Cours de C/C++ par la pratique

Hugues Talbot

24 octobre 2008

Table des matières

Chapitre 1

Comment démarrer ?

Jusqu'à présent, vous avez été un peu maternés avec les petits langages pour débutants que sont le Java et le Matlab.. Pour pouvoir mettre la ligne « je suis un programmeur émérite en C++ » sur le CV, il faut le mériter.

Bon ce n'est pas si dur en fait, voici comment faire :

1.1 Un programme simple en C pur

Pour se dépatouiller avec un programme en C pur, voici la procédure :

1. Commencer par démarrer la machine sous Linux. Vous pourrez continuer votre éducation à C++ sous Windows à la maison si vous le souhaitez ;
2. Logez vous sur votre compte habituel ;
3. Démarrez une console. Vous pouvez utiliser **konsole** par exemple ;
4. Trouvez un éditeur de texte pour programmeur. J'aime bien **Kate**, mais là encore c'est pour les débutants. Les vrais programmeurs utilisent **emacs** ou **vi**[1] ;
5. Tapez le programme suivant dans votre éditeur (et sauvegardez le dans un répertoire où vous pourrez le retrouver, par exemple `c++/hello.c` :

```
#include <stdio.h>

int main()
{
    printf{"Bienvenue au monde de C !\n");
    return(0);
}
```

6. Contrairement à Java, inutile de donner un nom particulier à votre fichier, mais il est préférable qu'il se termine par l'extension `.c` en C pur, et par `.cc` pour un programme en C++. Le programme précédent est en C pur.
7. Vous devez maintenant *compiler* votre programme. Contrairement à BlueJ qui est bien gentil, il n'y a pas de bouton à presser pour obtenir ce résultat. Il faut *tout* taper !!

[1] Exercice optionnel : taper l'exemple sous emacs, puis vi. Bon courage, surtout pour vi.

(a) Dans **konsole** (ou équivalent) déplacez vous vers le répertoire où ce trouve votre programme. Souvenez vous de votre cours de morpho, avec les commande Unix (`cd`, `mkdir`, etc) ;

(b) tapez la ligne suivante, en supposant que votre programme s'appelle `hello.c` :

```
gcc -g -o hello hello.c
```

(c) Cette ligne s'interprète de la façon suivante : compiler avec le compilateur GNU C (`gcc`), en utilisant les informations de débogage (*-g*) pour fournir *l'executable* `hello`, en partant du fichier `hello.c`. Simple, non ?

(d) Executez le programme résultant :

```
./hello
```

(e) le `./` sont nécessaire pour indiquer que l'executable est dans le répertoire présent. `hello` est le nom de l'executable. Sous Unix/Linux, il est inutile d'ajouter l'extention `.exe` pour un executable, mais vous pouvez le faire si vous voulez[2].

8. Voila, vous avez créé, compilé et executé votre premier programme en C !! Bravo.

1.2 Un programme en C++

Pour faire la même chose en C++ pas grand-chose ne change.

1. Le programme devient :

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Bienvenu au monde de C++ !" << endl;
    return(0);
}
```

2. Sauvegardez votre programme sous le nom (par exemple) de `hello.cc`.

3. Compilez le avec la commande suivante :

```
g++ -g -o hello hello.cc
```

4. Le reste est identique.

On constate que les deux langages ont des caractéristiques communes, mais des différences non triviales. C'est pourquoi il est à mon sens nécessaire d'apprendre les rudiments des deux langages. Nous reverrons ces deux programmes un peu plus loin plus en détail.

[2] mais c'est considéré de mauvais goût par certains

1.3 La suite

Les languages C et C++ sont munis chacun d'une riche suite d'outils. Pour réaliser des programmes simples, un terminal, un éditeur de textes et un compilateur sont suffisants dans un premier temps.

Pour répondre à des besoins ultérieurs, on aura besoin d'apprendre la compilation séparée (un programme complet dont le code source est réparti entre plusieurs fichiers), ce qui nécessite l'outil `make`. Les programmes un peu complexes contenant des bogues nécessiteront l'apprentissage du débogueur `gdb`, puis d'outils séparés pour les problèmes de mémoire, puis peut-être plus tard les outils pour produire une interface utilisateur graphigue, la documentation, etc.

1.3.1 L'outil make

Pour l'instant nous nous contenterons de l'outil `make`. Make est un langage de programmation en soi, dont le but est de créer des programmes ! Make fonctionne avec des « makefiles » qui sont simplement des fichiers texte (comme les programmes C ou C++), qui sont en fait des programmes dans le langage particulier de make. Pour des raisons historiques, il est préférable de nommer un fichier de commande make `makefile` ou `Makefile`, car ils sont trouvés automatiquement par la commande `make`.

Exemple de makefile

Voici un makefile générique en figure 1.1.

Ce makefile est livré avec l'ensemble des programmes. En décrire le fonctionnement est un peu long mais utile, mais tout d'abord, un point `extrèmement` important :

NOTE : les lignes *suivant immédiatement* une « *cible* », indiqués comme comportant le caractère `:`, par exemple :

