. En C++, le compteur de boucle n'existe qu'à l'intérieur de la boucle `for()`, alors qu'il existe après la boucle `while()`. Toutefois ces subtilités ne devraient pas vous gêner avant d'avoir davantage d'expérience.

Pour obtenir l'élément du tableau situé à un indice donné, il suffit d'inscrire le nom de la variable contenant le tableau, suivi de l'indice donné entre crochets. Dans l'exemple, le premier élément du tableau `notes` est donc obtenu par `notes[0]`, le deuxième par `notes[1]`, et ainsi de suite. Dans la boucle `for()` précédente, on donne le compteur de boucle plutôt qu'une valeur fixe : à mesure que le compteur va prendre les valeurs 0, 1, 2... On va ainsi parcourir le tableau en faisant référence à chacun de ses éléments, l'un après l'autre.

Essayez maintenant d'écrire le code qui va calculer la moyenne des notes. Il suffit pour cela de déclarer une variable destinée à recevoir la somme des notes, puis de la diviser par le nombre de notes. Voici une solution possible :

Python

```
somme = 0.0
for indice_note in range(nb_notes):
    somme += notes[indice_note]
moyenne = somme / nb_notes
print "Moyenne =", moyenne
```

C++

```
double somme = 0.0;
for(int indice_note = 0;
    indice_note < nb_notes;
    ++indice_note)
{
    somme += notes[indice_note];
}
double moyenne = somme / nb_notes;
std::cout << "Moyenne = "
            << moyenne << std::endl;
```

Il est ici particulièrement important de bien initialiser la variable `somme` à 0.0, afin que le calcul soit juste. S'il est obligatoire en Python de donner une valeur initiale à une variable au moment de sa déclaration, cela ne l'est pas en C++, mais c'est plus que vivement recommandé.

6.1.2. Si la taille est inconnue

Plus souvent que vous ne le souhaiteriez, vous ne connaîtrez pas à l'avance le nombre d'éléments à stocker dans un tableau. Par ailleurs, du point de vue de la convivialité d'un programme, on considère qu'il est préférable de décharger l'utilisateur du soin de compter le nombre de valeurs. Après tout, compter est encore ce que les ordinateurs font le mieux, or les ordinateurs sont là pour nous simplifier la tâche. Mais alors, comment faire avec nos tableaux ?

La solution consiste à déclarer un tableau vide, ne contenant aucun élément, puis à lui en ajouter au fur et à mesure que les éléments nous parviennent. Pour notre calcul de moyenne, cela signifie que l'on ne peut plus utiliser une boucle bornée pour demander

les notes : nous devons revenir à une boucle while classique. Voici comment procéder :

Python	C++
<pre>notes = [] print "Note ", len(notes)+1, "?", une_note = float(raw_input()) while (une_note >= 0.0): notes.append(une_note) print "Note ", len(notes)+1, "?", une_note = float(raw_input()) nb_notes = len(notes) print nb_notes, "notes."</pre>	<pre>std::vector<double> notes; double une_note = 0.0; std::cout << "Note " << notes.size()+1 << " ? "; std::cin >> une_note; while (une_note >= 0.0) { notes.push_back(une_note); std::cout << "Note " << notes.size()+1 << " ? "; std::cin >> une_note; } int nb_notes = notes.size(); std::cout << nb_notes << " notes.\n";</pre>

La première ligne de chacun des programmes déclare une variable notes de type tableau, ce tableau étant vide, c’est-à-dire qu’il ne contient aucun élément. Il va sans dire que si vous tentez de manipuler, ne serait-ce que consulter, une valeur dans l’un de ces tableaux, votre programme s’arrêtera brutalement en vous couvrant d’injures. D’une manière générale, si vous tentez d’accéder à un élément d’un tableau en donnant un indice supérieur ou égal au nombre d’éléments dans le tableau, alors vous allez au-devant de graves ennuis : dans le meilleur des cas, un plantage du programme, dans le pire, une corruption de données.

À la suite de cette déclaration, on amorce une boucle sur le même modèle que celle que nous avons utilisée dans le petit jeu du chapitre précédent. Comme il faut bien un moyen quelconque d’interrompre la boucle, on décide arbitrairement qu’on cesse de demander des valeurs dès qu’une note inférieure à 0 est donnée – ou, plus précisément, on continue tant que la note donnée est supérieure ou égale à 0.

Si une note valide a été donnée, on l’ajoute au tableau :

Python	C++
<pre>notes.append(une_note)</pre>	<pre>notes.push_back(une_note);</pre>

Cet ajout est effectué au moyen d'une *opération* disponible pour les types tableau. Nous avons déjà rencontré la notion d'opération, à la section 4.4.2 qui concernait le découpage d'une chaîne de caractères en Python. Ici, l'opération est `append()` pour Python, `push_back()` pour C++. L'effet est le même dans les deux langages : le tableau auquel l'opération est appliquée est agrandi d'une case, dans laquelle la valeur passée en paramètre (`une_note` dans l'exemple) est stockée. Les mots anglais *append* et *push_back* sont ici synonymes et signifient "*ajouter à la fin*". La dernière valeur ajoutée est donc le dernier élément du tableau.

Lorsque la boucle s'arrête, on peut obtenir le nombre d'éléments effectivement contenu dans le tableau, au moyen de la fonction `len()` en Python et de l'opération `size()` en C++ :

Python	C++
<pre>nb_notes = len(notes)</pre>	<pre>int nb_notes = notes.size();</pre>

Dès lors, la suite du programme est inchangée, il est à nouveau possible d'utiliser une boucle bornée pour les traitements suivants.

6.2. Plusieurs types en un : structures

Revenons à notre enseignant. Calculer une moyenne, c'est bien, mais il est encore plus intéressant de savoir quels sont les élèves au-dessus ou au-dessous de cette moyenne. Essayez d'écrire la boucle qui permet, par exemple, d'obtenir les notes inférieures à la moyenne. Elle pourrait ressembler à ceci :

Python	C++
<pre>for indice_note in range(nb_notes): if (notes[indice_note] < moyenne): print indice_note,</pre>	<pre>for(int indice_note = 0; indice_note < nb_notes; ++indice_note) { if (notes[indice_note] < moyenne) { std::cout << indice_note << " "; } }</pre>

Cela fonctionne, mais il y a un inconvénient : l'utilisateur obtient une suite d'indices, à lui ensuite de faire correspondre ces indices avec les noms des élèves. Tâche laborieuse, ingrate et source d'erreurs. Nous devons faire mieux.

Il faudrait pouvoir associer un nom à une note. C'est-à-dire mémoriser non seulement les notes mais également les noms des élèves les ayant obtenues. Comme si notre tableau ne contenait pas seulement des valeurs numériques mais aussi des chaînes de caractères.

Une telle association est possible par l'utilisation de *structures*, aussi appelées parfois *enregistrements*. Une structure est un nouveau type, créé de toutes pièces par le programmeur, qui rassemble en une seule entité des informations de différents types. Ces informations, appelées les *membres* de la structure, se comportent un peu comme des "sous-variables".

Créons donc une structure nommée *Eleve* dans le programme de moyenne, modifié ainsi :

Python

```
1. # -*- coding: utf8 -*-
2. class Eleve: pass
3.
4. eleves = []
5.
6. un_eleve = Eleve()
7.
8. print "Nom", len(eleves)+1, "?",
9. un_eleve.nom = raw_input();
10. while ( len(un_eleve.nom) > 0 ):
11.     print "Note", len(eleves)+1, "?",
12.     un_eleve.note = \
13.         float(raw_input())
14.     eleves.append(un_eleve)
15.     un_eleve = Eleve()
16.     print "Nom", len(eleves)+1, "?",
17.     un_eleve.nom = raw_input();
18.
19. nb_eleves = len(eleves)
```

C++

```
1. #include <iostream>
2. #include <vector>
3. #include <string>
4. struct Eleve
5. {
6.     std::string nom;
7.     double note;
8. };
9. int main(int argc, char* argv[])
10. {
11.     std::vector<Eleve> eleves;
12.     Eleve un_eleve;
13.     std::cout << "Nom "
14.               << eleves.size()+1
15.               << " ? ";
16.     std::getline(std::cin,
17.                 un_eleve.nom);
18.     while ( un_eleve.nom.size() > 0 )
19.     {
```



Python	C++
	<pre>20. std::cout << "Note " 21. << eleves.size()+1 22. << " ? "; 23. std::cin >> un_eleve.note; 24. eleves.push_back(un_eleve); 25. std::cout << "Nom " 26. << eleves.size()+1 27. << " ? "; 28. std::cin.ignore(); 29. std::getline(std::cin, 30. un_eleve.nom); 31. } 32. int nb_eleves = eleves.size();</pre>

Du côté Python, la déclaration d'une structure est introduite par le mot clé `class`. Du côté C++, on utilise le mot clé `struct`, suivi de la liste des membres de la structure entre accolades, comme s'il s'agissait de variables. N'oubliez pas le point-virgule à la fin de cette liste ! Remarquez que pour Python, les membres de la structure ne sont pas donnés à sa déclaration, mais "ajoutés" dès qu'on leur affecte une valeur, comme pour les variables classiques.

Répetons-le, la déclaration d'une structure équivaut à créer un nouveau type. Il est donc possible de créer des variables de ce type, comme ici `un_eleve`, et même de créer des tableaux de ce type, comme ici le tableau `eleves`. En Python, la déclaration d'une variable d'un type structuré se fait en utilisant le nom du type suivi de parenthèses, comme s'il s'agissait d'une fonction (ligne 6).

Lorsque vous avez une variable d'un tel type, vous pouvez accéder aux membres en utilisant la *notation pointée* : le nom de la variable, suivi d'un point, suivi du nom du membre. Par exemple, `un_eleve.nom` permet d'accéder au membre `nom` (en l'occurrence, une chaîne de caractères) contenu dans la variable de type `Eleve` nommée `un_eleve`.

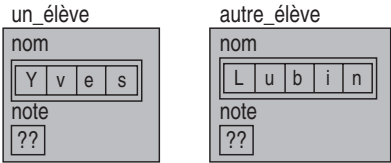
Par convention, on parle plutôt d'*instance* que de variable dans le cas des types structurés. Ainsi, on dit que `un_eleve` est une instance de la structure `Eleve`, ou plus simplement que `un_eleve` est une instance d'`Eleve`. De même, `un_eleve.note` désigne le membre `note` de l'instance `un_eleve` d'`Eleve`.

Naturellement, si vous créez deux instances d'un même type de structure, vous obtenez deux groupes de membres parfaitement distincts :

Python	C++
<pre>un_eleve = Eleve() autre_eleve = Eleve() un_eleve.nom = "Yves" autre_eleve.nom = "Lubin"</pre>	<pre>Eleve un_eleve; Eleve autre_eleve; un_eleve.nom = "Yves"; autre_eleve.nom = "Lubin";</pre>

À l'issue de ces instructions, on peut représenter la situation en mémoire comme à la Figure 6.3.

Figure 6.3
*Deux instances
pour deux élèves.*



Les instances (les variables) `un_eleve` et `autre_eleve` n'ont aucun lien entre elles, si ce n'est qu'elles partagent le même type, à savoir `Eleve`. Mais peut-être vous demandez-vous quelle est la signification des points d'interrogation dans le cadre qui représente le membre `note` ? Cela dépend en fait du langage considéré.

- En Python, ce membre n'existe pas encore, car aucune valeur ne lui a été affectée.
- En C++, comme pour toute variable, sa valeur n'est pas connue tant qu'aucune ne lui a été affectée. On ne sait donc pas ce que contient ce membre.

Ces deux points d'interrogation peuvent être gênants dans un programme : à la suite de la création d'une instance d'`Eleve`, quel que soit le langage, nous sommes en présence d'une forme d'incertitude. Or, les programmeurs détestent l'incertitude : elle est le chemin le plus sûr vers un programme défectueux. Heureusement, une solution existe.

6.2.1. Initialiser le contenu

L'objectif est ici de savoir exactement ce que contient une instance d'une structure, dès sa création, sans doute possible. Il nous faut pour cela initialiser les membres de la structure au moment même de sa création. Voici comme procéder :

Python

```
class Eleve:
    def __init__(self):
        self.nom = ""
        self.note = 0.0
```

Remarquez l'indentation : les trois lignes qui suivent la première "appartiennent" à la structure Eleve, elles en font partie.

C++

```
struct Eleve
{
    Eleve()
    {
        this->nom = "";
        this->note = 0.0;
    }
    std::string nom;
    double note;
};
```

Il suffit d'ajouter à la structure ce qu'on appelle un *constructeur*, c'est-à-dire une fonction spéciale chargée de *construire* toute instance de cette structure.

Python

Cette fonction spéciale porte un nom spécial qui est `__init__()`. Elle doit prendre (au moins) un paramètre spécial, portant le nom spécial `self`. Comme vous le constatez, tout cela est très spécialisé.

Le paramètre `self` représente l'instance elle-même, c'est-à-dire celle en cours de création. `self.nom` représente donc le membre `nom` de l'instance et celui d'aucune autre.

Ce paramètre `self` est l'équivalent de la variable implicite `this` du C++.

C++

Cette fonction spéciale doit porter le même nom que la structure, `Eleve()` dans notre exemple. Au sein de cette fonction, il existe implicitement une variable nommée `this` qui représente l'instance elle-même, c'est-à-dire celle en cours de création. Cette variable implicite est un peu particulière : pour accéder aux membres de la structure, il faut utiliser la flèche `->` plutôt que le point. Ainsi `this->nom` représente le membre `nom` de l'instance et celui d'aucune autre.

Cette variable implicite `this` est l'équivalent du paramètre `self` en Python.

À partir de maintenant, dès que vous créez une instance d'Eleve, les instructions qui ont été placées dans le constructeur seront exécutées. De cette façon, il n'y a plus aucune incertitude quant au contenu de l'instance nouvellement créée.

Par exemple :

Python	C++
<pre>un_eleve = Eleve() print un_eleve.note</pre>	<pre>Eleve un_eleve; std::cout << un_eleve.note << std::endl;</pre>
Sans constructeur, cet extrait provoquerait un plantage du programme, car alors le membre <code>note</code> n'existerait pas encore dans l'instance <code>un_eleve</code> .	Sans constructeur, cet extrait afficherait probablement une valeur parfaitement fantaisiste, car alors le membre <code>note</code> n'aurait pas été initialisé.

Ces deux extraits affichent simplement 0.0, qui est la note attribuée par défaut à un élève. C'est une très mauvaise note, mais au moins elle est parfaitement connue et déterminée.

6.3. Variables composées et paramètres

Pour terminer ce long chapitre consacré aux variables composées, revenons un instant au problème du passage de paramètres à une fonction. Plus précisément, au mécanisme par lequel une fonction reçoit un paramètre. Le but étant ici de créer une fonction pour calculer la moyenne des notes obtenues par les élèves, laquelle moyenne dépend naturellement de la liste des notes, donc du tableau d'élèves que nous avons créé. Autrement dit, nous allons devoir passer ce tableau en paramètre à notre fonction.

Toutefois, on ne transmet pas un objet composé à une fonction comme on transmet une simple valeur numérique. Il convient ici de s'interroger sur le "chemin" suivi par un paramètre.

6.3.1. Paramètre par valeur

Essayez les programmes suivants :

Python

```
# -*- coding: utf8 -*-
def Fonction(entier):
    entier = 2
    print "entier =", entier
un_entier = 1
Fonction(un_entier)
print "un_entier =", un_entier
```

C++

```
#include <iostream>
void Fonction(int entier)
{
    entier = 2;
    std::cout << "entier = "
                << entier << std::endl;
}
int main(int argc, char* argv[])
{
    int un_entier = 1;
    Fonction(un_entier);
    std::cout << "un_entier = "
                << un_entier << std::endl;
    return 0;
}
```

Les deux programmes afficheront :

```
entier = 2
un_entier = 1
```

Regardez la fonction définie dans ces programmes. Elle modifie le paramètre qu'elle reçoit en y plaçant une valeur, puis affiche la valeur contenue dans le paramètre. Comme on pouvait s'y attendre, la valeur affichée est bien celle qui a été stockée à la ligne précédente. Par contre, après l'appel de cette fonction, la variable qui lui a été passée en paramètre n'a pas été modifiée, comme le prouve l'affichage de sa valeur.

On parle dans ce cas d'un passage de paramètre *par valeur* : le paramètre entier que reçoit la fonction est comme une "vraie" nouvelle variable, qui contient une copie de la valeur contenue dans la variable un_entier passée à la fonction. Le temps de l'exécution de la fonction, on a effectivement deux variables, la variable "temporaire" (le paramètre) étant initialisée avec une copie du contenu de la variable "permanente". Celle-ci n'est en aucun cas modifiée par la fonction, quoi que celle-ci applique comme opération sur son paramètre.

6.3.2. Paramètre par référence

Mais voyons maintenant ce qui se passe dans le cas d'une structure ne contenant, par exemple, qu'un seul membre de type entier :

Python

```
# -*- coding: utf8 -*-
def Fonction(structure):
    structure.membre = 2
    print "structure.membre =", \
        structure.membre

class Une_Structure:
    def __init__(self):
        self.membre = 1

une_instance = Une_Structure()
print "une_instance.membre =", \
    une_instance.membre
Fonction(une_instance)
print "une_instance.membre =", \
    une_instance.membre
```

Affichage :

```
une_instance.membre = 1
structure.membre = 2
une_instance.membre = 2
```

C++

```
#include <iostream>
struct Une_Structure
{
    int membre;
    Une_Structure()
    {
        this->membre = 1;
    }
};
void Fonction(Une_Structure structure)
{
    structure.membre = 2;
    std::cout << "structure.membre = "
        << structure.membre
        << std::endl;
}
int main(int argc, char* argv[])
{
    Une_Structure une_instance;
    std::cout << "une_instance.membre = "
        << une_instance.membre
        << std::endl;
    Fonction(une_instance);
    std::cout << "une_instance.membre = "
        << une_instance.membre
        << std::endl;

    return 0;
}
```

Affichage :

```
une_instance.membre = 1
structure.membre = 2
une_instance.membre = 1
```

Cette fois, les deux programmes se comportent différemment. Si le programme C++ donne un résultat analogue au programme précédent, il en va tout autrement pour le

programme Python : il semble que la fonction ait effectivement modifié la valeur du membre `membre` de l'instance `une_instance` de la structure `Une_Structure`. Autrement dit, dans cette situation, en Python la fonction modifie l'instance qui lui est donnée en paramètre.

On parle alors de passage de paramètre *par référence* : la fonction ne reçoit plus une copie de la variable, mais en fait un nouveau point d'accès à celle-ci. Lorsque vous passez une variable composée (tableau ou structure) à une fonction en Python, vous ne transmettez qu'un nouveau point d'accès à cette variable, dont le contenu peut être modifié par la fonction.

En C++ au contraire, vous pouvez constater que l'on reste dans le cas d'un passage par valeur : la fonction reçoit bel et bien une copie de l'instance, toute action de la fonction sur son paramètre ne se retrouve pas dans la variable passée en paramètre. Les deux langages se comportent très différemment dans ce genre de situations.

Il n'est pas possible de reproduire en Python (du moins de manière simple) le comportement du C++ concernant le passage en paramètre de variables composées. Par contre, il est possible en C++ de préciser qu'on souhaite un passage de paramètre par référence. Il suffit pour cela de modifier légèrement la déclaration de la fonction comme suit :

```
void Fonction(Une_Structure& structure)
{
    /* des instructions */
}
```

Si vous ajoutez le symbole *éperluette* (ou "*et commercial*") à la suite du type du paramètre, alors le mécanisme du passage par référence sera utilisé. Essayez de modifier ainsi le programme C++ précédent : il se comportera alors comme le programme Python.

6.3.3. Mode de passage des paramètres

Dès que vous voulez passer des variables composées en paramètres à des fonctions, il est vivement recommandé d'utiliser le passage par référence. C'est automatique en Python, par contre vous devez le préciser explicitement en C++ au moyen de l'éperluette. Ce mécanisme évite la duplication des données en mémoire, qui survient lors du passage par valeur. Cette duplication n'est pas gênante s'il s'agit d'un simple entier ou d'un nombre en virgule flottante. Elle peut devenir assez coûteuse, en temps d'exécution ainsi qu'en encombrement de la mémoire, dans le cas de variables composées. Dupliquer un tableau contenant un million de valeurs n'est pas gratuit !

Incidemment, nous avons vu que les chaînes de caractères ressemblaient fortement à des tableaux de caractères. La remarque précédente s'applique également pour ce type de données : si vous devez passer une chaîne de caractères à une fonction, utilisez un passage par référence. Les Figures 6.4 et 6.5 schématisent la différence de comportement entre les deux modes de passage.

Figure 6.4

Passage de paramètre par valeur.

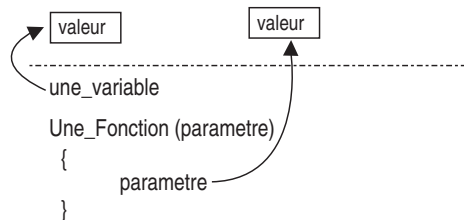
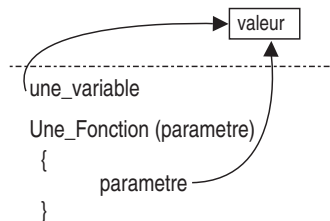


Figure 6.5

Passage de paramètre par référence.



Dernier petit raffinement, que vous verrez fréquemment utilisé dans du code C++ écrit par des programmeurs plus expérimentés. Pour diverses raisons, il peut être souhaitable de s'assurer qu'une fonction ne modifie pas le contenu du paramètre qui lui est passé. C'est une façon efficace de contrôler les modifications que subit une variable (d'un type composé) et ainsi de réduire la complexité d'un programme. D'un point de vue strictement théorique, on peut même dire qu'une fonction qui retourne une valeur ne devrait jamais modifier aucun de ses paramètres, bien qu'en pratique une telle rigueur soit rarement appliquée.

Si vous créez une fonction qui ne devrait jamais modifier ses paramètres, il suffit d'ajouter le mot clé `const` devant le nom du type du paramètre. Nous pouvons alors enfin construire une fonction calculant une moyenne à partir des notes des élèves :

```
double Moyenne(const std::vector<Eleve>& eleves)
{
    double somme = 0.0;
    for(int indice_note = 0;
        indice_note < eleves.size();
        ++indice_note)
```

```
{
    somme += eleves[indice_note].note;
}
return somme / eleves.size();
}
```

La fonction `Moyenne()` va recevoir un paramètre "doublement structuré", puisque nous lui passerons un tableau de structures `Eleve`. On utilise donc le passage par référence avec `&`. De plus, il est logique de considérer qu'une fonction calculant une moyenne ne doit en aucun cas modifier les valeurs qui lui sont données : on ajoute donc `const` devant le type du paramètre, qui est également le type du tableau contenant les élèves. Si vous tentez de modifier l'une des valeurs contenues dans le tableau, le compilateur vous avertira d'une erreur. Vous pouvez vérifier cela sur le petit programme de la section précédente qui montrait le passage par référence : ajoutez `const` dans la déclaration de la fonction, le compilateur refusera le programme avec un message dans le style de celui-ci :

```
$ g++ test_passage_reference.cpp
test_passage_reference.cpp: In function 'void Fonction(const Une_Structure&)':
test_passage_reference.cpp:12: error: assignment of data-member
'Une_Structure::membre' in read-only structure
```

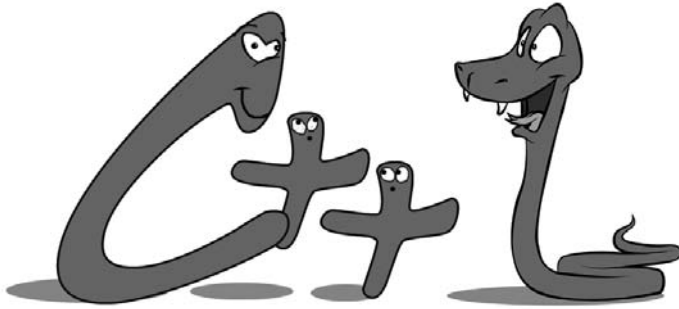
Ce qui signifie, en gros, *"tentative d'affectation d'un membre dans une structure en lecture seule"*.

Malgré tous ces raffinements, la fonction `Moyenne()` s'utilise tout naturellement dans le programme principal :

```
double moyenne = Moyenne(eleves);
```

Pour vous exercer, modifiez le programme en Python de façon à calculer la moyenne par une fonction. En sachant que le passage par référence sera automatique et qu'il n'existe pas d'équivalent au mot clé `const` en Python...

D'une manière générale, en C++, n'hésitez pas à utiliser le mot clé `const`. Il donne des informations précieuses au compilateur concernant vos intentions, ainsi il est mieux à même de vérifier que votre code est correct. Cela peut également permettre certaines optimisations aboutissant à une exécution plus rapide.



7

Les Tours de Hanoï

Au sommaire de ce chapitre :

- Modélisation du problème
- Conception du jeu
- Implémentation
- Résumé de la démarche

Le moment est venu de rassembler l'ensemble de ce que nous avons vu jusqu'ici pour construire un programme plus important. L'exemple choisi est celui du jeu des Tours de Hanoï, ou Tours de Brahma, jeu imaginé par le mathématicien français Édouard Lucas et inspiré d'une très ancienne légende.

Dans le grand et fameux temple de Bénarès, sous le dôme marquant le centre du monde, trône un socle de bronze sur lequel sont fixées trois aiguilles de diamant, chacune d'elles haute d'une coudée et fine comme la taille d'une guêpe.

Sur une de ces aiguilles, à la Création, Dieu empila soixante-quatre disques d'or pur, du plus grand au plus petit, le plus large reposant sur le socle de bronze.

Il s'agit de la tour de Brahma.

Jour et nuit, inlassablement, les moines déplacent les disques d'une aiguille vers une autre tout en respectant les immuables lois de Brahma qui les obligent à ne déplacer qu'un disque à la fois et à ne jamais le poser sur un disque plus petit.

Quand les soixante-quatre disques auront été déplacés de l'aiguille sur laquelle Dieu les déposa à la Création vers une des autres aiguilles, la tour, le temple et les brahmanes seront réduits en poussière, et le monde disparaîtra dans un grondement de tonnerre.

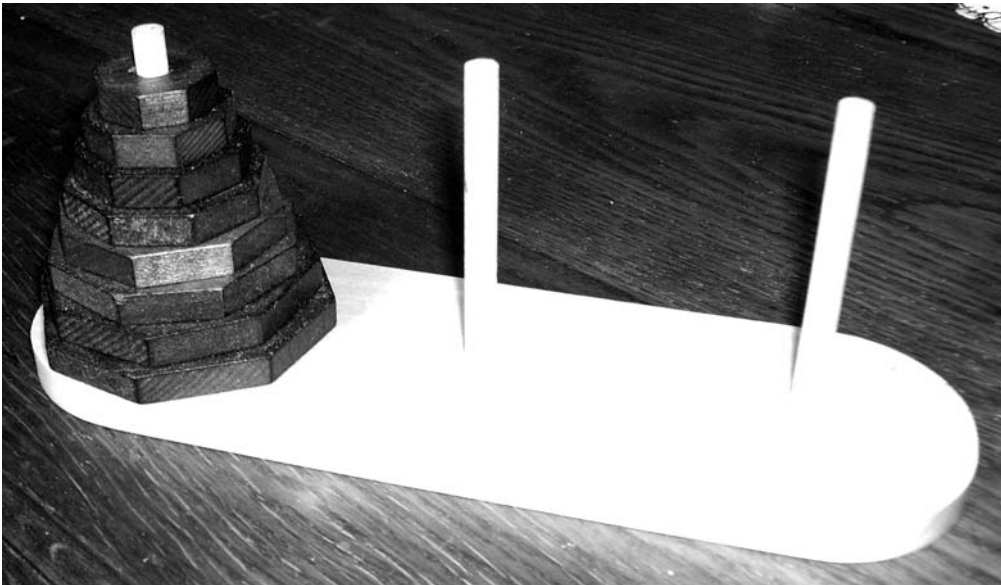


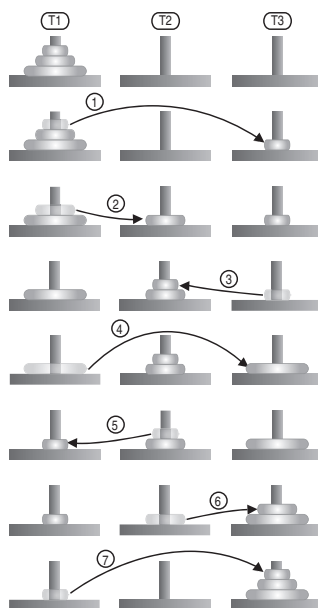
Figure 7.1

Modèle en bois du jeu des Tours de Hanoï.

L'intérêt de ce jeu, très couramment utilisé en programmation, est qu'il est suffisamment simple pour être compris sans difficulté, tout en étant suffisamment riche pour mettre à contribution toutes les techniques que nous avons rencontrées – et d'autres encore, comme nous le verrons dans la suite de cet ouvrage.

Première étape : comprendre à peu près de quoi il retourne. La légende parle de soixante-quatre disques, ce qui est assez considérable (nous verrons dans un instant pourquoi). La puissance des matériels disponibles ne cessant de croître, il est fréquent d'envisager des programmes destinés à manipuler de grandes quantités de données ou, du moins, nécessitant des quantités astronomiques de calculs. Dans ces situations, on commence par étudier le problème sur un ensemble plus petit. Pour notre exemple, prenons simplement trois disques. Saurez-vous trouver le bon enchaînement de mouvements ? La Figure 7.2 donne la solution.

Figure 7.2
*Solution au problème
avec trois disques.*



Nous avons dû effectuer sept mouvements. Et c'est la meilleure solution : il est impossible de résoudre le problème avec moins de mouvements. En fait, on peut démontrer (ce que nous ne ferons pas ici) que pour n disques, il faut effectuer au minimum $2^n - 1$ mouvements. Question : combien les moines de la légende devront-ils effectuer de mouvements ? Réponse : $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$, soit environ dix-huit milliards de milliards, en supposant qu'il n'y ait aucune erreur. Si les moines parviennent à déplacer un disque par seconde, le temps nécessaire au déplacement

de toute la tour sera d'un peu moins de 585 *milliards* d'années... Ce qui est rassurant, la fin du monde n'est pas pour demain ! Même simulés sur un ordinateur actuel, capable d'effectuer des dizaines de millions de mouvements par seconde, tous ces déplacements nécessiteraient tout de même des milliers d'années pour être énumérés. Ce petit exemple montre que parfois, un problème apparemment simple demeure hors de portée de nos machines, pourtant si puissantes.

7.1. Modélisation du problème

Voyons maintenant comment nous allons représenter les différents éléments. *A priori*, le jeu compte au moins deux types d'objets : le disque et la tour. Pour simplifier un peu les choses, nous pouvons décider qu'un disque est simplement représenté par un entier, dont la valeur est sa taille. Dans l'exemple de trois disques précédent, au tout début du jeu, le petit disque au sommet sera donc représenté par l'entier 1, et le grand tout en bas par l'entier 3.

Une tour est un empilement vertical de disques. Si on la fait tomber, on a un alignement horizontal de disques, c'est-à-dire, en fait, d'entiers. Une tour sera donc constituée d'un tableau d'entiers, le premier élément dans le tableau contenant la taille du disque le plus bas. Par convention, on peut décider qu'une valeur de 0 indique qu'il n'y a pas de disque. Ce tableau devra pouvoir contenir autant d'entiers qu'il y a de disques à déplacer. Toutefois, à un instant donné, une tour ne porte que quelques disques, voire aucun : il est donc également nécessaire de connaître le nombre de disques actuellement empilés sur une tour. On doit associer un tableau d'entiers et un compteur : nous sommes en présence d'une structure, qui pourrait être nommée *Tour* et avoir l'allure suivante :

Python

```
class Tour:
    def __init__(self, taille_max):
        self.nb_disques = 0
        self.disques = \
            [0 for e in range(taille_max)]
```

C++

```
struct Tour
{
    int nb_disques;
    std::vector<int> disques;
    Tour(int taille_max):
        nb_disques(0),
        disques(taille_max, 0)
    {
    }
};
```

Nous retrouvons ici les tableaux et les structures, dûment initialisés, que nous avons vus au chapitre précédent. Remarquez toutefois la présence d'un paramètre supplémentaire dans chacun des constructeurs (en Python et en C++) : il est possible de donner des informations à la structure au moment où on en crée une instance. Ainsi, l'exemple suivant crée une instance de la structure `Tour` dimensionnée pour recevoir cinq disques :

Python	C++
<pre>une_tour = Tour(5)</pre>	<pre>Tour une_tour(5);</pre>

En Python, l'information (le paramètre) est passée entre parenthèses après le nom de la structure, tandis qu'elle est passée après le nom de l'instance en C++. Finalement, le constructeur se comporte un peu comme une fonction qui serait intimement attachée à la structure.

De plus, nous avons utilisé une notation particulière en C++ pour initialiser les membres de la structure : ils sont donnés les uns après les autres, séparés par des virgules, accompagnés d'une paire de parenthèses contenant les informations nécessaires à leur initialisation. Cette liste est introduite par le caractère deux-points (:). Dans ce contexte, l'écriture `nb_disques(0)` signifie exactement "*initialiser le membre `nb_disques` à la valeur 0*". Le membre `disques`, qui est un tableau, est initialisé à partir de deux informations : la taille du tableau et la valeur à placer dans chaque case. Remarquez que cette écriture ressemble beaucoup à celle utilisée lors de la création d'une instance d'une structure, lorsque le constructeur de cette structure demande un paramètre. Cette similitude n'est pas fortuite. Mais nous reviendrons là-dessus.

Il existe un troisième type d'objet : le jeu lui-même. Celui-ci est caractérisé par trois tours, ainsi que par une hauteur maximale. En effet, il est souhaitable que l'utilisateur puisse choisir le nombre maximal de disques avec lesquels il veut jouer. Inutile de lui imposer un nombre trop élevé, ce qui le découragerait, ou trop faible, ce qui lui paraîtrait trivial.

Nous avons donc une autre structure, chargée de contenir l'état du jeu à tout instant :

Python

```

1. class Hanoi:
2.     def __init__(self, hauteur_max):
3.         self.hauteur = hauteur_max
4.         self.tours = \
5.             [Tour(hauteur_max) \
6.              for t in range(3)]
7.         for etage in range(hauteur_max):
8.             self.tours[0].disques[etage] = \
9.                 hauteur_max - etage
10.        self.tours[0].nb_disques = \
11.            hauteur_max

```

C++

```

1. struct Hanoi
2. {
3.     int hauteur;
4.     std::vector<Tour> tours;
5.     Hanoi(int hauteur_max):
6.         hauteur(hauteur_max),
7.         tours(3, Tour(hauteur_max))
8.     {
9.         for(int etage = 0;
10.            etage < hauteur_max;
11.            ++etage)
12.         {
13.             this->tours[0].disques[etage] =
14.                 hauteur_max - etage;
15.         }
16.         this->tours[0].nb_disques =
17.             hauteur_max;
18.     }
19. };

```

Cette structure `Hanoi` contient donc, en plus d'un entier, un tableau de structures contenant elles-mêmes un tableau. Voilà qui commence à devenir une imbrication complexe. Cette complexité transparaît dans l'initialisation de la première tour, qui est simplement "remplie" avec les disques demandés afin que le jeu soit prêt à démarrer dès sa création (lignes 7 à 11 en Python, lignes 9 à 17 en C++). Dans nos structures, `tours` est un tableau de structures `Tour`.

Notez qu'il est nécessaire de passer un paramètre au constructeur de `Hanoi` pour pouvoir construire les instances de `Tour` qu'elle contient, le constructeur de `Tour` attendant lui-même un paramètre. Dans le cas de l'initialisation du tableau en C++, ce paramètre est donné à la suite du type contenu dans le tableau : le membre `tours` sera donc un tableau de trois instances de la structure `Tour`, le constructeur de chacune de ces instances recevant le paramètre `hauteur_max`.

Ainsi, dans le cadre d'une instance de cette structure `Hanoi` :

- `tours[0]` est le premier élément de ce tableau, donc dans notre cas une variable (structure) de type `Tour`.

- `tours[0].nb_disques` désigne le membre nommé `nb_disques` dans la structure de type `Tour` se trouvant en première position dans le tableau `tours`.
- `tours[0].disques` désigne le membre nommé `disques` dans la structure de type `Tour` se trouvant en première position dans le tableau `tours`, ce membre étant lui-même un tableau.
- `tours[0].disques[etage]` désigne, enfin, l'élément à la position `etage` dans le tableau `disques` contenu dans la structure de type `Tour` se trouvant en première position dans le tableau `tours`.

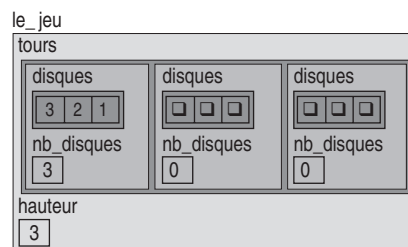
Prenez le temps de lire et de comprendre chacune de ces désignations. Nous avons des objets qui englobent d'autres objets. Dans une désignation, l'objet "le plus englobant" se trouve à gauche. À mesure qu'on parcourt la désignation vers la droite, on traverse les niveaux d'imbrication des objets, un peu comme on traverserait des strates géologiques ou les couches d'un oignon. On passe d'une couche à l'autre en utilisant soit les crochets (lorsque la couche d'où on vient est un tableau), soit le point suivi d'un nom de membre (lorsque la couche d'où on vient est une structure).

Imaginons que nous déclarions un jeu pour trois disques :

Python	C++
<code>le_jeu = Hanoi(3)</code>	<code>Hanoi le_jeu(3);</code>

On peut alors représenter, à la Figure 7.3, le contenu de la variable `le_jeu` que nous venons de déclarer, dans les deux langages.

Figure 7.3
Instance du jeu
pour trois disques.



Si nous voulons, par exemple, la taille du disque situé au troisième étage de la deuxième tour, nous devons écrire `le_jeu.tours[1].disques[2]`. Suivez le cheminement pour arriver à cette valeur.

7.2. Conception du jeu

Nous disposons maintenant de tout ce qui est nécessaire à l’affichage du jeu. Il est temps de nous pencher sur sa *logique interne*, c’est-à-dire sur la mise en œuvre des règles du jeu telles qu’énoncées dans la légende, la plus importante étant qu’il est interdit de poser un disque sur un autre plus petit.

Dans ce genre de situation, une méthode qui fonctionne assez bien (pour des programmes de taille modeste) consiste à exprimer littéralement une étape du jeu ou de l’action que doit effectuer le programme. Cette expression doit être aussi générale que possible, exempte de détails, éloignée de toute idée de programmation. Dans notre cas, cela donnerait quelque chose comme ceci :

1. Afficher l’état du jeu.
2. Déterminer le piquet de départ.
3. Déterminer le piquet d’arrivée.
4. Déplacer le disque du piquet de départ vers celui d’arrivée.
5. Recommencer tant que le jeu n’est pas terminé.

Remarquez qu’on utilise principalement des verbes à l’infinitif. Ces verbes vont se traduire en presque autant de fonctions, ce qui donne une première ébauche du programme, comme vu de très loin. Notez que la plupart de ces fonctions n’existent pas encore, elles se dégagent naturellement de la liste précédente. On peut dès maintenant écrire cette ébauche, en supposant disposer de toutes les variables nécessaires (dans notre exemple, une instance de la structure `Hanoi`) :

Python

```
while ( not Est_Fini(le_jeu) ) :  
    Afficher_Jeu(le_jeu)  
    tour_depart = \  
        Demander_Tour_Depart(le_jeu)  
    tour_arrivee = \  
        Demander_Tour_Arrivee(le_jeu)  
    Deplacer_Disque(le_jeu, \  
                    tour_depart, \  
                    tour_arrivee)
```

C++

```
while ( ! Est_Fini(le_jeu) )  
{  
    Afficher_Jeu(le_jeu);  
    int tour_depart =  
        Demander_Tour_Depart(le_jeu);  
    int tour_arrivee =  
        Demander_Tour_Arrivee(le_jeu);  
    Deplacer_Disque(le_jeu,  
                    tour_depart,  
                    tour_arrivee);  
}
```

On passe systématiquement l'instance de Hanoi, qui contient l'état du jeu, aux fonctions qui ont été envisagées : cela semble simplement logique que ces fonctions aient "besoin" de connaître l'état du jeu pour opérer. Pour la même raison, la fonction `Deplacer_Disque()` doit logiquement "savoir" de quel piquet à quel piquet elle doit effectuer le déplacement d'un disque.

Il est maintenant temps d'introduire une considération extrêmement importante en programmation, surtout dans un programme qui doit interagir avec un utilisateur (ce qui est en fait le cas de la plupart). Elle s'exprime sous la forme d'une règle simple : *Ne jamais faire confiance aux informations reçues de l'extérieur !*

Autrement dit, toujours vérifier que ce que donne l'utilisateur est cohérent. Dans notre cas, cela signifie qu'il est nécessaire de vérifier :

- Que le piquet de départ demandé est correct, c'est-à-dire qu'il ne s'agit pas d'un piquet vide ou d'un numéro de piquet aberrant (plus petit que 1 ou plus grand que 3).
- Que le piquet d'arrivée est correct, c'est-à-dire que le mouvement est autorisé selon les règles du jeu et qu'il s'agit d'un numéro valide (compris entre 1 et 3).