```
clean:
        rm ${CEXE}
```

sont *précédées* du caractère `TAB` *ET NON DE PLUSIEURS CARACTERES ESPACES.* En d'autres termes, le contenu du fichier est le suivant :

```
clean:
<TAB>rm ${CEXE}
```

où `<TAB>` indique le caractère tab (à gauche du A sur les claviers français).

CE POINT EST ESSENTIEL. Les makefiles ne fonctionneronts pas sans ceci.

Voici maintenant la description du makefile :

1. `CSOURCE` est une variable qui décrit les fichiers sources en C pur. Vous pouvez remplacer son contenu par les noms de vos fichiers C pur, au fur et à mesure que vous les créez. Par exemple, pour l'instant, vous pouvez simplement mettre
   `CSOURCE=hello.c`
2. `CEXE` est une variable qui décrit les fichiers executables C purs déduits des fichiers sources précédents. Dans la configuration présente, l'excutable pour chaque fichier C pur est le même nom de fichier sans extension. Le « `:=` » plutôt que le « `=` » simple indique qu'on peut préciser plusieurs lignes `CEXE`.

Listing 1.1 – Hello World en C

```
#
# Makefile for the C++ course examples
#
# Hugues Talbot 2006-2008
#

CSOURCE=creadlongstring.c  derefdouble.c getchar.c incf.c \
        tabmoyenne.c cprint1.c cscanf.c fprintf.c hello.c readwrite.c \
        gdb1.c gdb2.c gdb3.c

CEXE := ${CSOURCE:.c=}

CXXSOURCE=ccformat_exo1.cc  ccformat_sol1.cc  ccin.cc \
          ccprint2.cc       matmult1.cc       readline.cc \
          fstream.cc ofstream.cc

CXXEXE := ${CXXSOURCE:.cc=}

CFLAGS=-g -Wall

.c:
        ${CC} ${CFLAGS} $< -o $@

.cc:
        ${CXX} ${CFLAGS} $< -o $@

# 'test' is not a real target
.DUMMY: test

all: ${CEXE} ${CXXEXE}

clean:
        -rm ${CEXE}
        -rm ${CXXEXE}
        -rm *~

test:
        @echo "CSOURCE␣␣␣␣=␣" ${CSOURCE}
        @echo "CEXE␣␣␣␣␣␣␣=␣" ${CEXE}
        @echo "CXXSOURCE␣=␣" ${CXXSOURCE}
        @echo "CXXEXE␣␣␣␣␣=␣" ${CXXEXE}
```

3. `CXXSOURCE` et `CXXEXE` sont la même chose, pour les fichiers et programmes en C++.
4. `CFLAGS` spécifie les options à passer au compilateur. Ici on demande de générer les informations de débogage (`-g`) et de produire toutes les informations suspectes au cours de la compilation (`-Wall`, c-à-d afficher tous les « warnings »).
5. `.c`. Il s'agit ici de la ligne qui invoque le compilateur. On appelle cette ligne particulière une *directive*. La ligne suivante est une commmande, telle qu'on pourrait la taper dans un terminal (tel que nous l'avons fait auparavant). Dans celle-ci `$CC` est une variable remplie automatiquement qui invoke le compilateur par défaut, `$<` est la *dépendance* et `$` la cible (target). Dans le langage particulier de `make`, ces deux termes signifient respectivement le fichier source, et le fichier executable. Cette ligne (et la suivante pour le C++) sont les plus importantes de ce makefile. Cette directive et sa commande associée est invoquée automatiquement à chaque fois qu'un fichier se terminant en `.c` doit être recompilé.
6. `.cc` la même chose pour C++ et les fichiers se terminant en `.cc`.
7. `.DUMMY : test`. « test » est une cible possible (voir plus loin). Cette ligne indique que celle ci ne crée pas de nouveaux fichiers.
8. `all :` est la première vraie cible. C'est celle par défaut (voir plus loin). Elle crée les exécutables à partir des sources. On constate que cette cible n'est pas suivie de commande. C'est mystérieux, n'est-ce pas ? En fait les commandes sont implicites, et viennent des directives `.c` ou `.cc` précédentes.
9. `clean` est aussi une « `DUMMY` » cible, comme `test`, mais elle est si courante qu'on ne le précise pas : `make` le sait déjà. C'est une cible qui élimine les fichiers exécutables produits (pour une recompilation par exemple). Cette cible est suivie par des lignes de commandes, identiques à ce qu'on taperait à la ligne de commande, å un petit détail près : Les lignes suivante commencent par le caractère « `-` » : ceci veut dire que même si la commande en question retourne une erreur, la suivante sera quand même exécutée. Par défaut make s'arrête à la première erreur recontrée. C'est indispensable dans le contexte de la cible `clean`, pour pouvoir tout supprimer avec la commande « `rm` », et donc de ne pas s'arrêter si une ligne de commande demande à effacer des fichiers qui n'existent pas.
10. `test`. La fameuse cible dummy. Elle liste les fichier source et executable connues de ce fichier. Les commandes suivant cette ligne comportent le caractère « ». Ceci indique que la commande elle-même n'est pas reproduite en sortie, seulement son résultat.

NOTE : Dans un makefile, les lignes commençant par un point et comportant un « `:` » (par exemple `.c :` sont des `directives`. Toutes les autres lignes comportant un caractère « `:` » sont des `cibles explicites`.

Principe de fonctionnement de Make

Make est basé sur des principes de programmation en intelligence artificielle, où on précise un but à atteindre, mais non une série d'étapes successives. Ici les buts à atteindre sont simplement des créations de fichier. Make utilise cette génération pour vérifier qu'un executable à été produit à la suite d'une compilation. Il compare la date de création d'un fichier (executable) à celle du fichier source pour décider s'il y a lieu de relancer la compilation.

Dans ce contexte, un fichier executable sera une « cible » et un fichier source une « dépendance », indiquant qu'une cible est produite par l'action d'un compilateur (ou d'une commande en général) sur la dépendence.

NOTE : attention à ne pas confondre cible fichier, cible dans makefile, etc.

Make est invoqué par la commande `make`. On peut y rajouter des cibles :