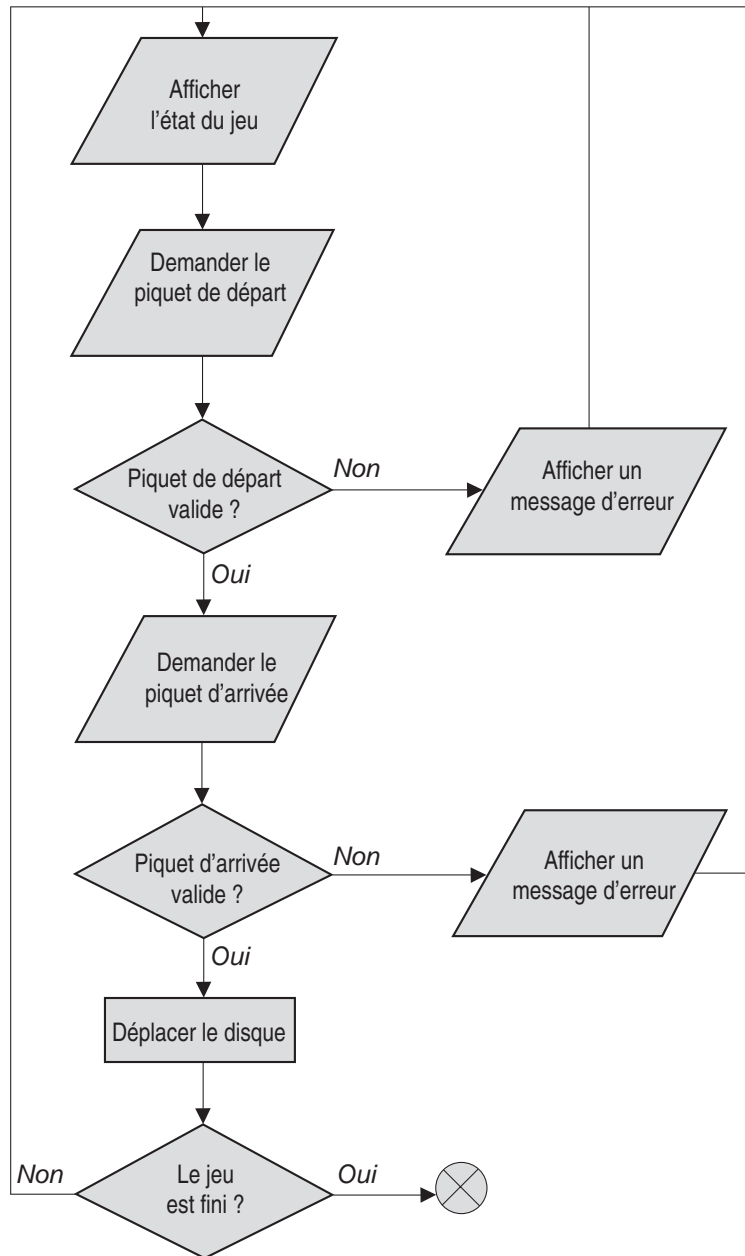
Il va donc falloir deux fonctions supplémentaires chargées d'effectuer ces vérifications. Elles retourneront une valeur booléenne (*vrai* ou *faux*) selon le cas. Le déroulement d'une étape du jeu devient alors :

1. Afficher l'état du jeu.
2. Demander le piquet de départ.
3. Si ce piquet est valide, alors :
 - demander le piquet d'arrivée ;
 - si ce piquet est valide, alors :
 - déplacer le disque ;
 - sinon, afficher un message d'erreur.
4. Sinon, afficher un message d'erreur.
5. Recommencer tant que le jeu n'est pas terminé.

Comme vous pouvez le constater, cette expression littérale laisse entrevoir la structure générale du programme. Une façon assez commune de représenter graphiquement un

tel déroulement est d'utiliser un *ordinogramme* (ou *algorithme*, ou *organigramme de programmation*), qui se présente comme à la Figure 7.4.

Figure 7.4
Ordinogramme du jeu.



Les flèches indiquent le déroulement des opérations. Les cadres aux bords inclinés sont utilisés pour représenter les interactions avec l'utilisateur, tandis que les formes en losange représentent les alternatives. Enfin, un cadre simple représente simplement des calculs "internes" au programme.

Ce type de représentation a longtemps été très en vogue et est toujours utilisé pour représenter des algorithmes simples ou une vision très générale d'un logiciel. Son principal intérêt est sa simplicité. Il présente toutefois certaines limites dès qu'il s'agit de traiter un problème complexe ou de grande taille. Aujourd'hui, on utilise plutôt le diagramme d'état, qui n'est qu'une partie d'une technique beaucoup plus vaste nommée UML (*Unified Modeling Language*, langage de modélisation unifié), particulièrement adaptée à la programmation orientée objet (que nous allons aborder très prochainement). Nous ne détaillerons pas ici ce procédé sophistiqué, qui nécessiterait à lui seul un ouvrage complet.

À partir de cette représentation, nous pouvons étoffer notre ébauche de programme :

Python

```
while ( not Est_Fini(le_jeu) ) :
    Afficher_Jeu(le_jeu)
    tour_depart = Demander_Tour_Depart(le_jeu)
    if ( Tour_Depart_Valide(le_jeu,
                           tour_depart) ) :
        tour_arrivee = Demander_Tour_Arrivee(le_jeu)
        if ( Tour_Arrivee_Valide(le_jeu,
                                tour_depart,
                                tour_arrivee) ) :
            Deplacer_Disque(le_jeu, \
                           tour_depart, \
                           tour_arrivee)
        else:
            print "Arrivée invalide !"
    else:
        print "Départ invalide !"
```

C++

```
while ( ! Est_Fini(le_jeu) )
{
    Afficher_Jeu(le_jeu);
    int tour_depart =
        Demander_Tour_Depart(le_jeu);
    if ( Tour_Depart_Valide(le_jeu,
                           tour_depart) )
    {
        int tour_arrivee =
            Demander_Tour_Arrivee(le_jeu);
        if ( Tour_Arrivee_Valide(le_jeu,
                                tour_depart,
                                tour_arrivee) )
        {
            Deplacer_Disque(le_jeu,
                           tour_depart,
                           tour_arrivee);
        }
        else
        {
            std::cout << "Arrivée invalide !\n";
        }
    }
    else
    {
        std::cout << "Départ invalide !\n";
    }
}
```

Dès que vous vous attaquez à un problème complexe, il est vivement recommandé d'en exprimer littéralement les étapes et éventuellement de construire un ou plusieurs ordigrammes. L'idée est de partir d'une vision très générale, de "haut niveau", puis de "descendre" progressivement dans les détails de chacun des éléments identifiés. On retrouve là une mise en pratique de la méthode cartésienne, ce que nous allons faire immédiatement.

7.3. Implémentation

Nous pouvons maintenant réaliser l'implémentation des diverses fonctions qui ont été prévues à la section précédente.

7.3.1. Afficher le jeu

Voyons comment nous allons coder la fonction d’affichage `Afficher_Jeu()`. Tout naturellement, l’utilisateur ne saurait se contenter d’un simple affichage de valeurs numériques. On pourrait en effet très simplement afficher ceci (ce qui est laissé en exercice au lecteur) :

```
Tour 0 : 3 2 1
Tour 1 : 0 0 0
Tour 2 : 0 0 0
```

C’est un reflet fidèle de la réalité, mais pas très pratique à utiliser et encore moins ludique. Il serait déjà plus sympathique d’avoir quelque chose comme ceci :

```
<=!>      !      !
<==!==>    !      !
<===!===>  !      !
```

Les colonnes de points d’exclamation représentent chacune un piquet, les disques étant figurés par une suite de caractères = entre chevrons. Chaque disque, ou plutôt chaque étage, devra ainsi être représenté individuellement en fonction du disque qui s’y trouve (ou non) et de la largeur maximale possible. Cette représentation étant une chaîne de caractères, il semble logique d’envisager la création d’une fonction `Chaine_Etage()` chargée de "produire" la chaîne en question. Voici quelle pourrait être cette fonction :

Python

```
def Chaine_Etage(taille,
                 largeur):
    espace = ' '*(largeur-taille)
    chaine = ""
    if ( taille > 0 ):
        disque = '='*taille
        chaine = espace + "<" + \
            disque + "!" + \
            disque + ">" + espace
    else:
        chaine = espace + " ! " + espace
    return chaine
```

C++

```
std::string Chaine_Etage(int taille,
                        int largeur)
{
    std::string
        espace(largeur-taille, ' ');
    std::string chaine;
    if ( taille > 0 )
    {
        std::string disque(taille, '=');
        chaine = espace + "<" +
            disque + "!" +
            disque + ">" + espace;
    }
    else
    {
        chaine = espace + " ! " + espace;
    }
    return chaine;
}
```


Cette fonction prend deux paramètres, le premier est la taille du disque se trouvant éventuellement à cet étage (ou 0, s'il n'y a pas de disque), le second la largeur totale de l'étage, ce qui correspond en fait à la plus grande taille possible de disque, soit le nombre de disques. À partir de ces deux paramètres, on construit une chaîne représentant un étage, éventuellement avec un disque.

Maintenant que nous savons représenter un étage, voyons comment représenter l'ensemble du jeu. Le mode texte pose une contrainte particulière : on ne peut afficher les caractères que ligne par ligne, la nouvelle ligne repoussant la précédente vers le haut. Cela pose deux problèmes :

- Si nous parcourons une tour "normalement", du premier étage au dernier, elle arrivera à l'écran la tête en bas : le premier étage sera la première ligne affichée, qui sera repoussée vers le haut, la ligne du dernier étage étant la dernière affichée – donc la plus basse à l'écran.
- Si nous affichons les tours l'une après l'autre, elles apparaîtront l'une en dessous de l'autre à l'écran, ce qui n'est guère gracieux ; il nous faudrait trouver un moyen pour qu'elles apparaissent côte à côte.

Vous le voyez, ce qui peut passer pour un problème simple – afficher les disques de chacune des tours – peut en réalité cacher des difficultés inattendues. Tentons de les résoudre.

Le premier point est assez simple : plutôt que de parcourir les étages du premier (en bas) au dernier (en haut), il suffit de les parcourir du dernier au premier. Plusieurs solutions sont possibles pour cela, par exemple utiliser une boucle bornée classique dans laquelle on déterminerait le numéro correct de l'étage, comme ceci :

Python	C++
<pre>for etage in range(nb_etages): etage_correct = nb_etages - etage - 1 # afficher l'étage</pre>	<pre>for(int etage = 0; etage < nb_etages; ++etage) { etage_correct = nb_etages - etage - 1; // afficher l'étage }</pre>

La "correction" du numéro d'étage tient compte du fait que, si `nb_etages` représente le nombre d'étages à afficher, alors les étages sont effectivement numérotés

de 0 à `nb_etages-1`. Une autre solution consiste à parcourir les numéros en ordre inverse :

Python

```
for etage in range(nb_etages, 0, -1):  
    # afficher l'étage
```

Cette écriture de `range()` utilisant trois paramètres permet de spécifier la valeur de départ, la valeur d'arrivée (qui n'est jamais atteinte) et le décalage à appliquer à chaque étape pour passer d'une valeur à la suivante.

C++

```
for(int etage = nb_etages-1;  
    etage >= 0;  
    --etage)  
{  
    // afficher l'étage  
}
```

Le compteur est cette fois décrémenté plutôt qu'incrémenté. Le symbole `>=` signifie "*supérieur ou égale*", qui est l'exacte négation de `<`.

La solution à la seconde difficulté est presque aussi simple : plutôt que d'afficher l'état de chaque étage de chaque tour, nous allons afficher l'état de chaque tour à chaque étage. Autrement dit, plutôt que d'examiner les tours l'une après l'autre et, pour chacune, d'examiner ses étages, nous allons examiner les étages et, pour chacun, examiner les tours à cet étage. C'est-à-dire que nous allons afficher l'état du jeu étage par étage. Implicitement, nous avons deux boucles imbriquées l'une dans l'autre :

Python

```
for etage in range(nb_etages, 0, -1):  
    for tour in range(3):  
        # afficher l'étage de la tour
```

C++

```
for(int etage = nb_etages;  
    etage >= 0;  
    --etage)  
{  
    for(int tour = 0; tour < 3; ++tour)  
    {  
        # afficher l'étage de la tour  
    }  
}
```

Il semble que nous ayons là le principe qui permettra à la fonction `Afficher_Jeu()` de fonctionner. Pour pouvoir afficher l'état du jeu, celle-ci a naturellement besoin de connaître, justement, l'état du jeu, qui se trouve mémorisé dans une structure de type `Hanoi`.

Elle devra donc recevoir un paramètre de ce type, en appliquant les principes que nous avons vus à la fin du chapitre précédent. Voici enfin cette fonction :

Python

```
def Afficher_Jeu(jeu):
    for etage in range(jeu.hauteur, 0, -1):
        for colonne in range(3):
            taille = jeu.tours[colonne].disques[etage-1]
            print " " + Chaine_Etage(taille, jeu.hauteur),
        print
```

C++

```
void Afficher_Jeu(const Hanoi& jeu)
{
    for(int etage = jeu.hauteur-1;
        etage >= 0;
        --etage)
    {
        for(int colonne = 0;
            colonne < 3;
            ++colonne)
        {
            int taille = jeu.tours[colonne].disques[etage];
            std::cout << " " << Chaine_Etage(taille, jeu.hauteur) << " ";
        }
        std::cout << std::endl;
    }
}
```

Remarquez l'utilisation d'un passage par référence constante, avec l'utilisation de `const` et de `&`.

Voyez comme la fonction `Chaine_Etage()` définie précédemment est mise à contribution. Chaque chaîne est, de plus, encadrée d'un espace de chaque côté, afin que la présentation soit un peu aérée.

7.3.2. Fonctions utilitaires

On désigne par ce terme des fonctions généralement très simples destinées à remplir des tâches assez élémentaires mais qui peuvent nécessiter plusieurs lignes de code. Pour simples qu'elles soient, ces portions de code présentes au milieu des instructions d'un algorithme compliqué sont comme une nuisance : en quelque sorte, elles brouillent la lecture de l'algorithme. On parle alors de *bruit*.

Ici, les fonctions permettant de demander les piquets de départ et d'arrivée sont typiquement des fonctions utilitaires. Les instructions qu'elles contiennent pourraient parfaitement être intégrées directement dans l'ébauche que nous avons construite, mais l'algorithme représenté par cette ébauche serait alors moins aisé à suivre. Voici l'une de ces deux fonctions :

Python	C++
<pre>def Demander_Tour_Depart(jeu): print "Tour de départ ?", tour_depart = int(raw_input()) return tour_depart - 1</pre>	<pre>int Demander_Tour_Depart(const Hanoi& jeu) { int tour_depart; std::cout << "Tour de départ ? "; std::cin >> tour_depart; return tour_depart - 1; }</pre>

Comme vous pouvez le constater, rien de bien terrible. La seule petite subtilité qui n'aura pas échappé au lecteur attentif est la valeur retournée : pourquoi soustraire 1 à la valeur donnée par l'utilisateur ?

Souvenez-vous que pour les langages Python et C++, le premier élément d'un tableau porte le numéro 0. Les tours sont donc numérotées de 0 à 2, les étages de 0 au nombre d'étages diminué de 1. Mais une telle numérotation est généralement perçue comme non naturelle : spontanément, on numéroterait plutôt les éléments d'un tableau en partant de 1, pas de 0.

Dans un souci de convivialité avec l'utilisateur, on lui permet ici d'utiliser une numérotation naturelle, en partant de 1. On s'attend donc à ce que l'utilisateur nous donne un nombre entre 1 et 3, qu'il nous faut transformer pour nos besoins internes de programmation en nombre entre 0 et 2. D'où la soustraction de 1 à la valeur donnée par l'utilisateur.

La fonction pour demander la tour d'arrivée est tout à fait similaire. Enfin, celle permettant de savoir si le jeu est terminé ou non se contente de vérifier si le nombre de disques sur le troisième piquet est égal au nombre total de disques, c'est-à-dire à la hauteur de la tour. Elle tient en une seule ligne :

Python	C++
<pre>def Est_Fini(jeu): return jeu.tours[2].nb_disques == \ jeu.hauteur</pre>	<pre>bool Est_Fini(const Hanoi& jeu) { return jeu.tours[2].nb_disques == jeu.hauteur; }</pre>

Répetons-le, le symbole pour tester l'égalité de deux valeurs est le double égal ==, pas le simple signe égal = utilisé pour l'affectation. La confusion des deux symboles est une erreur très courante, surtout lorsqu'on débute en programmation, donc prenez garde !

7.3.3. Vérification des entrées

Commençons par vérifier le numéro demandé pour le piquet de départ. Celui-ci n'est valide que s'il est compris entre 0 et 2 et qu'il y ait bien un disque sur ce piquet. La fonction est assez simple :

Python

```
def Tour_Depart_Valide(jeu,
                       depart):
    if ( depart < 0 ):
        return False
    if ( depart > 2 ):
        return False
    return \
        jeu.tours[depart].nb_disques > 0
```

C++

```
bool Tour_Depart_Valide(const Hanoi& jeu,
                       int depart)
{
    if ( (depart < 0) or
        (depart > 2) )
    {
        return false;
    }
    return
        jeu.tours[depart].nb_disques > 0;
}
```

Les deux versions de la fonction font exactement la même chose mais à l'aide de deux écritures différentes. La version en Python utilise deux tests pour vérifier l'intervalle du numéro de piquet donné, tandis que la version en C++ utilise un seul test composé. Le mot anglais *or* se traduit par *ou*, donc le test `(depart < 0) or (depart > 2)` vaut *vrai* (*true*) si l'un des deux membres au moins est vérifié. Notez que ces fonctions contiennent plusieurs instructions `return` : il est en effet possible de "sortir" d'une fonction en plusieurs endroits. Naturellement, dès qu'une instruction `return` est rencontrée et exécutée, les éventuelles instructions qui suivent ne sont pas exécutées. Ici, la fonction est assez simple, mais dans le cas d'une fonction complexe avec beaucoup d'embranchements (boucles ou alternatives), il est préférable de s'arranger pour n'avoir qu'une seule instruction `return`, placée à la fin : cela facilite la relecture et l'évolution du code.

Comme on pouvait s'y attendre, la vérification du piquet d'arrivée est un peu plus compliquée, car sa validité dépend du disque présent sur le piquet de départ et de celui éventuellement présent sur celui d'arrivée. C'est précisément à ce moment que nous allons assurer le respect de la règle qui dit qu'il est interdit de poser un disque sur un autre plus petit. Pour cela, nous devons trouver la taille des disques au sommet de chacune des deux tours. Sachant que la structure `Tour` contient un membre `nb_disques` qui est justement le nombre de disques présents sur un piquet, si cette valeur est supérieure à 0, alors le disque au sommet se trouve à l'indice `nb_disques-1` dans le tableau `disques` (la soustraction de 1 étant due, encore une fois, à la numérotation des éléments d'un tableau qui commence à 0).

À partir de tous ces éléments, vous devriez être à même d'écrire la fonction vérifiant le piquet d'arrivée. La voici, mais ne regardez pas la solution proposée avant d'avoir essayé !

Python

```
def Tour_Arrivee_Valide(jeu,
                        tour_depart,
                        tour_arrivee):
    if tour_arrivee < 0:
        return False
    if tour_arrivee > 2:
        return False
    nb_disques_arrivee = \
        jeu.tours[tour_arrivee].nb_disques
    if ( nb_disques_arrivee == 0 ):
        return True
    disque_arrivee = \
        jeu.tours[tour_arrivee]. \
            disques[nb_disques_arrivee-1]
    nb_disques_depart = \
        jeu.tours[tour_depart].nb_disques
    disque_depart = \
        jeu.tours[tour_depart]. \
            disques[nb_disques_depart-1]
    return \
        disque_arrivee > disque_depart
```

C++

```
bool Tour_Arrivee_Valide(const Hanoi& jeu,
                        int tour_depart,
                        int tour_arrivee)
{
    if ( (tour_arrivee < 0) or
        (tour_arrivee > 2) )
    {
        return false;
    }
    int nb_disques_arrivee =
        jeu.tours[tour_arrivee].nb_disques;
    if ( nb_disques_arrivee == 0 )
        return true;
    int disque_arrivee =
        jeu.tours[tour_arrivee].
            disques[nb_disques_arrivee-1];
    int nb_disques_depart =
        jeu.tours[tour_depart].nb_disques;
    int disque_depart =
        jeu.tours[tour_depart].
            disques[nb_disques_depart-1];
    return disque_arrivee > disque_depart;
}
```

La même chose aurait pu être écrite plus succinctement, sans utiliser les variables intermédiaires que sont `nb_disques_arrivee`, `disque_depart`, etc. Ces variables ont pour objet de clarifier l'écriture de chaque instruction. Sans elles, nous aurions quelque chose de relativement illisible, par exemple en Python :

```
return \
    jeu.tours[tour_depart].disques[jeu.tours[tour_depart].nb_disques-1] > \
    jeu.tours[tour_arrivee].disques[jeu.tours[tour_arrivee].nb_disques-1]
```

Pour finir, il serait pertinent de vérifier également le nombre de disques demandés par l'utilisateur au début du programme : on peut considérer qu'il faut au moins deux disques, mais pas plus d'une dizaine, le nombre optimal de déplacements pour dix disques étant de 1 023, ce qui est déjà beaucoup.

7.3.4. Déplacement d'un disque

Nous en arrivons au cœur du programme, la fonction qui va effectivement déplacer un disque d'un piquet vers un autre. Ou plutôt, simuler ce déplacement, qui va en réalité se traduire par des modifications opérées dans les tableaux de disques des tours.

Le principe est en fait assez comparable à celui utilisé dans la vérification du piquet d'arrivée. Sauf qu'au lieu de retourner une valeur booléenne nous allons modifier le contenu des instances de la structure `Tour` : l'instance qui correspond à la tour de départ perdra un disque, que celle qui correspond à la tour d'arrivée gagnera. Cette fonction ne devrait pas présenter trop de difficultés :

Python

```
def Deplacer_Disque(jeu, \
                    depart, \
                    arrivee):
    nb_disques_depart = \
        jeu.tours[depart].nb_disques
    nb_disques_arrivee = \
        jeu.tours[arrivee].nb_disques
    disque_depart = \
        jeu.tours[depart]. \
            disques[nb_disques_depart-1]
```

C++

```
void Deplacer_Disque(Hanoi& jeu,
                    int depart,
                    int arrivee)
{
    int nb_disques_depart =
        jeu.tours[depart].nb_disques;
    int nb_disques_arrivee =
        jeu.tours[arrivee].nb_disques;
    int disque_depart =
        jeu.tours[depart].
            disques[nb_disques_depart-1];
```



Python

```
jeu.tours[arrivee]. \
    disques[nb_disques_arrivee] = \
    disque_depart
jeu.tours[depart]. \
    disques[nb_disques_depart-1] = 0
jeu.tours[depart].nb_disques -= 1
jeu.tours[arrivee].nb_disques += 1
```

C++

```
jeu.tours[arrivee].
    disques[nb_disques_arrivee] = \
    disque_depart;
jeu.tours[depart].
    disques[nb_disques_depart-1] = 0;
jeu.tours[depart].nb_disques -= 1;
jeu.tours[arrivee].nb_disques += 1;
}
```

Remarquez l'antépénultième instruction, qui place la valeur 0 à l'endroit du tableau où se trouvait le disque qu'on vient de retirer. Cela permet de conserver le contenu de nos tableaux dans un état cohérent : un étage où il n'y a pas de disques est supposé contenir la valeur 0. Par ailleurs, la déclaration de la fonction dans la partie C++ n'utilise pas le qualificatif `const` devant le type `Hanoi` : en effet, le contenu de l'instance de `Hanoi` *doit* être modifié par la fonction (c'est son objet même), le paramètre ne peut donc pas être considéré comme invariable au sein de la fonction.

Le reste du programme est laissé en exercice au lecteur, tous les éléments sont désormais disponibles. Vous trouverez le code complet sur le site web de Pearson www.pearson.fr à la page dédiée à cet ouvrage.

7.4. Résumé de la démarche

Nous arrivons au terme de cette première partie. Le chapitre que vous venez de lire vous a donné un aperçu de la démarche usuelle lorsqu'on souhaite développer un programme qui n'est pas trivial :

- d'abord, modéliser les données du problème à traiter, en usant de structures et de tableaux pour rassembler les objets logiquement liés entre eux ;
- ensuite, concevoir les principaux algorithmes, en les exprimant littéralement, le cas échéant en dessinant des ordigrammes pour une présentation plus visuelle ;
- poursuivre la conception tant qu'il reste des étapes qui ne paraissent pas triviales ;
- enfin et seulement, écrire le code correspondant aux algorithmes précédemment conçus et aux fonctions prévues en cours d'analyse.

Naturellement, il n'est pas nécessaire de suivre un cheminement aussi précis lorsque vous avez à réaliser un petit programme rapide. Tout aussi naturellement, cette démarche est indispensable dans le cadre de tout projet important. Les grands logiciels que vous avez sans doute l'habitude de manipuler ont été conçus en suivant un paradigme analogue, généralement bien plus strict et détaillé.

Avec l'expérience, vous éprouverez de moins en moins le besoin de détailler les étapes d'un programme à mesure que vous analysez le problème et les algorithmes impliqués. Un programmeur ayant à peine une ou deux années de pratique serait capable de produire le programme que nous venons de réaliser en une heure, aboutissant à un résultat sans doute différent, mais assez ressemblant. N'oubliez pas, c'est en forgeant que l'on devient forgeron !

Vous constaterez également une forte tendance à vous jeter sur le clavier, pour écrire des kilomètres d'instructions qui vous viennent presque naturellement à l'esprit. Résistez à cette tentation : elle est souvent le plus court chemin vers l'inefficacité et la perte de temps. Il est de loin préférable de s'asseoir quelques minutes, avec un crayon et une feuille de papier, pour prévoir les étapes à réaliser – fût-ce au minimum – et généralement, à la fin, ce sera un gain de temps.

Partie 2

Développement logiciel

CHAPITRE 8. Le monde modélisé : la programmation orientée objet (POO)

CHAPITRE 9. Hanoï en objets

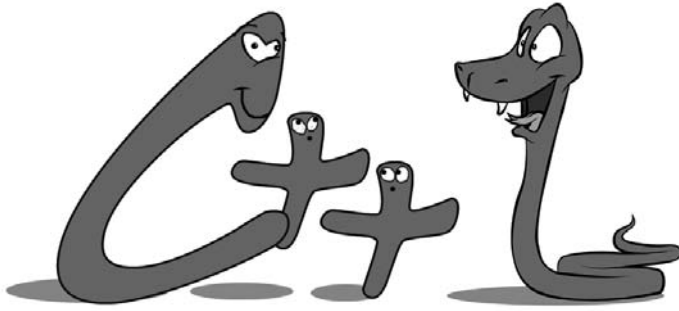
CHAPITRE 10. Mise en abîme : la récursivité

CHAPITRE 11. Spécificités propres au C++

CHAPITRE 12. Ouverture des fenêtres : programmation graphique

CHAPITRE 13. Stockage de masse : manipuler les fichiers

CHAPITRE 14. Le temps qui passe



8

Le monde modélisé : la programmation orientée objet (POO)

Au sommaire de ce chapitre :

- Les vertus de l'encapsulation
- La réutilisation par l'héritage
- Héritage et paramètres : le polymorphisme

Les programmes que nous avons créés jusqu'ici déclaraient des variables, définissaient des fonctions agissant sur ces variables, tout cela un peu pêle-mêle. Si c'est acceptable pour un petit programme, cela devient rapidement ingérable dès que l'on s'attaque à une application d'envergure. Aussi, le besoin d'organiser les choses, de regrouper ensemble les éléments liés logiquement, s'est-il rapidement imposé aux programmeurs.

Plusieurs techniques ont été élaborées pour permettre cette organisation. Par exemple, regrouper les fonctions en modules cohérents, auxquels un programme principal peut faire appel. Nous verrons plus tard cette notion de modules. Ici, nous allons étudier une autre technique, qui prend ses racines en 1967 avec le langage Simula, puis un peu plus tard avec SmallTalk : la *programmation par objets*.

Ce paradigme de programmation s'est véritablement imposé dans les années 1980, avec l'avènement du langage C++, extension du langage C, pour introduire dans ce dernier les principes de la programmation objet. D'autres langages ont également été étendus pour fournir les outils objet : Ada, Pascal, même Basic sont des exemples. Des langages objet dès leur création ont également été conçus, comme Java, Ruby, ou notre ami Python. Aujourd'hui, la programmation par objets est le paradigme de programmation le plus largement utilisé dans le monde.

Mais finalement, qu'est-ce qu'un objet ? Il s'agit avant tout d'un mécanisme d'*abstraction*, qui repose sur un principe fondamental. La *chose* manipulée par le programme, qu'il s'agisse de la modélisation d'un objet physique, comme une voiture, ou d'un objet abstrait, comme une fonction mathématique, est plus que la simple collection de ses caractéristiques. Pour modéliser correctement cette chose, il faut également prendre en compte les *méthodes*, les *règles* de son utilisation. Une voiture ne se réduit pas à une vitesse et une quantité d'essence : c'est également un accélérateur par lequel on agit sur ces deux quantités.

L'objet vise donc à fournir une représentation plus proche de la réalité de ce qu'on manipule : contenir en une même structure informatique, non seulement les données associées à la chose, mais aussi les moyens, les méthodes par lesquelles on peut agir dessus.

Un autre objectif essentiel de la programmation orientée objet est la *réutilisation du code*. C'est-à-dire, fournir des mécanismes facilitant la création de nouvelles structures à partir de plus anciennes, lesquelles sont adaptées, ajustées, *spécialisées*.

À la base de ces mécanismes se trouve la notion d'*héritage* ou de *dérivation* : un nouveau type objet est construit à partir d'un autre, dont il hérite les caractéristiques et les méthodes. On aboutit ainsi à une véritable taxonomie des "choses" que le programme doit manipuler, qui n'est pas sans rappeler celle des espèces animales.

8.1. Les vertus de l'encapsulation

Mais avant d'examiner le principe d'héritage, penchons-nous un instant sur celui de l'*encapsulation*. Ce terme étrange, qui évoque l'action de mettre une capsule (un bouchon) sur quelque chose, désigne une règle affirmant que, autant que possible, les données d'un objet ne doivent pas être directement accessibles. C'est-à-dire que l'utilisateur d'un objet, que cela soit vous-même ou un autre programmeur, ne doit pas pouvoir modifier (ni même consulter) les valeurs des variables contenues dans l'objet sans que cet accès soit contrôlé par l'objet lui-même. Si cela peut sembler bien contraignant au premier abord, ce principe s'est imposé de lui-même comme une nécessité absolue pour obtenir des programmes fiables, fonctionnant correctement et donnant des résultats cohérents.

Imaginons que vous vouliez modéliser un véhicule motorisé, comme une voiture, un bateau ou un vaisseau spatial. Pour simplifier les choses à l'extrême, nous n'allons considérer que deux caractéristiques de ce véhicule : sa vitesse et la quantité de carburant qu'il lui reste à un instant donné. Au moment de sa "création", notre véhicule est statique (sa vitesse est nulle) et n'emporte pas de carburant. Avec ce que nous avons vu jusqu'ici, nous pourrions modéliser cela ainsi :

Python

```
class Vehicule:
    def __init__(self):
        self.vitesse = 0.0
        self.carburant = 0.0
```

C++

```
struct Vehicule
{
    Vehicule()
    {
        this->vitesse = 0.0;
        this->carburant = 0.0;
    }
    double vitesse;
    double carburant;
};
```

Nous pouvons maintenant créer des véhicules, de n'importe quel type (croyons-nous). L'utilisation de cette structure paraît tout à fait triviale. Pour faire le plein d'un véhicule, rien de plus simple :

Python	C++
<pre>moto = Vehicule() moto.carburant = 100.0</pre>	<pre>Vehicule moto; moto.carburant = 100.0;</pre>

Puis nous pouvons "accélérer", sans problème, simplement en modifiant la vitesse de notre moto :

Python	C++
<pre>moto.vitesse = 90.0</pre>	<pre>moto.vitesse = 90.0;</pre>

Tout va bien. Nous pouvons continuer d'accélérer ainsi ou ralentir. La magie de tout cela, c'est que la quantité de carburant ne change pas.

Et c'est bien là le problème.

Nous venons de modéliser le véhicule idéal, capable de changer de vitesse à volonté sans consommer une goutte de carburant. Vous admettez que nous sommes bien loin de la réalité : lorsque vous conduisez une voiture, vous ne pouvez accélérer que par l'intermédiaire de la pédale d'accélérateur, action qui se traduit forcément par une diminution de la quantité disponible de carburant. Jusqu'à la panne sèche.

Autrement dit : notre modèle offre toutes les possibilités pour donner des résultats incohérents.

Le problème vient du fait que nous pouvons modifier directement les données de nos objets. Et pour l'heure, il ne saurait en être autrement, nous n'avons guère le choix. L'idéal serait de disposer d'une *méthode*, nommée `Acceleration()`, qui se chargerait elle-même de modifier les valeurs à la fois de la vitesse et du carburant restant.

Voici une solution possible :

Python

```
class Vehicule:
    def __init__(self):
        self.vitesse = 0.0
        self.carburant = 0.0
    def Remplir(self, quantite):
        self.carburant += quantite
    def Accelerer(self, delta):
        self.vitesse += delta
        self.carburant -= delta/10.0
    def Vitesse(self):
        return self.vitesse
    def Carburant(self):
        return self.carburant
# ...
moto = Vehicule()
moto.Remplir(50.0)
moto.Accelerer(90.0)
```

La notion de données privées en Python n'existe pas réellement, c'est-à-dire qu'il n'est pas "naturel" de chercher à limiter l'accès aux données d'une classe.

C++

```
class Vehicule
{
public:
    Vehicule():
        vitesse(0.0),
        carburant(0.0)
    { }
    void Remplir(double quantite)
    {
        this->carburant += quantite;
    }
    virtual
    void Accelerer(double delta)
    {
        this->vitesse += delta;
        this->carburant -= delta/10.0;
    }
    double Vitesse() const
    { return this->vitesse; }
    double Carburant() const
    { return this->carburant; }
protected:
    double vitesse;
    double carburant;
};
/* ... */
Vehicule moto;
moto.Remplir(50.0);
moto.Accelerer(90.0);
```

En C++, on utilise de préférence le mot clé `class` plutôt que `struct`. Tout ce qui se trouve dans une classe C++ est normalement



Python	C++
<p>Une technique consistant à faire précéder le nom d'une donnée par un double caractère de soulignement existe (par exemple, <code>self.__carburant</code>), mais elle n'est guère pratique et présente plus d'inconvénients que d'avantages.</p> <p>La protection des données en Python est donc essentiellement une affaire de convention entre le concepteur d'une classe et l'utilisateur de cette classe.</p>	<p>privé (<i>private</i>), c'est-à-dire que l'utilisateur de la classe ne peut y accéder. Les mots clés <code>public</code> et <code>protected</code> permettent d'indiquer si ce qui les suit est, respectivement, librement accessible ou d'un accès restreint à l'intérieur de la classe.</p> <p>Le sens du mot <code>virtual</code> qui précède la méthode <code>Accelerer()</code> sera explicité à la dernière section de ce chapitre.</p>

Notez les changements subtils (et moins subtils) par rapport aux déclarations précédentes. Désormais, lors de l'utilisation de la classe `Vehicule`, les données qui lui sont propres comme le carburant et la vitesse ne sont plus directement modifiables : il est nécessaire de passer par l'intermédiaire de *méthodes*, ici `Remplir()` et `Accelerer()`, pour altérer leurs valeurs. De cette façon, il est possible d'assurer la cohérence des données de la classe et ainsi d'éviter d'obtenir des résultats fantaisistes.

Les méthodes `Vitesse()` et `Carburant()` sont ce qu'on appelle des *accesseurs* : leur seule et unique vocation est de donner la valeur des données correspondantes, celles-ci n'étant pas accessibles puisque privées. Dans notre situation, cela se résume à retourner ce qui est contenu dans les données membres de l'instance. Mais cela peut également résulter d'un calcul plus ou moins complexe, comme nous allons le voir bientôt. Remarquez la présence du mot clé `const` en C++ : il indique que ces méthodes ne vont pas modifier le contenu de l'instance (c'est-à-dire modifier les valeurs des données membres). C'est une indication donnée au compilateur, qui va vérifier que cette règle est effectivement suivie. Si, par mégarde, vous tentez de modifier une donnée au sein d'une méthode qualifiée par le mot clé `const`, le compilateur vous le signalera par un message d'erreur. Ce sera alors très certainement le signe d'un problème de conception.

Ce principe consistant à limiter l'accès aux données d'une classe est donc désigné par le terme d'*encapsulation*. D'une manière générale, il est communément admis que toutes les données d'une classe devraient être privées, la classe fournissant des méthodes spécialisées, des accesseurs, à la fois pour obtenir les valeurs des données et pour les changer. De ce fait, les changements sont contrôlés, ce qui doit normalement induire une meilleure cohérence de l'ensemble.

Enfin, notre modèle est ici extrêmement simple. Dans la méthode `Accelerer()`, on ne devrait effectivement modifier la vitesse que s'il reste suffisamment de carburant pour cela. Le code est toujours potentiellement incohérent, mais au moins la structure est plus robuste. Le lecteur est invité à modifier cette méthode pour la rendre plus réaliste.

8.2. La réutilisation par l'héritage

Notre véhicule nous a permis de représenter une moto. Et nous pourrions représenter de même une voiture, un camion... Mais considérez le cas d'un avion, voire d'un sous-marin. Il serait intéressant de disposer de données supplémentaires, comme l'altitude ou la profondeur, ainsi que de méthodes supplémentaires, comme monter ou descendre. Ce sont des véhicules bien différents d'une moto, mais cela reste des véhicules.

Naturellement, nous pourrions simplement dupliquer le code de notre classe `Vehicule`, par un copier-coller trop facile, et ajouter ce qu'il nous manque. Mais, d'une part, l'expérience montre que la duplication de code est source d'erreurs et de dysfonctionnements (car on copie également les bogues contenus dans le code), d'autre part, il pourrait être intéressant de conserver la sémantique qui affirme qu'une moto, un avion et un sous-marin sont tous des véhicules. Ils font tous partie d'une même famille.

La solution consiste à utiliser l'*héritage*, c'est-à-dire à définir de nouvelles classes à partir de classes existantes. Voici à quoi pourrait ressembler une classe `Avion` :

Python

```
class Avion(Vehicule):
    def __init__(self):
        Vehicule.__init__(self)
        self.altitude = 0.0
    def Monter(self, delta):
        self.altitude += delta
    def Descendre(self, delta):
        self.altitude -= delta
    def Altitude(self):
        return self.altitude
    def Accelerer(self, delta):
        print "Avion accélère..."
        self.vitesse += delta
```

C++

```
class Avion: public Vehicule
{
public:
    Avion():
        Vehicule(),
        altitude(0.0)
    { }
    void Monter(double delta)
    {
        this->altitude += delta;
    }
}
```



Python

```
self.carburant -= \
    delta * \
    ((self.Altitude()+1.0)/1000.0)
```

C++

```
void Descendre(double delta)
{
    this->altitude -= delta;
}
virtual
void Accelerer(double delta)
{
    std::cout <<
        "Avion accélère...\n";
    this->vitesse += delta;
    this->carburant -= \
        delta *
        ((this->Altitude()+1.0)/1000.0);
}
double Altitude() const
{ return this->altitude; }
protected:
    double altitude;
};
```

Voyez comment est déclarée la nouvelle classe :

- En Python, `class Avion(Vehicule)` indique que la classe Avion *dérive* de la classe Vehicule, ou encore qu'elle en *hérite*.
- En C++, `class Avion: public Vehicule` indique que la classe Avion *dérive* de la classe Vehicule, ou encore qu'elle en *hérite*.

Tout cela signifie que la classe Avion "contient" tout ce que contient la classe Vehicule. Constatez la puissance du procédé : nous avons une nouvelle classe, offrant au total deux fois plus de méthodes que la précédente, mais sans devoir écrire deux fois plus de code. Le lecteur est invité à écrire le code nécessaire à une classe `Sous_Marin`, tout à fait similaire, mais qui aurait une profondeur plutôt qu'une altitude. Ou alors, imaginez une classe `Poisson_Volant`, ayant à la fois une altitude et une profondeur... ce qui imposerait de savoir à tout moment laquelle des deux valeurs est pertinente. Le monde naturel est une source inépuisable de surprises.

Remarquez tout de même que nous avons *surdéfini* la méthode `Accelerer()`. Cela signifie que nous avons défini dans la nouvelle classe une méthode de même nom, ayant le même aspect (on dit plutôt la même *signature*), c'est-à-dire prenant les mêmes paramètres dans le même ordre. Si nous déclarons une instance d'`Avion` et que nous invoquons la méthode `Accelerer()`, c'est bien celle surdéfinie dans `Avion` qui sera utilisée, pas celle venant de la classe mère `Vehicule`.

Par exemple :

Python	C++
<pre>robin = Avion() robin.Remplir(50.0) print robin.Carburant() robin.Accelerer(100.0) print robin.Carburant()</pre>	<pre>int main(int argc, char* argv[]) { Avion robin; robin.Remplir(50.0); std::cout << robin.Carburant() << std::endl; robin.Accelerer(100.0); std::cout << robin.Carburant() << std::endl; return 0; }</pre>

Les méthodes `Remplir()` et `Carburant()` dans les extraits précédents viennent directement de la classe `Vehicule`, bien qu'elles soient appliquées à une instance de la classe `Avion`. On dit que la méthode `Remplir()` est héritée de la classe `Vehicule`, ou encore que la classe `Avion` hérite de la méthode `Remplir()` à partir de la classe `Vehicule`. Par contre, `Accelerer()` est bien celle définie dans `Avion`.

Du côté du C++, il est préférable que toute méthode pouvant être ainsi surdéfinie dans une classe fille soit qualifiée par le mot clé `virtual`, comme c'est le cas ici. Nous allons voir immédiatement pourquoi.

8.3. Héritage et paramètres : le polymorphisme

En utilisant l'héritage, nous pouvons créer toute une famille de type de véhicules. Remarquez que nous n'avons défini qu'une méthode `Accelerer()`, pas de méthode permettant de définir exactement une vitesse. Ce n'est pas très grave : à partir d'une vitesse donnée, on peut atteindre une autre vitesse en accélérant ou en freinant (freiner équivalant à une accélération négative). On peut alors imaginer une fonction ressemblant à ceci :

Python

```
def Atteindre_Vitesse(un_vehicule, une_vitesse):  
    delta = une_vitesse - \  
            un_vehicule.Vitesse()  
    un_vehicule.Accelerer(delta)
```

C++

```
void Atteindre_Vitesse(Vehicule& un_vehicule,  
                      double une_vitesse)  
{  
    double delta = une_vitesse -  
                  un_vehicule.Vitesse();  
    un_vehicule.Accelerer(delta);  
}
```

Remarquez la présence de l'éperluette : on passe bien une référence sur une instance de `Vehicule`.

La question que l'on peut alors se poser est, si ce que nous venons d'écrire fonctionne pour une instance de `Vehicule`, comment faire pour une instance d'`Avion` ou pour toute autre classe dérivée de `Vehicule` ?

La bonne nouvelle est que le comportement est naturel, c'est-à-dire qu'il correspond à ce qu'on attend effectivement.

Par exemple :

Python	C++
<pre>moto = Vehicule() moto.Remplir(50.0) robin = Avion() robin.Remplir(50.0) Atteindre_Vitesse(moto, 90.0) Atteindre_Vitesse(robin, 100.0)</pre>	<pre>Vehicule moto; moto.Remplir(50.0); Avion robin; robin.Remplir(50.0); Atteindre_Vitesse(moto, 90.0); Atteindre_Vitesse(robin, 100.0);</pre>

Nous créons deux véhicules, l'un "générique" (la moto), l'autre un modèle d'avion. Puis nous cherchons pour chacun d'eux à atteindre une vitesse donnée, en utilisant la fonction `Atteindre_Vitesse()`.

En son sein, celle-ci va faire appel à la méthode `Accelerer()` présente dans chacune des classes `Vehicule` et `Avion`. Ce qui se passe ensuite est très naturel : si la fonction reçoit en paramètre une instance de `Vehicule`, alors c'est la méthode `Accelerer()` de `Vehicule` qui sera invoquée ; si elle reçoit en paramètre une instance d'`Avion`, alors c'est la méthode `Accelerer()` d'`Avion` qui sera invoquée. Si vous exécutez ce programme, vous obtiendrez l'affichage :

```
Vehicule accélère...
Avion accélère...
```

Ce qui est la preuve que c'est la "bonne" méthode qui est utilisée.

Cela peut sembler évident en Python, mais beaucoup moins en C++ : le type du paramètre passé à la fonction `Atteindre_Vitesse()` est une référence à une instance de `Vehicule`... et c'est justement en raison de cette référence que cela fonctionne, associé au fait que la classe `Avion` dérive de la classe `Vehicule`.

La variable `robin` est une instance d'`Avion`. Mais comme `Avion` dérive de `Vehicule`, `robin` est *aussi* une instance de `Vehicule`. On peut donc la passer à la fonction `Atteindre_Vitesse()`. Conséquence de la référence, un phénomène un peu étrange survient : la fonction "croit" recevoir une instance de `Vehicule`, mais c'est bel et bien une instance d'`Avion` qu'elle manipule, sans même le "savoir". Les méthodes invoquées dans la fonction ne peuvent qu'être des méthodes présentes dans la classe `Vehicule`, mais ce sont bien les méthodes de la classe `Avion` qui seront invoquées. S'il a été nécessaire de surdéfinir une méthode, comme c'est le cas de la méthode `Accelerer()`, la présence du mot clé `virtual` garantit que, dans cette situation, c'est la

méthode surdéfinie qui sera invoquée. C'est pourquoi toute méthode d'une classe destinée à être surdéfinie dans une classe dérivée doit être qualifiée par le mot clé `virtual`.

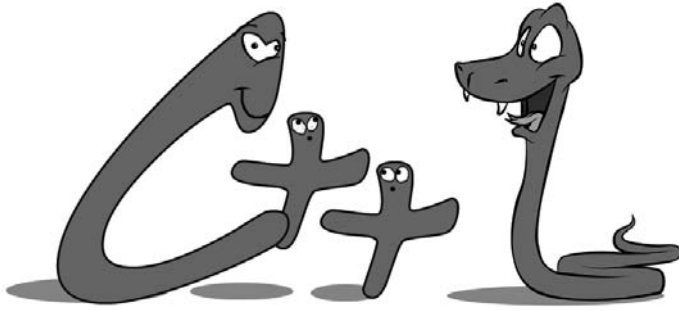
Faites les expériences suivantes. D'abord, retirez le mot clé `virtual` précédant la déclaration de la méthode `Accelerer()` dans la classe `Vehicule`, puis compilez et exécutez à nouveau le programme. Ensuite, remplacez le mot clé `virtual`, mais retirez l'éperluette dans la déclaration de la fonction `Atteindre_Vitesse()`, puis compilez et exécutez à nouveau le programme. Vous obtiendrez ce résultat :

```
Vehicule accélère...  
Vehicule accélère...
```

Seule la méthode `Accelerer()` de la classe `Vehicule` est invoquée.

Avec la combinaison de `virtual` et de l'éperluette, on peut dire que le paramètre `un_vehicule` passé à la fonction change d'aspect selon la nature *réelle* de la variable qui lui est donnée. Ce phénomène est désigné par le terme de *polymorphisme*, qui signifie littéralement "pouvant prendre plusieurs formes". Pour la fonction `Atteindre_Vitesse()`, le paramètre `un_vehicule` est polymorphe : selon le type de la variable qui lui est donnée, il aura en réalité tantôt la forme d'un `Vehicule`, tantôt la forme d'un `Avion`, ou toute autre forme définie par une classe dérivant de `Vehicule` (ou même d'`Avion`).

En langage Python, le polymorphisme est implicite et automatique. En langage C++, il nécessite l'emploi des références pour les paramètres et du mot clé `virtual` pour les méthodes susceptibles d'être surdéfinies dans des classes dérivées. Il s'agit là d'un mécanisme fondamental et extrêmement puissant de la programmation orientée objet, comme nous allons le constater bientôt.



9

Hanoï en objets

Au sommaire de ce chapitre :

- Des tours complètes
- Un jeu autonome
- L'interface du jeu
- Le programme principal

Il est maintenant temps que transformer le programme du jeu des Tours de Hanoi, créé au Chapitre 7, afin de lui appliquer ce que nous venons de voir. Toutes les techniques ne seront pas utilisées immédiatement, mais cela va vous donner une vue d'ensemble de ce à quoi peut ressembler un programme objet complet.

9.1. Des tours complètes

Pour commencer, voyons la classe nous permettant de représenter une tour, ou plutôt l'aiguille portant la tour. La structure présentée au Chapitre 7 se limitait au seul constructeur. Nous avons maintenant la possibilité de faire bien mieux, en ajoutant de nombreuses petites méthodes pratiques.

Python

```
class Tour:
    def __init__(self, taille_max):
        self.nb_disques = 0
        self.disques = \
            [0 for e in range(taille_max)]
    def Est_Vide(self):
        return self.nb_disques == 0
    def Est_Pleine(self):
        return \
            self.nb_disques == \
            len(self.disques)
    def Sommet(self):
        if self.Est_Vide():
            return 0
        else:
            return self.disques [self.nb_disques-1]
    def Disque_Etage(self, etage):
        if ( self.Est_Vide() or \
            (etage >= self.nb_disques) ):
            return 0
        else:
            return self.disques[etage]
    def Peut_Empiler(self, disque):
        if self.Est_Vide():
            return True
        else:
            return (self.Sommet() > disque)
    def Empiler(self, disque):
        self.disques[self.nb_disques] = disque
        self.nb_disques += 1
    def Depiler(self):
        self.nb_disques -= 1
        self.disques[self.nb_disques] = 0
```

Pour Python, il importe de bien respecter l'indentation : c'est en effet elle qui va assurer, d'une part, que la définition d'une méthode appartient à une classe, d'autre part, que des instructions appartiennent à une méthode.

Par ailleurs, si vous comparez ce code avec la version C++ qui suit, vous constatez qu'il n'existe pas de signes particuliers pour indiquer qu'une méthode ne modifie pas l'instance ou qu'une donnée ne doit pas être utilisée directement. Python ne possède pas, en effet, de moyen simple et immédiat pour mettre en œuvre ces techniques normalement destinées à améliorer la robustesse du programme.

En contrepartie, le code est plus simple et plus "compact".

C++

```

class Tour
{
public:
    Tour(int taille_max):
        nb_disques(0),
        disques(taille_max, 0)
    { }
    bool Est_Vide() const
    {
        return this->nb_disques == 0;
    }
    bool Est_Pleine() const
    {
        return
            this->nb_disques ==
            this->disques.size();
    }
    int Sommet() const
    {
        if ( this->Est_Vide() )
            return 0;
        else
            return
                this->disques [this->nb_disques-1];
    }
    int Disque_Etage(int etage) const
    {
        if ( this->Est_Vide() or
            (etage >= this->nb_disques) )
            return 0;
        else
            return this->disques[etage];
    }
    bool Peut_Empiler(int disque) const
    {
        if ( this->Est_Vide() )
            return true;
        else
        {
            if ( this->Est_Pleine() )
                return false;
            else
                return (this->Sommet() > disque);
        }
    }
    void Empiler(int disque)
    {
        this->disques[this->nb_disques] = disque;
        this->nb_disques += 1;
    }
}

```

Côté C++, remarquez comme les méthodes purement informatives, comme `Est_Vide()`, sont toutes qualifiées par le mot clé `const`.

Comme nous l'avons vu au chapitre précédent, cela indique que ces méthodes ne doivent pas modifier l'instance sur laquelle elles sont appliquées. Grâce à quoi, des erreurs d'inattention peuvent être détectées, comme utiliser un simple signe `=` (affectation) à la place du signe `==` (comparaison d'égalité).

Par ailleurs, les données de la classe sont placées dans une section `private` (*privée*), contrairement à la section `protected` (*protégée*) vue au chapitre précédent. La conséquence est une meilleure protection de ces données : non seulement, elles ne sont pas directement utilisables à l'extérieur de la classe mais, en plus, même une classe dérivant de celle-ci ne pourrait pas les manipuler. On aboutit ainsi à une encapsulation complète de ces données.

Enfin, notez qu'il est fait une certaine économie sur les accolades, notamment dans les structures alternatives (`if`). En l'absence d'accolades, seule l'unique instruction suivant immédiatement le test introduit par `if` est exécutée si la condition est vérifiée. Ainsi :

```

if ( une_condition )
    une_instruction;
    une_autre_instruction;

```



C++	
<pre> void Depiler() { this->nb_disques -= 1; this->disques[this->nb_disques] = 0; } private: int nb_disques; std::vector<int> disques; }; // class Tour </pre>	<p>est équivalent à :</p> <pre> if (une_condition) { une_instruction; } une_autre_instruction; </pre> <p>Rappelez-vous que l'indentation en C++ est purement décorative et n'a aucune valeur pour le langage.</p>

Les méthodes ajoutées dans cette classe devraient parler d'elles-mêmes. Décrivons-les brièvement :

- `Est_Vide()` permet de savoir si une tour est vide ou non, c'est-à-dire si elle contient au moins un disque ou non.
- `Est_Pleine()`, symétrique de la précédente, permet de savoir si tous les étages disponibles sont occupés.
- `Sommet()` donne la taille du disque présent au sommet de la tour, `Disque_Etage()` donne la taille du disque présent à un étage donné ; une taille nulle signifie qu'aucun disque n'est présent à l'endroit demandé.
- `Peut_Empiler()` fait partie de la modélisation de la règle du jeu : elle indique si la tour peut recevoir la taille de disque qu'on lui donne en paramètre.
- `Empiler()` et `Depiler()` permettent, respectivement, d'ajouter ou de retirer un disque au sommet de la tour. Elles sont ici particulièrement sommaires : dans un véritable programme, il faudrait ajouter des tests pour vérifier que ces opérations sont légitimes (par exemple, on ne peut dépiler un disque d'une tour vide). Ces tests sont laissés en exercice au lecteur.

Peut-être toutes ces méthodes assez élémentaires vous semblent-elles inutiles. Après tout, elles contiennent si peu d'instructions qu'on pourrait tout aussi bien les réécrire lorsque le besoin se présente. Mais voyons comment, maintenant, nous allons modéliser le jeu lui-même.

9.2. Un jeu autonome

Vous l'aurez compris, nous allons maintenant étendre la structure `Hanoi` du Chapitre 7, comme nous avons étendu la structure `Tour`. Mais ne perdez pas de vue la raison d'être de cette structure : modéliser le jeu des Tours de Hanoï... et rien d'autre. En particulier, cela signifie que nous n'allons y intégrer aucune méthode réalisant un affichage quelconque : nous allons bel et bien créer un *moteur* de jeu, parfaitement indépendant de la manière dont il sera présenté à l'utilisateur. Nous verrons plus loin l'importance que cela aura.

Vous retrouverez dans cette classe certaines des fonctions que nous avons créées dans notre premier programme de jeu, naturellement adaptées au nouveau contexte. Un changement important réside dans la création d'une méthode `Initialiser()`, dont le but est tout simplement de préparer le jeu dans un état initial en fonction d'un certain nombre de disques demandés.

Voici donc notre nouvelle classe `Tour` :

Python

```
class Hanoi:
    def __init__(self):
        self.hauteur_max = 0
        self.tours = []
    def Initialiser(self, nb_disques):
        self.tours = [Tour(nb_disques), \
                       Tour(nb_disques), \
                       Tour(nb_disques)]
        for disque in range(nb_disques, 0, -1):
            self.tours[0].Empiler(disque)
        self.hauteur_max = nb_disques
    def Hauteur(self):
        return self.hauteur_max
    def Une_Tour(self, num_tour):
        return self.tours[num_tour]
```

Dans le constructeur, l'écriture `self.tours = []` permet de déclarer la donnée membre `tours` comme un tableau, initialement vide (ne contenant aucun élément).

Au sein de la méthode `Initialiser()`, le tableau des tours est tout simplement recréé.

Un nouveau tableau, dont on donne explicitement la liste des éléments entre crochets, lui est affecté.



Python

```

def Tour_Depart_Valide(self, depart):
    if (depart < 0) or (depart > 2):
        return False
    return (not self.tours[depart].Est_Vide())
def Tour_Arrivee_Valide(self, depart, arrivee):
    if (arrivee < 0) or (arrivee > 2):
        return false;
    disque_depart = self.tours[depart].Sommet()
    return
        self.tours[arrivee].Peut_Empiler(disque_depart)
def Deplacer_Disque(self, depart, arrivee):
    disque = self.tours[depart].Sommet()
    self.tours[depart].Depiler()
    self.tours[arrivee].Empiler(disque)
def Jeu_Termine(self):
    return self.tours[2].Est_Pleine()

```

C++

```

class Hanoi
{
public:
    Hanoi():
        hauteur_max(0),
        tours()
    {
    }
    void Initialiser(int nb_disques)
    {
        this->tours.clear();
        this->tours.resize(3, Tour(nb_disques));
        for(int disque = nb_disques;
            disque > 0;
            --disque)
        {
            this->tours[0].Empiler(disque);
        }
        this->hauteur_max = nb_disques;
    }
    int Hauteur() const
    {
        return this->hauteur_max;
    }
    const Tour& Une_Tour(int i) const

```

Au sein de la méthode Initialiser(), on utilise deux méthodes propres au type `std::vector`.

`clear()` a pour effet de vider le tableau, c'est-à-dire de détruire tous les éléments qu'il contient : tout se passe alors comme si on avait un tableau vide après son application.

`resize()`, au contraire, permet de spécifier la taille du tableau, c'est-à-dire le nombre d'éléments qu'il pourra contenir.

C++

```

{
    return this->tours[i];
}
bool Tour_Depart_Valide(int depart) const
{
    if ( (depart < 0) or (depart > 2) )
    {
        return false;
    }
    return (not this->tours[depart].Est_Vide());
}

bool Tour_Arrivee_Valide(int depart,
                        int arrivee) const
{
    if ( (arrivee < 0) or (arrivee > 2) )
    {
        return false;
    }
    int disque_depart =
        this->tours[depart].Sommet();
    return
        this->tours[arrivee].Peut_Empiler(disque_depart);
}

void Deplacer_Disque(int depart,
                    int arrivee)
{
    int disque = this->tours[depart].Sommet();
    this->tours[depart].Depiler();
    this->tours[arrivee].Empiler(disque);
}

bool Jeu_Termine() const
{
    return this->tours[2].Est_Pleine();
}

private:
    int hauteur_max;
    std::vector<Tour> tours;
}; // class Hanoi

```

Le deuxième paramètre de `resize()` est une valeur initiale à recopier dans chacun des nouveaux éléments du tableau. Vous pouvez constater que `resize()` présente un aspect tout à fait analogue à celui que nous avons rencontré jusqu'ici lors de la déclaration d'un tableau : une taille suivie d'une valeur initiale.

Comparez maintenant l'écriture des méthodes telles que `Tour_Arrivee_Valide()`, avec celle que nous avons vue dans les fonctions éponymes du Chapitre 7. Vous pouvez constater que le code donné ici est plus lisible : l'utilisation des méthodes

fournies par la classe `Tour` permet d'obtenir une écriture plus compacte et plus expressive. C'est l'un des avantages "annexes" de la programmation orientée objet : l'obtention de programmes plus lisibles car plus expressifs, donc plus aisés à maintenir dans le temps et à faire évoluer.

La méthode `Une_Tour()` est un peu particulière : elle donne accès à l'une des tours du jeu en cours aux utilisateurs de la classe `Hanoi`. Remarquer en C++ l'utilisation du mot clé `const` : la référence sur une tour renvoyée est ainsi qualifiée pour garantir que l'utilisateur ne pourra pas modifier cette tour. En effet, normalement seule l'instance de la classe `Hanoi` doit pouvoir modifier les tours, seule façon d'apporter une garantie (relative) à la robustesse et la cohérence de l'ensemble. Incidemment, cela signifie qu'à partir de cette instance constante de la classe `Tour` obtenue, l'utilisateur ne pourra utiliser que des méthodes de `Tour` elles-mêmes qualifiées par `const` : les seules garantissant l'intégrité des données de la tour.

9.3. L'interface du jeu

Répetons-le, un principe essentiel de la programmation consiste à bien séparer l'interface d'un programme, c'est-à-dire la partie chargée d'interagir avec l'utilisateur, du moteur de calcul, c'est-à-dire la partie contenant effectivement la mise en œuvre des règles et contraintes du problème à résoudre. L'importance de cette séparation sera pleinement démontrée dans quelques chapitres. Pour l'heure, nous allons utiliser la programmation par objet pour traduire cette distinction, en créant une classe `Hanoi_Texte` chargée précisément de (presque) tous les aspects de l'affichage du programme. Cette classe est ici donnée dans son intégralité, afin que vous puissiez comparer son contenu avec celui des fonctions créées au Chapitre 7.

Python

```
class Hanoi_Texte:
    def __init__(self, jeu):
        self.jeu_hanoi = jeu
        self.nb_mouvements = 0
    def Jeu(self):
        return self.jeu_hanoi
    def Demander_Tour_Depart(self):
        print "Tour de départ ?",
        tour_depart = int(raw_input())
        return tour_depart-1
```



Python

```
def Demander_Tour_Arrivee(self):
    print "Tour d'arrivée ?",
    tour_arrivee = int(raw_input())
    return tour_arrivee-1
def Afficher_Jeu(self):
    for etage in \
        range(self.Jeu().Hauteur(), -1, -1):
        for tour in range(3):
            print " " + \
                self.Chaine_Etage(tour, etage),
            print
def Nb_Mouvements(self):
    return self.nb_mouvements;
def Etape(self):
    tour_depart = self.Demander_Tour_Depart()
    if self.Jeu().Tour_Depart_Valide (tour_depart):
        tour_arrivee =
            self.Demander_Tour_Arrivee()
        if self.Jeu().Tour_Arrivee_Valid (tour_depart, \
            tour_arrivee):
            self.Jeu().Deplacer_Disque(tour_depart, \
                tour_arrivee)
            self.nb_mouvements += 1
        else:
            print "Arrivée invalide !"
    else:
        print "Départ invalide !"
def Chaine_Etage(self, tour, etage):
    largeur_max = self.Jeu().Hauteur()
    disque =
        self.Jeu().Une_Tour(tour).Disque_Etage(etage)
    espace = ' '*(largeur_max-disque)
    chaine = ""
    if ( disque > 0 ):
        ligne = '='*disque
        chaine = espace + "<" + ligne + "!" +
            ligne + ">" + espace
    else:
        chaine = espace + " ! " + espace
    return chaine
```


C++

```

class Hanoi_Texte
{
public:
    Hanoi_Texte(Hanoi& jeu):
        jeu_hanoi(jeu),
        nb_mouvements(0)
    {
    }
    Hanoi& Jeu()
    {
        return this->jeu_hanoi;
    }
    const Hanoi& Jeu() const
    {
        return this->jeu_hanoi;
    }
    int Demander_Tour_Depart()
    {
        int tour_depart;
        std::cout << "Tour de départ ? ";
        std::cin >> tour_depart;
        return tour_depart - 1;
    }
    int Demander_Tour_Arrivee()
    {
        int tour_arrivee;
        std::cout << "Tour d'arrivée ? ";
        std::cin >> tour_arrivee;
        return tour_arrivee - 1;
    }
    void Afficher_Jeu() const
    {
        for(int etage = this->Jeu().Hauteur();
            etage >= 0;
            --etage)
        {
            for(int tour = 0; tour < 3; ++tour)
            {
                std::cout << " "
                    << this->Chaine_Etage(tour, etage)
                    << " ";
            }
            std::cout << std::endl;
        }
    }
}

```

C++

```
int Nb_Mouvements() const
{
    return this->nb_mouvements;
}
void Etape()
{
    int tour_depart = this->Demander_Tour_Depart();
    if ( this->Jeu().Tour_Depart_Valide(tour_depart) )
    {
        int tour_arrivee = \
            this->Demander_Tour_Arrivee();
        if (
            this->Jeu().Tour_Arrivee_Valide(tour_depart,
            tour_arrivee) )
        {
            this->Jeu().Deplacer_Disque(tour_depart,
            tour_arrivee);
            this->nb_mouvements += 1;
        }
        else
        {
            std::cout << "Arrivée invalide !\n";
        }
    }
    else
    {
        std::cout << "Départ invalide !\n";
    }
}
std::string Chaine_Etape(int tour,
                        int etage)const
{
    int largeur_max = this->Jeu().Hauteur();
    int disque =
        this->Jeu().Une_Tour(tour).Disque_Etage(etage);
    std::string espace(largeur_max-disque, ' ');
    std::string chaine;
    if ( disque > 0 )
    {
        std::string ligne(disque, '=');
        chaine = espace + "<" + ligne + "!" +
            ligne + ">" + espace;
    }
}
```



C++

```
        else
        {
            chaine = espace + " ! " + espace;
        }
        return chaine;
    }
protected:
    Hanoi& jeu_hanoi;
    int nb_mouvements;
}; // Hanoi_Texte
```

Quelques remarques s'imposent. Tout d'abord, le constructeur de cette classe `Hanoi_Texte` attend un paramètre, qui doit être une instance de la classe `Hanoi` – c'est-à-dire, une instance du jeu à afficher. Il serait en effet absurde de vouloir afficher un jeu qui n'existe pas. Cela signifie que l'instance du jeu devrait être créée "en dehors" de l'instance de la classe d'interface. On gagne ainsi une certaine souplesse, en ayant la possibilité de faire afficher un même jeu par différentes interfaces (en supposant que cela présente un intérêt quelconque).

Remarquez qu'en C++ cette instance du jeu est passée par une référence et stockée en tant que référence : cela signifie que la donnée `jeu_hanoi` permettra effectivement d'accéder au jeu déclaré par ailleurs, cette donnée n'est pas une autre instance du jeu. Une telle référence est implicite en Python : dans ce langage, tout paramètre représentant une variable structurée est passé par référence.

Enfin, voyez comment l'utilisation d'une référence sur la classe `Tour` dans la méthode `Chaine_Etage()`, en C++, se limite à l'utilisation de la méthode `Disque_Etage()`, elle-même qualifiée par `const`. Si vous retirez ce qualificatif dans la déclaration de cette méthode dans `Tour`, vous obtiendrez une erreur à la compilation, car la référence retournée par `Une_Tour()` (de la classe `Hanoi`) est qualifiée par `const` : seules les méthodes constantes sont autorisées sur une référence constante. Encore une fois, il s'agit là d'une mesure de protection pour éviter des erreurs de conception. Un tel mécanisme est toutefois absent en Python.

9.4. Le programme principal

Maintenant que l'essentiel des fonctionnalités du programme est organisé en classes et méthodes, le programme principal se résume finalement à sa plus simple expression :

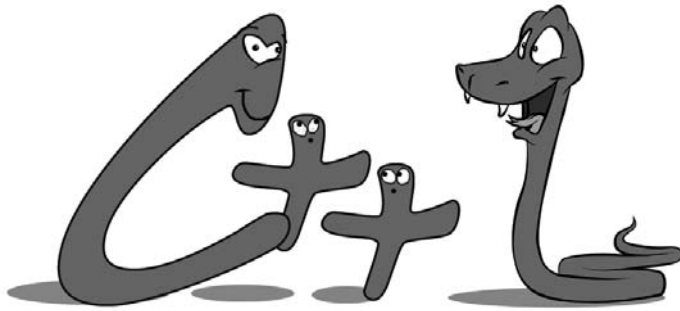
Python

```
print "Nombre de disques ?",
hauteur_max = int(raw_input())
le_jeu = Hanoi()
le_jeu.Initialiser(hauteur_max);
interface = Hanoi_Texte(le_jeu)
while (not le_jeu.Jeu_Termine()):
    interface.Afficher_Jeu()
    interface.Etape()
interface.Afficher_Jeu()
print "Réussit en", \
    interface.Nb_Mouvements(), \
    "mouvements !"
```

C++

```
int main(int argc, char* argv[])
{
    int hauteur_max;
    std::cout << "Nombre de disques ? ";
    std::cin >> hauteur_max;
    Hanoi le_jeu;
    le_jeu.Initialiser(hauteur_max);
    Hanoi_Texte interface(le_jeu);
    while ( not le_jeu.Jeu_Termine() )
    {
        interface.Afficher_Jeu();
        interface.Etape();
    }
    interface.Afficher_Jeu();
    std::cout << "Réussit en "
        << interface.Nb_Mouvements()
        << " mouvements !\n";
    return 0;
}
```

Sa structure fondamentale est la même que celui du Chapitre 7. Toutefois, l'algorithme en est infiniment plus simple, tout étant assuré par les diverses classes que nous avons créées. Il en devient de fait beaucoup plus facile à lire.



10

Mise en abîme : la récursivité

Au sommaire de ce chapitre :

- La solution du problème
- Intégration au grand programme

Non, nous n'allons pas étudier ici un procédé cinématographique. Maintenant que nous avons un programme permettant à un utilisateur de jouer aux Tours de Hanoï, il est temps de proposer une fonctionnalité essentielle pour un jeu de réflexion : afficher la solution pour résoudre le problème.

10.1. La solution du problème

Avant de nous lancer dans une programmation effrénée, il vaut mieux se demander comment nous pouvons résoudre le problème, déjà en tant qu'êtres humains. Et si possible, trouver une solution optimale, en un minimum de mouvements. Et sachant également que nous ne prétendons pas ne serait-ce qu'effleurer les principes conduisant à l'intelligence artificielle.

Une approche possible consisterait à examiner systématiquement tous les mouvements possibles, afin de déterminer la succession la plus rapide pour parvenir au résultat. Toutefois, nous nous heurterions rapidement à ce qu'on appelle parfois l'*explosion combinatoire* : le nombre de possibilités est tellement vaste que même un ordinateur puissant ne pourrait les examiner toutes en un temps raisonnable.

Nous allons donc devoir nous reposer sur notre intuition, être intelligents et transmettre notre intelligence au programme. Posons-nous la question : comment moi, être humain, puis-je résoudre ce problème ?

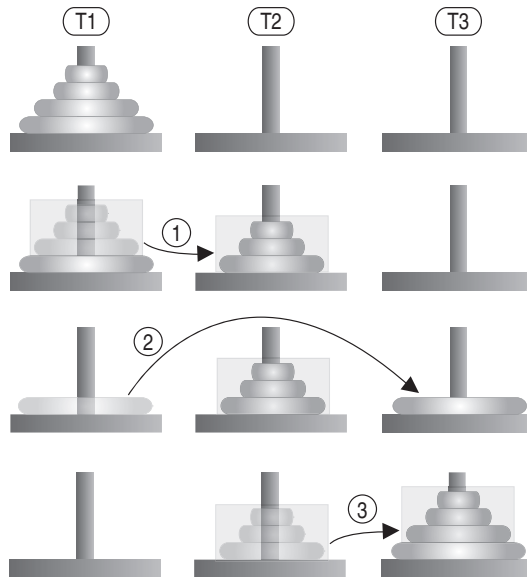
Pour fixer les idées, prenons par exemple quatre disques. Parmi eux, un est remarquable : le plus grand. Il ne peut être posé sur aucun autre, il est donc le plus contraignant à déplacer. Une solution idéale ne devrait déplacer ce disque qu'une seule fois. Nommons T1, T2 et T3 les trois piquets, la pile de disques étant au départ sur T1. Idéalement, donc, le plus grand disque sera déplacé de T1 à T3 en une seule fois. Comme il ne peut être posé sur aucun autre disque – car il est le plus grand –, cela ne pourra se faire que si aucun disque n'est présent sur T3. Autrement dit, on ne pourra effectuer ce déplacement particulier que si le grand disque est seul sur T1, et que tous les autres disques sont sur T2. Incidemment, nous venons de réduire le problème : pour pouvoir déplacer nos quatre disques de T1 vers T3, il faut d'abord déplacer (seulement) trois disques de T1 vers T2. Puis déplacer le grand sur T3. Puis redéplacer les trois disques de T2 vers T3. Ces trois étapes sont illustrées à la Figure 10.1.

Le grand disque présente une autre particularité : il est le seul sur lequel on peut poser n'importe lequel des autres disques. Donc, dans le cadre d'un déplacement des trois disques supérieurs, il n'a aucune incidence : tout se passe comme s'il n'était pas là.

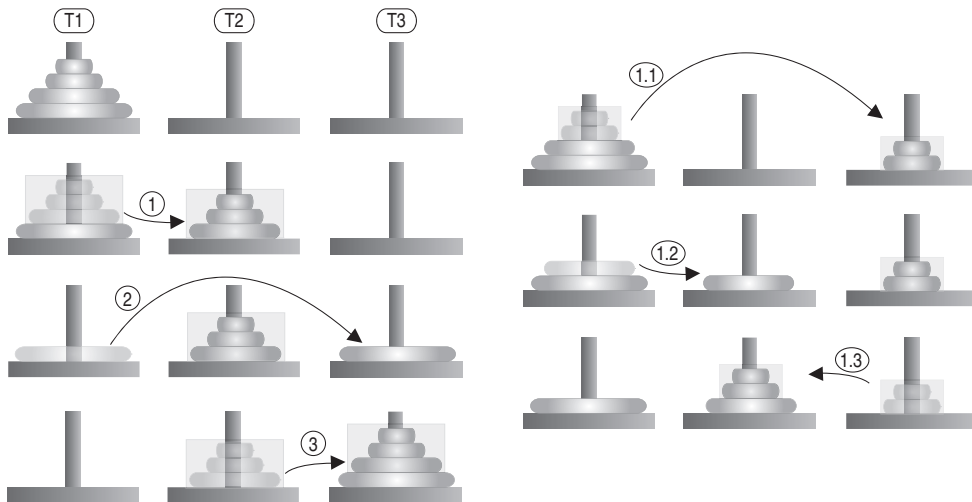
Cette remarque nous amène à un constat intéressant : on peut traiter le problème du déplacement des trois disques, exactement de la même manière que nous avons traité

Figure 10.1

Étapes pour une réduction
du problème à quatre
disques.



celui des quatre. La Figure 10.2, dans sa partie droite, montre comment la première des trois étapes peut elle-même être décomposée en trois "sous-étapes" : déplacer les deux disques supérieurs, puis le disque inférieur, puis redéplacer les deux disques supérieurs. La troisième étape de la partie gauche pourrait être décomposée de la même façon.

**Figure 10.2**

Étapes pour une réduction du problème à quatre disques.

Généralisons. Nous avons comme objectif de déplacer n disques d'un piquet T1 vers un piquet T3, par l'intermédiaire d'un piquet T2 (appelé le *pivot*). La marche à suivre est alors :

1. Déplacer $n-1$ disques de T1 vers T2.
2. Déplacer un disque (le dernier restant) de T1 vers T3.
3. Déplacer $n-1$ disques de T2 vers T3.

Les étapes 1 et 3 peuvent elles-mêmes se décomposer selon le même principe, sauf que les rôles des piquets sont intervertis. Par exemple, dans la première, on va de T1 vers T2, donc forcément par l'intermédiaire de T3. Voici la marche à suivre, donnée sur deux niveaux :

1. Déplacer $n-1$ disques de T1 vers T2 :
 - déplacer $n-2$ disques de T1 vers T3,
 - déplacer un disque de T1 vers T2,
 - déplacer $n-2$ disques de T3 vers T2.
2. Déplacer un disque de T1 vers T3.
3. Déplacer $n-1$ disques de T2 vers T3 :
 - déplacer $n-2$ disques de T2 vers T1,
 - déplacer un disque de T2 vers T3,
 - déplacer $n-2$ disques de T1 vers T3.

Et ainsi de suite. Nous tenons là une idée de solution. Maintenant, voyons comment toutes ces idées se traduisent en programmation.

10.2. Une fonction récursive

Nous allons commencer par un petit programme tout simple, qui affiche la solution au problème sans s'occuper de l'esthétique.

L'idée de la fonction récursive est la suivante : étant donné un problème de taille N , on suppose savoir le résoudre si la taille est 1, ainsi que disposer d'une fonction permettant de le résoudre à la taille $N-1$. La solution pour la taille N s'appuie donc sur la solution à la taille $N-1$. Les lecteurs ayant un penchant pour les mathématiques auront reconnu là une expression du principe de récurrence.

Un exemple concret dans la vie courante se rencontre dans le cas d'un emprunt bancaire ou d'un placement financier. Si vous placez la somme de 100 au taux de 5 % par an, quel sera votre capital au bout de dix ans ? Exactement 5 % de plus que ce qu'il était au bout de neuf ans. Et au bout de neuf ans ? Exactement 5 % de plus que ce

qu'il était au bout de huit ans. Et ainsi de suite. Si on suppose disposer d'une fonction `Capital()` donnant le capital au bout de n années (passées en paramètre) à partir d'une mise initiale de x (passée en paramètre) en fonction d'un taux t (passé en paramètre), une écriture naturelle reprenant le principe précédent serait :

Python	C++
<pre>def Capital(annees, mise, taux): return (1.0 + (taux/100.0)) * \ Capital(annees-1, mise, taux)</pre>	<pre>double Capital(int anneess, double mise, double taux) { return (1.0 + (taux/100.0)) * Capital(annees-1, mise, taux); }</pre>

La fonction `Capital()` s'utilise elle-même pour calculer son résultat. Écrite ainsi, elle présente toutefois un inconvénient majeur : elle va s'appeler elle-même à l'infini, sans jamais s'arrêter – ou du moins, en ne s'arrêtant qu'au moment où le programme aura épuisé la mémoire disponible, tout appel de fonction consommant une petite quantité de mémoire.

Il faut ajouter un test, pour retourner le résultat lorsqu'on peut le calculer directement. Ici, lorsqu'on demande le capital au bout d'une seule année. Ce test est ce que l'on appelle parfois la *condition d'arrêt* de la récursivité. Voici un programme complet :

Python	C++
<pre>def Capital(annees, mise, taux): if (anneess == 1): return (1.0 + (taux/100.0)) * mise else: return (1.0 + (taux/100.0)) * \ Capital(annees-1, mise, taux) print Capital(10, 100.0, 5.0)</pre>	<pre>#include <iostream> double Capital(int anneess, double mise, double taux) { if (anneess == 1) return (1.0 + (taux/100.0)) * mise; else return (1.0 + (taux/100.0)) * Capital(annees-1, mise, taux); } int main(int argc, char* argv[]) { std::cout << Capital(10, 100.0, 5.0) << std::endl; return 0; }</pre>

Encore une fois, les mathématiciens auront reconnu une exponentielle. Mais l'important est ici de comprendre le mécanisme en œuvre : une fonction s'utilisant elle-même.

Essayez maintenant d'écrire un programme minimaliste affichant les déplacements nécessaires pour résoudre le problème des Tours de Hanoï. Cela pourrait ressembler à ceci :

Python	C++
<pre>def Hanoi_Solution(nb_disques, depart, pivot, arrivee): if (nb_disques == 1): print depart, "->", arrivee else: Hanoi_Solution(nb_disques-1, \ depart, \ arrivee, \ pivot) Hanoi_Solution(1, \ depart, \ pivot, \ arrivee) Hanoi_Solution(nb_disques-1, \ pivot, \ depart, \ arrivee) Hanoi_Solution(3, 1, 2, 3)</pre>	<pre>#include <iostream> void Hanoi_Solution(int nb_disques, int depart, int pivot, int arrivee) { if (nb_disques == 1) std::cout << depart << " -> " << arrivee << std::endl; else { Hanoi_Solution(nb_disques-1, depart, arrivee, pivot); Hanoi_Solution(1, depart, pivot, arrivee); Hanoi_Solution(nb_disques-1, pivot, depart, arrivee); } } int main(int argc, char* argv[]) { Hanoi_Solution(4, 1, 2, 3); return 0; }</pre>

La fonction `Hanoi_Solution()` attend en paramètre le nombre de disques à déplacer, la tour de départ, la tour d'arrivée et la tour intermédiaire utilisée comme pivot. Celle-ci pourrait être déterminée automatiquement, mais cela alourdirait inutile notre exemple (toutefois, le lecteur est invité à écrire les quelques tests nécessaires). Voyez comme on retrouve exactement le principe de solution esquissé à la première section de ce chapitre.

10.3. Intégration au grand programme

Voyons maintenant comme intégrer tout cela à notre grand programme. Le calcul d'une solution au jeu semble bien faire partie de la partie "interne" du programme, aussi serait-on tenté d'ajouter une méthode effectuant ce calcul dans la classe `Hanoi`. Cependant, ce calcul est indépendant du jeu : il n'a pas besoin des données contenues dans la classe. De plus, se pose le problème de l'affichage du résultat : il n'est évidemment pas question d'afficher quoi que ce soit en dehors de la classe `Hanoi_Texte` prévue à cet effet.

Pour ces raisons, naturellement discutables, le choix a été fait de placer le calcul d'une solution dans une fonction totalement indépendante. De plus, le résultat de ce calcul sera stocké au fur et à mesure dans un paramètre supplémentaire, un tableau dans lequel sera placée la suite de mouvements déterminée.

Python

```
def Hanoi_Solution(nb_disques, \
                  depart, \
                  pivot, \
                  arrivee, \
                  soluce):
    if ( nb_disques == 1 ):
        soluce.append( (depart, arrivee) )
    else:
        Hanoi_Solution(nb_disques-1,depart, arrivee, pivot, soluce)
        Hanoi_Solution(1, depart, pivot,arrivee, soluce)
        Hanoi_Solution(nb_disques-1,pivot, depart, arrivee, soluce)
```

Le paramètre `soluce` est attendu comme un tableau. À mesure qu'on avance dans le calcul des déplacements, on ajoute à la fin de ce tableau (*append*) une paire de valeurs, ce que l'on appelle un *tuple* en Python.

Un tuple est écrit comme une liste de valeurs, séparées par des virgules, encadrée de parenthèses. Les parenthèses en apparence surnuméraires dans l'instruction mise en évidence ici ne sont donc pas une erreur, elles permettent effectivement d'écrire un tuple composé de deux éléments.

C++

De même qu'en Python, nous allons stocker des paires de valeurs dans un tableau.

```
void Hanoi_Solution(int nb_disques,
                   int depart,
                   int pivot,
                   int arrivee,
                   std::vector<std::pair <int, int> >& soluce)
{
    if ( nb_disques == 1 )
        soluce.push_back(std::make_pair(depart, arrivee));
    else
    {
        Hanoi_Solution(nb_disques-1, depart, arrivee, pivot, soluce);
        Hanoi_Solution(1, depart, pivot, arrivee, soluce);
        Hanoi_Solution(nb_disques-1, pivot, depart, arrivee, soluce);
    }
}
```

En C++, une paire de valeurs peut être stockée dans une instance du type générique `std::pair<>`, en donnant dans les chevrons les types de chacune des valeurs (ici des entiers). Une instance d'un tel type est obtenue par la fonction `std::make_pair()`, qui prend en paramètres les deux valeurs à stocker. Enfin, l'ajout d'un élément à la fin d'un tableau est obtenu par la méthode `push_back()` du type générique `std::vector<>` que nous connaissons depuis longtemps.