```
make
make all
make hello
make test
make clean
(etc.)
```

les cibles `all`, `test`, `clean` sont des cibles prévues dans le makefile, mais `hello` n'en est pas une. En fait comme le makefile donné indique comment générer un fichier executable (sans extension) à partir d'un fichier `.c` ou `.cc`, la commande `make` va chercher dans le répertoire courant s'il existe un fichier `hello.c` ou `hello.cc`. Si oui il va les compiler avec la commande appropriée.

La cible par défaut est la cible `all`, car c'est la première spécifiée explicitement dans le makefile donné en exemple, donc `make` et `make all` ont dans notre contexte le même effet.

Remarques sur make

Make est un langage délicat mais puissant. Un bon développeur doit pouvoir s'en servir, autant commencer tout de suite ! Ne vous découragez pas.

A l'instar du langage C++ les outils Unix sont ésotériques, d'aspect rebutant au début, mais puissants et efficaces une fois qu'on commence à les maîtriser, au point de ne plus pouvoir s'en passer. Par exemple ce document entre vos mains est produit à l'aide de make.

1.3.2 Autres outils

Ils seront introduits au fur et à mesure du cours en fonction des besoins.

Chapitre 2

Introduction

Ce support de TP est une introduction à C/C++ par la pratique. On suppose que le lecteur connaît le langage Java.

Pour la suite, les exemples sont illustrés sur le système Linux/Unix, mais sont sensés fonctionner également sous Windows ou MacOS/X.

2.1 Pourquoi parler à la fois de C et de C++

Pour plusieurs raisons :

- On pourrait se contenter de faire un cours de C++ pur, mais certaines fonctions importantes ne sont disponibles qu'en C, il faut donc en parler de toutes manières.
- C est un langage à part entière, beaucoup plus simple que C++ mais avec ses idiomes et ses propres difficultés.
- Certains aspects difficiles de C++ se comprennent mieux avec un éclairage par le langage C qui est plus accessible parfois.

D'autre part il est important de parler de C++ car ce langage est extrêmement puissant, souple et très demandé dans l'industrie.

2.2 C, C++ et Java

2.2.1 Ce qui est pareil

Les trois langages ont en commun :

- Un style commun (avec des `()` et des {} par exemple).
- les fonctions, les structures de contrôle (`if`, `else`, `while`, `for`, etc). On ne les répétera pas ;
- Les types numériques signés (plus ou moins).

2.2.2 Ce qui diffère entre C/C++ et Java

Beaucoup de choses ! entre autres :

- Java n'a pas de type non signés (`unsigned int` par exemple) sauf pour le type byte.
- Tout est une classe dans Java. Ce n'est pas nécéssaire en C/C++
- Les noms de fichiers ne sont pas imposés (En Java, nom de classe et nom de fichier doivent être identiques)

- La gestion de la mémoire (automatique en Java, manuelle en C/C++)
- Les chaînes de caractères font 16 bits par caractères en Java, 8 en C. C++ supporte les deux.
- Le style de programmation, bien plus contraignant et verbeux en Java
- La librairie standard Java, beaucoup plus vaste !
- Et encore bien plus...

On ne peut guère s'appuyer sur Java pour fonder un cours de C/C++, mais on ne part pas de zéro tout de même.

2.2.3 Ce qui diffère entre C et C++

Bien des choses, mais presque tout programme C est automatiquement un programme C++. Ce dernier est plus contraignant sur les types, et donc certains programmes C valident ne compilent pas en C++. En général ces derniers sont souvent mal écrits (font des suppositions hasardeuses par exemple).

Un des buts de ce TP et des suivants est de présenter ces différences et similarités par l'exemple.

2.3 Style

En C et en C++, il est extrêmement facile de se tromper, même pour le programmeur expérimenté. Il est important de respecter les points suivants :

2.3.1 Indentation

Un code lisible est moins facilement faux. Alignez vos déclarations et indentez les blocks logiques. Profitez des possibilités d'indentation de votre éditeur pour garder votre code propre.

Exercice 1 (Erreur logique) *Quelle est l'erreur dans le code du listing 2.1 ? Trouvez la sans éditer*

2.3.2 Commentez

Pour que les autres et vous-même plus tard compreniez ce que vous voulez dire, commentez votre code abondamment. Dans un programme professionnel, on compte (normalement !) autant de lignes de commentaires que de lignes exécutables.

Comme en Java, utilisez `//` pour commenter une fin de ligne, et le couple `/* */` pour commenter plusieurs lignes.

N'utilisez pas le préprocesseur (`#ifdef / #endif`) pour commenter. Ces macros sont strictement pour éliminer temporairement des parties de programme pour débogage.

En C standard ANSI 1989, les commentaires `//` ne sont pas légaux, mais la plupart des compilateurs les respectent, et d'autre part ces commentaires sont devenus part du langage depuis la norme ISO-C99.

Listing 2.1 – Erreur logique

```
//
// Erreur ?
#include <stdio.h>

int main()
{
    int a, b;

    /* code intermédiaire non montré */
    if (a == 0)
        if (b == 0)
            print("a et b sont nuls\n");
    else
        print("ni a ni b ne sont nuls\n");

}
```

Chapitre 3

Bases

Quelques généralités pour s'y retrouver :

3.1 Déclaration / définition

En C/C++ on doit normalement *déclarer* les choses avant de les utiliser : les variables, les classes, les fonctions, etc. Le C est plus « coulant » que le C++ sur ce point.

Une *déclaration* est une ligne du type

```
// déclaration de variable
extern int a; // indique une variable non définie dans le fichier en cours
// déclaration de type
typedef struct {
    int a;
    float b;
} couple; // crée un nouveau type de données : un couple entier/flottant
// déclaration de fonction (prototype)
float[] matrixmult(float[] a, float[] b, int size);
```

Une *définition* est l'endroit du code où ce qui a été déclaré est « fabriqué », exemple :

```
// définition de fonction
int additionne(int a, int b)
{
    int c;  // déclaration ET définition de variable
    c= a+b; // ceci est une expression.
    return (c);
}
```

3.2 Types

Le C/C++ comme Java est un langage typé, c'est à dire que toutes les variables sont attribuées d'un type (entier, flottant, classe, etc).

3.2.1 Types en C

Le C n'est pas un langage typé « fort » au sens ou on peut convertir un type en un autre, parfois automatiquement. Par exemple :

```
int   a = 10 ;
float b = 20.0 ;

float c = a * b; // dans cette expression, a est automatiquement converti en float.
```

Les règles de « promotion » des types sont complexes mais vont généralement du type le moins capable au plus capable, donc ici d'entier vers flottant.

Transtypage En C/C++ On peut forcer un type vers un autre, même s'ils ne sont pas compatibles a-priori, en utilisant l'opérateur de transtypage, encore appelé « moulage » (ou *cast* en anglais). On « moule » un flottant en entier par l'opération suivante :

```
float f = 3.1415;
int   d;

d = (int)f;
```

Le transtypage s'effectue avec le type de destination entre parenthèses.

En C++, l'opérateur de transtypage le plus courant a un une syntaxe différente, plus verbeuse mais plus claire pour qui connaît les templates :