Le contenu de ce tableau est ensuite exploité par la classe `Hanoi_Texte` pour afficher effectivement la solution. Cette fois, nous allons ajouter une méthode dans cette classe, nommée `Solution()`, dédiée à cette tâche. Sa première action sera de réinitialiser le jeu (par la méthode `Initialiser()` de la classe `Hanoi`), afin de pouvoir simuler un déroulement normal.

Python

```

def Solution(self):
    print "-"*40
    print "SOLUTION"
    print "-"*40
    soluce = []
    self.Jeu().Initialiser (self.Jeu().Hauteur())
    self.Jeu().Solution(soluce)
    self.Afficher_Jeu()
    print "-"*40
    for mouvement in soluce:
        self.Jeu().Deplacer_Disque(mouvement[0],
                                   mouvement[1])

        self.nb_mouvements += 1
        self.Afficher_Jeu()
        print "-"*40
void Solution()
{
    std::string ligne(40, '-');
    std::cout << ligne << std::endl;
    std::cout << "SOLUTION\n";
}

```

Remarquez la forme particulière de la boucle `for` utilisée ici : la variable de boucle `mouvement` va automatiquement et successivement prendre la valeur de chacun des éléments du tableau `soluce`, sans qu'il soit nécessaire de passer par un intervalle (`range`).

À chaque "tour de boucle", `mouvement` prendra donc la valeur de l'un des tuples qui auront été stockés dans le tableau `soluce`. Les éléments d'un tuple sont obtenus comme ceux d'un tableau, en donnant leur indice (le premier étant 0) entre crochets

C++

```

std::cout << ligne << std::endl;
std::vector<std::pair<int, int> >soluce;
this->Jeu().Solution(soluce);
this->Jeu().Initialiser(this->Jeu().Hauteur());
this->Afficher_Jeu();
std::cout << ligne << std::endl;

```



C++

```

    for(int ind_mouvement = 0;
        ind_mouvement < soluce.size();
        ++ind_mouvement)
    {
        this->Jeu().Deplacer_Disque(soluce [ind_mouvement].first,
soluce[ind_mouvement].second);
        this->nb_mouvements += 1;
        this->Afficher_Jeu();
        std::cout << ligne << std::endl;
    }
}

```

En C++, les deux valeurs contenues dans une instance de `std::pair<>` peuvent être récupérées par les membres `first` (pour la première) et `second` (pour la seconde). C'est l'un des très rares exemples où des données membres sont directement accessibles.

Enfin, une méthode `Solution()` est également ajoutée dans la classe `Hanoi`, dont le rôle se limite à invoquer la fonction globale `Hanoi_Solution()` en lui donnant notamment le nombre total de disques :

Python

```

def Solution(self, soluce):
    Hanoi_Solution(self.hauteur_max, 0, 1, 2, soluce)

```

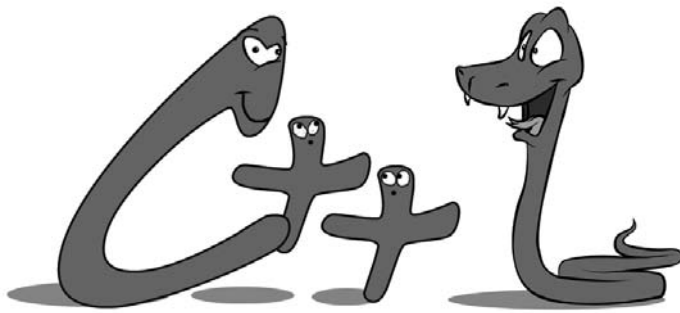
C++

```

void Solution(std::vector<std::pair<int,int> >& soluce) const
{
    Hanoi_Solution(this->hauteur_max, 0, 1, 2, soluce);
}

```

Nous avons maintenant un programme complet, permettant de jouer au jeu des Tours de Hanoï et capable d'afficher la solution du problème.



11

Spécificités propres au C++

Au sommaire de ce chapitre :

- Les pointeurs
- Mémoire dynamique
- Compilation séparée
- Fichiers de construction

Le chapitre suivant abordera les techniques propres à la programmation graphique, afin de créer des programmes permettant l'utilisation de la souris pour agir sur une interface plus attrayante. Il est toutefois nécessaire de nous arrêter un instant pour examiner quelques aspects propres aux langages de programmation compilés, en général, et au langage C++, en particulier. Ces aspects auront en effet un rôle majeur dans ce qui va suivre.

11.1. Les pointeurs

Imaginez que vous soyez l'incarnation vivante d'un programme. Vous avez connaissance d'au moins une autre personne, qui serait l'incarnation d'une information dans l'ordinateur et aussi l'élue de votre cœur. Pour joindre cette personne, vous ne disposez que du téléphone : celui-ci est comme une variable dans un programme, il vous permet d'avoir accès à une information.

Mais vous ne savez pas *où* se trouve physiquement cette autre personne. Vous pouvez la consulter et lui transmettre des données, mais il est par exemple impossible de demander à un fleuriste de lui livrer un bouquet : vous ne connaissez pas son *adresse*. Ce qui peut parfois être bien malheureux. Or, toute personne possède une adresse, une localisation, même si elle n'est pas fixe.

Il en va de même dans le cas d'une variable dans un programme. Le nom que vous utilisez est comme un téléphone, qui vous permet d'accéder à une donnée sans savoir à quel emplacement elle se trouve en mémoire. Dans le cadre du langage Python, cette information vous est complètement masquée : elle est inaccessible (du moins par des moyens simples).

En C++, on désigne par le terme de *pointeur* une variable qui contient la position d'une autre variable en mémoire. Cette position est désignée par le terme d'*adresse*, que l'on peut comparer à une adresse postale pour une personne, ou plutôt à une borne kilométrique le long d'une route : en pratique, une adresse est un nombre entier, généralement assez grand. Notez que la valeur même de ce nombre est le plus souvent sans aucun intérêt pour le programmeur, sauf dans certaines situations très particulières que nous nous garderons d'évoquer.

Les pointeurs en C++ sont des variables comme les autres, donc ayant un type. Si un pointeur contient l'adresse d'une variable de type `int`, alors on dit qu'il est de type "*pointeur sur int*". On le note `int*`, c'est-à-dire le nom du type dont on veut connaître les adresses des variables, suivie d'une étoile (astérisque). On dit alors que le type `int*` *pointe* sur des variables de type `int`.

Lorsque vous disposez d'une variable, vous pouvez obtenir son adresse en faisant précéder son nom par le caractère éperluette :

```
int une_variable = 0;
int* pointeur = &une_variable;
```

L'utilisation de ce caractère n'est pas sans rappeler la notion de référence, que nous avons vue à la fin du Chapitre 6. Ce n'est pas un hasard : ces deux notions sont assez proches, mais elles s'utilisent dans des contextes différents.

Naturellement, si vous tentez de prendre l'adresse d'une variable d'un certain type pour la stocker dans un pointeur d'un autre type, le compilateur vous insultera copieusement. Par exemple, l'extrait suivant sera refusé par un compilateur C++ :

```
int une_variable = 0;
double* pointeur = &une_variable;
```

Là où cela devient vraiment subtil, c'est qu'une telle affectation est possible dans le cadre de la programmation objet : vous pouvez affecter à une variable d'un type pointeur l'adresse d'une variable d'un type dérivé du type pointé. Voyez cet extrait de code :

```
class Mere
{
};
class Derivee: public Mere
{
};
/* ... */
Derivee une_instance;
Mere* pointeur = &une_instance;
```

Cela fonctionne parfaitement. C'est même une technique très largement utilisée pour la mise en œuvre du polymorphisme, que nous avons exploré au chapitre précédent. Nous verrons très bientôt des exemples concrets d'utilisation.

Précisons dès maintenant que l'utilisation des pointeurs est une source infinie de dysfonctionnements dans un programme. Au cours de vos expériences, vous remarquerez qu'il est bien trop aisé de mettre un désordre infernal au point de vous retrouver, d'une part, avec des pointeurs qui pointent sur "rien" ou sur une donnée erronée, et d'autre part, dans l'incapacité de comprendre ce qui se passe, votre propre code devenant un labyrinthe digne de Dédale.

11.2. Mémoire dynamique

Vous pouvez alors vous demander quel est l'intérêt de toute cette histoire de pointeurs, puisque cela semble être un nid à problèmes. Ce qui n'est pas faux.

Rappelons que le langage C++ tire son héritage du langage C, qui est un langage dit de "bas niveau", c'est-à-dire relativement proche de l'électronique de l'ordinateur. La majorité des systèmes d'exploitation, que cela soit Windows, Linux ou Mac OS, sont écrits principalement en langage C. Dans ces contextes particuliers, l'utilisation des pointeurs est une obligation pour avoir la maîtrise nécessaire de l'organisation des éléments en mémoire. Il serait ainsi impossible d'écrire un système d'exploitation en Python, ce langage occultant complètement la notion de pointeurs.

Plus communément, les pointeurs permettent d'effectuer ce qu'on appelle une *allocation dynamique de mémoire*. Lorsque vous créez une variable dans un programme ou une fonction, elle reçoit un emplacement particulier dans la mémoire de l'ordinateur pour y contenir les données qu'elle représente : on dit que de la mémoire est allouée à la variable. Cette allocation est automatique, dès la déclaration de la variable (on parle d'ailleurs de variables automatiques). Dans le cas d'une variable déclarée dans une fonction, nous l'avons vu, elle "disparaît" dès que la fonction se termine : techniquement, cela se traduit par la *libération* de la mémoire allouée, qui devient dès lors utilisable par une autre variable, voire la même variable mais lors d'un autre appel de la fonction.

Dans bien des situations, toutefois, il est souhaitable de maîtriser plus finement la manière dont la mémoire est allouée et libérée. Par exemple, si on ne sait pas à l'avance (lors de l'écriture du programme) de quelle quantité on a besoin ou si on souhaite créer une variable dont la durée de vie sera supérieure à celle de la fonction dans laquelle elle a été créée. Voyez l'extrait suivant :

```
int* pointeur = 0;
pointeur = new int;
```

On commence par déclarer une variable pointeur, donc destinée à contenir l'adresse d'une zone de mémoire. À ce moment-là, aucun espace n'est effectivement réservé en mémoire : le pointeur est dit *invalide*, sa valeur n'ayant pas de sens car arbitraire. Afin de clarifier les choses, on lui attribue dès la déclaration la valeur 0 (zéro), valeur universelle représentant une adresse mémoire invalide.

La seconde ligne réserve effectivement un emplacement dans la mémoire de l'ordinateur, ici destiné à contenir un nombre entier, par le mot clé `new` (techniquement, il s'agit en fait d'un opérateur, comme le sont `+`, `/`, etc.). L'écriture `new int` "produit"

une adresse mémoire, que nous mémorisons soigneusement dans la variable préalablement déclarée : la variable `pointeur` contient dès lors l'adresse d'une zone mémoire valide, destinée à recevoir la valeur d'un nombre entier.

Mais ce qui nous intéresse réellement, c'est justement de placer une valeur dans cette zone mémoire. À partir d'un pointeur, on peut accéder à l'information qui se trouve derrière en le faisant précéder d'une étoile, par exemple :

```
*pointeur = 1;  
int entier = 2 * *pointeur;
```

Remarquez que dans la seconde ligne, les espaces ont une certaine importance pour la lisibilité, bien qu'ils ne soient pas réellement nécessaires. Ces deux lignes sont parfaitement équivalentes à la précédente :

```
int entier = 2**pointeur;  
int entier = 2*(*pointeur);
```

À vous de trouver la notation qui vous semble la plus lisible.

Par l'utilisation d'une allocation dynamique à l'aide de `new`, on obtient une zone mémoire qui va perdurer au-delà de la fonction dans laquelle elle a eu lieu, jusqu'à la fin du programme. À moins, naturellement, qu'on décide de libérer cette zone mémoire explicitement :

```
delete pointeur;
```

Le mot clé `delete` (qui est également un opérateur, en réalité) a pour effet de libérer une zone mémoire préalablement réservée par `new`, afin qu'elle puisse être subséquentement utilisée par d'autres variables ou d'autres allocations dynamiques. Il convient de prendre garde à deux points importants :

- La valeur de la variable `pointeur` (non pas le contenu de la zone mémoire pointée) n'est en rien modifiée par `delete` : vous n'avez aucun moyen immédiat de savoir si une zone réservée a été libérée ou non.
- Le contenu de la zone mémoire pointée, par contre, peut dès cet instant être modifié : vous n'avez aucune garantie que ce contenu a encore un sens dès que l'opérateur `delete` a été appliqué.

Lorsque vous utilisez `delete`, une bonne habitude est de systématiquement placer une valeur invalide dans la variable `pointeur` :

```
delete pointeur;  
pointer = 0;
```

Ainsi, vous pourrez tester par la suite si un pointeur a été libéré ou non, et donc savoir si les données sur lesquelles il pointe sont cohérentes ou non.

Vous voyez qu'il y a dans ces mécanismes de nombreuses sources de soucis. Un problème assez répandu, et généralement difficile à corriger, est ce qu'on appelle la *fuite de mémoire* (*memory leak* en anglais). Cela se produit lorsqu'un programme réserve fréquemment de la mémoire, avec `new`, mais néglige de libérer les zones réservées lorsqu'elles ne sont plus utiles. Au fil du temps, le programme consomme de plus en plus de mémoire alors qu'il n'en a pas besoin. Jusqu'au moment où le système n'est plus capable de fournir davantage de mémoire, ce qui résulte au mieux en un plantage du programme, au pire en un plantage du système. Prenez donc toujours bien soin de vérifier que toute mémoire que vous réservez est symétriquement libérée dès qu'elle n'est plus utile.

11.3. Compilation séparée

À mesure que vos programmes deviendront plus complexes, vous vous trouverez confronté au problème de la taille des fichiers de code source : vous aurez de plus en plus de difficulté à vous y retrouver dans un fichier contenant plusieurs milliers de lignes. Il deviendra bientôt nécessaire de "ventiler" le code source du programme dans plusieurs fichiers, le mieux étant un fichier par type objet (classe).

Si un tel morcellement du code source est assez naturel et simple à mettre en œuvre en Python, il en va tout autrement en C++. Pour un langage compilé, la distribution du code source en plusieurs fichiers nécessite en général d'effectuer une *compilation séparée* suivie d'une *édition des liens*.

La compilation séparée consiste à compiler chaque fichier de code source indépendamment du reste. Au sens strict, la compilation ne produit pas de fichier exécutable : le code source est transformé en un fichier dit "objet", dans lequel les instructions contenues dans le fichier source ont été pour la plupart transformées en codes binaires compréhensibles par le processeur. Ce fichier n'est toutefois pas exécutable en tant que tel, car il ne contient ni les instructions présentes dans les autres fichiers sources, ni les informations "administratives" nécessaires à son exécution par le système d'exploitation.

À l'issue de cette compilation, les différents fichiers objet sont assemblés en une seule entité cohérente, chacun étant lié à l'ensemble. Cette phase est nommée *édition des liens*, car c'est à ce moment que les relations entre chacun des éléments sont effectivement établies et vérifiées.

En réalité, ces deux étapes sont présentes depuis notre premier programme. Jusqu'ici, elles ont été traitées ensemble, car les exemples étaient suffisamment simples. Nous allons maintenant nous attaquer à des activités plus sophistiquées, aussi la connaissance de ce mécanisme est-elle nécessaire.

Reprenons la salutation, le programme qui demande le nom de l'utilisateur avant de l'afficher. On pourrait séparer ce programme pourtant trivial en quatre fichiers différents :

- un fichier contenant une fonction pour demander le nom ;
- un fichier contenant une fonction pour saluer un nom reçu en paramètre ;
- un fichier contenant la fonction principale `main()`, nécessaire à tout programme C++, qui utiliserait les deux fonctions précédentes ;
- un fichier d'en-tête qui réaliserait la liaison entre les trois précédents.

Un fichier d'en-tête, en langages C/C++, est un fichier d'extension `.h` (ou `.hpp`, parfois `.hxx`) ne contenant en général que des *déclarations*, c'est-à-dire ne faisant qu'annoncer qu'un certain objet existe. Dans notre cas, on pourrait créer un fichier `salutation.h` qui contiendrait ceci :

```
#ifndef SALUTATION_H__
#define SALUTATION_H__ 1
#include <string>
void Demander_Nom(std::string& nom);
void Saluer(const std::string& nom);
#endif // SALUTATION_H__
```

Ce fichier ne contient que des déclarations de fonctions, pas de véritable code. Remarquez au début et à la fin du fichier les lignes commençant par `#ifndef`, `#define` et `#endif` : il s'agit de directives de compilation, destinées à éviter que le fichier soit inclus plus d'une fois au sein d'une implémentation. La raison de cela est explicitée un peu plus loin.

L'implémentation de la fonction `Demander_Nom()` pourrait se situer dans un fichier `demande_nom.cpp` ressemblant à ceci :

```
#include <iostream>
#include "salutation.h"
void Demander_Nom(std::string& nom)
{
    std::cout << "Quel est votre nom ? ";
    std::cin >> nom;
}
```

Remarquez la deuxième ligne : on fait référence à notre fichier d'en-tête. En fait, tout se passe comme si le fichier `salutation.h` était intégralement recopié à l'intérieur du fichier `demande_nom.cpp`. Du point de vue du compilateur, il devient alors ceci :

```
#include <iostream>
#ifndef SALUTATION_H__
#define SALUTATION_H__ 1
#include <string>
void Demander_Nom(std::string& nom);
void Saluer(const std::string& nom);
#endif // SALUTATION_H__
void Demander_Nom(std::string& nom)
{
    std::cout << "Quel est votre nom ? ";
    std::cin >> nom;
}
```

C'est là le véritable sens de cette directive `#include` que nous utilisons depuis le début : elle signifie, littéralement, "recopier à cet endroit le contenu du fichier auquel il est fait référence". Vous l'aurez compris, ce qui est vrai pour notre fichier d'en-tête `salutation.h` l'est également pour les fichiers `iostream` et `string` : ce sont bel et bien, eux aussi, des fichiers d'en-tête, situés à un emplacement prédéfini par le compilateur lui-même. Les deux lignes `#include` restantes seront donc remplacées par les contenus de ces fichiers. Et ainsi de suite jusqu'à ce qu'il n'en reste plus une seule.

Cette opération consistant à remplacer les fichiers mentionnés dans des directives `#include` par leur contenu est ce que l'on appelle *l'étape de prétraitement* (*pre-processing* en anglais). Ce n'est qu'après cette étape que la compilation va effectivement pouvoir avoir lieu. Les directives `#ifndef`, `#define` et `#endif` sont également interprétées lors de cette étape.

Ces trois directives, justement, permettent d'éviter une situation qui pourrait être délicate : celle où un fichier d'en-tête est inclus (par `#include`) dans un autre, lui-même inclus dans un programme... avec le premier. On pourrait ainsi aboutir à une situation où le contenu d'un même fichier d'en-tête est inclus plusieurs fois avant la compilation. Si dans certains cas, ce n'est guère gênant, dans la plupart des situations cela peut aboutir à des incohérences subtiles. Le fait "d'encadrer" le contenu d'un fichier d'en-tête à l'aide des directives `#ifndef`, `#define` et `#endif`, permet

de détecter si un contenu a déjà été inclus ou non et donc de ne pas l'inclure une seconde fois. Cela est fait simplement en définissant une sorte d'identifiant du fichier, par convention son nom en majuscules suivi de deux caractères de soulignement, comme c'est fait ici.

Si vous êtes curieux, vous pouvez consulter le résultat intermédiaire de cette étape, qui normalement ne vous est jamais présenté :

```
> g++ -E demander_nom.cpp -o demander_nom.e
```

Le résultat est stocké dans le fichier `demander_nom.e`, lequel est, vous le constatez, considérablement plus volumineux que votre fichier de départ. Vous avez en fait sous les yeux ce qui sera effectivement compilé.

La compilation, justement, est effectuée en passant une option spéciale au compilateur :

```
> g++ -c demander_nom.cpp -o demander_nom.o
```

L'option `-c` indique que l'on souhaite effectuer seulement la compilation (et le prétraitement, naturellement) et rien d'autre. Le nom du fichier objet résultant utilise l'extension `.o`, par convention (parfois `.obj` si on utilise d'autres compilateurs).

Nous créons et compilons de la même manière l'implémentation de la fonction `Saluer()`, dans `saluer.cpp` :

```
#include <iostream>
#include "salutation.h"
void Saluer(const std::string& nom)
{
    std::cout << "Bonjour, "
               << nom << std::endl;
}

> g++ -c saluer.cpp -o saluer.o
```

Et enfin la fonction principale, dans `salutation.cpp` :

```
#include "salutation.h"
int main(int argc, char* argv[])
{
    std::string nom;
    Demander_Nom(nom);
}
```



```
    Saluer(nom);  
    return 0;  
}
```

```
> g++ -o salutation.cpp -o salutation.o
```

Cette fonction utilise les deux fonctions `Demander_Nom()` et `Saluer()` : cela n'est possible que parce qu'elles ont été *déclarées* (annoncées) dans le fichier d'en-tête `salutation.h`, lequel est justement inclus ici.

Nous disposons donc maintenant de trois fichiers objet. Pour obtenir enfin un fichier exécutable, nous pouvons effectuer l'édition des liens ainsi :

```
> g++ demander_nom.o saluer.o salutation.o -o salutation
```

Finalement, les étapes nécessaires à la construction d'un fichier exécutable peuvent être schématisées comme à la Figure 11.1.

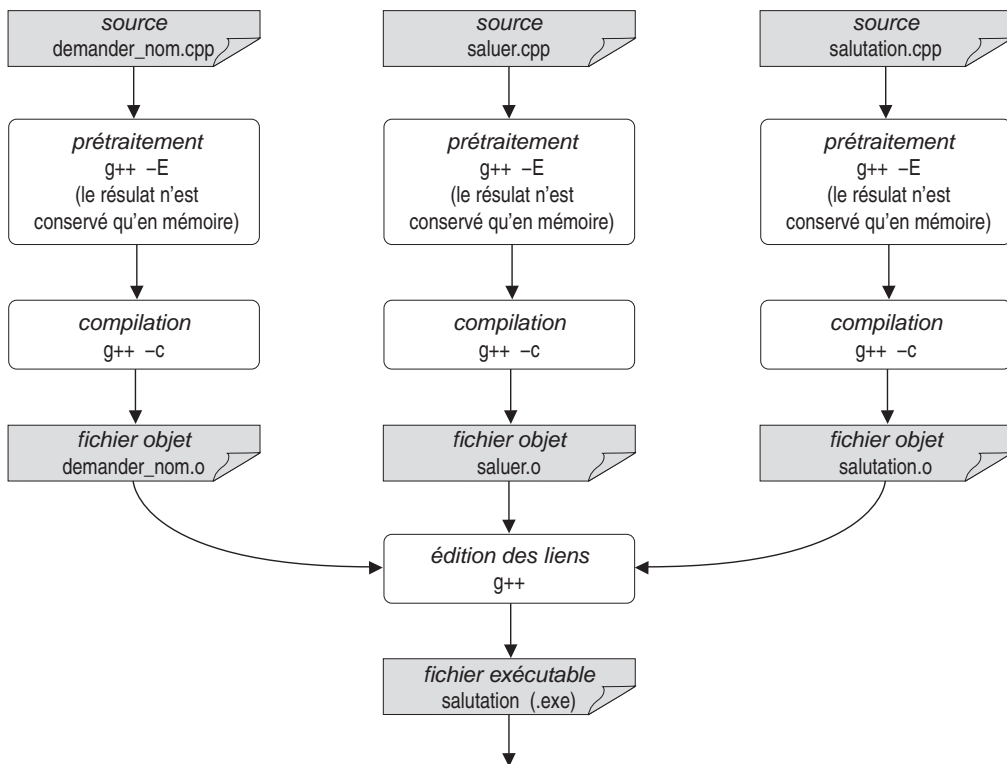


Figure 11.1

Étapes pour la construction d'un fichier exécutable.

Il ne devrait échapper à personne qu'il s'agit là d'un processus relativement laborieux, pouvant être entaché d'erreurs à chacune des étapes. Et encore ne s'agit-il que d'un exemple extrêmement modeste. Heureusement, des solutions existent pour nous simplifier la vie, que nous allons examiner dès maintenant.

11.4. Fichiers de construction

Nous avons vu qu'un programme pouvait être décomposé en plusieurs fichiers. C'est même presque une règle dans le cas de logiciels véritablement complexes : le code source de votre traitement de texte favori se répartit en plusieurs *milliers* de fichiers ! Compiler chacun d'eux "à la main", comme nous l'avons fait précédemment, n'est tout simplement pas envisageable.

Dans ces cas-là, on fait appel à ce que l'on appelle un *programme de construction* (aussi appelé *moteur de production*). Il s'agit d'un logiciel spécialement dédié à la tâche consistant à produire... des logiciels. La structure du programme qu'on souhaite construire est décrite dans un *fichier de construction*, ainsi que les éventuelles règles particulières devant être appliquées. À partir de ce fichier de construction, le moteur de production va prendre en charge automatiquement la compilation des différents fichiers source puis l'édition des liens finale pour obtenir un logiciel exécutable.

Le plus répandu des moteurs de production est aujourd'hui l'utilitaire GNU make. Il est normalement presque toujours disponible sur tous les systèmes de type Unix (ce qui inclut GNU/Linux et Mac OS). Une version accompagne l'ensemble MinGW sous Windows. Par convention, le fichier de construction qu'utilisera l'utilitaire make se nomme simplement *Makefile* (sans extension).

Si on reprend l'exemple de la section précédente, nous pourrions décrire ainsi notre programme de salutation dans un fichier de construction nommé *Makefile* :

```
CXX = g++
TARGET = salutation
OBJECTS = demander_nom.o saluer.o salutation.o

%.o: %.cpp
⇒ $(CXX) -c $^ -o $@

all: $(OBJECTS)
⇒ $(CXX) -o $(TARGET) $(OBJECTS)
```

Nous n'allons pas rentrer dans le détail des règles d'écriture des fichiers de construction, qui peuvent être assez complexes. L'exemple précédent donne toutefois une idée générale. Pour commencer, les flèches que vous voyez représentent un caractère de tabulation : c'est absolument obligatoire, impossible d'utiliser simplement des espaces. Concernant le contenu, on commence par renseigner quelques variables usuelles (CXX pour le compilateur à utiliser, TARGET pour le résultat final que l'on souhaite obtenir, OBJECTS pour la liste des fichiers intermédiaires), puis on définit quelques règles sous la forme :

```
cible: dépendances
⇒ commande à exécuter
```

Par "dépendances" on entend ici des fichiers à partir desquels la cible va être générée, en exécutant la commande indiquée. La cible nommée `a11` est un peu particulière : c'est la cible par défaut si aucune n'est précisée lors de l'utilisation de `make`.

Si vous placez ce fichier `Makefile` dans le même répertoire que celui contenant les fichiers de notre programme de salutation, l'utilisation de `make` est facile. Il suffit d'exécuter la commande :

```
$ make
```

Ou si vous êtes sous Windows :

```
> mingw32-make
```

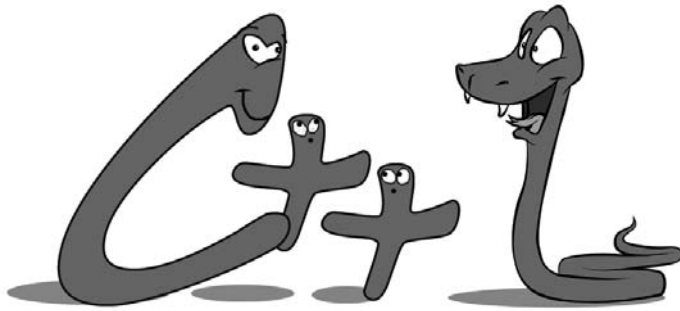
Vous devriez alors obtenir quelque chose comme ceci :

```
g++ -c demander_nom.cpp -o demander_nom.o
g++ -c saluer.cpp -o saluer.o
g++ -c salutation.cpp -o salutation.o
g++ -o salutation demander_nom.o saluer.o salutation.o
```

Vous pouvez constater que chacun des fichiers est compilé indépendamment, puis l'exécutable final est créé par l'édition des liens. On retrouve des commandes analogues à celles que nous avons utilisées à la section précédente. Faites l'expérience suivante : relancer immédiatement la commande `make`.

Normalement, presque rien ne devrait se passer. C'est là un autre intérêt d'utiliser un outil de construction : seuls les fichiers devant être recompilés le sont effectivement. Modifiez l'un des fichiers source, simplement en ajoutant un espace ou une ligne vide, puis relancez `make`. Seul le fichier modifié sera recompilé.

Nous verrons bientôt que nous n'aurons même pas à écrire nous-même de fichier de construction. En effet, ils peuvent être assez délicats à mettre au point. Ne vous laissez pas tromper par l'exemple donné ici, il est d'une simplicité caricaturale (et, en fait, pas tout à fait correct, par souci de compréhension). Des outils existent pour créer ces fichiers de construction, à partir d'autres fichiers encore plus simples.



12

Ouverture des fenêtres : programmation graphique

Au sommaire de ce chapitre :

- La bibliothèque Qt
- Installation des bibliothèques Qt et PyQt
- Le retour du bonjour
- Un widget personnalisé et boutoné
- Hanoï graphique

Tous nos programmes étaient jusqu'à maintenant des programmes en *mode texte*, c'est-à-dire qu'ils ne font qu'afficher et recevoir des caractères, rien d'autre. Ce qui n'a rien d'infamant. L'interface graphique, qui nous semble aujourd'hui si commune, n'a été imaginée dans les laboratoires de l'entreprise Xerox qu'aux alentours de 1973, avant d'être largement développée et diffusée par les entreprises Apple (ordinateurs Macintosh, 1984), Sun (stations de travail graphiques, 1983), SGI (affichage 3D, 1984) et Microsoft (environnement Windows 1.0 en 1985 et 3.0 en 1990). Aujourd'hui, même les systèmes les plus récents que sont Mac OS X, Windows Vista ou les nombreux environnements graphiques disponibles pour systèmes Linux (et Unix), s'inscrivent dans la lignée des concepts âgés de plus de trente-cinq ans. Tout au long de cette période, de très nombreux programmes de premier ordre, comme les applications comptables, sont restés longtemps cantonnés à un affichage à base de caractères.

Mais aujourd'hui, il paraît naturel pour la majorité des utilisateurs de disposer du confort d'une interface graphique, manipulable à l'aide d'une souris. Il n'y a d'ailleurs guère d'autre solution lorsque la nature même des informations qu'on souhaite afficher est fondamentalement graphique.

Aussi allons-nous désormais découvrir quelques-unes des possibilités qui s'offrent à nous. Vous pourrez constater que, si les programmes gagnent alors en convivialité et attractivité, ils gagnent aussi en complexité. Il s'agit là d'une règle qui semble s'affirmer : plus un logiciel est agréable et aisé d'utilisation, plus il est complexe dans sa conception et son écriture.

12.1. La bibliothèque Qt

Un langage de programmation, en lui-même, n'offre généralement pas de possibilités pour réaliser un programme graphique (à quelques rares exceptions près). Le graphisme n'est atteint le plus souvent qu'en utilisant des bibliothèques d'outils, adaptées à un langage donné et à un environnement graphique donné. En effet, la *manière* de programmer graphiquement diffère assez considérablement selon que l'on se trouve sous un système Linux, Windows ou Mac OS, chacun d'eux pouvant éventuellement proposer différents contextes (X-Window ou *framebuffer* sur Linux, par exemple).

Aussi, allons-nous devoir utiliser une bibliothèque spécialisée. Le choix a été fait de la bibliothèque Qt (prononcez "kiouti"), créée et développée depuis de nombreuses

années par la société norvégienne TrollTech (<http://www.trolltech.com>, récemment passée sous le contrôle de Nokia). Plusieurs raisons ont guidé ce choix :

- Il s'agit d'une bibliothèque en langage C++ disponible sur tous les systèmes d'exploitation majeurs.
- Elle propose une myriade d'outils divers et puissants, tout en conservant une certaine cohérence et une logique interne, qui facilite son apprentissage.
- La documentation qui l'accompagne est d'une grande qualité (quoique uniquement en anglais), aspect essentiel pour pouvoir réaliser un projet sans passer des nuits à chercher une information.
- Elle est employée quotidiennement par des millions d'utilisateurs, ne serait-ce qu'au travers de l'environnement graphique KDE couramment utilisé sur systèmes Linux, dont elle constitue les fondations.
- Une version libre (au sens du logiciel libre) est disponible gratuitement, de même que plusieurs adaptations dans d'autres langages que le C++. Il est ainsi possible de l'utiliser en langage Python, au travers de l'adaptation PyQt, elle-même de grande qualité.

Au-delà de toutes ces bonnes raisons, ne perdez pas de vue que le choix d'une bibliothèque, comme celui d'un langage de programmation, obéit également à des motivations subjectives. Même si cela peut vous paraître encore un peu abstrait, vous constaterez que *l'esthétique* d'une bibliothèque a également son importance.

Gardez à l'esprit que cet ouvrage n'est en aucun cas un manuel de la bibliothèque Qt, dont la documentation complète représente plusieurs fois le volume que vous tenez entre les mains. À l'issue de l'installation des bibliothèques, vous trouverez cette documentation dans un répertoire nommé *doc* à l'endroit où Qt a été installée. Nous ne décrirons pas tous les outils utilisés dans le moindre détail, aussi n'hésitez pas à aller consulter cette documentation. La lecture est une part non négligeable de la programmation !

12.2. Installation des bibliothèques Qt et PyQt

Au moment où ces lignes sont écrites, la version libre 4.3.3 est la dernière de Qt. Vous trouverez son code source sur le site FTP <ftp://ftp.trolltech.com/qt/source>, pour toutes les plates-formes.

La bibliothèque PyQt, l'adaptation en Python de Qt, nécessite l'installation d'un petit utilitaire pour pouvoir être compilée. Nommé SIP, vous le trouverez à partir de la page <http://www.riverbankcomputing.co.uk/sip/download.php>. PyQt est, quant à elle, disponible à partir de la page <http://www.riverbankcomputing.co.uk/pyqt/download.php>.

12.2.1. Windows

Téléchargez le fichier `qt-win-opensource-src-4.3.3.zip` à partir du site FTP indiqué précédemment, puis décompressez son contenu dans le répertoire `C:\Programmes`. Renommez le répertoire nouvellement créé en `qt433`, afin d'économiser quelques touches de clavier par la suite.

À partir d'une ligne de commande, placez-vous dans ce répertoire, puis exécutez la commande suivante :

```
C:\Programmes\qt433> configure -platform win32-g++ -release -qt-sql-sqlite
➡-qt-zlib -qt-libpng -qt-libmng -qt-libtiff -qt-libjpeg -no-dsp -no-vcproj
➡-no-qt3support -no-openssl -no-qdbus
```

Cela va paramétrer Qt pour votre système et préparer sa compilation, que vous lancerez à l'aide de la commande :

```
C:\Programmes\qt433> mingw32-make
```

Puis, allez préparer quelques cafés. En plus de la bibliothèque elle-même, d'une taille assez considérable, de nombreux outils annexes et des exemples seront également compilés. Le processus complet peut demander plusieurs heures, selon la puissance de votre machine.

Cela fait, modifiez comme suit le fichier de commande `prog.bat` que nous avons créé au premier chapitre, qui permet d'obtenir un environnement de développement adéquat (attention, la ligne commençant par `set PATH` est bien une seule et unique ligne !) :

```
set QTDIR=C:\Programmes\qt433
set QMAKESPEC=win32-g++
set
PATH=%QTDIR%\bin;C:\Python25;C:\MinGW\bin;C:\MinGW\libexec\gcc\mingw32\3.4.5;%PATH%
C:
cd \Programmes
cmd.exe
```

Fermez la ligne de commande puis relancez-la, afin de bénéficier du nouvel environnement. Essayez d'exécuter la commande `qtdemo` afin de voir quelques démonstrations et exemples de Qt.

Récupérez maintenant le fichier sip-4.7.4.zip à partir de la page <http://www.riverbankcomputing.co.uk/sip/download.php>. Décompactez ce fichier, nommez le répertoire en sip474, à partir duquel vous exécutez les commandes :

```
C:\Programmes\sip474> python configure.py -p win32-g++  
C:\Programmes\sip474> mingw32-make  
C:\Programmes\sip474> mingw32-make install
```

Nous pouvons maintenant installer PyQt. Récupérez le fichier PyQt-win-gpl-4.3.3.zip sur la page <http://www.riverbankcomputing.co.uk/pyqt/download.php>. Comme précédemment, décompactez-le dans C:\Programmes, puis nommez le répertoire nouveau pyqt433. À partir de ce répertoire, exécutez simplement :

```
C:\Programmes\pyqt433> python configure.py  
C:\Programmes\pyqt433> mingw32-make  
C:\Programmes\pyqt433> mingw32-make install
```

Vous êtes maintenant prêt à développer des logiciels graphiques aussi bien en C++ qu'en Python.

12.2.2. Linux

Si vous utilisez une distribution majeure récente (datant de l'année 2006), il est certain que les bibliothèques Qt et PyQt sont disponibles parmi les paquetages qu'elle propose. En particulier, cherchez les paquetages dont les noms ressemblent à ceux-ci :

- pour Qt, libqt4-dev, libqt4-core, libqt4-gui ;
- pour PyQt, python-qt4, pyqt4-dev-tools, python-qt4-gl, python-qt4-doc.

Installez tous ces paquetages, ainsi que ceux desquels ils dépendent.

12.2.3. Mac OS X

Téléchargez le fichier qt-mac-opensource-src-4.3.3.tar.gz à partir du site FTP de TrollTech, puis placez-le dans votre répertoire personnel. À partir de la ligne de commande, décompressez-le puis configurez Qt comme suit :

```
$ tar xzf qt-mac-opensource-src-4.3.3.tar.gz  
$ cd qt-mac-opensource-src-4.3.3  
$ ./configure -prefix $HOME/opt/qt433 -qt-sql-sqlite -no-qt3support -qt-zlib  
➡ -qt-libtiff -qt-libmng -qt-libjpeg -no-framework -no-dwarf2
```

Enfin, lancez la compilation de Qt, non sans avoir prévu quelques cafés ou un divertissement quelconque, car cette opération peut prendre un certain temps :

```
$ make -j2 && make install
```

Il est ensuite nécessaire de modifier votre environnement, afin que la bibliothèque Qt soit dûment reconnue et localisée. Créez pour cela un fichier nommé `qt_env.sh` dans votre répertoire Programmes, contenant les lignes suivantes :

```
$ export QTDIR=$HOME/opt/qt433
$ export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
$ export PATH=$QTDIR/bin:$PATH
```

Désormais, chaque fois que vous lancerez une ligne de commande, il vous suffira d'invoquer ce fichier afin de mettre à jour l'environnement :

```
$ source Programmes/qt_env.sh
```

Cela fait, vous pouvez vérifier que votre installation fonctionne correctement en exécutant le programme de démonstration de Qt :

```
$ qtdemo
```

Téléchargez maintenant le fichier `sip-4.7.4.tar.gz` sur le site <http://www.riverbankcomputing.co.uk/sip/download.php>. Décompactez-le puis, à partir du répertoire nouvellement créé, exécutez les commandes :

```
$ python configure.py
$ make
$ make install
```

Il ne nous reste maintenant plus qu'à installer PyQt. Récupérez le fichier `PyQt-mac-gpl-4.3.3.zip` sur la page <http://www.riverbankcomputing.co.uk/pyqt/download.php>. Comme précédemment, décompactez-le puis, à partir du répertoire créé, exécutez simplement :

```
$ python configure.py
$ make
$ make install
```

Vous êtes maintenant prêt à développer des logiciels graphiques aussi bien en C++ qu'en Python.

12.3. Le retour du bonjour

Il est toujours préférable de commencer la découverte d'une nouvelle technique ou d'un nouvel outil par des réalisations simples. Nous avons entamé notre initiation à la programmation en saluant le monde, faisons de même pour découvrir la programmation graphique !

Voici le programme.

Python

```
import sys
from PyQt4 import QtCore, QtGui
appli = QtGui.QApplication(sys.argv)
bijour = QtGui.QLabel("Bonjour, Monde !")
bijour.show()
sys.exit(appli.exec_())
```

C++

```
#include <Qapplication>
#include <Qlabel>
int main(int argc, char* argv[])
{
    QApplication appli(argc, argv);
    QLabel bijour("Bonjour, Monde !");
    bijour.show();
    return appli.exec();
}
```

Vous pouvez constater que, si ces programmes demeurent extrêmement courts, ils sont néanmoins beaucoup plus longs que leurs équivalents purement textuels. Examinons un instant leur contenu.

Pour commencer, tout programme graphique s'appuyant sur Qt doit contenir une instance et une seule de la classe `QApplication` (ici la variable `appli`). Celle-ci représente en quelque sorte le cadre général dans lequel le programme va s'exécuter. Elle maintient également la boucle d'événements dont il a déjà été question.

Du point de vue de Python, cette classe fait partie du *module* `QtGui`. Le terme module pour Python est plus ou moins l'équivalent du terme bibliothèque : un regroupement en une seule entité de fonctions et de classes utilisables par un autre programme. La première ligne du programme précédent, `import sys`, signale que nous allons utiliser le contenu du module nommé `sys`. La deuxième rend disponibles les contenus des modules `QtCore` et `QtGui`, qui sont, en fait, des sous-modules du module général `PyQt4`.