```
d = static_cast<int>(f);
```

Le C++ possède trois autres opérateurs de transtypage :

- `dynamic_cast` réservé aux pointeurs et références sur des classes. Ce transtypage effectue des vérifications de validité.
- `reinterpret_cast` n'effectue aucune vérification, et est le plus proche du transtypage C. Il est à déconseillé dans le contexte du C++.
- `const_cast` permet de convertir un type constant (ou volatile) en un type normal, à l'exclusion de toute autre conversion.

Entiers

Il existe au moins 4 types d'entiers :

- `char`, le type de caractère, est en fait un petit entier (8 bits, un octet) normalement non signé
- `short`, un type d'entier court (normalement 2 octets)
- `int`, le type entier normal (normalement 4 octets)
- `long`, le type entier long, capable d'accueillir le résultat d'une multiplication entre deux ints (4 ou 8 octets suivant le système).

De plus ces types peuvent être affublé des qualificatifs `signed` et `unsigned`. Comme on l'imagine, un entier unsigned ne peut être négatif. Il existe aussi le type `long long` (un entier très long), qui est généralement à éviter.

Le standard ne spécifie que les tailles relatives de ces types, et non les tailles absolues. Le dernier standard C (C99) spécifie des entiers à taille absolue avec des noms différents (`int32` : un entier de 32 bits signé par exemple).

Flottants

Il existe 2 types de flottants, tout deux signés :
– `float` : un flottant simple précision codé sur 32 bits.
– `double` : un flottant double précision (d'où le nom) codé sur 64 bits.

Anciennement faire des calculs sur des `float` était plus rapide, mais ce n'est plus vrai depuis longtemps. En revanche un tableau de `double` prend bien deux fois plus de place qu'un tableau de `float`. Le codage des float/double est formalisé par la norme IEEE 754 et bénéficie du support matériel de l'unité centrale.

Pour pouvoir réaliser des calculs sur les flottants plus compliqués que les 4 opérations, on doit généralement inclure le header `math.h` (ou `cmath` en C++), par exemple :

```
#include <math.h>

double d = 2.0;
double s = sqrt(d); // calcul de la racine carrée
```

Exercice 2 (Type de fonction par défaut) *Question : que se passerait-il dans le cas suivant ?*

```
// sans inclure <math.h>
double s = sqrt(2);
```

La réponse est subtile.

3.2.2 Tableaux

Comme on va très vite parler des tableaux, donc voici comment on les déclare :

```
int a[100]; // un tableau de 100 entiers
double b[10][10] // une matrice à 2 dimensions de doubles.
```

Voici comment on les utilise dans des fonctions :

```
int sumtrace(double mat[][], int size)
{
    double s = 0;
    for (int i = 0 ; i < size ; ++i) {
        s += mat[i][i];
    }
    return s;
}
```

La syntaxe des tableaux en 2 dimensions peut sembler étrange mais vient de la très grande proximité des notions de tableau et de `pointeurs` que nous verrons plus tard.

3.3 Pièges

Les pièges sont nombreux en C/C++. Pour les trouver plus facilement, utiliser les avertissements de votre compilateur. Pour le compilateur GNU (gcc, g++) il faut utiliser l'option `-Wall` (all warnings).

Le piège numéro 1 est le même qu'en Java, c'est le suivant :

```
if (a = 2) {
    /* faire quelque chose */
}
```

Exercice 3 (test piège) *Que fait ce code ? Pouvez vous suggérer une méthode éviter le problème associé ?*

3.4 Préprocesseur

Le C et le C++ sont dotés d'un préprocesseur qui est une sorte de méta-langage de macros.

Toutes les directives du préprocesseur commencent par le caractère `#`.

3.4.1 Inclusion

La directive la plus courante est le `#include` qui inclus un fichier dans un autre. On peut mettre n'importe quoi dans un fichier inclus, y compris du code exécutable, mais par convention on n'y place que des déclarations. Un fichier qui est inclus s'appelle un « header » et est normalement un fichier qui se termine en `.h` (ou parfois `.hh`, `.H`, `.hxx` ou encore `.h++` pour le C++)

Il y a deux façon d'invoquer `#include` :

– `#include <someheader.h>` réservé aux headers du système.

– `#include "someheader.h"` pour vos propres headers, ceux que vous créerez.

Il est usuel, dans un projet assez grand, de créer ses propres headers et d'y mettre les définitions qui sont partagées entre plusieurs fichiers.

3.4.2 Directives préprocesseurs

On peut définir des macros en préprocesseur avec la directive `#define`, c'est à dire des expressions qui ressemblent à des fonctions (normalement très courtes) qui sont substituées directement dans le texte du programme. Une macro très courante est :

```
#define MIN(a,b) ((a) < (b)) ? (a) : (b)
```

La construction étrange avec `?` et `:` est un test simple en une ligne. De manière générale, l'expression

```
(test) ? si_oui : si_non
```

est exactement équivalente à

```
if (test)
    si_oui;
else
    si_non;
```

mais le test s'écrit en une ligne. Une macro doit être exprimée en une seule ligne, mais peut être définie sur plusieurs lignes en les continuant avec le caractère '\'. On pourrait écrire la macro `MAX` de la façon suivante :