Pour le C++, il est nécessaire d'ajouter la ligne `#include <QApplication>` pour pouvoir l'utiliser. Notez bien qu'il est nécessaire et indispensable de créer une instance de `QApplication` **avant** d'utiliser **n'importe laquelle** des possibilités graphiques de

Qt : si vous ne le faites pas, votre programme plantera irrémédiablement avant même d'afficher la moindre fenêtre.

Ensuite, nous créons un label (variable `bi_jour`), c'est-à-dire un élément graphique dont le seul objet est d'afficher un texte, sur lequel l'utilisateur n'a pas de possibilité d'action directe. Tout élément textuel d'une interface graphique doit ainsi être contenu dans un objet graphique, d'une manière ou d'une autre. Ce qui tranche radicalement avec les programmes que nous avons réalisés jusqu'ici, où les chaînes de caractères étaient simplement affichées. Ici, ce n'est pas "vous" qui affichez, c'est l'objet contenant votre chaîne de caractères.

On demande ensuite à cet élément graphique, que l'on appelle communément un *widget* (contraction des mots *window gadget*, en français *gadget de fenêtre*) ou un *contrôle*, de justement s'afficher. À leur création, les widgets sont normalement masqués, il faut donc demander explicitement leur apparition à l'écran : c'est le rôle de la méthode `show()` (*montrer* en anglais).

Enfin, on démarre la fameuse boucle d'événements au moyen de la méthode `exec()` de la classe `QApplication`. C'est elle qui va recevoir tous les événements et les transmettre aux éléments appropriés du programme. Par exemple, si une fenêtre vient occulter celle de votre programme, celui-ci recevra un événement l'informant qu'il est masqué. S'il redevient visible, un nouvel événement sera transmis, lequel sera dirigé vers les différents composants pour leur demander de se redessiner à l'écran.

Nous en avons presque terminé avec notre premier programme graphique. La version en Python doit normalement s'exécuter directement, simplement en invoquant la commande :

```
$ python bonjour_gui.py
```

Par contre, la version C++ doit être compilée, ce qui peut devenir rapidement assez complexe pour un programme graphique utilisant une bibliothèque aussi sophistiquée que Qt (cela est vrai pour la plupart des bibliothèques graphiques). Juste pour satisfaire votre curiosité, voici à quoi pourrait ressembler la compilation de ce programme tout simple (exemple sur système Windows) :

```
> g++ -c -O2 -frtti -fexceptions -mthreads -Wall -DUNICODE -  
DQT_LARGEFILE_SUPPORT -DQT_DLL -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB -  
DQT_THREAD_SUPPORT -DQT_NEEDS_QMAIN -I"qt433\include\QtCore" -  
I"qt433\include\QtCore" -I"qt433\include\QtGui" -I"qt433\include\QtGui" -
```

```
I"qt433\include" -I"c:\Programmes\qt433\include\ActiveQt" -I".moc" -I"." -  
I"qt433\mkspecs\win32-g++" -o .objs\bonjour_gui.o bonjour_gui.cpp  
  
> g++ -enable-stdcall-fixup -Wl,-enable-auto-import -Wl,-enable-runtime-pseudo-  
reloc -Wl,-s -mthreads -Wl -Wl,-subsystem,windows -o bin\bonjour_gui.exe .objs/  
bonjour_gui.o -L"c:\Programmes\qt433\lib" -lmingw32 -lqtmain -lQtGui4 -lQtCore4
```

Non seulement cette compilation se fait en deux étapes, mais en plus les lignes de commande à exécuter sont particulièrement chargées. Il est tout simplement hors de question de saisir des commandes aussi complexes à chaque compilation. La solution consiste à utiliser un fichier de construction, dans le genre de ceux évoqués au chapitre précédent.

Par chance, Qt fournit un moyen d'obtenir aisément ce fichier à partir de quelques informations beaucoup plus simples contenues dans un fichier, dit *fichier projet*, d'extension `.pro`. Voici le contenu d'un tel fichier, que nous nommerons ici `bonjour_gui.pro`, que vous pouvez saisir avec votre éditeur de texte favori :

```
TEMPLATE = app  
  
QT += gui  
  
CONFIG += release  
  
OBJECTS_DIR = .objs  
  
MOC_DIR = .moc  
  
DESTDIR = bin  
  
SOURCES += bonjour_gui.cpp
```

Cela pourrait presque ressembler à un programme Python dans lequel on ne ferait qu'affecter des valeurs dans des variables. La première, `TEMPLATE`, indique de quel type de projet il s'agit. Ici, nous voulons une application, donc on donne la valeur `app`. D'autres valeurs sont possibles, comme `lib` pour créer une bibliothèque. Les deux variables suivantes, `QT` et `CONFIG`, précisent quelques paramètres pour la compilation. Ici, on demande simplement à utiliser les outils graphiques de Qt (`gui`) et à obtenir une version optimisée (`release`). Notez l'utilisation du symbole `+=`, qui a pour effet *d'ajouter* des valeurs à ces variables, comme si on effectuait une concaténation de chaînes de caractères (ce qui est en réalité le cas, mais pour l'heure vous n'avez pas à vous en soucier). Les trois variables suivantes concernent l'organisation des répertoires, notamment le stockage des fichiers intermédiaires utilisés lors de la phase de compilation : ces fichiers sont créés par le compilateur lui-même pour ses besoins propres. On demande à ce qu'ils soient rangés dans deux sous-répertoires (par `MOC_DIR` et `OBJECTS_DIR`), afin de ne pas encombrer le répertoire contenant les

fichiers source. DESTDIR indique le répertoire de destination final, c'est-à-dire celui qui contiendra effectivement le fichier exécutable résultant de la compilation.

Enfin, la variable la plus importante est SOURCES : placez dans celle-ci tous les fichiers de code source constituant votre programme. Une solution simple à la compilation séparée évoquée au chapitre précédent.

Une fois ce fichier créé, on fait appel à l'utilitaire qmake fourni par Qt :

```
$ qmake bonjour_gui.pro
```

Le résultat est un fichier de construction (nommé normalement Makefile), que nous pouvons utiliser immédiatement par la commande :

```
$ make
```

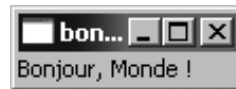
ou si vous êtes sous système Windows :

```
> mingw32-make
```

À l'issue de quoi, vous devriez obtenir un fichier exécutable nommé `bonjour_gui` (avec l'extension `.exe` sur Windows) dans un sous-répertoire nommé `bin`, comme nous l'avons demandé dans le fichier projet. Le nom de ce fichier exécutable est déduit du nom du fichier projet, à moins que vous ne l'ayez précisé en affectant une valeur à la variable TARGET. Mais inutile de compliquer les choses.

Figure 12.1

Premier programme graphique.



La Figure 12.1 montre à quoi devrait ressembler ce premier programme graphique sur système Windows. Si vous êtes un peu dépité par son aspect élémentaire, considérez le chemin parcouru depuis le début de cet ouvrage...

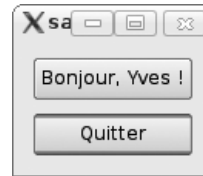
12.4. Un widget personnalisé et boutoné

Naturellement, dans l'immense majorité des cas, votre programme ne saurait se contenter d'un simple label affichant un texte statique. Il devra offrir un minimum d'interactions à l'utilisateur et adapter son affichage. Reprenons l'exemple de la salutation. Nous nous proposons maintenant de réaliser un programme à peine moins minimaliste (voir Figure 12.2). Deux boutons sont offerts. Le premier affiche au

départ simplement "Bonjour !" ; lorsque l'utilisateur clique dessus, le programme lui demande son nom puis l'affiche dans le bouton. Le deuxième permet simplement de quitter le programme.

Figure 12.2

Salutations à la souris.



Chacun de ces boutons est un widget devant réagir à une action de l'utilisateur. Mais, en fait, ce sont au total *trois widgets* qui seront créés et affichés : les deux boutons, plus un widget les "contenant" et assurant leur disposition harmonieuse, autant que possible. Chaque widget peut, en effet, en contenir d'autres, créant ainsi une sorte d'emboîtements des éléments graphiques les uns dans les autres. Un logiciel complexe, comme un traitement de texte, peut ainsi contenir des centaines de widgets, dont une partie n'existent que pour en contenir d'autres et les agencer correctement. Par exemple, les boutons d'une barre d'outils sont tous des widgets, "contenus" dans le widget qu'est la barre d'outils, elle-même "contenue" avec d'autres dans le widget qu'est la fenêtre générale du logiciel.

Il existe différentes techniques pour réaliser cela. Celle présentée ici n'est pas toujours la plus simple, mais certainement celle qui offre le plus de souplesse. Elle consiste à créer un nouveau widget, en s'appuyant sur la classe de base `QWidget` fournie par Qt. Tous les contrôles graphiques de Qt héritent de cette classe – y compris la classe `QLabel` vue précédemment.

À partir de maintenant, il est vivement recommandé de placer les fichiers composant chacun des programmes que nous allons réaliser dans un répertoire particulier : un répertoire par programme. Éventuellement, distinguez les fichiers propres à Python et les fichiers propres à C++, afin de bien vous y retrouver. Car dès à présent, nos programmes ne seront plus stockés dans un unique fichier.

En effet, lorsque l'on réalise une application complexe dans laquelle on crée de nouveaux widgets, il est d'usage de placer chacun d'eux dans un fichier qui lui est dédié. Si cela peut paraître superflu pour cet exemple, qui reste simple, cette séparation devient indispensable dès que vous vous attaquez à quelque chose de plus sophistiqué. Sans elle, vous aboutirez à des fichiers contenant des milliers de lignes au sein desquelles il deviendra extrêmement difficile de naviguer.

12.4.1. En Python

La situation la plus simple est, comme souvent, en langage Python. La séparation est élémentaire : un widget, un fichier, plus un fichier supplémentaire pour le programme principal. Voici la déclaration de notre widget personnalisé, dans un fichier nommé `salutations.py` :

```
from PyQt4 import QtCore, QtGui
class Salutation(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.bt_bonjour = QtGui.QPushButton("Bonjour !", self)
        self.bt_quitter = QtGui.QPushButton("Quitter", self)
        layout = QtGui.QVBoxLayout(self)
        layout.addWidget(self.bt_bonjour)
        layout.addWidget(self.bt_quitter)
        self.connect(self.bt_bonjour, \
                     QtCore.SIGNAL("clicked()"), \
                     self.demanderNom)
        self.connect(self.bt_quitter, \
                     QtCore.SIGNAL("clicked()"), \
                     self, \
                     QtCore.SLOT("close()"))
    def demanderNom(self):
        nom = QtGui.QInputDialog.getText(self, "Salutation", "Quel est votre nom ?")
        self.bt_bonjour.setText(QtCore.QString("Bonjour, %1 !").arg(nom[0]))
```

Ne vous laissez pas impressionner par l'apparente longueur de ce code, c'est en fait assez simple. On déclare une classe `Salutation`, en signalant qu'elle dérive de la classe `QWidget` contenue dans le module `QtGui` (lui-même issu du module `PyQt4` et importé par la ligne précédente). Le constructeur de notre classe prend (au moins) un paramètre `parent`, désignant le widget dans lequel il sera éventuellement contenu, la valeur par défaut étant aucun (`None`). La première action de ce constructeur est d'invoquer celui de la classe ancêtre : ce n'est pas syntaxiquement obligatoire, mais il est plus que vivement recommandé de **toujours** commencer par invoquer ce constructeur ancêtre.

Ensuite, nous créons les deux boutons en instanciant deux instances de la classe `QPushButton` : celle-ci représente, en effet, un simple bouton à l'écran sur lequel l'utilisateur peut cliquer. Le premier paramètre d'instanciation est le texte à afficher dans le bouton, tandis que le second est le widget dans lequel sera contenu chaque bouton. Comme nous sommes précisément en train de construire un widget destiné à contenir ces boutons, on donne donc la valeur `self`, l'instance elle-même de `Salutation` en cours de création. Du point de vue des boutons (qui sont également des widgets), cette

valeur sera reçue par le paramètre parent de leur propre constructeur. On établit ainsi une relation parent-enfant entre les widgets composant une application (un parent pouvant avoir plusieurs enfants, mais un enfant ne pouvant avoir qu'un seul parent).

Les trois lignes qui suivent concernent l'*agencement* de ces widgets à l'écran. Il est toujours possible de positionner "à la main" les éléments graphiques, en donnant leurs coordonnées et leurs dimensions. Mais l'expérience a montré que cette manière de faire était loin d'être satisfaisante : les fenêtres et les boîtes de dialogue étaient alors d'une taille fixe, impossible à modifier, ce qui est parfois gênant. Aussi, utilisons-nous ici un *layout*, c'est-à-dire un outil fourni par Qt pour agencer automatiquement les éléments graphiques. En l'espèce, il s'agit d'une instance de la classe `QVBoxLayout`. Cette instance (nommée `layout`) est attachée à notre widget personnalisé (qui lui est donné en paramètre) : elle va alors en contrôler automatiquement certaines contraintes concernant son aspect. On ajoute à `layout` les deux boutons préalablement créés par la méthode `addWidget()`. Cet "agenceur" particulier a pour effet de placer les widgets qu'on lui donne en colonne (le *VBox* dans `QVBoxLayout` signifiant *boîte verticale*), le premier ajouté étant celui du haut. Nous n'avons ainsi pas à nous préoccuper de la position ni de la taille des boutons : ces propriétés, particulièrement rébarbatives à régler, seront automatiquement prises en charge par `layout`.

Nous en arrivons à la partie essentielle de la création de notre widget : paramétrer la réaction du programme lorsque l'utilisateur clique sur chacun des boutons. Pour cela, Qt utilise la métaphore des *signaux* et des *slots*. Le terme anglais de *slot* n'a pas, dans ce contexte, de bon équivalent en français ; pensez à un *emplacement*, une *cible*, voire une *prise* (au sens prise de courant). L'idée est qu'un objet Qt (par exemple, un bouton) peut "émettre" des signaux (par exemple, "*on m'a cliqué dessus*"). Ces signaux peuvent ensuite être captés par d'autres objets Qt, en provoquant l'exécution de méthodes particulières, les fameux *slots*. À noter qu'un même objet peut à la fois émettre et recevoir des signaux. La seule véritable contrainte étant que les classes impliquées doivent obligatoirement dériver, directement ou non, de la classe `QObject`. Ce qui est le cas de la classe `QWidget` sur laquelle nous nous appuyons ici.

L'association entre un signal émis par un contrôle et une méthode devant y répondre est réalisée par la méthode `connect()`. Celle-ci demande trois ou quatre paramètres, selon la situation. Le premier est l'instance émettrice du signal. Le deuxième est un objet décrivant le signal concerné, obtenu en utilisant la fonction `SIGNAL()` du module `QtCore`, laquelle attend une chaîne de caractères.

Ensuite :

- Si la fonction ou méthode devant être exécutée lorsque le signal est émis est une méthode que vous avez vous-même définie, au sein de la même classe, le troisième et dernier paramètre est simplement le nom de cette méthode, précédé de l'instance concernée, comme c'est le cas ici pour la méthode `demandeNom()`.
- S'il s'agit au contraire d'une méthode propre à Qt, c'est-à-dire venant d'une classe de base de Qt comme `QWidget`, alors le troisième paramètre est l'instance devant recevoir ce signal et le quatrième une référence sur la méthode concernée, obtenue par un appel à la fonction `SLOT()` du module `QtCore`.

Dans notre exemple, la méthode `close()` vient directement de la classe `QWidget`, aussi utilisons-nous l'écriture à quatre paramètres.

Il ne nous reste plus qu'à utiliser cette classe graphique, ce qui est fait dans un programme principal, dans un fichier nommé `salutation_gui.py` :

```
1. import sys
2. from PyQt4 import QtCore, QtGui
3. import salutations
4.
5. app = QtGui.QApplication(sys.argv)
6. salut = salutations.Salutation()
7. salut.show()
8. sys.exit(app.exec_())
```

Voyez comment nous signalons l'accès aux outils (unique dans notre cas) que nous avons définis dans le fichier `salutations.py` : simplement avec la ligne `import salutations` (ligne 3). Nous pouvons dès lors créer une instance de la classe `Salutation` (ligne 6) et lancer la boucle d'événements.

12.4.2. En C++

Comme l'on pouvait s'y attendre, la situation est sensiblement plus complexe en C++. Non pas au niveau du code lui-même, mais dans la façon de l'organiser. En premier lieu, nous devons déclarer la classe `Salutation` dans un fichier d'en-tête, de la façon suivante :

```
1. #ifndef SALUTATIONS_H_
2. #define SALUTATIONS_H_
3. #include <QtGui>
4. class Salutation: public QWidget
5. {
6.     Q_OBJECT
7.     public:
8.         Salutation(QWidget* parent = 0);
```

```

9.   public slots:
10.    void demanderNom();
11.   private:
12.    QPushButton* bt_bonjour;
13.    QPushButton* bt_quitter;
14. };
15. #endif // SALUTATIONS_H__

```

Remarquez que, pour la première fois, ce fichier ne contient aucun code. De la même manière que nous avons, au chapitre précédent, déclaré des fonctions sans en donner le code, ici nous décrivons simplement ce que contient la classe Salutation. Quelques remarques :

- Le mot `Q_OBJECT` à la ligne 6 est nécessaire au fonctionnement interne de Qt : dès que vous créez une classe dérivant directement ou non de la classe `QObject` (ce qui est le cas de la classe `QWidget`), vous devez inscrire `Q_OBJECT` ainsi (c'est ce qu'on appelle une *macro*, qui sera interprétée par le préprocesseur).
- L'indication de section `public slots:`, ligne 9, n'est pas une écriture standard du langage C++ : elle est utilisée par les mécanismes de Qt permettant d'associer un signal à une méthode devant être exécutée lorsqu'il est émis. Ne cherchez pas à utiliser une telle écriture dans un programme n'utilisant pas la bibliothèque Qt, cela ne compilera tout simplement pas.

Nous devons maintenant donner le code devant être exécuté par les différentes méthodes de notre classe, dans un fichier `salutations.cpp` :

```

#include "salutations.h"
Salutation::Salutation(QWidget* parent):
    QWidget(parent),
    bt_bonjour(0)
{
    this->bt_bonjour = new QPushButton("Bonjour !", this);
    this->bt_quitter = new QPushButton("Quitter", this);
    QVBoxLayout* layout = new QVBoxLayout(this);
    layout->addWidget(this->bt_bonjour);
    layout->addWidget(this->bt_quitter);
    this->connect(this->bt_bonjour,
        SIGNAL(clicked()),
        SLOT(demanderNom()));
    this->connect(this->bt_quitter,
        SIGNAL(clicked()),
        this,
        SLOT(close()));
}
void Salutation::demanderNom()
{
    QString nom = QInputDialog::getText(this,

```

```

        "Salutations",
        "Quel est votre nom ?");
    this->bt_bonjour->setText(QString("Bonjour, %1 !").arg(nom));
}

```

Voyez comment ce code est donné : chacune des méthodes est répétée, son nom précédé du nom de la classe, séparé par opérateur :: (qu'on appelle *l'opérateur de résolution de portée*). En dehors de cette écriture particulière, le code même est tout à fait analogue à celui utilisé dans la version en langage Python. Remarquez, toutefois, qu'on ne donne pas de chaînes de caractères aux fonctions SIGNAL() et SLOTS() (qui sont en réalité des macrofonctions, destinées au préprocesseur), mais simplement les noms des méthodes concernées. De même, on utilise intensivement les possibilités de création d'objets dynamiques, comme nous l'avons vu au chapitre précédent.

En termes de code, il ne nous reste plus qu'à écrire le programme principal, dans un fichier `salutation_gui.cpp` :

```

#include <QApplication>
#include "salutations.h"
int main(int argc, char* argv[])
{
    QApplication appli(argc, argv);
    Salutation salut;
    salut.show();
    return appli.exec();
}

```

Enfin, on termine par la création d'un fichier projet, nommé par exemple `salutations.pro` :

```

TEMPLATE = app
Qt += gui
CONFIG += release
OBJECTS_DIR = .objs
MOC_DIR = .moc
DESTDIR = bin
HEADERS += salutations.h
SOURCES += salutations.cpp \
           salutation_gui.cpp

```

Voyez comme on donne explicitement la liste des fichiers d'en-tête (dans la variable `HEADERS`) et de code (dans la variable `SOURCES`) que nous avons créés. D'une manière générale, vous aurez autant de fichiers de code que de fichiers d'en-tête, plus un contenant le programme principal.

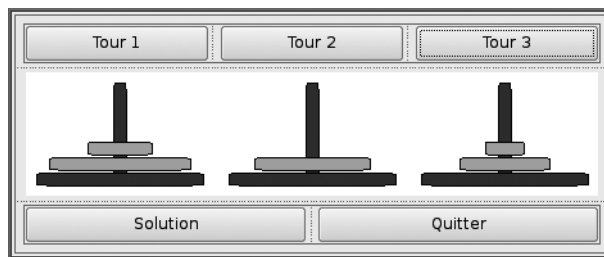
Vous pouvez maintenant générer un fichier de construction à l'aide de l'utilitaire `qmake`, construire votre programme par la commande `make` (ou `mingw32-make` sur Windows), pour enfin pouvoir l'exécuter, le fichier exécutable se trouvant dans un sous-répertoire `bin`.

12.5. Hanoï graphique

Il est maintenant temps de réaliser une version graphique de notre programme des Tours de Hanoï. Nous allons l'étudier de façon exhaustive (du moins la partie graphique), puisque c'est le premier programme aussi complexe que nous réalisons. Par la suite, seules certaines parties pertinentes du code seront examinées.

La Figure 12.3 montre à quoi ressemblera notre version graphique des Tours de Hanoï.

Figure 12.3
Hanoï graphique.



Les rectangles représentent les différents *layouts* (outils d'agencement) qui seront utilisés : on peut en distinguer un grand principal, qui en contient deux autres plus petits pour les boutons. Il est en effet possible d'imbriquer les layouts les uns dans les autres, ce qui rend possible la création d'une interface pouvant être particulièrement complexe sans avoir trop à se préoccuper de la position et de la taille de ses éléments.

12.5.1. En Python

Pour commencer, créez un fichier nommé `hanoi.py` et contenant la classe `Hanoi` que nous avons vue précédemment (ainsi que la fonction calculant une solution). Ne nous attardons pas sur cette partie déjà connue.

La partie centrale du programme est une représentation de l'état du jeu à un moment donné. Plusieurs solutions sont possibles pour obtenir une telle représentation. Le choix a été fait ici de recourir tout simplement au dessin, c'est-à-dire que nous allons afficher une image qui sera créée à la demande. La création et l'affichage même de

cette image seront pris en charge par un widget dédié, nommé Dessin. Voici donc le fichier dessin.py, contenant la définition de ce widget :

```

1. # -*- coding: utf8 -*-
2. from PyQt4 import QtCore, QtGui
3. import hanoi
4. hauteur_disque = 10
5. rayon_unite = 10
6. rayon_aiguille = 5
7. espace_inter_disques = 2
8. espace_inter_tours = 20
9. marge = 8
10. couleur_disques = QtGui.QColor(255, 160, 96)
11. couleur_supports = QtGui.QColor(128, 48, 16)
12. couleur_fond = QtGui.QColor(255, 255, 255)
13. def rayon_max_tour(taille_max):
14.     return \
15.         (taille_max + 1) * rayon_unite + \
16.         rayon_aiguille
17.
18. def hauteur_max_tour(taille_max):
19.     return \
20.         taille_max * (hauteur_disque + espace_inter_disques) + \
21.         2 * (2*rayon_aiguille)
22.
23. class Dessin(QtGui.QWidget):
24.     def __init__(self, le_jeu, parent=None):
25.         QtGui.QWidget.__init__(self, parent)
26.         self.jeu = le_jeu
27.         largeur_max = \
28.             3 * (2*rayon_max_tour(self.jeu.Hauteur())) + \
29.             2 * espace_inter_tours + \
30.             2 * marge
31.         hauteur_max = \
32.             hauteur_max_tour(self.jeu.Hauteur()) + \
33.             2 * marge
34.         self.setFixedSize(largeur_max, hauteur_max)
35.         self.image = QtGui.QPixmap(largeur_max, hauteur_max)
36.         self.Mettre_A_Jour()
37.
38.     def Centre_Etage(self, tour, etage):
39.         centre_tour_x = \
40.             marge + \
41.             (1+2*tour)*rayon_max_tour(self.jeu.Hauteur()) + \
42.             tour * espace_inter_tours
43.         base_y = self.height() - (marge + (2*rayon_aiguille))
44.         etage_y = base_y - \
45.             (espace_inter_disques+hauteur_disque)*(etage+1) + \
46.             (hauteur_disque/2)
47.         return (centre_tour_x, etage_y)
48.
49.     def Mettre_A_Jour(self):
50.         self.image.fill(couleur_fond);

```

```

51.     p = QtGui.QPainter(self.image);
52.     # dessin des supports
53.     p.setPen(QtGui.QColor(0, 0, 0))
54.     p.setBrush(couleur_supports)
55.     base_y = self.height() - marge
56.     for tour in range(3):
57.         centre_tour_x = \
58.             marge + \
59.             (1+2*tour)*rayon_max_tour(self.jeu.Hauteur()) + \
60.             tour * espace_inter_tours
61.         rayon = rayon_max_tour(self.jeu.Hauteur())
62.         # aiguille
63.         p.drawRoundRect(centre_tour_x - rayon_aiguille, \
64.                         base_y - hauteur_max_tour(self.jeu.Hauteur()), \
65.                         2 * rayon_aiguille, \
66.                         hauteur_max_tour(self.jeu.Hauteur()), \
67.                         20, 20)
68.         # base
69.         p.drawRoundRect(centre_tour_x - rayon, \
70.                         base_y - (2*rayon_aiguille), \
71.                         2 * rayon, \
72.                         2 * rayon_aiguille, \
73.                         20, 20)
74.         # dessin des disques
75.         p.setBrush(couleur_disques)
76.         for tour in range(3):
77.             nb_disques = self.jeu.Une_Tour(tour).Nombre_Disques()
78.             for etage in range(nb_disques):
79.                 disque = self.jeu.Une_Tour(tour).Disque_Etage(etage)
80.                 centre = self.Centre_Etage(tour, etage)
81.                 self.Dessiner_Disque(disque, centre, p)
82.         p.end()
83.
84.     def Dessiner_Disque(self, disque, centre, p):
85.         x = centre[0] - rayon_aiguille - disque*rayon_unite
86.         y = centre[1] - hauteur_disque/2
87.         largeur = 2*(disque*rayon_unite + rayon_aiguille)
88.         p.drawRoundRect(x, y, largeur, hauteur_disque, 20, 20)
89.
90.     def paintEvent(self, event):
91.         p = QtGui.QPainter(self)
92.         p.drawPixmap(0, 0, self.image)
93.         p.end()
94.         QtGui.QWidget.paintEvent(self, event)

```

Les premières lignes (4 à 12) ne sont destinées qu'à "nommer" les différentes dimensions que nous allons utiliser par la suite. Pour la clarté du programme, il est en effet préférable d'utiliser des noms que des valeurs brutes : les noms des variables devraient parler d'eux-mêmes. Par exemple, la formule à la ligne 87, qui calcule la largeur d'un disque à l'écran, est plus parlante ainsi que si nous avions écrit :

```
largeur = 2*(disque * 10 + 5)
```


L'unité de ces dimensions est le *pixel* (contraction de *picture element*, élément d'image), c'est-à-dire le point le plus petit que votre écran puisse afficher. Lorsqu'on dit qu'un écran a une résolution de $1\,024 \times 768$ (par exemple), on indique par là qu'on peut afficher un tableau de pixels, d'une largeur de 1 024 colonnes de 768 pixels, ou d'une hauteur de 768 lignes de 1 024 pixels (ce qui en fait au total un peu moins d'un demi-million). Plus la résolution est élevée, plus les pixels sont "petits", la taille réelle d'un pixel dépendant naturellement de la taille réelle de l'écran.

Incidentement, le fait d'utiliser ainsi des variables pour stocker les dimensions permet de faciliter grandement toute modification ultérieure, si vous voulez que la représentation du jeu soit plus grande ou plus petite.

Les trois dernières variables sont destinées à contenir les couleurs qui seront appliquées pour le dessin (voir, ci-après, l'encadré consacré au codage des couleurs). Ces couleurs sont représentées par des instances de la classe `QColor`, une classe fournie par la bibliothèque Qt et obtenue dans le module `QtGui`.

Suivent deux fonctions "utilitaires", permettant de calculer l'espace total occupé par chaque tour en fonction des valeurs précédentes et de la hauteur maximale choisie par l'utilisateur. Il ne s'agit que de calculs élémentaires ne présentant pas de difficultés particulières. À une petite subtilité près : par convention choisie voilà bien des années (même des décennies), les coordonnées de chaque pixel de l'écran sont exprimées à partir du coin supérieur gauche de l'écran (ou de l'image si on dessine dans une image). Dans une résolution de $1\,024 \times 768$, la colonne (ou l'abscisse, ou la coordonnée x) de chaque pixel va de la gauche vers la droite, de 0 à 1 023 ; la ligne de chaque pixel (ou l'ordonnée, ou la coordonnée y) va de haut en bas, de 0 à 767. L'ordonnée est inversée par rapport aux représentations graphiques usuelles. Le pixel de coordonnées (0, 0) est celui tout en haut à gauche, celui de coordonnées (1 023, 0) celui tout en haut à droite, (0, 767) est tout en bas à gauche et (1023, 767) dans le coin en bas à droite. Cette inversion de l'axe des ordonnées peut être déroutante dans un premier temps, mais vous vous y ferez vite avec un peu de pratique. Notamment, cherchez à comprendre les formules utilisées dans le programme.

Les couleurs de l'écran : codage RGB

Les couleurs innombrables que nos écrans modernes sont capables d'afficher sont constituées du mélange de trois couleurs de base, le rouge (*red*), le vert (*green*) et le bleu (*blue*), dans cet ordre, d'où le nom de codage RGB (en anglais) ou RVB (en français). Une couleur est obtenue en mélangeant ces trois couleurs fondamentales selon diverses proportions, tout comme un peintre le fait à l'aide de son pinceau. Par exemple, le jaune est obtenu en mélangeant à parts égales du rouge et du vert. Il existe de multiples façons de spécifier la

quantité, ou *l'intensité*, de chacune des couleurs de base ; toutefois, la plus utilisée consiste à donner un nombre entier entre 0 et 255, la valeur 0 correspondant à une intensité nulle (noir) et la valeur 255, à une intensité maximale. Ainsi, un jaune pur serait donné par la combinaison (255, 255, 0). Une combinaison comme (128, 128, 0) produirait un jaune plus sombre, tandis que (255, 255, 128) donnerait un jaune plus clair.

En effet, le mélange des trois couleurs à pleine intensité, soit la combinaison (255, 255, 255), donne un blanc parfait. Au contraire, le noir absolu est donné par (0, 0, 0). D'un extrême à l'autre, vous disposez de 16 777 216 couleurs différentes, exactement.

Notez qu'il existe d'autres façons de donner la proportion de chacune de ces couleurs, par exemple par le biais d'un nombre décimal entre 0.0 et 1.0. L'exactitude de la représentation dépend en général du matériel disponible : ainsi, certaines cartes graphiques associées à des écrans de très haut de gamme peuvent afficher des *milliards* de couleurs.

Enfin, d'autres représentations des couleurs existent, comme le codage HSV (fondé sur un cercle chromatique) ou le codage CMYK (mélange de cyan, magenta, jaune et noir, couramment utilisé sur les imprimantes). Nous ne les aborderons pas ici.

Nous en arrivons à la classe `Dessin`, le widget chargé d'afficher une image représentant l'état du jeu. Le constructeur de cette classe reçoit en paramètre une instance de la classe `Hanoi`, nécessaire afin de calculer les diverses dimensions et de tracer la représentation. La méthode `setFixedSize()` (ligne 34), issue de la classe `QWidget`, permet de fixer une taille au widget : ainsi, les layouts ne pourront pas changer sa taille, ce qui permet d'éviter des effets disgracieux sur le dessin (il serait cependant possible d'adapter le dessin à un changement de taille, mais cela deviendrait vraiment compliqué). Enfin, l'image elle-même est représentée par une instance de la classe `QPixmap`, classe optimisée spécialement pour être affichée à l'écran. Son constructeur attend en paramètre les dimensions de l'image, d'abord la largeur, puis la hauteur.

L'essentiel se passe dans la méthode `Mettre_A_Jour()` (à partir de la ligne 49). C'est là que va effectivement être dessinée l'image représentant le jeu. La première action est de "vider" l'image, c'est-à-dire d'obtenir une image vierge, remplie de la couleur de fond choisie au début du fichier (ici du blanc). Cela est réalisé par la méthode `fill()` de la classe `QPixmap` : tous les pixels de l'image sont alors blancs, prêts à recevoir un nouveau dessin.

Pour dessiner, il faut un pinceau, justement fourni par la classe `QPainter` de Qt. Celle-ci contient de nombreuses méthodes permettant de tracer les points individuels, mais aussi des rectangles, des ellipses, des courbes de Bézier... La plupart des formes pouvant être dessinées acceptent une couleur de bord et une couleur de remplissage. La première est définie par la méthode `setPen()` de `QPainter`, la seconde par la méthode `setBrush()`.

Les supports des disques sont dessinés en premier. Il importe ici de comprendre que le dessin que nous réalisons est tout à fait comparable à celui qu'on ferait avec de l'encre sur une feuille de papier : ce qui est dessiné apparaîtra toujours "par-dessus" ce qui se trouvait préalablement au même endroit.

Le dessin lui-même est effectué par la méthode `drawRoundRect()` de `QPainter`, qui permet d'obtenir des rectangles aux coins arrondis. Les valeurs numériques qui lui sont passées sont des coordonnées et des dimensions exprimées en pixels, toujours avec l'origine des coordonnées dans le coin supérieur gauche.

Enfin, la dernière méthode intéressante est la méthode `paintEvent()`, héritée de la classe `QWidget`. C'est elle qui est invoquée lorsqu'il est nécessaire de redessiner le widget à l'écran, que cela soit lors de sa première apparition ou lorsqu'il se retrouve exposé après avoir été caché par une autre fenêtre. Nous utilisons à nouveau une instance de `QPainter`, simplement pour "plaquer" l'image sur le widget. Remarquez que nous invoquons pour terminer la méthode `paintEvent()` de la classe ancêtre, ce qui est généralement recommandé.

La classe principale représentant l'ensemble de la fenêtre, ainsi que le programme principal permettant de lancer l'application, se trouve dans un fichier `hanoi_gui.py` :

```
# -*- coding: utf8 -*-
import sys
from PyQt4 import QtCore, QtGui
import hanoi
import dessin

class Hanoi_Gui(QtGui.QWidget):
    def __init__(self, le_jeu, parent):
        QtGui.QWidget.__init__(self, parent)
        self.jeu = le_jeu
        self.nb_mouvements = 0
        self.tour_depart = -1
        colonne = QtGui.QVBoxLayout(self)
        ligne_boutons_tours = QtGui.QHBoxLayout()
        self.bt_tour_1 = QtGui.QPushButton("Tour 1", self)
        ligne_boutons_tours.addWidget(self.bt_tour_1)
        self.connect(self.bt_tour_1,
                     QtCore.SIGNAL("clicked()"),
                     self.Tour_Choisie)
        self.bt_tour_2 = QtGui.QPushButton("Tour 2", self)
        ligne_boutons_tours.addWidget(self.bt_tour_2)
        self.connect(self.bt_tour_2,
                     QtCore.SIGNAL("clicked()"),
                     self.Tour_Choisie)
```

```

self.bt_tour_3 = QtGui.QPushButton("Tour 3", self)
ligne_boutons_tours.addWidget(self.bt_tour_3)
self.connect(self.bt_tour_3,
              QtCore.SIGNAL("clicked()"),
              self.Tour_Choisie)
colonne.addLayout(ligne_boutons_tours)
self.dessin = dessin.Dessin(self.jeu, self)
colonne.addWidget(self.dessin)
ligne_boutons_fonctions = QtGui.QHBoxLayout()
bt_solution = QtGui.QPushButton("Solution", self)
ligne_boutons_fonctions.addWidget(bt_solution)
self.connect(bt_solution,
              QtCore.SIGNAL("clicked()"),
              self.Solution)
bt_quitter = QtGui.QPushButton("Quitter", self)
ligne_boutons_fonctions.addWidget(bt_quitter)
self.connect(bt_quitter,
              QtCore.SIGNAL("clicked()"),
              QtGui.QApp,
              QtCore.SLOT("quit()"))
colonne.addLayout(ligne_boutons_fonctions)
self.dessin.Mettre_A_Jour()

def Jeu(self):
    return self.jeu

def Tour_Choisie(self):
    if ( self.tour_depart < 0 ):
        # on a choisi la tour de départ
        self.tour_depart = self.Bouton_Tour_Choisie(self.sender())
        if ( self.tour_depart < 0 ):
            return
    if ( self.Jeu().Tour_Depart_Valide(self.tour_depart) ):
        if ( self.tour_depart != 0 ):
            self.bt_tour_1.setEnabled(
                self.Jeu().Tour_Arrivee_Valide(self.tour_depart, 0))
        if ( self.tour_depart != 1 ):
            self.bt_tour_2.setEnabled(
                self.Jeu().Tour_Arrivee_Valide(self.tour_depart, 1))
        if ( self.tour_depart != 2 ):
            self.bt_tour_3.setEnabled(
                self.Jeu().Tour_Arrivee_Valide(self.tour_depart, 2))
    else:
        self.tour_depart = -1;
        QtGui.QMessageBox.warning(self,
                                   self.trUtf8("Tours de Hanoi"),
                                   self.trUtf8("Tour de départ invalide !"))

```



```
app.trUtf8("Nombre de disques ?"),
3, 2, 8)

le_jeu = hanoi.Hanoi()
le_jeu.Initialiser(hauteur_max[0])
gui = Hanoi_Gui(le_jeu, None)
gui.show()
sys.exit(app.exec())
```

L'idée est que l'utilisateur choisit les tours de départ et d'arrivée à l'aide de trois boutons prévus à cet effet. Ces trois boutons sont connectés à une unique méthode, `Tour_Choisie()`, dont le rôle sera de vérifier les choix et de réaliser effectivement le déplacement d'un disque (ainsi que de demander à l'instance de `Dessin`, que la classe contient, de se remettre à jour). Une petite subtilité a été introduite : en fonction de la tour choisie comme point de départ, les autres boutons sont éventuellement grisés (méthode `setEnabled()` issue de `QWidget`) afin que l'utilisateur ne puisse pas choisir un déplacement interdit.

Autre subtilité, la méthode `Bouton_Tour_Choisie()` qui permet de retrouver le numéro d'une tour à partir du bouton cliqué. Cela est permis par le fait qu'il existe une méthode `sender()`, chargée de fournir une référence sur l'objet, l'émetteur, ayant provoqué l'exécution de la méthode `Tour_Choisie()` (qui est un slot dans le vocable Qt). Sans cette possibilité, il aurait été nécessaire de connecter chacun des boutons à une méthode différente, afin de les identifier.

En regardant ce code, sans doute vous êtes-vous demandé ce qu'était cette fonction (une méthode en réalité, fournie par la classe de base `QObject`) `trUtf8()` fréquemment employée pour les chaînes de caractères à afficher. Son rôle est de transformer la chaîne de caractères qui lui est donnée en paramètre en une instance de la classe `QString`. Cette classe de Qt est également un type permettant la manipulation de chaînes de caractères, mais la chaîne est codée selon le standard Unicode. Cela donne l'accès à un catalogue beaucoup plus vaste de caractères, facilitant ainsi l'affichage dans des langues possédant d'autres alphabets (langues arabes ou asiatiques, mais également certaines langues européennes comme le polonais ou le norvégien). Qt offre en effet un support assez complet de l'*internationalisation*, c'est-à-dire un ensemble d'outils permettant d'adapter un programme en différentes langues. Nous n'aborderons pas cet aspect ici.

Il n'aura pas échappé au lecteur attentif que rien ne semble prévu pour montrer, à l'utilisateur, la solution au problème. Nous verrons cela au Chapitre 14, qui mettra en œuvre une petite animation pour le déplacement des disques.

12.5.2. En C++

Comme on pouvait s'y attendre, la situation en C++ est un peu plus complexe. Pour commencer, il est nécessaire de séparer en deux fichiers les déclarations de nos classes et le code qu'elles devront effectivement exécuter. Voici le fichier `dessin.h`, fichier d'en-tête pour la classe `Dessin` (chargée d'afficher l'état du jeu) :

```
#ifndef DESSIN_H__
#define DESSIN_H__ 1
#include <QWidget>
#include <QPixmap>
#include "hanoi.h"
class Dessin: public QWidget
{
    Q_OBJECT
public:
    Dessin(const Hanoi& le_jeu,
           QWidget* parent = 0);
    QPoint Centre_Etage(int tour,
                        int etage) const;

public slots:
    void Mettre_A_Jour();
protected:
    void Dessiner_Disque(int disque,
                        const QPoint& centre,
                        QPainter& p) const;
    virtual void paintEvent(QPaintEvent* event);
private:
    QPixmap image;
    const Hanoi& jeu;
}; // class Dessin
#endif // DESSIN_H__
```

Rien de bien exceptionnel ici. La classe `QPoint`, fournie par Qt, représente simplement les coordonnées d'un point à l'écran. Elle apporte un moyen commode pour "transporter" des coordonnées d'une fonction à l'autre.

Remarquez, tout de même, la façon dont nous conservons l'instance sur le jeu (donnée jeu). On utilise une référence constante, façon de garantir que cette classe d'affichage ne va pas modifier le jeu lui-même.

Voyons maintenant l'implémentation de tout cela, dans le fichier `dessin.cpp`.

```
#include <QPainter>
#include "dessin.h"
namespace
{
```

```
const int hauteur_disque = 10;
const int rayon_unite = 10;
const int rayon_aiguille = 5;
const int espace_inter_disques = 2;
const int espace_inter_tours = 20;
const int marge = 8;
const QColor couleur_disques(255, 160, 96);
const QColor couleur_supports(128, 48, 16);
const QColor couleur_fond(Qt::white);
int rayon_max_tour(int taille_max)
{
    return
        (taille_max + 1) * rayon_unite +
        rayon_aiguille;
}
int hauteur_max_tour(int taille_max)
{
    return
        taille_max * (hauteur_disque + espace_inter_disques) +
        2 * (2*rayon_aiguille);
}
}
```

On retrouve ici les diverses constantes décrivant la géométrie de notre affichage, ainsi que les deux fonctions utilitaires `rayon_max_tour()` et `hauteur_max_tour()`. Voyez comment chacune des valeurs est qualifiée par `const`, toujours pour leur garantir une certaine cohérence et éviter des modifications intempestives. De plus, tous ces éléments sont contenus dans un bloc namespace `{ }`.

Le mot clé `namespace` signifie *espace de nommage*. Il permet de définir une sorte de catégorie au sein de laquelle les différents éléments doivent avoir un nom unique. Depuis le début de cet ouvrage, nous utilisons couramment un espace de nommage : celui nommé `std`, réservé pour les outils de la bibliothèque standard du langage C++. Et c'est bien là le sens des écritures telles que `std::cout` : on désigne l'objet nommé `cout`, dans la catégorie nommée `std`. De cette façon, vous pouvez parfaitement créer vous-même un objet nommé `cout` au sein de votre programme : il n'y aura pas de confusion sur les noms.

Nous utilisons ici un espace de nommage anonyme, c'est-à-dire sans nom. Sa fonction est uniquement d'indiquer que ce qu'il contient est propre au fichier dans lequel il se trouve et ne saurait être accessible depuis une autre partie du programme. On retrouve une exigence de cohérence et de séparation nette entre les différentes parties d'un programme.

Mais poursuivons dans ce fichier `dessin.cpp`.


```

Dessin::Dessin(const Hanoi& le_jeu,
               QWidget* parent):
    image(0,0),
    jeu(le_jeu)
{
    const int largeur_max =
        3 * (2*rayon_max_tour(this->jeu.Hauteur())) +
        2 * espace_inter_tours +
        2 * marge;
    const int hauteur_max =
        hauteur_max_tour(this->jeu.Hauteur()) +
        2 * marge;
    this->setFixedSize(largeur_max, hauteur_max);
    image = QPixmap(largeur_max, hauteur_max);
    this->Mettre_A_Jour();
}

QPoint Dessin::Centre_Etage(int tour,
                             int etage) const
{
    const int centre_tour_x =
        marge +
        (1+2*tour)*rayon_max_tour(this->jeu.Hauteur()) +
        tour * espace_inter_tours;
    const int base_y = this->height() - (marge + (2*rayon_aiguille));
    const int etage_y = base_y -
        (espace_inter_disques+hauteur_disque)*(etage+1) +
        (hauteur_disque/2);
    return QPoint(centre_tour_x, etage_y);
}

void Dessin::Mettre_A_Jour()
{
    this->image.fill(couleur_fond);
    QPainter p(&(this->image));
    // dessin des supports
    p.setPen(Qt::black);
    p.setBrush(couleur_supports);
    const int base_y = this->height() - marge ;
    for(int tour = 0; tour < 3; ++tour)
    {
        const int centre_tour_x =
            marge +
            (1+2*tour)*rayon_max_tour(this->jeu.Hauteur()) +
            tour * espace_inter_tours;
        const int rayon = rayon_max_tour(this->jeu.Hauteur());
        // aiguille

```

```

        p.drawRoundRect(centre_tour_x - rayon_aiguille,
                        base_y - hauteur_max_tour(this->jeu.Hauteur()),
                        2 * rayon_aiguille,
                        hauteur_max_tour(this->jeu.Hauteur()),
                        20, 20);

    // base
    p.drawRoundRect(centre_tour_x - rayon,
                    base_y - (2*rayon_aiguille),
                    2 * rayon,
                    2 * rayon_aiguille,
                    20, 20);
}
// dessin des disques
p.setPen(Qt::black);
p.setBrush(couleur_disques);
for(int tour = 0; tour < 3; ++tour)
{
    const int nb_disques = this->jeu.Un_Tour(tour).Nombre_Disques();
    for(int etage = 0; etage < nb_disques; ++etage)
    {
        const int disque = this->jeu.Un_Tour(tour).Disque_Etage(etage);
        QPoint centre = this->Centre_Etage(tour, etage);
        this->Dessiner_Disque(disque, centre, p);
    }
}
p.end();
}
void Dessin::Dessiner_Disque(int disque,
                             const QPoint& centre,
                             QPainter& p) const
{
    const int x = centre.x() - rayon_aiguille - disque*rayon_unite;
    const int y = centre.y() - hauteur_disque/2;
    const int largeur = 2*(disque*rayon_unite + rayon_aiguille);
    p.drawRoundRect(x, y, largeur, hauteur_disque, 20, 20);
}
void Dessin::paintEvent(QPaintEvent* event)
{
    QPainter p(this);
    p.drawPixmap(0, 0, this->image);
    p.end();
    QWidget::paintEvent(event);
}

```

Comme vous pouvez le constater, le code est parfaitement comparable à celui que nous avons vu dans la version Python. Aussi n'allons-nous pas le détailler davantage. Les différences essentielles sont imposées par le changement de langage, notamment dans l'utilisation explicite de références et de pointeurs en C++.

Le fichier `hanoi_gui.h` contient la déclaration de la classe `Hanoi_Gui`, qui est, comme précédemment, la classe représentant l'interface de notre programme :

```
#ifndef HANOI_GUI_H__
#define HANOI_GUI_H__ 1
#include <QWidget>
#include <QPushButton>
#include "hanoi.h"
#include "dessin.h"
class Hanoi_Gui: public QWidget
{
    Q_OBJECT
public:
    Hanoi_Gui(Hanoi& le_jeu,
              QWidget* parent = 0);
    Hanoi& Jeu();
    const Hanoi& Jeu() const;
public slots:
    void Tour_Choisie();
    void Solution();
protected:
    int Bouton_Tour_Choisie(QObject* bt) const;
private:
    Dessin*      dessin;
    QPushButton* bt_tour_1;
    QPushButton* bt_tour_2;
    QPushButton* bt_tour_3;
    int          tour_depart;
    int          nb_mouvements;
    Hanoi&       jeu;
}; // class Hanoi_Gui
#endif // HANOI_GUI_H__
```

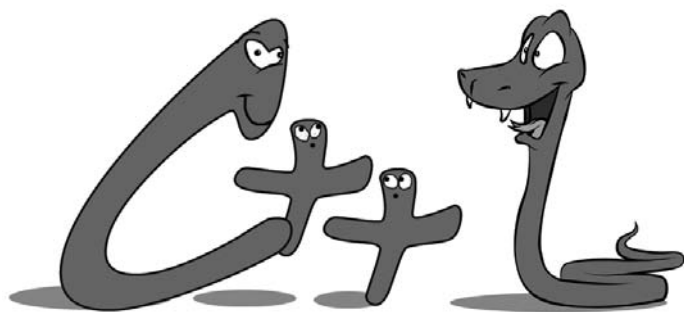
Encore une fois, rien de bien spécial ici. L'implémentation des méthodes ne présente aucune difficulté particulière, le code étant presque équivalent à celui du programme Python (toujours aux différences de syntaxe près). Aussi, pouvons-nous passer directement au programme principal, dans `main.cpp` :

```
#include <QApplication>
#include <QInputDialog>
#include "hanoi.h"
#include "hanoi_gui.h"
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    int hauteur_max;
```

```
    hauteur_max =
        QDialog::getInteger(0,
                            app.trUtf8("Tours de Hanoi"),
                            app.trUtf8("Nombre de disques ?"),
                            3, 2, 8);

    Hanoi le_jeu;
    le_jeu.Initialiser(hauteur_max);
    Hanoi_Gui gui(le_jeu);
    gui.show();
    return app.exec();
}
```

Et même ce programme principal est tout ce qu'il y a de plus classique ! Si vous prenez l'habitude de structurer ainsi vos programmes, même les plus complexes, vous ne devriez pas rencontrer de problèmes majeurs pour vous y retrouver.



Stockage de masse : manipuler les fichiers

Au sommaire de ce chapitre :

- Écriture
- Lecture
- Aparté : la ligne de commande

On désigne habituellement par *stockage de masse* l'enregistrement de données sur un support physique, généralement de grande capacité, et capable de conserver ces données même sans être alimenté électriquement. Par exemple, un disque dur (dont vous pouvez voir l'intérieur à la Figure 13.1 – mais ne faites pas cela, un disque dur ouvert ne fonctionne plus !). La différence essentielle avec les données que vous pouvez stocker en mémoire lors de l'exécution d'un programme est justement que le stockage de masse permet d'en mémoriser beaucoup plus et de façon permanente.

Figure 13.1

*Vue de l'intérieur
d'un disque dur.*



Les données ainsi stockées sont organisées en fichiers, chacun possédant sa propre organisation "interne" définie par le programme l'ayant créé. On distingue habituellement deux types de fichiers :

- Les *fichiers texte*, dont le contenu est constitué uniquement de caractères lisibles par un être humain. Ces caractères sont organisés en lignes, chaque fin de ligne étant marquée par un caractère spécial (ou deux caractères spéciaux, selon le système sur lequel vous vous trouvez). Les fichiers contenant les codes source de vos programmes, une page Web visualisée dans un navigateur, sont des fichiers texte.
- Les *fichiers binaires*, dont le contenu est comparable à ce qui se trouve réellement dans la mémoire de l'ordinateur lors de l'exécution du programme. Les informations sont alors généralement plus "compactes" que dans le cas d'un fichier texte et ne sont pas lisibles par un œil humain (à moins qu'il ne soit extrêmement entraîné).

Les fichiers résultants d'une compilation, les photos issues d'un appareil numérique ou d'un scanner, sont des fichiers binaires.

Le type de fichier à utiliser dépend du programme que vous développez. D'une manière générale, les fichiers binaires sont privilégiés lorsque l'encombrement sur disque et la rapidité de lecture et d'écriture sont des considérations importantes. On préférera plutôt les fichiers texte si on souhaite obtenir un fichier pouvant être lu facilement par n'importe quel éditeur de texte.

Nous allons ici nous limiter délibérément aux fichiers texte, car ce sont encore les plus simples à manipuler. Comme vous allez le constater, il n'y a pas grande différence avec les techniques d'entrées-sorties que nous avons vues depuis le Chapitre 4.

13.1. Écriture

Pour commencer, nous allons créer un programme qui permet de stocker dans un fichier la liste des déplacements nécessaires pour résoudre le jeu des Tours de Hanoï. Cela peut être utile pour transmettre cette information capitale à un ami, sans devoir noter scrupuleusement ces déplacements à la main, avec tous les risques d'erreurs que cela comporte.

Voici donc le programme `ecrire_solution` :

Python

```
# -*- coding: utf8 -*-
import hanoi
print "Nombre de disques ?",
hauteur_max = int(raw_input())
le_jeu = hanoi.Hanoi()
le_jeu.Initialiser(hauteur_max);
solution = []
le_jeu.Solution(solution)
fichier = file("solution", "w")
for mouvement in solution:
    print >> fichier, \
        mouvement[0]+1, \
        mouvement[1]+1
fichier.close()
```

C++

```
#include <iostream>
#include <fstream>
#include "hanoi.h"
int main(int argc, char* argv[])
{
    int nb_disques = 3;
    std::cout << "Nombre de disques ? ";
    std::cin >> nb_disques;
    Hanoi jeu;
    jeu.Initialiser(nb_disques);
    std::vector<std::pair<int, int> >solution;
    jeu.Solution(solution);
    std::ofstream fichier("solution");
    for(int mouvement = 0;
        mouvement < solution.size();
        ++mouvement)
```



Python	C++
	<pre> { fichier << solution[mouvement].first+1 << " " << solution[mouvement].second+1 << std::endl; } fichier.close(); return 0; } </pre>

L'écriture dans un fichier se fait en créant une instance d'un type particulier : le type `file` en Python, le type `std::ofstream` en C++. La simple création de cette instance provoque l'*ouverture* du fichier, c'est-à-dire que le programme signale au système d'exploitation qu'il s'apprête à manipuler un fichier. Le nom de ce fichier est passé en argument à la création de l'instance (vous l'aurez compris, `file` et `std::ofstream` sont en réalité des classes).

Le type C++ `std::ofstream` représente un flux de sortie vers un fichier (o pour *out*, sortie ; f pour *file*, fichier ; *stream* est l'anglais de flux). Cela signifie que l'instance ne permet que d'écrire dans le fichier, pas d'en lire le contenu. Pour Python, le "sens d'ouverture" (lecture ou écriture) est déterminé par un second paramètre passé au constructeur de `file`, une chaîne de caractères. Ici, la chaîne "w" (pour *write*, écrire) indique que nous allons écrire dans le fichier : l'instance ne nous permettra pas de lire depuis le fichier.

Dans les deux cas, si un fichier du même nom existe déjà (dans le même répertoire), il est tout simplement "écrasé", c'est-à-dire que le contenu précédent est irrémédiablement perdu. Si ce n'est pas ce que vous souhaitez, si vous voulez ajouter des lignes à un fichier existant sans perdre celles déjà présentes :

- En C++, passez en second paramètre la valeur `std::ios::app`.
- En Python, passez la chaîne "w+" en second paramètre.

L'écriture en elle-même est tout à fait comparable à l'affichage à l'écran que nous avons vu depuis le début de cet ouvrage. Elle est même parfaitement équivalente en C++, tandis que Python nécessite l'emploi d'une notation particulière pour l'instruction `print` : la faire suivre de deux chevrons fermants (`>>`), suivis de l'instance de `file` précédemment créée, avant la liste de ce que l'on veut inscrire.

Lorsque l'écriture est terminée, il convient de *fermer* le fichier, au moyen de la méthode `close()`. On signale ainsi au système d'exploitation que l'on a terminé. Ces deux opérations symétriques, l'ouverture et la fermeture d'un fichier, sont assez importantes pour le système sous-jacent : elles permettent la mise en œuvre de diverses techniques d'optimisation qui conduisent à un meilleur "répondant" du système.

13.2. Lecture

Maintenant que nous avons stocké dans un fichier les mouvements nécessaires à la résolution du problème, il serait tout aussi intéressant de relire ce fichier afin d'en extraire cette précieuse information. Voici le programme `lire_solution` :

Python

```
# -*- coding: utf8 -*-
fichier = file("solution", "r")
while ( fichier ):
    ligne = fichier.readline()
    mouvement = ligne.split()
    if ( len(mouvement) == 2 ):
        print int(mouvement[0]), \
              "->", \
              int(mouvement[1])
fichier.close()
```

C++

```
#include <iostream>
#include <fstream>
int main(int argc, char* argv[])
{
    std::ifstream fichier("solution");
    while ( not fichier.eof() )
    {
        std::string ligne;
        int depart;
        int arrivee;
        fichier >> depart;
        fichier >> arrivee;
        std::cout << depart << " -> "
                  << arrivee << "\n";
    }
    fichier.close();
    return 0;
}
```

La lecture se fait comme l'écriture, en obtenant une instance sur un type dédié à cette tâche. On retrouve le type `file` en Python, cette fois construit avec en second paramètre la chaîne `"r"` (pour *read*, lire). En C++, on utilise plutôt le type `std::ifstream` (i pour *input*, entrée). Si vous remplacez mentalement l'utilisation des objets fichier dans ces programmes par les moyens usuels pour recevoir des données de l'utilisateur, vous constaterez une grande similarité avec ce que nous avons lors de l'étude de l'interactivité, notamment la saisie de plusieurs valeurs (voir la section 4 du Chapitre 4).

Le problème essentiel consiste à "détecter" lorsqu'on arrive à la fin du fichier. En effet, cela n'aurait pas de sens de chercher à lire au-delà. C'est pour cela qu'on utilise une boucle `while` pour la lecture : le fichier est lu ligne à ligne (pour Python, en utilisant la méthode `readline()` de l'instance de `file`) ou information par information (pour C++, en utilisant les chevrons comme nous le faisons habituellement avec `std::cin`), *tant que* nous ne sommes pas à la fin. Dans le cas de Python, il suffit de tester l'instance de `file`, test qui échouera dès qu'on aura terminé la lecture. Dans le cas de C++, on fait appel à la méthode `eof()` de `std::ifstream` (pour *End Of File*, fin de fichier), laquelle retourne vrai (*true*) dès que l'on est effectivement à la fin du fichier.

Incidemment, remarquez que nous n'avons absolument pas utilisé les outils réalisés pour les Tours de Hanoi. Un bon exercice consisterait à modifier ce programme, afin de lui faire "simuler" la solution du jeu, comme nous l'avons vu au Chapitre 10.

13.3. Aparté : la ligne de commande

Probablement serez-vous tenté un jour d'écrire un programme qui lit les lignes d'un fichier, effectue une certaine opération sur chacune, puis écrit un nouveau fichier résultat de la transformation. C'est ce qu'on appelle un filtre. Idéalement, ce filtre pourrait fonctionner par la ligne de commande, de la même façon que les commandes classiques que sont `copy`, `cd` ou `mkdir`. Autrement dit, il doit être possible de passer des paramètres au programme, qui sont par nature inconnus au moment de son écriture.

Il existe un moyen assez simple pour cela, tant en Python qu'en C++. Modifions le programme d'écriture précédent, afin qu'il accepte en paramètre le nombre de disques et le fichier devant les stocker :

Python

```
# -*- coding: utf8 -*-
import sys
import hanoi
hauteur_max = int(sys.argv[1])
le_jeu = hanoi.Hanoi()
le_jeu.Initialiser(hauteur_max);
solution = []
le_jeu.Solution(solution)
fichier = file(sys.argv[2], "w")
for mouvement in solution:
    print >> fichier, \
        mouvement[0]+1, \
```

C++

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "hanoi.h"
int main(int argc, char* argv[])
{
    int nb_disques = atoi(argv[1]);
    Hanoi jeu;
    jeu.Initialiser(nb_disques);
    std::vector<std::pair<int,
        ➤int> > solution;
```



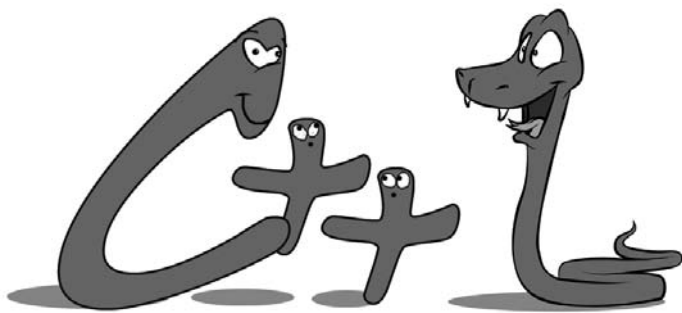
Python	C++
<pre>mouvement[1]+1 fichier.close()</pre> <p>Exemple d'utilisation (solution pour trois disques stockée dans un fichier <code>soluce.txt</code>):</p> <pre>\$ python ecriture.py 3 soluce.txt</pre>	<pre>jeu.Solution(solution); std::ofstream fichier(argv[2]); /* inchangé... */ return 0; }</pre> <p>Exemple d'utilisation (solution pour trois disques stockée dans un fichier <code>soluce.txt</code>):</p> <pre>\$./ecriture 3 soluce.txt</pre> <p>ou sur Windows :</p> <pre>> ecriture 3 soluce.txt</pre>

Les lignes modifiées ont été mises en évidence. Et vous voyez maintenant le sens des paramètres `argc` et `argv` passés à la fonction principale `main()` en C++.

Le paramètre `argv` est, à strictement parler, un tableau de pointeurs sur données de type `char`, soit un tableau de tableaux de caractères – autrement dit, un tableau de chaînes de caractères "brutes", ne fournissant aucune des facilités du type `std::string` que nous utilisons depuis le début. Chacune de ces chaînes contient l'un des paramètres passés au programme lors de son exécution. L'équivalent en Python est le tableau de chaînes de caractères `argv`, auquel on accède par le module `sys`. Le paramètre C++ `argc`, quant à lui, contient le nombre de paramètres. Normalement, ce nombre vaut toujours au moins 1, le premier paramètre (celui d'indice 0) étant, par convention, le nom du programme lui-même. Pour retrouver cette valeur en Python, il suffit de demander la taille du tableau par `len(sys.argv)`.

Chacun des éléments de ces tableaux peut donc être vu comme une chaîne de caractères. Or, dans notre cas, nous attendons en premier paramètre un nombre entier. Si cela est simple en Python, par l'utilisation de la pseudo-fonction `int()`, en C++ nous devons faire appel à une fonction de conversion spécialisée pour ce genre de situation : la fonction `atoi()`, qui attend en paramètre un type `char*` (une chaîne de caractères standard du langage C) et retourne un entier. Cette fonction est disponible dans l'en-tête `<cstdlib>`.

Naturellement, dans un véritable programme, il conviendrait de vérifier qu'on ne cherche pas à obtenir plus de paramètres qu'il n'en a été passé. Si vous ne donnez qu'un seul paramètre aux programmes précédents, ceux-ci planteront à coup sûr.



14

Le temps qui passe

Au sommaire de ce chapitre :

- Réalisation d'un chronomètre
- Animation

Comme annoncé au Chapitre 12, nous allons ajouter à la version graphique des Tours de Hanoï l'affichage de la solution au problème. Une façon de faire serait de tout simplement simuler les déplacements en mettant à jour l'affichage après chacun d'eux. Mais cela poserait au moins deux problèmes :

- Des changements limités à la position d'un unique disque d'une image à l'autre sont assez difficiles à suivre pour l'œil humain : un tel affichage ne serait donc pas très attrayant, voire pas suffisamment démonstratif pour l'utilisateur.
- Plus grave, les machines actuelles sont tellement puissantes et performantes que les différentes images se succéderaient à une vitesse bien supérieure à ce que l'œil humain est capable de suivre : un utilisateur ne verrait qu'un bref clignotement entre le début et la fin de la démonstration.

Bref, il faut faire mieux que cela. Une idée consiste alors à animer les déplacements des disques, c'est-à-dire à proposer une simulation plus proche de la réalité. Mais qui dit animation, dit succession d'images dans le temps... Nous devons donc avoir une certaine maîtrise du temps.

Dans le monde de la programmation, la notion de temps est parfois assez floue et incertaine. Notamment dans le cas de systèmes d'exploitation qui "font comme si" ils permettaient à plusieurs programmes de s'exécuter simultanément. En réalité, sur une machine ne disposant que d'un unique processeur, un seul programme s'exécute à un instant donné. Sur une machine équipée de deux processeurs ou d'un processeur à double cœur, deux programmes au plus peuvent s'exécuter simultanément. Et on entend ici "programme" au sens large, ce qui inclut le système lui-même.

Ce qui se passe en réalité, c'est que le système "prend la main" à intervalles réguliers et attribue une certaine quantité de temps à chacun des programmes qui s'exécutent, l'un après l'autre. Cette opération est tellement rapide qu'elle passe presque inaperçue (sauf dans le cas d'une activité particulièrement intense). Mais cela signifie surtout que le temps écoulé entre le moment où votre programme démarre et le moment où il s'arrête, n'est pas équivalent au temps "réel" durant lequel son code a effectivement été exécuté – le second est généralement inférieur au premier.

Un mot justement, au sujet du terme *temps réel*. Un système est dit temps réel si son bon fonctionnement dépend de réponses devant impérativement être données dans un intervalle de temps précis, et ce quelle que soit la charge du système. Cet intervalle n'est pas nécessairement court (bien que cela soit souvent le cas). Par exemple, sur une voiture, le système évitant le blocage des roues en cas de freinage brutal est

un système temps réel : quelles que soient les tâches en cours d'exécution sur le calculateur, celui-ci *doit* impérativement donner une réponse en un temps donné et fixé.

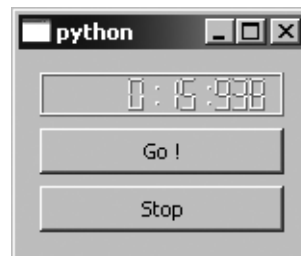
Cette notion ne doit pas être confondue avec celle de *haute performance*. Celle-ci consiste à effectuer un grand nombre de calculs en un minimum de temps, sans que l'exactitude du résultat dépende de ce temps. Les jeux dits temps réel n'ont (le plus souvent) rien de temps réel : ils s'efforcent d'utiliser au mieux la puissance disponible pour fournir des réactions aussi rapides que possible. Mais que la charge du système augmente, par exemple du fait d'une autre application consommatrice de puissance, et les temps de réponse du jeu seront inexorablement augmentés.

Fermons cette parenthèse pour nous pencher sur notre problème.

14.1. Réalisation d'un chronomètre

Avant de nous attaquer au gros morceau du programme des Tours de Hanoï, commençons modestement par la réalisation d'un simple chronomètre : un bouton pour démarrer, un bouton pour arrêter et l'affichage du temps écoulé. La Figure 14.1 montre le programme que nous allons réaliser.

Figure 14.1
*Un chronomètre
élémentaire.*



Lorsque l'utilisateur clique sur le bouton Go, le programme déclenche un minuteur, un objet tout à fait comparable à ces horribles instruments qu'on trouve communément dans les cuisines : vous demandez un temps d'attente, subissez le tic-tac, puis sursautez à l'infernale sonnerie. La classe `QTimer` de Qt propose une fonction semblable, sauf qu'elle est beaucoup plus précise (de l'ordre de quelques dizaines de millisecondes) et que la sonnerie prend la forme d'un signal auquel le programme va réagir. Les calculs sur des durées sont, quant à eux, effectués à l'aide de la classe `QTime`.

Voici le code du programme chrono, limité à la classe contenant l'interface :

Python

```
class Chrono(QtGui.QWidget):
    def __init__(self, \
                 le_jeu, \
                 parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.colonne = \
            QtGui.QVBoxLayout(self)
        self.affichage = \
            QtGui.QLCDNumber(12, self)
        self.colonne.addWidget( \
            self.affichage)
        self.bt_go = \
            QtGui.QPushButton("Go !", self)
        self.colonne.addWidget(self.bt_go)
        self.connect(self.bt_go, \
                     QtCore.SIGNAL("clicked()"), \
                     self.Go)
        self.bt_stop = \
            QtGui.QPushButton("Stop", self)
        self.colonne.addWidget(self.bt_stop)
        self.connect(self.bt_stop, \
                     QtCore.SIGNAL("clicked()"), \
                     self.Stop)
        self.chrono = QtCore.QTimer(self)
        self.connect(self.chrono, \
                     QtCore.SIGNAL("timeout()"), \
                     self.timeout)
    def Go(self):
        self.debut = \
            QtCore.QTime.currentTime()
        self.chrono.start(20)
    def Stop(self):
        self.chrono.stop()
        self.timeout()
    def timeout(self):
        maintenant = \
            QtCore.QTime.currentTime()
        ecart = \
            QtCore.QTime().addMsecs( \
```

C++

```
Chrono::Chrono(QWidget* parent):
    QWidget(parent),
    chrono(0),
    affichage(0)
{
    QVBoxLayout* colonne =
        new QVBoxLayout(this);
    this->affichage =
        new QLCDNumber(12, this);
    colonne->addWidget(this->affichage);
    QPushButton* bt_go =
        new QPushButton("Go !", this);
    colonne->addWidget(bt_go);
    connect(bt_go,
            SIGNAL(clicked()),
            this,
            SLOT(Go()));
    QPushButton* bt_stop =
        new QPushButton("Stop", this);
    colonne->addWidget(bt_stop);
    connect(bt_stop,
            SIGNAL(clicked()),
            this,
            SLOT(Stop()));
    this->chrono = new QTimer(this);
    connect(this->chrono,
            SIGNAL(timeout()),
            this,
            SLOT(timeout()));
}
void Chrono::Go()
{
    this->debut = QTime::currentTime();
    this->chrono->start(20);
}
void Chrono::Stop()
{
    this->chrono->stop();
    this->timeout();
}
```



Python

```
self.debut.msecsTo(maintenant))
self.affichage.display( \
    ecart.toString("m:ss:zzz"))
```

C++

```
void Chrono::timeout()
{
    QTime maintenant =
        QTime::currentTime();
    QTime ecart = QTime().addMSecs(
        this->debut.msecsTo(maintenant));
    this->affichage->display(
        ecart.toString("m:ss:zzz"));
}
```

Le signal qui nous intéresse dans la classe `QTimer` est `timeout()`, émis lorsque la durée demandée est écoulée et connecté à une méthode éponyme dans la classe `Chrono`. Le minuteur est déclenché par un appel à la méthode `start()`, laquelle prend en paramètre la durée d'attente en millisecondes. Si vous examinez le code de la méthode `Go()` de la classe `Chrono`, vous voyez qu'on demande une durée de 20 millisecondes, autrement dit on souhaite afficher la durée écoulée 50 fois par seconde. Dans la réalité, il est assez peu probable qu'on obtienne un rythme aussi élevé, encore moins de façon fiable. Les systèmes d'exploitation auxquels nous nous intéressons ne sont pas des systèmes temps réel. Cela signifie que le système va "faire de son mieux" pour nous délivrer un message (un signal) toutes les vingt millisecondes, mais sans aucune garantie.

La conséquence est qu'on ne peut pas savoir avec certitude à quel rythme notre méthode `timeout()` sera invoquée. Autrement dit, on ne peut pas calculer le temps écoulé simplement en ajoutant vingt millisecondes à un compteur.

C'est là qu'intervient la classe `QTime`. Au moment où on démarre le minuteur, on mémorise l'heure courante (méthode `currentTime()` de `QTime`, utilisable sans créer d'instance). Lorsque la méthode `timeout()` est invoquée, on calcule le temps écoulé en millisecondes entre la nouvelle heure courante et celle qui a été mémorisée. Avec les outils dont nous disposons, c'est la seule façon d'avoir une mesure qui ne soit pas trop entachée d'erreurs.

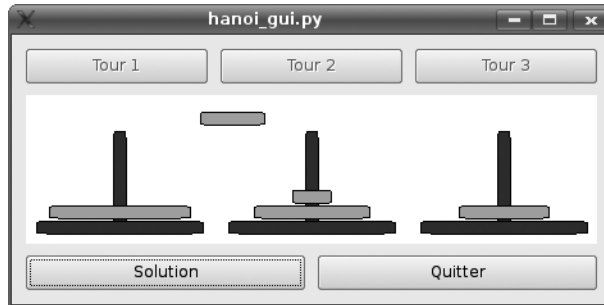
Accessoirement, remarquez l'utilisation de la classe `QLCDNumber`, laquelle propose un affichage de type montre à quartz.

14.2. Animation

Fort de notre expérience, nous pouvons maintenant attaquer le gros morceau : modifier le programme des Tours de Hanoï, pour enfin pouvoir montrer la solution à l'utilisateur. La Figure 14.2 montre le programme alors qu'il est en train d'afficher cette solution. Remarquez que les boutons permettant de choisir une tour ont été désactivés.

Figure 14.2

*Une solution animée
pour Hanoï.*



Pour commencer, nous allons mémoriser les positions de chaque disque dans un tableau. Cela nous permettra par la suite de savoir précisément où chacun se trouve et de facilement modifier cette position. La méthode `Mettre_A_Jour()` de la classe `Dessin` se trouve réduite à simplement calculer ces positions (version Python) :

```
def Mettre_A_Jour(self):
    if ( self.anim_chrono.isActive() ):
        self.anim_chrono.stop()
        self.emit(QtCore.SIGNAL("Fin_Deplacement"))
    for tour in range(3):
        nb_disques = self.jeu.Une_Tour(tour).Nombre_Disques()
        for etage in range(nb_disques):
            disque = self.jeu.Une_Tour(tour).Disque_Etage(etage)
            centre = self.Centre_Etage(tour, etage)
            self.positions[disque-1] = list(centre)
    self.Dessiner()
```

Laissons de côté pour l'instant les trois premières lignes de cette méthode modifiée, bien que le nom de la variable `anim_chrono` vous laisse probablement déjà deviner certains changements à venir. Voyez comment les positions des disques sont mémorisées dans le tableau `positions`.

Le dessin effectif de l'image a été déplacé dans une méthode `Dessiner()`, qui ressemble beaucoup à l'ancienne méthode `Mettre_A_Jour()` : son contenu est pratiquement

le même, la seule différence est qu'elle parcourt le tableau `positions` plutôt que de calculer le centre de chaque disque. Voici un extrait de son contenu (en C++) :

```
void Dessin::Dessiner()
{
    this->image.fill(couleur_fond);
    /* lignes identiques non recopiées... */
    // dessin des disques
    p.setPen(Qt::black);
    p.setBrush(couleur_disques);
    for(int disque = 0; disque < this->jeu.Hauteur(); ++disque)
        this->Dessiner_Disque(disque+1,
                               this->positions[disque],
                               p);
    p.end();
}
```

L'animation elle-même est répartie en deux méthodes. Une première, `Deplacer_Disque()`, lance l'animation ainsi qu'un minuteur, comme pour le chronomètre de la section précédente. La seconde, `timeout()`, sera invoquée périodiquement par ce minuteur pour modifier la position courante du disque en cours de déplacement et mettre à jour l'image. Voici `Deplacer_Disque()`, en Python :

```
def Deplacer_Disque(self, tour_depart, tour_arrivee):
    self.anim_disque = self.jeu.Une_Tour(tour_depart).Sommet() - 1
    self.anim_depart = tuple(self.positions[self.anim_disque])
    etage_arrivee = \
        self.jeu.Une_Tour(tour_arrivee).Nombre_Disques()
    self.anim_arrivee = self.Centre_Etage(tour_arrivee, etage_arrivee)
    self.anim_deplacement = (0, -1)
    self.anim_chrono.start(10)
```

Le membre donnée de la classe `Dessin` nommé `anim_disque` contient le disque en cours de déplacement, c'est-à-dire en fait sa taille. Dans `anim_depart`, on mémorise les coordonnées de ce disque au début de l'animation, tandis qu'`anim_arrivee` reçoit les coordonnées "cibles" auxquelles on veut amener ce disque. La donnée `anim_deplacement`, pour sa part, contient le déplacement à effectuer pour obtenir la nouvelle position du disque au cours de l'animation : déplacement horizontal et déplacement vertical. Initialement, le disque va monter le long de l'aiguille : le déplacement horizontal est donc nul, tandis que le déplacement vertical est négatif (rappelez-vous que la coordonnée verticale va de haut en bas). Ensuite, il sera déplacé latéralement, jusqu'à se positionner au-dessus de l'aiguille de destination (déplacement `(1, 0)` ou `(-1, 0)`, selon qu'il va vers la droite ou vers la gauche). Enfin, il descendra le long de cette aiguille (déplacement `(0, 1)`).

Cette donnée `anim_deplacement` permet donc de savoir "où nous en sommes". Elle est utilisée à cette fin dans la méthode `timeout()`, éventuellement modifiée, à chaque étape de l'animation. La dernière ligne de la méthode `Deplacer_Disque()` montre qu'on demande que le minuteur émette un signal toutes les 10 millisecondes... ce qui a fort peu de chances d'arriver, en fait.

Voici les actions qui sont effectuées à chaque étape de l'animation :

```
void Dessin::timeout()
{
    QPoint distance = this->positions[this->anim_disque] -
                      this->anim_arrivee;
    if ( this->anim_deplacement.y() < 0 )
    {
        // déplacement vertical vers le haut
        if ( this->positions[this->anim_disque].y() <=
            marge + hauteur_disque )
        {
            // on est en haut, on prépare le déplacement latéral
            this->positions[this->anim_disque].ry() = marge + hauteur_disque;
            if ( this->anim_depart.x() < this->anim_arrivee.x() )
                this->anim_deplacement = QPoint(1, 0);
            else
                this->anim_deplacement = QPoint(-1, 0);
        }
        else
            this->positions[this->anim_disque] += this->anim_deplacement;
    }
    else if ( this->anim_deplacement.y() == 0 )
    {
        // déplacement latéral
        if ( std::abs(distance.x()) < 2 )
        {
            // fin du déplacement latéral, on descend le disque
            this->positions[this->anim_disque].rx() = this->anim_arrivee.x();
            this->anim_deplacement = QPoint(0, 1);
        }
        else
            this->positions[this->anim_disque] += this->anim_deplacement;
    }
    else
    {
        // déplacement vertical vers le bas
        if ( (std::abs(distance.x())+std::abs(distance.y())) < 2 )
        {
            // déplacement terminé
            this->anim_chrono->stop();
            this->positions[this->anim_disque] = this->anim_arrivee;
            emit Fin_Deplacement();
        }
    }
}
```

```

        else
            this->positions[this->anim_disque] += this->anim_deplacement;
        }
        this->Dessiner();
        this->update();
    }

```

On commence par calculer le chemin qu'il reste à parcourir en calculant la différence entre la position actuelle du disque et la position cible établie précédemment. Puis on déplace le disque en modifiant sa position courante, ou alors on modifie le déplacement pour passer d'une étape à l'autre.

Ce code peut sembler compliqué mais, si vous le suivez calmement, vous verrez qu'il ne l'est pas tant que cela. La partie la plus intéressante est la ligne contenant le mot `emit`. Ce n'est rien d'autre que le moyen proposé par Qt pour émettre explicitement un signal (lequel doit avoir été préalablement déclaré dans une section `signals`: au sein de la classe, dans le cas du langage C++). Cette instruction est ici utilisée pour littéralement signaler que le déplacement du disque est terminé, donc que l'animation est arrivée à son terme. Le même signal est émis dans la méthode `Mettre_A_Jour()`, dont nous avons vu la version en Python au début de cette section. Nous allons voir comment ce signal est utilisé par le reste de l'application.

Car nous devons maintenant modifier légèrement la classe `Hanoi_Gui`, qui contient l'ensemble de l'interface, ne serait-ce que pour bénéficier de la nouvelle fonctionnalité de `Dessin`: celle qui anime le déplacement d'un disque. Un premier changement se situe au sein de la méthode `Tour_Choisie()`, dans le cas où on a effectivement choisi une tour d'arrivée valide :

```

def Tour_Choisie(self):
    if ( self.tour_depart < 0 ):
        # on a choisi la tour de départ
    # lignes identiques non recopiées...
    else:
        # on a choisi la tour d'arrivée
        tour_arrivee = self.Bouton_Tour_Choisie(self.sender())
        if ( tour_arrivee != self.tour_depart ):
            if ( self.Jeu().Tour_Arrivee_Valide(self.tour_depart, \
                                                tour_arrivee) ):
                self.dessin.Deplacer_Disque(self.tour_depart, tour_arrivee)
                self.Jeu().Deplacer_Disque(self.tour_depart, tour_arrivee)
                self.nb_mouvements += 1
            else:
                # etc.

```

On ne se contente plus de mettre à jour le dessin : désormais, on lui demande de déplacer le disque, c'est-à-dire de faire fonctionner l'animation.

Mais la véritable évolution se situe dans la méthode `Solution()`, chargée de montrer la solution à l'utilisateur. Comme nous l'avons fait dans la version en pur mode texte (voir le Chapitre 10), cette méthode commence par calculer la solution pour la mémoriser dans un tableau (version en C++) :

```
void Hanoi_Gui::Solution()
{
    this->jeu.Initialiser(this->jeu.Hauteur());
    this->dessin->Mettre_A_Jour();
    this->jeu.Solution(this->solution);
    this->nb_mouvements = 0;
    this->bt_tour_1->setEnabled(false);
    this->bt_tour_2->setEnabled(false);
    this->bt_tour_3->setEnabled(false);
    this->en_simulation = true;
    this->Fin_Deplacement();
}
```

Les boutons de choix des tours sont désactivés (grâce à la méthode `setEnabled()` issue de `QWidget`), manière simple d'éviter une interférence de l'utilisateur durant l'animation. La donnée `en_simulation` est de type booléen (valant soit *vrai* soit *faux*) et elle n'est utilisée que pour indiquer que l'on se trouve dans un état particulier, à savoir la simulation du jeu pour afficher la solution. L'utilisation de variables telles que celle-ci pour savoir à tout instant dans quel "état" se trouve un programme est une technique couramment utilisée, qui évite bien souvent de sombrer dans des abîmes de réflexion face à une situation complexe. N'hésitez donc pas à en faire usage.

Le compteur du nombre de mouvements, `nb_mouvements`, va être utilisé pour parcourir le tableau contenant la liste des mouvements de la solution (tableau contenu dans la variable `solution`). Le lancement réel de la simulation est effectué par un appel à `Fin_Deplacement()`, que voici :

```
void Hanoi_Gui::Fin_Deplacement()
{
    if ( not this->en_simulation )
        return;
    if ( this->Jeu().Jeu_Termine() )
    {
        this->en_simulation = false;
        this->bt_tour_1->setEnabled(true);
        this->bt_tour_2->setEnabled(true);
        this->bt_tour_3->setEnabled(true);
    }
    else
    {
        std::pair<int, int> deplacement =
            this->solution[this->nb_mouvements];
```