```
#define MAX(a,b) \
    if ((a) > (b)) \
        (a) \
    else \
        (b)
```

Ici dans la macro `MIN`, si `a` est plus petit que `b` alors le résultat du test est `a`, sinon `b`. Notez que le type de a n'est pas spécifié. Ce test marche car la macro fonctionne par substitution de caractère par le préprocesseur. En d'autre termes, avant la compilation, l'expression

```
int a = MIN(2, 5);
```

est substitué par l'expression

```
int a = ((2) < (5)) ? (2) : (5);
```

Et on a bien le résultat voulu : 2. Notez que l'un de `(a)` ou `(b)` est évalué deux fois, ce qui peut avoir des effets secondaires. Dans le code du listing 3.1

Listing 3.1 – Piège des macros

```
#include <stdio.h>

#define MIN(a,b) (a) < (b) ? (a) : (b)

int f()
{
    // une variable static est persistante
    // sa valeur est gardée d'un appel au suivant.
    static int count = 0;

    return ++count;
}

int main()
{
    printf("le min de (f() et 1) vaut %d\n"
           "Le min de (1 et f()) vaut %d\n", MIN(f(), 1), MIN(1, f()));
}
```

on a le résultat suivant :

```
$ ./incf
le min de (f() et 1) vaut 1
Le min de (1 et f()) vaut 2
```

Ce qui est pour le moins curieux... Les macros préprocesseur sont une grande source de pièges difficiles, on tâchera de les éviter si possible. En C++ et en C99 on peut presque toujours les remplacer par une fonction `inline`, c'est à dire substituée en place comme les macros, mais en respectant la syntaxe des fonctions C normales. Dans les cas où l'on souhaite conserver l'indépendance de la macro quant aux types, on peut en C++ utiliser une fonction `template`, que nous verrons en section 7.6.

3.4.3 Compilation conditionnelle

Le préprocesseur permet de ne compiler qu'une partie d'un fichier avec l'opération logique `#define` du préprocesseur.

```
#ifdef SOMEVAR

/* le code ici n'est compilé que si SOMEVAR est définie par le préprocesseur */

#endif // SOMEVAR
```

Ce mécanisme est particulièrement utile dans les headers, pour éviter qu'un header soit inclus de manière multiple.

Contenu dans le header `myheader.h`

```
#ifndef MYHEADER
#define MYHEADER

/* déclarations dans le header */
/* ces déclarations ne seront inclues qu'une fois par fichier */

#endif // MYHEADER
```

3.4.4 Autres

Entre autres :

- `#pragma` permet d'invoquer des caractéristiques du compilateur
- `#LINE et #FILE` permettent d'écrire certains messages de débogage.
- `##` est une directive de concaténation qui permet par exemple la programmation générique en C pur, mais est très difficile d'emploi.

Chapitre 4

Entrées – sorties

Bien qu'on rassemble souvent les deux langages sous une même dénomination, Les exemples suivants sont illustratifs des styles de programmation différents.

4.1 Hello

Un premier programme en C est donné en listing 4.1.

Listing 4.1 – Hello World en C

```c
// Programme qui imprime "hello world" 'a la console
//

// bibliotheque standard d'entree-sorties
#include <stdio.h>

// fonction principale -- point d'entree
// Les parenthese vident signifient : arguments arbitraires
int main()
{
    // la commande, qui est une fonction externe de libc
    printf("Hello world\n");
    return(0);
}
```

La version C++ est donnée en listing 4.2.

Pour compiler sous Linux, on exécute les commandes suivantes (le caractère '$' représente l'invite de commande et n'est pas à taper :

1. Pour C pur :

```
$ cc -Wall -g -o hello hello.c
$ ./hello
Hello world
```

2. Pour C++ :

Listing 4.2 – Hello World en C++

```
//
// Programme qui imprime hello a la console
//

// Notez la bibliotheque differente
#include <iostream>

// les fonctions du namespace std qu'on va utiliser
using std::cout;
using std::endl;

int main()
{
    // cout est ici un stream de sortie
    cout << "Hello world" << endl;
}
```

```
$ c++ -Wall -g -o hello hello.c
$ ./hello
Hello world
```

Dans les deux cas `-Wall` est une *option* qui demande au compilateur de signaler toutes les erreurs potentielles. Cette option est spécifique au compilateur GNU, si vous en utilisez un autre, il possède sans doute une option similaire. Dans tous les cas cette option est **fortement** recommandée.

Exercice 4 (Prise en main) *Entrez les deux programmes ci-dessus dans un éditeur, compilez et exécutez les.*

4.2 Lecture/écriture

Les entrées-sorties sont parmi les opérations les plus courantes en programmation. Nous allons donner une série d'exemples à la fois en C et en C++

4.2.1 Sortie standard en C

La fonction principale de sortie sur la console est, comme on l'a vu, la fonction `printf` qui fait partie de la bibliothèque standard C. Elle permet de réaliser des affichages formatés, comme dans l'exemple `cformat1.c` du listing 4.3.

Le résultat de l'exécution de ce programme est le suivant :