```
        this->dessin->Deplacer_Disque(deplacement.first, deplacement.second);  
        this->Jeu().Deplacer_Disque(deplacement.first, deplacement.second);  
        this->nb_mouvements += 1;  
    }  
}
```

Cette méthode est connectée au signal éponyme de la classe `Dessin` : elle est donc invoquée à l'issue de chaque animation du déplacement d'un disque. Dans notre exemple, si on n'est pas en train d'afficher la solution du jeu, aucune action particulière n'est à effectuer : c'est là qu'entre en jeu la donnée `en_simulation`, comme vous pouvez le constater sur les deux premières lignes.

Ensuite, si le jeu n'est pas terminé (autrement dit, si tous les disques n'ont pas été déplacés, façon de détecter la fin de la simulation), on recherche le déplacement suivant à partir du tableau `solution` et du compteur `nb_mouvements`. Ce déplacement est simulé, autant pour le dessin que pour le jeu.

L'évolution dans les mouvements est assurée par le signal `Fin_Deplacement()`, émis par la classe `Dessin`, à l'issue d'un déplacement. C'est pourquoi l'affichage de la solution n'utilise pas de boucle en tant que telle, contrairement à la version texte du Chapitre 10. Vous voyez là une traduction du fait que la programmation graphique est une programmation dite *événementielle*. Ici, l'événement est l'expiration d'un délai, d'une attente contrôlée par un minuteur. Cet événement en déclenche d'autres, qui à leur tour finissent par aboutir au relancement de ce minuteur. On a bien ainsi une boucle (une même série d'actions répétées à plusieurs reprises), bien qu'elle ne soit pas explicitement écrite.

Partie 3

Pour aller plus loin

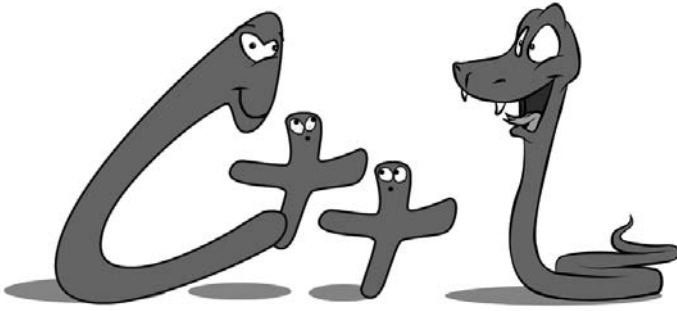
CHAPITRE 15. Rassembler et diffuser : les bibliothèques

CHAPITRE 16. Introduction à OpenGL

CHAPITRE 17. Aperçu de la programmation parallèle

CHAPITRE 18. Aperçu d'autres langages

CHAPITRE 19. Le mot de la fin



15

Rassembler et diffuser : les bibliothèques

Au sommaire de ce chapitre :

- Hiérarchie d'un logiciel
- Découpage de Hanoï

Au cours des chapitres précédents, à mesure que nous avançons dans la réalisation de notre programme, nous avons dûment dupliqué les parties communes – en particulier, la classe `Hanoi` qui contient le moteur interne du jeu, indépendamment de toute interface.

Au fil du temps, vous créerez vous-même des outils (fonctions ou classes) que vous serez amené à réutiliser dans vos nouveaux projets. Il est généralement préférable de s'appuyer sur des outils existants, plutôt que de sans cesse "réinventer la roue". De plus, si vous utilisez un seul et même outil (ou un seul et même ensemble d'outils) dans plusieurs de vos programmes, toute correction ou amélioration en son sein profitera automatiquement à tous ces programmes : la maintenance et même l'évolutivité s'en trouvent grandement facilitées.

Ce double besoin, réutiliser le travail déjà effectué et mutualiser les opérations de maintenance, a donné naissance à la notion de *bibliothèque*. Au sens "programmation" du terme, une bibliothèque est un ensemble de fonctions et de classes rassemblées en un objet unique et cohérent, lesquelles fonctions et classes peuvent être utilisées par différents programmes, voire d'autres bibliothèques. Par exemple, Qt est une bibliothèque de classes en langage C++ : nous avons utilisé ces classes, sans pour autant devoir dupliquer quelque partie de code source de Qt que ce soit.

Techniquement, une bibliothèque est composée (en général) d'une partie binaire, ressemblant à un fichier exécutable et donc incompréhensible, et d'une partie "visible" qui sera utilisée par les clients de la bibliothèque pour exploiter ses services. Cette partie visible est désignée sous le terme d'*API*, pour *Application Programming Interface*, en français *interface de programmation*. En C++, une API se présente sous la forme de fichiers d'en-tête.

Ce chapitre concerne essentiellement le langage C++ et la compilation d'une bibliothèque grâce aux facilités offertes par Qt. Pour Python, pratiquement tout fichier peut être vu comme une minibibliothèque, ce que l'on nomme *module* dans le jargon propre à Python. Pour l'utiliser en tant que telle, il suffit généralement de copier les fichiers à un endroit approprié dépendant du système.

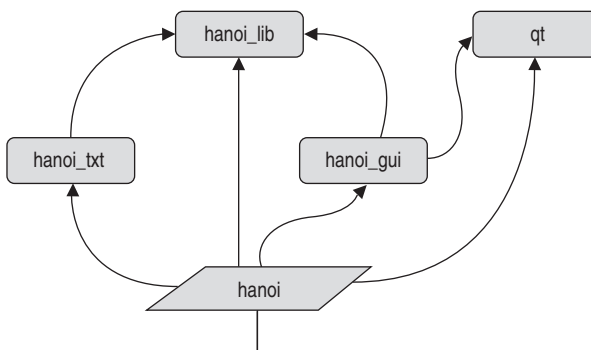
Nous allons récupérer le travail effectué au cours des chapitres précédents, afin de transformer les éléments de notre programme des Tours de Hanoï en autant de bibliothèques. Si cela est certainement inutile sur un programme aussi restreint, l'important est l'idée sous-jacente et la technique utilisée.

15.1. Hiérarchie d'un logiciel

Tout logiciel non trivial est organisé de façon hiérarchique, le programme principal dépendant de bibliothèques elles-mêmes dépendant d'autres bibliothèques. La Figure 15.1 présente la hiérarchie que nous allons mettre en place pour le programme des Tours de Hanoï, les flèches se lisant "... dépend de...". La notion de dépendance est importante lorsqu'on manipule des bibliothèques : dire que A dépend de B signifie que A utilise des fonctions fournies par B, donc, incidemment, que A ne peut fonctionner sans B. Par ailleurs, Qt a été inséré au schéma comme référence, bien qu'elle ne fasse pas partie elle-même de "nos" bibliothèques.

Figure 15.1

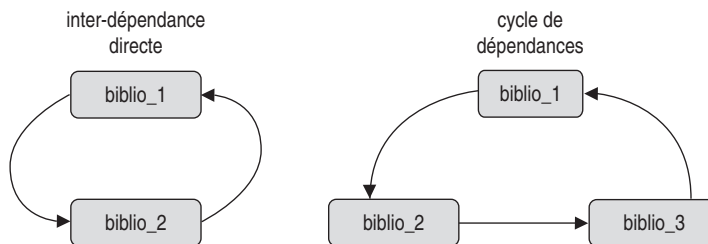
Hiérarchie des bibliothèques pour le programme Hanoï.



Répetons-le, cet exemple est extrêmement simple. On peut donc s'y retrouver sans difficulté. Mais dans un programme de plus grande envergure, une telle hiérarchie peut rapidement devenir très complexe. Si on n'y prend garde, on peut même arriver à la situation où deux bibliothèques sont interdépendantes, c'est-à-dire où une bibliothèque A dépend d'une bibliothèque B laquelle, en même temps, dépend de A. Mais ce qui arrive le plus souvent est l'apparition d'un cycle de dépendances, où plusieurs bibliothèques dépendent indirectement les unes des autres, formant une boucle infernale. La Figure 15.2 illustre ces deux cas.

Figure 15.2

Exemples d'interdépendances.



Inutile de préciser que, dans ce genre de situations, on se retrouve rapidement confronté à des problèmes inextricables, comparables à la grande question de savoir qui de l'œuf ou de la poule est apparu le premier. Aussi, prenez soin d'éviter cela dans vos développements.

15.2. Découpage de Hanoi

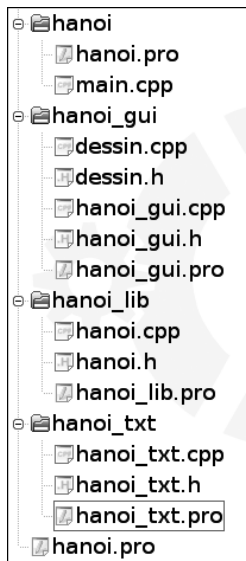
Le découpage que nous allons effectuer est le suivant :

- `hanoi_lib` (le suffixe `lib` signifiant *library*, l'anglais de bibliothèque) contiendra la classe `Hanoi`, le moteur interne du jeu.
- `hanoi_txt` contiendra l'interface en mode texte `Hanoi_Txt`.
- `hanoi_gui` contiendra l'interface en mode graphique `Hanoi_Gui`.
- Enfin, `hanoi` sera tout simplement le programme principal qui fera usage des bibliothèques précédentes.

Vous l'aurez compris, nous allons obtenir un programme pouvant s'exécuter aussi bien en mode texte qu'en mode graphique. La Figure 15.3 montre comment les fichiers vont être répartis en différents répertoires.

Figure 15.3

*Répartition des fichiers
pour un programme
Hanoi modulaire.*



Dans un répertoire quelconque, créez quatre sous-répertoires nommés comme les bibliothèques que nous allons créer. Créez également un fichier, nommé `hanoi.pro`, ayant le contenu suivant :

```
TEMPLATE = subdirs
SUBDIRS = hanoi_lib \
          hanoi_txt \
          hanoi_gui \
          hanoi
```

Il s'agit d'un fichier projet "maître" (la variable `TEMPLATE` vaut `subdirs`), dont le seul rôle est de construire les sous-projets référencés. Chacun des sous-projets se trouve dans un répertoire, dont le nom est donné à la variable `SUBDIRS`. Dans chacun de ces répertoires, l'utilitaire `qmake` va chercher un fichier projet portant le même nom (mais avec l'extension `.pro`, naturellement). L'ordre a son importance : les sous-projets seront compilés dans l'ordre où ils apparaissent, qui découle des dépendances entre les bibliothèques. Cela n'aurait en effet pas de sens de chercher à compiler `hanoi_gui` avant `hanoi_lib`, sachant que la première a besoin de la seconde.

Voici par exemple le contenu du fichier `hanoi_lib.pro`, dans le sous-répertoire `hanoi_lib` :

```
TEMPLATE = lib
CONFIG -= qt
CONFIG += release console
MOC_DIR = .moc
OBJECTS_DIR = .objs
DESTDIR = ../bin
HEADERS += hanoi.h
SOURCES += hanoi.cpp
```

Cette fois-ci, la variable `TEMPLATE` vaut `lib`, la valeur qu'il faut donner lorsqu'on souhaite créer une bibliothèque. La ligne `CONFIG -= qt` a pour effet de "retirer" la bibliothèque Qt des dépendances du projet : en effet, cette bibliothèque de base n'utilise aucun des services de Qt. Enfin, la ligne `DESTDIR` indique dans quel répertoire on va créer le fichier bibliothèque (qui n'est pas un fichier exécutable), en donnant un chemin *relatif* au répertoire dans lequel se trouve le fichier projet. Autrement dit, le résultat de la compilation se retrouvera dans un répertoire `bin` voisin des quatre répertoires que vous venez de créer.

Petite remarque concernant la spécification de chemins. Il est vivement recommandé de toujours utiliser la notation Unix pour séparer les noms des répertoires, donc de toujours utiliser la barre oblique (ou *slash*) `/`, même si vous vous trouvez sur un système

Windows (qui utilise normalement la barre inverse \, ou *backslash*). Le compilateur, tout comme qmake, s'adaptera de lui-même.

Voyons maintenant le fichier projet pour construire l'interface texte :

```
TEMPLATE = lib
CONFIG -= qt
CONFIG += release console
MOC_DIR = .moc
OBJECTS_DIR = .objs
LIBS += -L../bin -lhanoi_lib
DESTDIR = ../bin
INCLUDEPATH += ..
HEADERS += hanoi_txt.h
SOURCES += hanoi_txt.cpp
```

On retrouve les mêmes éléments que précédemment avec, en plus, deux nouvelles variables : INCLUDEPATH et LIBS.

La première permet d'indiquer au compilateur les répertoires dans lesquels chercher les fichiers d'en-tête. En effet, le fichier d'en-tête `hanoi.h`, qui contient l'API (la partie visible et utilisable par d'autres) de la bibliothèque `hanoi_lib`, ne se trouve plus dans le même répertoire. Une convention parmi d'autres, appliquée ici, consiste à indiquer, dans le code source, le fichier d'en-tête précédé du répertoire qui le contient, lequel répertoire se trouve également être le nom de la bibliothèque à laquelle correspond cet en-tête. Ainsi, le début du fichier `hanoi_txt.h` ressemble à ceci :

```
#ifndef HANOI_TXT_H__
#define HANOI_TXT_H__ 1
#include <string>
#include <hanoi_lib/hanoi.h>
class Hanoi_Text
{
public:
    Hanoi_Text(Hanoi& jeu);
/*
    etc.
*/
}; // Hanoi_Text
#endif // HANOI_TXT_H__
```

Pour trouver effectivement l'en-tête `hanoi.h`, il est en fait nécessaire de "remonter" d'un cran dans les répertoires, par rapport à l'endroit où se trouve `hanoi_txt.h`. Et peut-être voudrez-vous un jour déplacer la bibliothèque `hanoi_lib` à un endroit complètement différent. Le fait de renseigner INCLUDEPATH permet de savoir à partir de quels répertoires il convient de chercher les en-têtes. Ainsi, le compilateur

cherchera en réalité `<../hanoi_lib/hanoi.h>`. Si vous déplacez `hanoi_lib`, il suffira d'adapter `INCLUDEPATH`, sans qu'il soit besoin de toucher au code source. Vous pouvez rapprocher cette variable de la célèbre variable d'environnement `PATH`, utilisée pour la recherche des fichiers exécutables.

Remarquez, par ailleurs, que dans le code source nous utilisons également la convention Unix pour la séparation des noms des répertoires.

La variable `LIBS` permet d'indiquer les bibliothèques dont dépend le projet. Elle suit pour cela une syntaxe particulière : d'abord l'indication de répertoires où chercher les bibliothèques, chaque répertoire devant être précédé de `-L` ; ensuite, les noms des bibliothèques, chacun précédé de `-l` (la lettre "l" en minuscule). Remarquez que l'on donne bien le *nom* de chaque bibliothèque, non pas le *nom du fichier* qui la contient. Ce nom de chaque bibliothèque est déduit du nom du fichier projet.

Pour le programme principal, qui dépend des trois autres bibliothèques, la ligne `LIBS` dans le fichier `hanoi.pro` ressemble à ceci :

```
LIBS += -L../bin -lhanoi_lib -lhanoi_txt -lhanoi_gui
```

Et voici le contenu abrégé du programme principal :

```
#include <string>
#include <iostream>
#include <QApplication>
#include <QInputDialog>
#include <hanoi_lib/hanoi.h>
#include <hanoi_txt/hanoi_txt.h>
#include <hanoi_gui/hanoi_gui.h>
int main(int argc, char* argv[])
{
    bool mode_texte = true;
    if ( argc > 1 )
    {
        if ( std::string(argv[1]) == "gui" )
            mode_texte = false;
    }
    if ( mode_texte )
    {
        // interface texte
    }
    else
    {
        // interface graphique
    }
}
```

Ne soyez pas effrayé par la profusion de directives `#include` au début : il est très commun d'en avoir beaucoup, surtout dans un logiciel de grande taille.

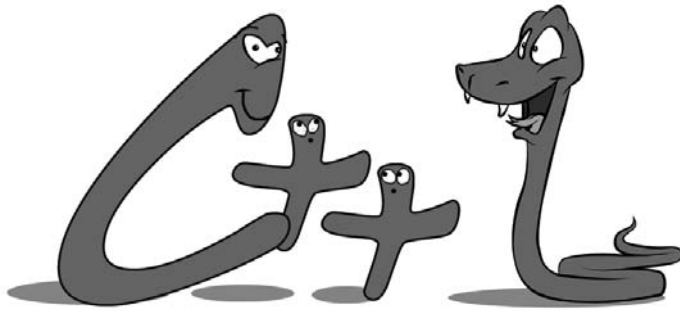
Voyez comme nous exploitons la ligne de commande, suivant ce que nous avons vu à la fin du Chapitre 13. Si on trouve le mot "gui" en premier paramètre, alors on utilise l'interface graphique fondée sur Qt. Sinon, on utilise l'interface en mode texte.

Vous trouverez le code source complet sur le site web de Pearson France. Mais c'est un bon exercice que d'essayer de remplir par vous-même les blancs qui n'ont pas été traités ici.

Une fois tout cela en place, vous pouvez vous placer dans le répertoire qui contient tous les autres (celui contenant le fichier projet référençant les sous-projets, le premier que nous avons vu) et exécuter tout simplement :

```
$ qmake  
$ make
```

Remplacez `make` par `mingw32-make` si vous êtes sous Windows. À l'issue d'une compilation qui devrait être assez rapide, vous obtiendrez dans un nouveau sous-répertoire bin trois fichiers contenant les bibliothèques, ainsi qu'un fichier exécutable `hanoi`. À vous de le tester maintenant !



Introduction à OpenGL

Au sommaire de ce chapitre :

- Installation
- Triangle et cube de couleur
- La GLU et autres bibliothèques

Tout ce que nous avons fait jusqu'ici a consisté à "dessiner" dans un plan, comme sur une feuille de papier. Ce qui semble tout à fait naturel, étant donné que nos écrans n'offrent que deux dimensions (sachez que des recherches sont en cours pour proposer des dispositifs d'affichage qui soient réellement en trois dimensions, c'est-à-dire en volume).

Pourtant, que cela soit dans le monde des jeux vidéo ou dans celui de l'image de synthèse (utilisée pour les effets spéciaux au cinéma, par exemple), les exemples abondent de programmes affichant des objets en trois dimensions sur un écran en deux dimensions. Nous allons, au fil de ce chapitre, approcher l'une des techniques disponibles pour obtenir un tel résultat.

De nos jours, il existe essentiellement deux grandes bibliothèques concurrentes facilitant grandement l'affichage d'objets en volume (ou objets 3D) sur un écran :

- Direct3D, bibliothèque définie et développée par Microsoft pour ses systèmes Windows et ses consoles de jeu, faisant partie de l'ensemble plus vaste DirectX (voir <http://msdn2.microsoft.com/en-us/directx/>).
- OpenGL, bibliothèque définie initialement par SGI pour ses stations de travail et désormais maintenue par un consortium de plusieurs entreprises (dont Microsoft s'est retiré il y a peu), destinée à être utilisable sur (presque) toutes les plates-formes (voir <http://www.opengl.org>).

Nous pourrions discuter pendant quelques centaines de pages des mérites et inconvénients de chacune de ces deux bibliothèques, dans la continuité d'une longue série de débats parfois passionnés entre les "pro-Direct3D" et les "pro-OpenGL", débats qui pouvaient parfois prendre l'allure de guerre de religion. Nous nous épargnerons cette peine, en constatant simplement que notre exigence initiale d'écrire un programme unique pouvant fonctionner sur plusieurs plates-formes nous impose d'elle-même l'utilisation d'OpenGL.

Il s'agit donc d'une bibliothèque, définie en langage C, grâce à laquelle un programme peut afficher des objets 3D sur un écran, qui est, par nature, une surface plane. La bibliothèque prend en charge la plupart des calculs (compliqués) permettant de projeter un point défini dans un espace 3D à une certaine position sur l'écran, ainsi que bien d'autres aspects, comme de ne pas afficher les objets masqués par d'autres ou représenter un effet d'éclairage par des sources lumineuses.

Depuis quelques années, une part croissante de ces calculs est effectuée directement par la carte graphique plutôt que par le processeur central de votre ordinateur. On parle alors de carte graphique accélérée 3D. La spécialisation du processeur graphique (ou

GPU) fait qu'il devient possible d'obtenir des performances inimaginables il n'y a pas si longtemps.

Mais voyons comment nous pouvons utiliser cette bibliothèque OpenGL.

16.1. Installation

La bibliothèque OpenGL et les éléments nécessaires à son utilisation en C++ sont normalement toujours disponibles, pratiquement quel que soit le système que vous utilisez. Il n'en va pas de même avec Python : il est nécessaire d'installer un petit module, nommé PyOpenGL.

Si vous utilisez Windows ou Mac OS, téléchargez le fichier http://peak.telecommunity.com/dist/ez_setup.py. Exécutez ensuite ce fichier :

```
$ python ez_setup.py
```

Il suffit ensuite d'exécuter la commande suivante pour installer PyOpenGL :

```
$ easy_install PyOpenGL
```

Cela va prendre en charge le téléchargement et l'installation du module.

Si vous utilisez un système Linux, installez simplement le paquetage PyOpenGL, il devrait être disponible quelle que soit votre distribution.

16.2. Triangle et cube de couleur

Précisons dès maintenant que OpenGL est une bibliothèque qui peut sembler assez pauvre à première vue, en ce sens qu'elle ne fournit pas d'outils de haut niveau pour des opérations apparemment aussi simples que, par exemple, afficher un texte. Nous n'allons donc pas pouvoir saluer le monde pour nos premières expérimentations.

De plus, l'affichage 3D lui-même doit s'effectuer dans un *contexte OpenGL*, lequel doit être fourni par le système graphique du système d'exploitation. Chaque système proposant naturellement ses propres fonctions pour obtenir ce contexte, ce qui complique encore la tâche.

Heureusement, la bibliothèque Qt propose un widget spécialement destiné à l'affichage utilisant OpenGL, par la classe `QGLWidget`. Nous pouvons donc obtenir aisément un contexte OpenGL, une zone rectangulaire à l'écran dans laquelle nous

pourrons "dessiner" en trois dimensions, simplement en dérivant cette classe et en surchargeant quelques méthodes.

16.2.1. Le triangle

Comme premier exemple, voici comment afficher un simple triangle en OpenGL en nous basant sur QGLWidget :

Python	C++
<pre># -*- coding: utf8 -*- import sys from PyQt4 import QtCore, QtGui, QtOpenGL from OpenGL.GL import * class Triangle(QtOpenGL.QGLWidget): def __init__(self, parent): QtOpenGL.QGLWidget.__init__(self, \ parent) def initializeGL(self): glEnable(GL_DEPTH_TEST) def resizeGL(self, w, h): glViewport(0, 0, w, h) glMatrixMode(GL_PROJECTION) glLoadIdentity() def paintGL(self): glClearColor(0.0, 0.0, 0.0, 1.0) glClear(GL_COLOR_BUFFER_BIT \ GL_DEPTH_BUFFER_BIT) glMatrixMode(GL_MODELVIEW) glLoadIdentity() glShadeModel(GL_SMOOTH) glBegin(GL_TRIANGLES) glColor3d(1.0, 0.0, 0.0) glVertex3d(0.0, 0.0, 0.0) glColor3d(0.0, 1.0, 0.0) glVertex3d(1.0, 0.0, 0.0) glColor3d(0.0, 0.0, 1.0) glVertex3d(0.0, 1.0, 0.0) glEnd()</pre>	<pre>#include "triangle.h" Triangle::Triangle(QWidget* parent): QGLWidget(parent) { } void Triangle::initializeGL() { glEnable(GL_DEPTH_TEST); } void Triangle::resizeGL(int w, int h) { glViewport(0, 0, w, h); glMatrixMode(GL_PROJECTION); glLoadIdentity(); } void Triangle::paintGL() { glClearColor(0.0, 0.0, 0.0, 1.0); glClear(GL_COLOR_BUFFER_BIT); glClear(GL_DEPTH_BUFFER_BIT); glMatrixMode(GL_MODELVIEW); glLoadIdentity(); glShadeModel(GL_SMOOTH); glBegin(GL_TRIANGLES); glColor3d(1.0, 0.0, 0.0); glVertex3d(0.0, 0.0, 0.0); glColor3d(0.0, 1.0, 0.0); glVertex3d(1.0, 0.0, 0.0); glColor3d(0.0, 0.0, 1.0); glVertex3d(0.0, 1.0, 0.0); glEnd(); }</pre>

Nous n'avons montré ici que le contenu d'une classe `Triangle`, laquelle dérive de `QGLWidget`. Remarquez les modules que nous importons par rapport aux programmes précédents en Python : `OpenGL.GL` pour les fonctions OpenGL elles-mêmes, `QtOpenGL` (à partir de `PyQt4`) pour la classe `QGLWidget`. Côté C++, une simple ligne `#include <QGLWidget>` suffit dans le fichier d'en-tête correspondant à cette classe `Triangle` (essayez de reconstruire vous-même cet en-tête, c'est un bon exercice). Par contre, toujours pour le C++, il est nécessaire de préciser, dans le fichier projet, que nous allons utiliser les fonctionnalités OpenGL, par les variables `CONFIG` et `QT` :

```
CONFIG += release opengl
QT += gui opengl
```

Le programme principal se réduit à sa plus simple expression, ne faisant rien d'autre que créer une instance de `Triangle`, l'afficher et lancer la boucle d'événements, comme pour tout programme utilisant Qt.

Mais examinons un peu le contenu de cette classe `Triangle`, justement. Trois méthodes doivent être surchargées à partir de `QGLWidget` :

- `initializeGL()`, appelée une seule fois, est destinée à paramétrer le contexte OpenGL (dans la pratique, pour une application complexe, on ne l'utilise que rarement).
- `resizeGL()` est invoquée chaque fois que le widget change de taille ; en effet, la façon d'afficher un "monde" en trois dimensions dépend fortement des dimensions du rectangle plan dans lequel il doit s'inscrire.
- `paintGL()`, enfin, est la méthode invoquée lorsqu'il est nécessaire de dessiner effectivement les objets en trois dimensions : c'est en son sein que s'effectue le véritable travail d'affichage.

Toutes les fonctions de la bibliothèque OpenGL commencent par le suffixe `gl`. La première que nous rencontrons, `glEnable()` dans `initializeGL()`, est une fonction assez générique permettant d'activer certaines fonctionnalités. Son miroir est `glDisable()`, pour au contraire en désactiver certaines. Par exemple, dans notre cas, on demande à OpenGL d'activer le *test de profondeur*, qui a en gros pour effet de ne pas afficher les objets qui se trouveraient "derrière" d'autres objets, selon la direction dans laquelle on regarde.

Sans doute est-il temps de voir comment OpenGL passe de la modélisation d'un monde en trois dimensions à une représentation en deux dimensions sur l'écran.

Ce que nous allons voir maintenant est un peu simplifié par rapport à la réalité, mais cela devrait vous donner une idée.

Pour cela, considérez un point dans l'espace. Un point au sens mathématique du terme, c'est-à-dire n'ayant pas d'épaisseur. Une tête d'épingle vue d'un peu loin, par exemple. Ce point est localisé à l'aide de coordonnées, tout comme les pixels sur l'écran, sauf que ces coordonnées sont (le plus souvent) des nombres à virgule flottante et qu'elles sont au nombre de trois – d'où le terme trois dimensions, ou 3D en abrégé. Ces coordonnées sont exprimées par rapport à un repère, lequel est défini par la *matrice modèle/vue* (*model/view matrix* en anglais). Le terme matrice renvoie à l'objet mathématique utilisé, une matrice de transformation : il s'agit grossièrement d'un tableau de quatre lignes et de quatre colonnes, contenant donc seize valeurs numériques. D'une certaine façon, cette matrice détermine la position et l'orientation de votre œil dans l'espace. Chaque point nécessaire à la description des objets à représenter est ainsi spécifié par ses coordonnées dans cet espace 3D.

Imaginez maintenant que, juste devant votre œil, vous placiez une immense plaque de verre. Sur cette plaque, à l'aide d'un feutre, vous tracez un point à l'endroit où vous voyez la tête d'épingle. Et vous faites de même avec tous les objets que vous voyez et que vous voulez représenter. Vous obtenez finalement un dessin plat, en deux dimensions, d'un ensemble d'objets en trois dimensions. Ce que vous venez de réaliser est une *projection*, consistant à "écraser" le monde en trois dimensions sur un plan en deux dimensions. La taille de la plaque de verre détermine directement les limites du monde que vous représentez. La façon dont cette projection doit être effectuée est déterminée par la matrice de projection (*projection matrix* en anglais). Pour notre tête d'épingle, le résultat est le passage de coordonnées 3D en coordonnées 2D.

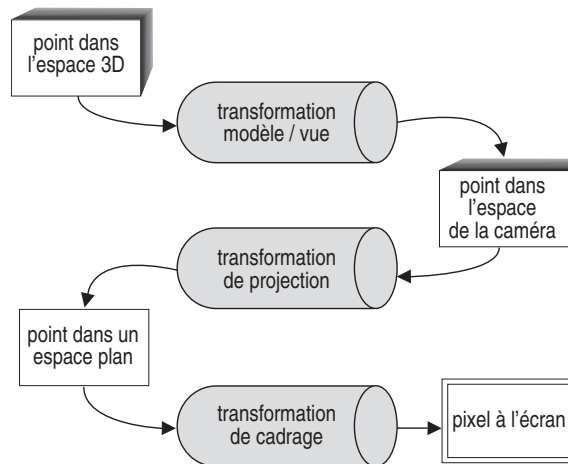
Enfin, il reste à obtenir une image, de la même façon qu'on obtient une photo : il faut définir les dimensions de cette photo, dans quelle zone de la feuille de papier elle va se situer... C'est ce qu'on appelle le cadrage. Cette opération est régie par la fonction `glViewport()`, c'est-à-dire la façon dont l'image (plane) finale sera insérée dans la fenêtre graphique. Ses paramètres sont les coordonnées du coin inférieur gauche et les dimensions d'un rectangle, par rapport au widget contenant le contexte OpenGL. Le plus souvent, on donne simplement (0, 0) comme origine et les largeur et hauteur du widget, comme ici. L'image finale, résultant de la transformation des objets 3D en un dessin plan en deux dimensions, sera insérée dans ce rectangle. Petite subtilité, source de nombreuses confusions : contrairement à l'usage dans la programmation

d'une interface graphique, OpenGL considère que l'origine des coordonnées se situe dans le coin *inférieur* gauche, non pas dans le coin supérieur gauche.

Le cadrage est la dernière opération de transformation géométrique, conclusion d'une chaîne illustrée à la Figure 16.1.

Figure 16.1

La chaîne des transformations OpenGL.



Généralement, cadrage et projection sont définis dans la méthode `resizeGL()`, car ces deux transformations dépendent directement des dimensions du widget. `glMatrixMode()` indique à OpenGL sur quelle matrice on s'apprête à agir : la constante `GL_PROJECTION` désigne la matrice de projection. `glLoadIdentity()` place dans celle-ci la matrice identité, c'est-à-dire que le cadre de notre widget représente exactement le volume d'espace que nous allons projeter.

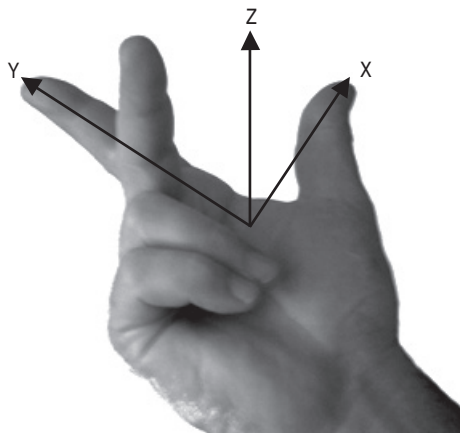
Enfin, le dessin effectif est effectué dans la méthode `paintGL()`. Les deux premières fonctions utilisées, `glClearColor()` et `glClear()`, ont pour effet de "nettoyer" le contexte d'affichage OpenGL en le remplissant de noir. Puis on positionne notre œil, en donnant la matrice modèle/vue (à l'aide de `glMatrixMode()` et `glLoadIdentity()`).

La combinaison de transformations telle que nous l'avons définie ici place l'origine des coordonnées 3D au centre du widget, l'axe X (première coordonnée) allant de la gauche vers la droite, entre -1.0 et 1.0 , l'axe Y (deuxième coordonnée) allant de bas en haut, entre -1.0 et 1.0 , l'axe Z (troisième coordonnée) "jaillissant" de l'écran comme pour vous rentrer dans l'œil. Pour vous représenter ce système de coordonnées, utilisez votre main droite, poing fermé. Écartez le pouce vers l'extérieur de la main : il représente l'axe X. Tendez complètement l'index, qui devient ainsi perpendiculaire au pouce : il représente

l'axe Y. Enfin, déployez les deux dernières phalanges du majeur, qui devient ainsi perpendiculaire à la fois au pouce et à l'index : il représente l'axe Z.

Figure 16.2

Repère direct ou repère main droite.



Votre main ainsi ouverte représente un repère direct, aussi appelé repère main droite. Si vous placez le pouce de façon qu'il pointe vers la droite de votre écran et l'index vers le haut, vous voyez que le majeur semble "sortir" de l'écran.

Toutes les opérations effectuées jusqu'ici n'ont fait que préparer le contexte OpenGL pour le dessin. Celui-ci est effectué sur la base de primitives, c'est-à-dire d'objets élémentaires qu'OpenGL "sait" effectivement représenter. Ils sont très peu nombreux, les plus utilisés étant le triangle et la ligne brisée. Tout autre objet plus complexe, comme un cylindre, une sphère ou un simple cercle, est en réalité obtenu en dessinant un nombre plus ou moins élevé de ces primitives de base. Comme illustration, voyez la Figure 16.3, qui montre une même scène du monde virtuel *SecondLife* : l'image de gauche est celle qui est normalement exposée à l'utilisateur, l'image de droite montre la décomposition en triangles de chacun des objets affichés.

Le dessin d'une primitive (ou plusieurs) est réalisé en donnant les coordonnées de chacun de ses sommets, dans une suite d'instructions encadrées par la paire `glBegin()` / `glEnd()`. La première annonce le commencement d'un dessin, son paramètre identifiant quelle primitive on s'apprête à dessiner. La seconde signale la fin du dessin.

Entre les deux, on trouve une suite d'appels à la fonction `glVertex3d()`, chacun d'eux définissant les coordonnées d'un sommet de la primitive. Dans notre exemple, comme nous dessinons un simple triangle, trois appels sont nécessaires.

Figure 16.3

Un monde virtuel en OpenGL : ce que vous voyez et la face cachée.



Vous pouvez constater qu'avant chaque appel à `glVertex3d()` nous invoquons `glColor3d()` : cette fonction permet de spécifier la couleur de chacun des sommets. Le modèle de couleur est ici aussi le modèle RGB (voir la section 5 du Chapitre 12), sauf que les proportions de rouge, vert et bleu sont données par un nombre à virgule flottante compris entre 0.0 (noir) et 1.0 (pleine intensité). Après un appel à `glColor3d()`, tous les sommets suivants seront de la couleur indiquée. Donc, nous indiquons un premier

sommet en rouge, un deuxième en vert et un troisième en bleu. Le résultat est un triangle montrant un dégradé entre ces trois couleurs, comme le montre la Figure 16.4.

Figure 16.4

*Dégradé de couleur
dans un triangle.*



16.2.2. Le cube

Notre triangle est sans doute très esthétique, mais il reste désespérément plat : nous aurions pu obtenir le même résultat en utilisant les possibilités offertes par la classe `QPainter` de Qt. Voyons maintenant comment nous pouvons passer réellement en trois dimensions.

Nous ne pouvons plus nous contenter des matrices identité pour la projection et la vue. Commencez donc par modifier ainsi la méthode `resizeGL()` (en Python) :

```
def resizeGL(self, w, h):
    glViewport(0, 0, w, h)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(-1.0, 2.0, -1.0, 2.0, -2.0, 3.0)
```

La fonction `glOrtho()` définit une projection orthogonale, c'est-à-dire dans laquelle deux droites parallèles dans l'espace restent parallèles une fois projetées. Ce type de projection est adapté pour des représentations techniques mais ne correspond pas à la réalité. La vision humaine est mieux représentée par une projection utilisant des points de fuite. Par exemple, si vous conduisez une voiture sur une route longue et droite, les bords de celle-ci semblent s'incurver pour se rejoindre à l'horizon. Une telle perspective peut être obtenue avec `glFrustum()`, mais elle est plus délicate à mettre en œuvre. Les quatre paramètres passés à `glOrtho()` déterminent les limites du volume visualisé.

Voici maintenant `paintGL()` (en C++), dans laquelle on dessine trois faces d'un cube :