```
$ ./cprint1
Message : Une valeur quelconque          : 10 * 3.14 = 31.415. Etrange non ?
```

Exercice 5 (Formatage en C) *Compilez et exécutez le programme du listing 4.3. Lisez le manuel de printf (commande* `man 3 printf`*). Interprétez le résultat, en particulier pourquoi avons nous le résultat numérique étrange ? Corrigez le.*

Listing 4.3 – Formatage en C

```
//
// Un exemple de formatage en C
//
#include <stdio.h>

int main()
{
    int a = 10;
    float f = 3.1415;
    const char *s = "Une valeur quelconque";

    printf("Message : %-30s : %d * %.2f = %g. Etrange non ?\n", s, a, f, a*f);

    return 0;
}
```

Notez que ce programme en C donne explicitement une valeur de retour (0) au moyen du mot réservé `return`. Sous le shell Unix, la valeur 0 est interprétée comme signifiant « succès », et une valeur positive comme une valeur d'échec. Nous verrons plus loin comment utiliser cette caractéristique.

Note : Il existe d'autres fonctions de sortie standard à la console en C, dont `puts, fputs` et `putc` qui sont plus efficaces et plus spécialisées. Il est utile de les connaître.

4.2.2 Sortie standard en C++

En C++, on utilise la classe de stream `cout` (**c**onsole **out**put) et l'opérateur `< <`. Le formatage s'opère avec des méthodes liées à `cout`. Un exemple est donné dans le listing 4.4

Dans cet exemple on a utilisé la classe standard `string` et non un tableau de caractères. Nous verrons plus loin l'intérêt de cette démarche.

On constate que le formatage au moyen de `cout` est beaucoup plus verbeux. En revanche, une fois que la syntaxe est connue, elle est plus facile à comprendre. On peut d'ailleurs simplifier les directives de formatage avec les fonctions contenues dans le header `iomanip`.

Exercice 6 (Formatage en C++) *Complétez le programme du listing 4.5 pour afficher le contenu du vecteur* `values` *et sa somme totale de la manière suivante :*

```
valeur[0] =     10.3000
valeur[1] =     43.6450
valeur[2] =    242.3000
valeur[3] =     35.8000
valeur[4] =     75.8890
valeur[5] =    204.1120
valeur[6] =    100.0010
valeur[7] =  13434.4434
-----------------------
Somme =      14146.4902
```

Listing 4.4 – Formatage en C++

```
//
// formatage de sortie en C++
//

#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::ios;
using std::string;

int main()
{
    int a = 10;
    float f = 3.1415;
    string s = "Une valeur quelconque";

    cout << "Message : " ;
    cout.width(30); // largeur de message
    cout.setf(ios::left, ios::adjustfield); // ajuste à gauche
    cout << s ;
    cout.width(0); // signifie 'autant que nécessaire'
    cout.setf(ios::fixed, ios::floatfield); // passe en format virgule fixe
    cout.precision(2);
    cout << a << " * "
         << f << " = " ;
    // passe en format virgule général
    cout.setf(ios::fixed|ios::scientific, ios::floatfield);
    cout.precision(6);
    cout << a*f << endl;

    return 0;
}
```

Ajoutez des valeurs au vecteur, et voyez comment la somme évolue. Que se passe t-il si on a plus de 9 valeurs ?

facultatif : *faire la même chose en C pur.*

Listing 4.5 – Exercise de formatage en C++

```
//
// formatage de sortie en C++
//

#include <iostream>
#include <iomanip>
#include <string>

using std::cout;
using std::endl;
using std::ios;
using std::setw;
using std::setfill;
using std::setprecision;
using std::string;

int column(float valeurs[], int nbval)
{
    /*
     * remplir la fonction
     *
     */

    return 0;
}

int main()
{
    float values[] = {
        10.3, 43.645, 242.3, 35.8, 75.889, 204.112, 100.001, 13434.443
    };
    column(values, sizeof(values)/sizeof(float));
    return 0;
}
```

4.2.3 Entrée standard en C

L'entrée de données standards en C s'opère par la fonction `scanf`, comme dans l'exemple du listing 4.6.

Ce listing illustre certains des points difficiles du C. La fonction `scanf` doit retourner ce qui est entré, mais on ne peut pas le faire par la valeur de sortie de la fonction (c-à-d du type `a= scanf()`), car il faudrait alors autant de types de `scanf` que de types de données. Les concepteurs ont opté pour une seule fonction à la `printf` qui fait tout, au prix d'une complexité certaine.

Listing 4.6 – Lecture en C