```
void Cube::paintGL()
{
```

```

glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
glClear(GL_DEPTH_BUFFER_BIT);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotated(-45.0, 1.0, 0.0, 0.0);
glRotated(-30.0, 0.0, 0.0, 1.0);
glShadeModel(GL_SMOOTH);
glBegin(GL_QUADS);
    // première face (avant)
    glColor3d(0.0, 0.0, 0.0);
    glVertex3d(0.0, 0.0, 0.0);
    glColor3d(1.0, 0.0, 0.0);
    glVertex3d(1.0, 0.0, 0.0);
    glColor3d(1.0, 0.0, 1.0);
    glVertex3d(1.0, 0.0, 1.0);
    glColor3d(0.0, 0.0, 1.0);
    glVertex3d(0.0, 0.0, 1.0);
    // deuxième face (côté)
    glColor3d(1.0, 0.0, 0.0);
    glVertex3d(1.0, 0.0, 0.0);
    glColor3d(1.0, 1.0, 0.0);
    glVertex3d(1.0, 1.0, 0.0);
    glColor3d(1.0, 1.0, 1.0);
    glVertex3d(1.0, 1.0, 1.0);
    glColor3d(1.0, 0.0, 1.0);
    glVertex3d(1.0, 0.0, 1.0);
    // troisième face (dessus)
    glColor3d(0.0, 0.0, 1.0);
    glVertex3d(0.0, 0.0, 1.0);
    glColor3d(1.0, 0.0, 1.0);
    glVertex3d(1.0, 0.0, 1.0);
    glColor3d(1.0, 1.0, 1.0);
    glVertex3d(1.0, 1.0, 1.0);
    glColor3d(0.0, 1.0, 1.0);
    glVertex3d(0.0, 1.0, 1.0);
glEnd();
glBegin(GL_LINES);
    glColor3d(0.0, 0.0, 0.0);
    glVertex3d(1.0, 0.0, 1.0);
    glVertex3d(1.0, 0.0, 0.0);
    glVertex3d(1.0, 0.0, 1.0);
    glVertex3d(1.0, 1.0, 1.0);
    glVertex3d(1.0, 0.0, 1.0);
    glVertex3d(0.0, 0.0, 1.0);
glEnd();
}

```

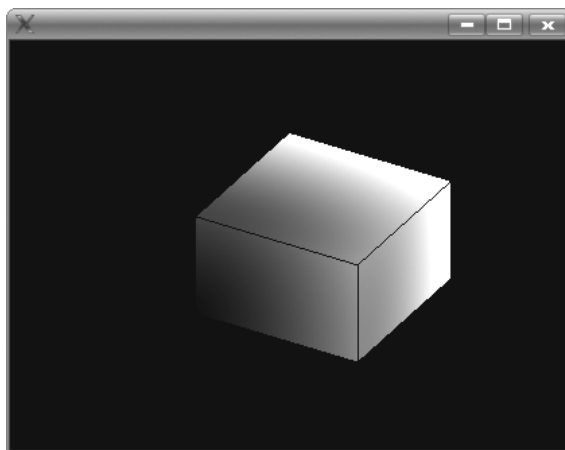
Les appels à la fonction `glRotated()` ont pour effet de faire tourner notre œil autour d'un axe donné – c'est-à-dire, en pratique, de modifier la matrice modèle/vue.

Notez que ces transformations se composent les unes avec les autres : le premier appel à `glRotated()` fait tourner autour de l'axe X, le deuxième autour d'un axe Z **résultant de la rotation précédente**. Par des combinaisons plus ou moins complexes, vous pouvez ainsi obtenir des effets sophistiqués. Vous pouvez également obtenir très facilement une image parfaitement noire, ne montrant rien du tout : les combinaisons de transformations peuvent être assez délicates à manipuler, au point de vous retrouver perdu quelque part dans l'espace.

Les instructions qui suivent dessinent tout simplement trois rectangles (primitive `GL_QUADS`), puis trois segments de droite (primitive `GL_LINES`), ces derniers n'ayant d'autre but que de souligner les bords du cube. Les rectangles nécessitent chacun quatre points, tandis que les segments n'en nécessitent que deux. Voyez comment plusieurs primitives d'un même type sont dessinées au sein d'un unique bloc `glBegin()/glEnd()`. La Figure 16.5 montre le résultat de ce programme.

Figure 16.5

Le cube en couleur.



16.3. La GLU et autres bibliothèques

Nous n'avons évoqué ici que quelques fonctions de la bibliothèque OpenGL. Des centaines d'autres sont disponibles. Pourtant, la plupart restent d'assez "bas niveau" : si OpenGL fournit tous les outils pour dessiner une simple sphère, aucune fonction ne permet de la dessiner facilement.

Aussi, de nombreuses bibliothèques ont-elles été créées "par-dessus" OpenGL, afin de fournir des outils de plus haut niveau. La première d'entre elles est la *GLU* (pour

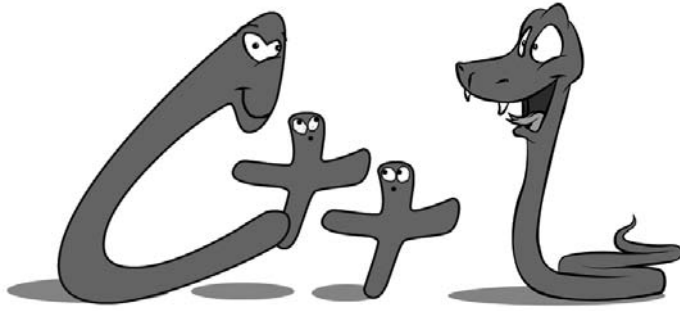
OpenGL Utilities), qui offre plusieurs possibilités pour le dessin d'objets géométriques plus complexes, le réglage des matrices de transformation selon diverses situations, etc. L'importance de cette bibliothèque est telle qu'elle est normalement toujours disponible aux côtés de toutes les implémentations d'OpenGL.

Concernant l'affichage de texte, encore une fois, OpenGL ne fournit aucune facilité. Pourtant, une spécification a été établie, nommée GLC, pour définir une API permettant l'affichage de caractères dans un contexte OpenGL. Une implémentation libre et multi-plate-forme de cette spécification est offerte par la bibliothèque QuesoGLC (<http://quesoglc.sourceforge.net/>).

À mesure que vous progresserez dans l'utilisation d'OpenGL, vous ressentirez inévitablement le besoin de structurer le monde que vous voudrez représenter, d'associer des éléments entre eux, de permettre à l'utilisateur de les sélectionner à la souris, voire de pouvoir changer librement l'angle de vue... En résumé, vous aurez le besoin d'un véritable *moteur 3D* de haut niveau. Citons quelques bibliothèques.

- Coin3D (<http://www.coin3d.org/>) utilise une représentation hiérarchique d'une scène 3D pour son affichage, permettant ainsi la représentation d'ensembles complexes et structurés. Elle offre, en plus, diverses options pour l'interaction avec les objets de la scène, ce qui simplifie considérablement le travail de développement. Ajoutons qu'elle s'intègre fort bien à la bibliothèque Qt. Coin3D est adaptée pour les applications à vocation technique.
- OpenSceneGraph (<http://www.openscenegraph.org/projects/osg>) offre des fonctionnalités analogues à celles de Coin3D, mais avec une encapsulation plus poussée des possibilités avancées d'OpenGL. Elle est également plus généraliste, permettant la création aussi bien d'applications scientifiques que de jeux.
- Soya3D (<http://home.gna.org/oomadness/en/soya3d/index.html>) est une bibliothèque spécialement dédiée au langage Python pour la création d'applications 3D, aussi bien jeux que programmes de visualisation ou simulation scientifique. Et ce qui ne gâte rien, les développeurs de Soya3D sont principalement français.

Gardez à l'esprit que ces bibliothèques ne sont pas des jouets destinés à un enfant... Pour puissantes qu'elles soient, elles peuvent être plus ou moins difficiles à appréhender. Si vous souhaitez vous lancer dans le développement d'une application 3D complexe, que cela soit un jeu ou un programme scientifique, n'hésitez pas à passer du temps pour expérimenter les possibilités offertes et déterminer ainsi laquelle vous conviendra le mieux.



17

Aperçu de la programmation parallèle

Au sommaire de ce chapitre :

- Solution simple pour Hanoï
- Solution parallélisée
- Précautions et contraintes

Si vous vous intéressez un tant soit peu au marché des ordinateurs, il ne vous aura pas échappé que la tendance actuelle en matière de processeurs est à la multiplication des cœurs, c'est-à-dire en fait à fournir des ordinateurs contenant plusieurs processeurs au lieu d'un seul.

Les ordinateurs équipés de plusieurs processeurs ne sont pas une nouveauté. Cela fait des dizaines d'années que ces configurations sont couramment utilisées dans les domaines scientifiques et techniques, particulièrement gourmands en puissance de calcul. Un exemple célèbre est la machine Deep Blue, premier ordinateur à vaincre le grand maître des échecs Garry Kasparov en 1997 : elle contenait 30 processeurs généraux, assistés de 480 processeurs spécialisés pour les échecs. Toutefois, les systèmes équipés de plusieurs processeurs demeuraient très onéreux et réservés à des domaines assez spécialisés, tels que les prévisions météo.

Dans le monde des micro-ordinateurs destinés au grand public, durant de nombreuses années, la course à la puissance s'est traduite par une course à la fréquence : chaque année, le nouveau processeur était plus rapide que celui de l'année précédente, cette vitesse étant grossièrement exprimée par la fréquence de l'horloge interne, qui représentait plus ou moins le nombre d'instructions élémentaires (très élémentaires) que le processeur était capable d'effectuer par seconde. Les progrès ont été considérables, depuis le premier 8086 du début des années 1980, affichant une fréquence de 4,77 MHz, aux derniers processeurs atteignant des fréquences de l'ordre de 4 GHz : en une vingtaine d'années, la fréquence a été multipliée par mille.

Mais malgré toute l'habileté des ingénieurs, les lois physiques sont têtues. Il est devenu impossible de miniaturiser davantage, d'augmenter la fréquence davantage, du moins pour un coût raisonnable. D'une certaine manière, on peut dire que la puissance brute a atteint ses limites, au moins jusqu'à la prochaine innovation technologique majeure. L'augmentation de la fréquence n'est ainsi plus la voie à suivre pour augmenter la vitesse de calcul de nos processeurs "domestiques".

Alors, les fabricants de processeurs se sont tournés vers une autre voie : multiplier les cœurs au sein d'un même processeur, c'est-à-dire, en fait, fournir plusieurs processeurs en un seul. Dès lors, du point de vue du programmeur, une nouvelle possibilité pour améliorer les performances de son programme consiste à exploiter les multiples processeurs disponibles en effectuant des traitements en parallèle, c'est-à-dire, littéralement, en faisant plusieurs choses à la fois.

17.1. Solution simple pour Hanoï

Comme illustration, nous allons reprendre la simulation de la solution de notre jeu des Tours de Hanoï. La solution ne sera pas réellement calculée, mais simplement nous allons compter le nombre de mouvements effectués. Voici le programme, tout ce qu'il y a de plus classique :

```
#include <cstdlib>
#include <iostream>
#include <QTime>
void Hanoi_Simple(int nb_disques,
                  int depart,
                  int pivot,
                  int arrivee,
                  long long& nb_mouvements)
{
    if ( nb_disques == 1 )
    {
        ++nb_mouvements;
    }
    else
    {
        Hanoi_Simple(nb_disques - 1,
                    depart, arrivee, pivot,
                    nb_mouvements);
        Hanoi_Simple(1,
                    depart, pivot, arrivee,
                    nb_mouvements);
        Hanoi_Simple(nb_disques - 1,
                    pivot, depart, arrivee,
                    nb_mouvements);
    }
}
int main(int argc, char* argv[])
{
    int nb_disques = atoi(argv[1]);
    long long nb_mouvements = 0;
    QTime chrono;
    chrono.start();
    Hanoi_Simple(nb_disques, 1, 2, 3, nb_mouvements);
    std::cout << nb_mouvements << " "
               << chrono.elapsed() / 1000.0 << std::endl;
    return 0;
}
```

On retrouve la classe `QTime`, fournie par Qt, que nous utilisons ici pour mesurer le temps nécessaire à la simulation. Le type `long long` utilisé dans ce programme est un type entier signé, mais d'une plus grande capacité que le type habituel `int`. Son utilisation est nécessaire, car nous allons dénombrer un très grand nombre de mouvements (plusieurs milliards), valeurs qu'un simple `int` ne pourrait pas représenter.

Lorsqu'on exécute ce programme en lui donnant en paramètre 34 disques, on obtient 17 179 869 183 mouvements en environ cent secondes. Naturellement, cette durée peut varier d'une machine à l'autre, selon sa puissance.

17.2. Solution parallélisée

L'immense majorité des langages de programmation ne fournissent par eux-mêmes aucun support pour la mise en œuvre d'une programmation parallèle. Une exception notable est le langage Ada, qui prévoyait la définition de tâches pouvant s'exécuter simultanément dès sa conception, à la fin des années 1970. Mais les langages Python et C++ doivent explicitement s'appuyer sur des bibliothèques externes.

Naturellement, il existe différentes façons de "faire du parallélisme" ; de plus, pratiquement chaque système propose son propre ensemble de fonctions pour cela. La solution que nous allons exposer ici s'appuie sur la bibliothèque Qt, qui a l'immense mérite de fournir une interface de programmation unique pour tous les systèmes.

Le principe consiste à créer, au sein d'un même programme, plusieurs flux d'instructions. Les programmes que nous avons réalisés jusqu'ici ne contenaient qu'un seul flux, les boucles et les alternatives n'étant en fait que des "sauts" dans ce flux. Nous allons créer deux flux qui vont s'exécuter en parallèle. On utilise plus couramment le terme anglais de *thread*, dont flux n'est qu'une traduction approximative. Ainsi, parle-t-on de programmation *multithread*, ou encore d'un programme *multithreadé*.

Dans notre exemple consistant à compter le nombre de mouvements nécessaires au déplacement de n disques, chaque thread sera chargé de simuler le déplacement de $n-1$ disques. La création d'un thread se fait simplement en dérivant la classe `QThread`

et en surchargeant la méthode `run()`, laquelle doit contenir les instructions à exécuter. Voici le programme (légèrement abrégé) :

```
#include <cstdlib>
#include <iostream>
#include <QTime>
#include <QCoreApplication>
#include <QThread>
void Hanoi_Simple(int nb_disques,
                  int depart,
                  int pivot,
                  int arrivee,
                  long long& nb_mouvements)
{
    /* inchangée par rapport au précédent */
}
class Hanoi_Thread: public QThread
{
public:
    Hanoi_Thread(int nb_disques,
                 int depart,
                 int pivot,
                 int arrivee):
        nb_disques(nb_disques),
        depart(depart),
        pivot(pivot),
        arrivee(arrivee),
        nb_mouvements(0)
    {
    }
    virtual void run()
    {
        Hanoi_Simple(this->nb_disques,
                     this->depart,
                     this->pivot,
                     this->arrivee,
                     this->nb_mouvements);
    }
    int nb_disques;
    int depart;
    int pivot;
    int arrivee;
    long long nb_mouvements;
}; // class Hanoi_Thread
void Hanoi_Parallele(int nb_disques,
                    int depart,
                    int pivot,
                    int arrivee,
                    long long& nb_mouvements)
```

```
{
    Hanoi_Thread thread_1(nb_disques-1,
                          depart,
                          arrivee,
                          pivot);

    thread_1.start();
    Hanoi_Thread thread_2(nb_disques-1,
                          pivot,
                          depart,
                          arrivee);

    thread_2.start();
    thread_1.wait();
    thread_2.wait();
    nb_mouvements = thread_1.nb_mouvements +
                    1 +
                    thread_2.nb_mouvements;
}
int main(int argc, char* argv[])
{
    int nb_disques = atoi(argv[1]);
    QCoreApplication app(argc, argv);
    long long nb_mouvements = 0;
    QTime chrono;
    chrono.start();
    Hanoi_Parallele(nb_disques, 1, 2, 3, nb_mouvements);
    std::cout << nb_mouvements << " "
               << chrono.elapsed() / 1000.0 << std::endl;
    return 0;
}
```

Pour des raisons qui sont propres à Qt, nous devons impérativement créer une instance de la classe `QCoreApplication` avant de créer toute instance de `QThread` (ou d'une classe dérivée). Cette classe représente en quelque sorte l'application elle-même, de la même manière que la classe `QApplication`, rencontrée pour les programmes graphiques, sauf que `QCoreApplication` est utilisable dans une application non graphique. Incidemment, remarquons que `QApplication` dérive de `QCoreApplication`.

Les threads sont créés dans la fonction `Hanoi_Parallele()`, tout simplement à travers la création de deux instances de la classe `Hanoi_Thread`, celle-ci dérivant de `QThread`. Lorsqu'une instance est créée, les instructions que nous avons placées dans la méthode `run()` ne sont pas encore exécutées : le thread est bel et bien créé, mais il

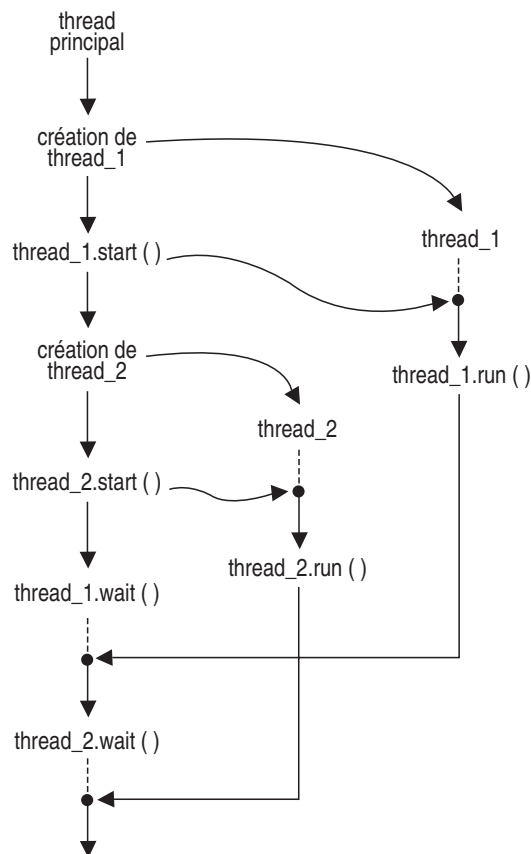
est inactif. Il ne s'exécutera effectivement qu'après avoir invoqué la méthode `start()` (issue de `QThread`).

Cette méthode `start()` se termine très rapidement, quelle que soit la longueur du traitement effectué au sein de la méthode `run()`. Aussi, est-il généralement nécessaire d'attendre que ce traitement se termine : c'est le rôle de la méthode `wait()`. En pratique, invoquer cette méthode `wait()` suspend l'exécution du programme tant que l'instance de `QThread` à partir de laquelle elle a été invoquée n'a pas terminé son exécution. Celle-ci est terminée au moment où l'on sort de la méthode `run()`.

La Figure 17.1 schématise le déroulement des opérations. Les lignes en pointillés symbolisent les moments où les différents threads sont inactifs, c'est-à-dire en attente de "quelque chose". Remarquez la présence d'un thread principal : en réalité, tout programme contient au moins un thread.

Figure 17.1

Hanoi parallélisé.



L'exécution de ce programme, sur une machine équipée d'un processeur à double cœur (donc comme si elle disposait de deux processeurs), donne 17 179 869 183 mouvements en environ cinquante secondes. Il est heureux de constater que l'on obtient le même nombre de mouvements. Mais le plus intéressant est que le temps d'exécution a bien été divisé par deux : par une réorganisation somme toute assez simple de l'algorithme, nous exploitons la puissance du matériel disponible et augmentons ainsi considérablement les performances.

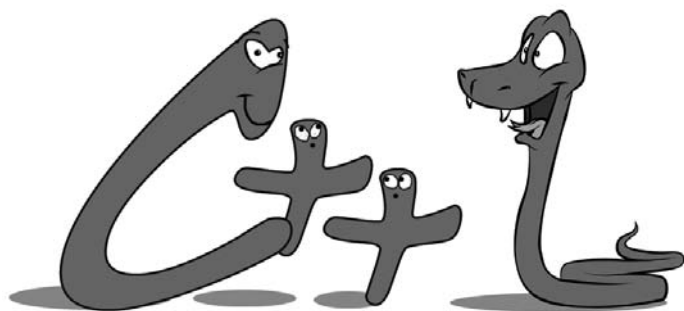
17.3. Précautions et contraintes

Naturellement, il est bien rare qu'un algorithme se prête aussi aisément au parallélisme. Dans la majorité des cas, la mise en œuvre d'une programmation parallèle peut être extrêmement complexe.

Les problèmes surgissent dès qu'il devient nécessaire que deux threads s'exécutant en parallèle puissent lire et modifier une même zone mémoire. On peut alors se trouver confronté à des problèmes subtils où un thread lit une valeur, tandis que l'autre modifie cette valeur : le premier risque de se retrouver avec une valeur erronée, conduisant presque inmanquablement à un dysfonctionnement. De plus, il est pratiquement impossible de savoir dans quel ordre sont exécutées les instructions contenues dans chacun des threads : si l'ordre au sein d'un thread est connu, il est impossible de savoir si des instructions d'autres threads ne vont pas venir s'intercaler. Et cet ordre peut varier d'une exécution à l'autre d'un même programme, ce qui conduit à une grande incertitude.

Pour pallier ce problème, des techniques existent qui consistent à fixer des moments où les threads sont sérialisés, c'est-à-dire explicitement ordonnés, comme s'il s'agissait de fonctions classiques (non parallèles). Mais ces techniques ne sont elles-mêmes pas sans risques : elles peuvent conduire à des situations où deux threads s'attendent l'un l'autre, provoquant un blocage complet du programme. Celui-ci n'est pas "planté" à proprement parler, il est tout simplement pris dans une attente infinie.

Dans un programme non trivial, l'utilisation de la programmation parallèle peut donc rapidement devenir très délicate. Elle n'est simple que dans les cas où chaque thread ne manipule qu'un ensemble limité des données qui lui est dédié. Dès que deux threads doivent lire ou modifier la même zone mémoire, les problèmes peuvent surgir, parfois de façon tout à fait inattendue. Aussi, soyez particulièrement précautionneux si vous vous lancez dans ce genre d'aventure, n'hésitez pas à vous documenter abondamment.



18

Aperçu d'autres langages

Au sommaire de ce chapitre :

- Ada
- Basic
- C#
- OCaml
- Pascal
- Ruby

Pour terminer, nous allons évoquer ici quelques autres langages de programmation parmi les plus répandus ou les plus célèbres. Nous n'entrerons pas dans les détails – cet ouvrage n'est pas une encyclopédie ! –, mais nous verrons simplement comment pourrait s'écrire notre programme donnant la solution du problème des Tours de Hanoï.

Sans doute est-il également temps de dévoiler une petite vérité. Vous trouverez dans ce qui suit des langages dits *interprétés*, d'autres dits *compilés*. En réalité, aujourd'hui, pratiquement aucun langage n'est réellement interprété. Tous les langages modernes dits interprétés s'appuient en réalité sur une *machine virtuelle*, véritable dénomination de l'interpréteur. Les programmes en langage Python que nous avons écrits tout au long de cet ouvrage, au moment de leur exécution, sont en fait transformés en une représentation intermédiaire et binaire nommée *bytecode*. Ce n'est pas exactement une compilation, mais ce n'est pas non plus une "bête" interprétation des instructions une à une. Cette représentation est ensuite exécutée par la machine virtuelle, dont le rôle est en fait d'exécuter ce bytecode en lui fournissant un environnement parfaitement défini – une sorte de modèle d'ordinateur idéal. Le célèbre langage Java, par exemple, se réfère clairement à sa JVM (pour *Java Virtual Machine*, machine virtuelle Java) pour s'exécuter.

Notez bien que la sélection proposée ici est purement subjective et ne saurait traduire ni les préférences de l'auteur, ni la popularité de tel ou tel langage de programmation. Signalons tout de même que la plupart font partie du groupe des langages impératifs.

On désigne par ce terme les langages de programmation consistant en la description d'un état du programme, représenté par ses variables, ainsi que les opérations à exécuter pour modifier cet état. Il s'agit du plus ancien paradigme de programmation et de loin le plus répandu. Les langages Python et C++ font partie de ce groupe.

Enfin, les langages évoqués maintenant appliquent tous les principes de la programmation orientée objet.

18.1. Ada

Le langage Ada a été développé à la fin des années 1970 par une équipe française dirigée par Jean Ichbiah, au sein de l'entreprise française CII-Honeywell Bull. Ce langage fait suite à un appel d'offres international lancé par le département de la Défense des États-Unis d'Amérique, qui cherchait alors à réduire le nombre de langages de programmation utilisés afin de gagner en cohérence et en efficacité globale.

Les contraintes imposées étaient extrêmement sévères, l'objectif étant moins la rapidité du développement et de l'exécution que la robustesse et la fiabilité des programmes obtenus. Le nom Ada a été choisi en hommage à lady Augusta Ada King, comtesse de Lovelace, seule fille légitime du poète anglais lord Byron. Lady Ada vécut de 1815 à 1852, brèves années durant lesquelles elle entretint une correspondance soutenue avec l'inventeur de la machine analytique Charles Babbage. Elle a longtemps été perçue comme l'auteur du premier programme informatique.

Sans plus attendre, voici un exemple de programme en Ada :

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Hanoi_Ada is
  procedure Solution(nb_disques: in integer;
                    depart    : in integer;
                    pivot     : in integer;
                    arrivee   : in integer) is
  begin
    if nb_disques = 1
    then
      Put(depart);
      Put(" -> ");
      Put(arrivee);
      New_Line;
    else
      Solution(nb_disques-1, depart, arrivee, pivot);
      Solution(1, depart, pivot, arrivee);
      Solution(nb_disques-1, pivot, depart, arrivee);
    end if;
  end Solution;
  nb_disques: integer;
begin
  Put("Nombre de disques ? ");
  Get(nb_disques);
  Solution(nb_disques, 1, 2, 3);
end Hanoi_Ada;
```

La syntaxe du langage emprunte beaucoup au langage Pascal, qui avait lui-même été conçu comme le langage idéal pour apprendre la programmation. Sa caractéristique principale, qui rebute plus d'un programmeur, est une extrême rigueur sur le plan des types : s'il est possible en C++ d'affecter la valeur d'une variable d'un type entier à une variable d'un type en virgule flottante, cela est interdit en Ada sans utiliser explicitement une conversion de type (ou *transtypage*). Cette contrainte très forte, associée à d'autres, fait que l'écriture d'un programme en Ada est généralement perçue comme

plus difficile que dans aucun autre langage. En contrepartie, une fois que le programme passe l'étape de la compilation, il est presque garanti qu'il s'exécute correctement.

Le langage Ada offre des facilités pour la programmation parallèle ou le mélange de plusieurs langages au sein d'un même logiciel. Il est aujourd'hui utilisé dans le cadre de missions critiques, comme la fusée Ariane 5, la gestion du trafic aérien ou ferroviaire ou par certaines institutions financières.

Le compilateur le plus répandu pour le langage Ada est celui développé par la société AdaCore, nommé GNAT, désormais intégré à la suite de compilation GCC (celle que nous avons utilisée depuis le début de cet ouvrage).

18.2. Basic

Ce langage de programmation, qui fut un moment peut-être le plus populaire, a été inventé en 1964. Le mot "BASIC" est un acronyme de *Beginner's All-purpose Symbolic Instruction Code*, qui signifie quelque chose comme "langage symbolique généraliste pour débutants". L'un des fondements du Basic était en effet qu'il soit simple à manipuler, notamment par des non-informaticiens.

Les premières versions du langage furent de fait assez... basiques. Les lignes du programme devaient être numérotées, il n'existait pas de boucles bornées (for) ni de sous-programmes. L'outil principal pour passer d'un endroit à l'autre dans un programme consistait à utiliser l'instruction GOTO, suivie du numéro de ligne à exécuter. On aboutissait ainsi à un programme non structuré, comparable à un plat de spaghettis... Ce qui fit dire à de nombreux informaticiens que le langage Basic faisait plus de mal que de bien et produisait plus de "mauvais" programmes qu'aucun autre.

Pourtant, ce langage a connu une popularité considérable, du fait de sa simplicité et de sa diffusion dans pratiquement tous les ordinateurs personnels, dès la fin des années 1970. Il a depuis considérablement évolué, abandonnant la numérotation des lignes et intégrant jusqu'aux notions de la Programmation Orientée Objet. L'une de ses incarnations les plus connues est le langage VisualBasic, développé par Microsoft, mais de nombreuses autres existent comme FreeBasic (multiplate-formes) ou Gambas (principalement sous Linux).

```
Sub Solution(ByVal nb_disques As Integer, ByVal depart As Integer, ByVal pivot
➡As Integer, ByVal arrivee As Integer)
    If nb_disques = 1 Then
```

```
Print Using "# -> #"; depart, arrivee
Else
  Solution(nb_disques-1, depart, arrivee, pivot)
  Solution(1, depart, pivot, arrivee)
  Solution(nb_disques-1, pivot, depart, arrivee)
End If
End Sub

Dim nb_disques As integer
Input "Nombre de disques ? ", nb_disques
Solution(nb_disques, 1, 2, 3)
```

En-dehors de ses "défauts" intrinsèques, qui font de Basic l'un des langages les plus décriés parmi les informaticiens professionnels ou chercheurs, un problème majeur est la prolifération des "dialectes" du langage. Le programme précédent a été testé en utilisant FreeBasic, mais il n'est pas garanti qu'il fonctionne avec un autre compilateur ou interpréteur Basic.

18.3. C#

Le langage C# (prononcez "ci-sharp") a été développé par la société Microsoft en 2001 afin de fournir un langage dédié à sa plate-forme .NET (prononcez "dotte-net"), laquelle constitue l'équivalent d'une machine virtuelle pour les systèmes Windows. Du point de vue de nombreux observateurs, C# et .NET ont été développés pour concurrencer directement le langage Java créé par Sun. Si le seul compilateur "officiel" de C# est Visual C#, développé par Microsoft uniquement pour les systèmes Windows, d'autres compilateurs sont apparus pour d'autres plates-formes. Citons notamment les projets Mono et DotGNU.

```
using System;
namespace Hanoi
{
  class Solution
  {
    public static void Solution(int nb_disques,
                                int depart,
                                int pivot,
                                int arrivee)
    {
      if ( nb_disques == 1 )
      {
        System.Console.Write(depart);
        System.Console.Write(" -> ");
      }
    }
  }
}
```

```
        System.Console.WriteLine(arrivee);
    }
    else
    {
        Solution(nb_disques-1, depart, arrivee, pivot);
        Solution(1, depart, pivot, arrivee);
        Solution(nb_disques-1, pivot, depart, arrivee);
    }
}
static void Main(string[] args)
{
    System.Console.Write("Nombre de disques ? ");
    int nb_disques = System.Convert.ToInt32(System.Console.ReadLine());
    Solution(nb_disques, 1, 2, 3);
    System.Environment.Exit(0);
}
}
```

La syntaxe générale du langage s'inspire fortement de celle de Java, tout en reprenant des éléments au C++ et au Pascal. Cette dernière influence vient probablement du fait que le créateur de C#, Anders Hejlsberg, fut également l'auteur du compilateur qu'on trouve dans les produits TurboPascal puis Delphi de Borland ; il fut responsable du développement de ce dernier avant de rejoindre Microsoft.

18.4. OCaml

Le langage OCaml, contraction d'*Objective Caml*, a été créé en 1996 par Xavier Leroy, directeur de recherche à l'INRIA. C'est, en fait, l'implémentation la plus avancée du langage de programmation Caml, développé depuis 1985. Il s'agit essentiellement d'un langage fonctionnel. La programmation fonctionnelle considère un programme comme une suite d'évaluations de fonctions mathématiques, plutôt que comme une suite de changements d'état comme le fait la programmation impérative. Bien que déroutante au premier abord, elle est particulièrement bien adaptée dans les situations où il est nécessaire de manipuler des structures de données complexes et récursives.

OCaml propose toutefois des possibilités permettant de mélanger les paradigmes impératif et fonctionnel : il entre donc dans la catégorie des langages multiparadigmes. Dans une certaine mesure, Python entre également dans cette catégorie, car il offre quelques possibilités de programmation fonctionnelle.

```
let rec solution nb_disques depart pivot arrivee =
  if nb_disques = 1
  then
    begin
      print_int(depart);
      print_string(" -> ");
      print_int(arrivee);
      print_newline();
    end
  else
    begin
      solution (nb_disques-1) depart arrivee pivot;
      solution 1 depart pivot arrivee;
      solution (nb_disques-1) pivot depart arrivee;
    end;;
print_string("Nombre de disques ? ");
let nb_disques = read_int() in
solution nb_disques 1 2 3;
exit 0
```

18.5. Pascal

Le langage Pascal doit son nom au philosophe Blaise Pascal, auquel il rend hommage. Créé au début des années 1970, son objectif initial était d'être utilisé dans l'enseignement de la programmation en mettant l'accent sur la rigueur et la clarté des programmes, rôle qu'il remplit encore de nos jours. Il a connu une évolution assez remarquable, notamment par la diffusion des outils TurboPascal puis Delphi par la société Borland. Ainsi est-il utilisé dans l'industrie, pour le développement d'applications professionnelles aussi diverses que variées.

```
program Hanoi(input, output);
var nb_disques: integer;
  procedure Solution(nb_disques: integer;
                    depart    : integer;
                    pivot     : integer;
                    arrivee   : integer);
  begin
    if nb_disques = 1
    then
      writeln(depart, ' -> ', arrivee)
    else
      begin
        Solution(nb_disques-1, depart, arrivee, pivot);
```



```
        Solution(1, depart, pivot, arrivee);
        Solution(nb_disques-1, pivot, depart, arrivee);
    end
end;
begin
    write('Nombre de disques ? ');
    readln(nb_disques);
    writeln;
    Solution(nb_disques, 1, 2, 3);
END.
```

Si vous comparez ce programme avec celui donné en langage Ada, vous remarquerez une certaine similarité : cela n'est pas un hasard. En effet, le Pascal a été une source d'inspiration pour le langage Ada, ce dernier en ayant conservé l'exigence de clarté (entre autres).

En-dehors du célèbre Delphi de Borland, citons le projet FreePascal, proposant un compilateur Pascal efficace, que l'on retrouve dans l'environnement graphique Lazarus. FreePascal et Lazarus présentent l'intérêt d'être multiplate-formes, en plus d'être libres et gratuits.

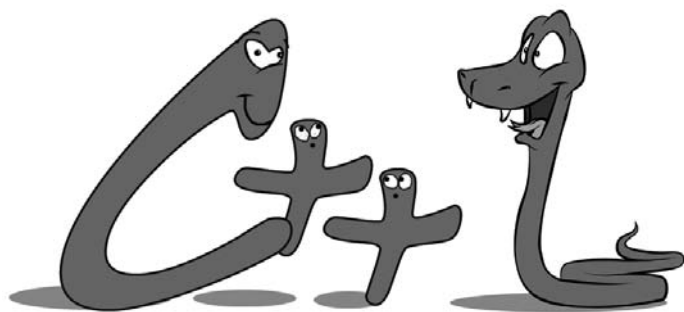
18.6. Ruby

Le langage Ruby a été créé en 1993 par Yukihiro Matsumoto, alors frustré de ne pas trouver satisfaction dans les langages interprétés existants (principalement Perl et Python). Il a été influencé par les concepts présents dans les langages Eiffel et Ada. En particulier, tout élément d'un programme Ruby est un objet (ou une classe) au sens de la programmation objet, même les variables numériques simples. Ainsi Ruby est-il l'un des rares langages purement objet, bien que permettant une programmation procédurale.

```
def Solution(nb_disques, depart, pivot, arrivee)
  if ( nb_disques == 1 )
    print depart.to_s + " -> " + arrivee.to_s + "\n"
  else
    Solution(nb_disques-1, depart, arrivee, pivot)
    Solution(1, depart, pivot, arrivee)
    Solution(nb_disques-1, pivot, depart, arrivee)
  end
end
```

```
print "Nombre de disques ? "  
nb_disques = gets.to_i  
Solution(nb_disques, 1, 2, 3)
```

La simplicité de la syntaxe n'est pas sans rappeler celle des langages Ada et Python. Aujourd'hui, Ruby est surtout connu dans le monde des applicatifs Web, par le biais de l'environnement RoR (*Ruby On Rails*) destiné à faciliter la création de véritables logiciels s'exécutant sur un serveur distant et s'affichant dans un navigateur Web. Dans ce domaine, il offre une forte concurrence au célèbre PHP.



19

Le mot de la fin

Nous voilà arrivés au terme de notre initiation à la programmation. Naturellement, beaucoup d'aspects ont été passés sous silence. En particulier, nous n'avons pas abordé certaines structures de données, comme les arbres ou les graphes, essentielles pour modéliser correctement bien des domaines de la réalité. De même, nous n'avons pas traité des environnements de développement, qui sont des logiciels spécialement conçus pour... l'écriture de logiciels. Il est toutefois préférable d'avoir au préalable "tâté" de la ligne de commande, comme nous l'avons fait, pour tirer le maximum de ces outils sophistiqués.

Si vous voulez progresser, n'hésitez pas à vous plonger dans les nombreuses documentations qui existent, aussi bien en librairie que sur Internet. Un bon moyen de trouver des idées et d'apprendre des techniques consiste également à examiner le code source d'un programme écrit par quelqu'un d'autre. C'est très formateur, bien que cela ne soit évidemment possible que dans le cas de logiciels Open Source.

C'est également un aspect de la programmation que nous avons laissé de côté : l'épineux problème des licences logicielles. En caricaturant un peu, on distingue deux grandes familles.

- Les logiciels propriétaires, ceux que vous trouvez généralement en vente, aussi bien en magasin que sur Internet. Lorsque vous achetez un logiciel propriétaire, vous n'achetez en réalité qu'un droit d'utilisation du logiciel, pas le logiciel en lui-même. En particulier, il vous est le plus souvent interdit de le copier, de le distribuer, de le modifier, et même ne serait-ce que tenter de comprendre la façon dont il fonctionne.
- Les logiciels libres, qui sont principalement diffusés par Internet. Ceux-ci suivent une philosophie pratiquement à l'opposé de la précédente : lorsque vous obtenez un logiciel (que cela soit contre paiement ou gratuitement), vous obtenez également le droit de le diffuser, de l'étudier, de le modifier. La seule contrepartie demandée (mais pas vraiment exigée) est de faire bénéficier tout le monde de vos améliorations.

Notez que cette distinction s'établit sur la philosophie du logiciel, pas sur son prix. Un logiciel propriétaire peut être gratuit, un logiciel libre peut être payant. Même si dans la pratique, c'est plutôt l'inverse qui est vérifié.

Pratiquement tous les logiciels mentionnés dans cet ouvrage sont des logiciels libres. Les deux exceptions majeures étant les systèmes d'exploitation Windows et Mac OS X (bien que ce dernier s'appuie largement sur le système Unix libre BSD).

Si cette question des licences logicielles vous semble une argutie juridique sans grand intérêt, nous ne saurions toutefois trop vous recommander d'y prêter quelque attention. Une activité très en vogue aux États-Unis, qui arrive doucement en Europe, consiste à lancer des procès à tort et à travers dans le seul but d'obtenir des indemnités financières en réponse à des préjudices, parfois bien illusoires. Si vous voulez utiliser tel outil ou telle bibliothèque, consultez toujours soigneusement sa licence d'utilisation. Et posez-vous la question de la destinée de votre programme : s'il ne s'agit que de réaliser un profit maximal à court terme ou s'il s'agit de rendre service au plus grand nombre.

C'est maintenant à vous de jouer. Attrapez votre clavier, dominez-le, pour créer les programmes qui vous feront rêver. Comme vous avez pu le constater, l'aventure n'est pas sans écueils. Soyez ambitieux dans vos projets, sans être irréaliste.

Commencez doucement, montez en puissance à mesure que vous gagnez en expérience. Il en va de la programmation comme de bien d'autres domaines : un moment survient où vous dépasserez la technique pour enfin pouvoir vous exprimer pleinement. Alors, vous apparaîtra l'art de la programmation.

Glossaire

API

Acronyme d'*Application Programming Interface*. Ensemble des déclarations de fonctions (ou classes) destinées à l'utilisateur d'une bibliothèque, pour en utiliser les services.

Bibliothèque

Ensemble d'outils (fonctions et classes) constituant un tout cohérent et regroupé en une seule entité. Divers langages utilisent plutôt les termes de modules (Python), paquetages (Ada) ou unité (Pascal), quoique ces notions ne soient pas exactement équivalentes. Les outils d'une bibliothèque sont utilisés par l'intermédiaire de son API.

Classe

Type de donnée, au même titre qu'un entier ou une chaîne de caractères, défini par les moyens de la programmation orientée objet.

Compilation

Opération consistant à transformer un fichier contenant le code source d'un programme écrit dans un certain langage de programmation, en une représentation binaire exploitable par le processeur de l'ordinateur.

Constructeur

Méthode spéciale dans une classe, exécutée lorsqu'on crée une instance de cette classe.

Instance

Variable dont le type est une classe, comme 10 est une instance du type entier ou "bonjour" est une instance du type chaîne de caractères.

Méthode

Fonction définie à l'intérieur d'une classe, agissant sur les données de celle-ci.

Thread

Flux d'instructions dans un programme. Tout programme d'exécution contient au moins un thread. Il est possible de créer plusieurs threads, ce qui donne un programme multithreadé pouvant exploiter la puissance offerte par plusieurs processeurs.

Type

Ensemble de valeurs et opérations pouvant leur être appliquées. De là est généralement déduite la façon de stocker ces valeurs dans la mémoire de l'ordinateur.

Variable

Nom donné à une zone mémoire contenant une information et moyen d'accéder à cette zone. Le plus souvent, à une variable est associé un type, définissant la nature de l'information et la manière de la stocker.

Index

A

Accesseur [108](#)
Adresse [140](#)
Aléatoire [49](#)
Allocation [142](#)
API [208](#)
Append [135](#)
argc [191](#)
argv [191](#)

B

Bytecode [238](#)

C

Cast [43](#)
Code source [10](#)
Concaténation [25](#)

Conteneur [61](#)
Couleurs [172](#)

D

Débordement [21](#)
Déclarations [145](#)
Dérivation [105](#)
Directive
 #define [145](#)
 #endif [145](#)
 #ifndef [145](#)
 #include [146](#)

E

Éditeur de texte [4](#)
Édition des liens [144](#)
Enregistrements [68](#)
Éperluette [75](#)

Espace de nommage [179](#)

Événementielle [203](#)

F

Fichiers

binaires [186](#)

fermer [189](#)

ouverture [188](#)

textes [186](#)

Fonctions [31](#)

Fuite de mémoire [144](#)

H

Hasard [48](#)

Héritage [105](#)

I

#include [146](#)

Indentation [32](#)

L

Library [210](#)

M

Machine virtuelle [238](#)

Macro [167](#)

make [149](#)

Méthode [106](#)

Mode texte [154](#)

Modèle/vue [220](#)

Module [159](#)

N

Namespace [179](#)

Notation scientifique [22](#)

O

Opération [67](#)

Ordinogramme [88](#)

Organigramme [88](#)

P

Paramètres

par référence [75](#)

par valeur [73](#)

Pixel [172](#)

Pointeur [140](#)

Polymorphisme [114](#)

Portée [35](#)

Précision [22](#)

Private [117](#)

Procédures [31](#)

Projection [220](#)

Protected [117](#)

Q

QLCDNumber [197](#)

QTime [195](#)

QTimer [195](#)

R

Répétitive [56](#)

RGB [172](#)

RVB [172](#)

S

Signal [165](#)

Signature [111](#)

Slot [165](#)

Surdéfinir [111](#)

T

Temps réel [194](#)

Thread [232](#)

Transtypage [43](#), [239](#)

Tuple [135](#)

Type [19](#)

 constructeur [71](#)

 double [20](#)

 instance [69](#)

 int [20](#)

 notation pointée [69](#)

 prétraitement [146](#)

 std

 string [20](#)

 vector<> [62](#)

U

UML [89](#)

V

Valeur de retour [37](#)

Le Programmeur

Initiation à la programmation avec Python et C++

La programmation à portée de tous !

En moins de 300 pages, Yves Bailly réussit la prouesse de vous présenter dans un style clair et concis toutes les notions fondamentales de la programmation et de vous apprendre à construire progressivement un véritable programme, le jeu des Tours de Hanoï.

L'ouvrage aborde aussi bien les aspects des langages interprétés (Python) que ceux des langages compilés (C++). Les différentes problématiques sont illustrées simultanément dans ces deux langages, établissant ainsi un parallèle efficace et pédagogique.

Les bases acquises, vous apprendrez les techniques de la programmation objet et créez des interfaces graphiques attrayantes. Enfin, en dernière partie, quelques pistes vous aideront à poursuivre votre apprentissage, de l'affichage en trois dimensions à la programmation parallèle, sans oublier une petite présentation d'autres langages courants.

Très concret, extrêmement pédagogique, cet ouvrage est accessible à tous, et n'exige pas de connaissances préalables en programmation.

À propos de l'auteur

Yves Bailly est l'auteur de nombreux articles parus dans *Linux Magazine* et *Linux Pratique*, consacrés à l'utilisation de la bibliothèque graphique Qt et à la découverte des langages Python, Ada et C++.

- Mise en place
- Stockage de l'information : les variables
- Des programmes dans un programme : les fonctions
- L'interactivité : échanger avec l'utilisateur
- Ronds-points et embranchements : boucles et alternatives
- Des variables composées
- Les Tours de Hanoï
- Le monde modélisé : la programmation orientée objet (POO)
- Hanoï en objets
- Mise en abîme : la récursivité
- Spécificités propres au C++
- Ouverture des fenêtres : programmation graphique
- Stockage de masse : manipuler les fichiers
- Le temps qui passe
- Rassembler et diffuser : les bibliothèques
- Introduction à OpenGL
- Aperçu de la programmation parallèle
- Aperçu d'autres langages

Niveau : Débutant / Intermédiaire

Catégorie : Programmation

Configuration : Multiplate-forme

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

ISBN : 978-2-7440-4086-3