```
//
// Entrée de données en pur C
//

#include <stdio.h>

int main()
{
    int   a;
    float f;
    char  s[100];

    printf("Entrez un entier : ");
    scanf("%i", &a);
    printf("Vous avez entré %d\n", a);
    printf("Entrez un nombre à virgule flottante : ");
    scanf("%f", &f);
    printf("Vous avez entré %f\n", f);
    printf("Entrez une chaîne de caractères\n");
    scanf("%s", s);
    printf("Vous avez entré %s\n", s);

    return 0;
}
```

D'abord le type de donné est spécifié avec des codes qui diffèrent entre `printf` et `scanf`, par exemple le `%d` par rapport au `%i` pour spécifier un entier. Ensuite chaque variable doit être spécifiée par *référence*, et c'est ça la signification du caractère '&' dans le listing. Finalement une référence à une chaîne de caractère en C est le nom de la chaîne elle-même, contrairement aux autres types. Il est facile de se tromper !

Exercice 7 (Utilisation de scanf) *On peut demander à* `scanf` *de lire plus d'un type à la fois. Réécrivez l'exemple donné pour lire d'un coup les 3 types (entier, flottant et chaîne de caractère) au moyen d'un seul appel à* `scanf`*.*

Exercice 8 (Lecture d'une chaîne avec espaces) *Vous pouvez noter que lorsque vous entrez une chaîne avec espace dans le programme du listing 4.6, seul le premier mot est lu. Comment remédier à ce problème ?*

Il y a plusieurs solutions, lisez la page du manuel sur scanf et tentez de lire les mots les uns après les autres. Il vous faudra sans doute une fonction pour concaténer des chaînes de caractère : essayez `strcat`*. Cette solution est-elle acceptable ?*

4.2.4 Entrée standard en C++

L'entrée standard en C++ se fait par la classe `cin` (**c**onsole **in**put) et par l'opérateur `>>` comme dans l'exemple du listing 4.7

On voit que l'entrée en C++ est plus simple et intuitive, mais on ne sait toujours pas lire une chaîne de caractère avec espaces avec `>>`. Cependant on peut le faire avec `getline()` comme dans le listing 4.8

Vous en savez assez pour faire un petit exercice un peu plus complet :

Exercice 9 (Entrée de matrice) *Écrivez un programme qui lit deux matrice* 2×2 *sur l'entrée standard, qui en calcule le produit et l'affiche à l'écran de manière raisonnable. Il est conseillé de représenter les matrices par des tableaux simples et de faire les calculs d'indices à la main, mais vous pouvez aussi utiliser les types de tableaux à deux indices.*

4.2.5 Fichiers en C

Les entrées-sorties à la console sont utiles dans une certaine mesure, mais on a souvent besoin de lire/écrire des données en quantité importantes (des images par exemples). Pour cela on passe généralement par des fichiers sur disque.

En C pur, il y a trois façons générales de lire/écrire dans un fichier, soit caractère par caractère, soit par accès avec tampon, soit par accès sans tampon. Un tampon permet de limiter les accès au système d'exploitation et donc d'être relativement plus efficace dans certains cas.

Accès caractère par caractère

Les accès caractères par caractères sont une extension simple des entrées/sorties consoles, où on suppose qu'on a affaire à un ou des fichiers obtenu par redirection de l'entrée/sortie standard. Le listing 4.9 propose une illustration.

Il s'utilise de la façon suivante (sous Unix/Linux et Windows) :

Listing 4.7 – Lecture en C++

```
//
// Entrée de données en pur C
//

#include <iostream>
#include <string>

using std::cin;
using std::cout;
using std::endl;
using std::string;

int main()
{
    int    a;
    float  f;
    string  s;

    cout << "Entrez␣un␣entier␣:␣";
    cin >> a;
    cout << "Entrez␣un␣nombre␣en␣virgule␣flottante␣:␣" ;
    cin >> f;
    cout << "Entrez␣une␣chaîne␣de␣caractères␣:␣";
    cin >> s;

    cout << "Vous␣avez␣tapé␣:\n"
         << "(entier␣:␣" << a << ")\n"
         << "(flottant␣:␣" << f << ")\n"
         << "(chaîne␣:␣" << s << ")" << endl;


    return 0;
}
```

Listing 4.8 – Lecture d'une ligne en C++

```
//
// Entrée de données en pur C
//

#include <iostream>
#include <string>

using std::cin;
using std::cout;
using std::endl;
using std::string;

int main()
{
    string s;

    cout << "Entrez une phrase avec des espaces" << endl;
    getline(cin, s);
    cout << "Vous avez entré : (" << s << ")" << endl;


    return 0;
}
```

Listing 4.9 – `getchar.c` Entrée/sortie par caractères

```
/*
 * Ce programme compte le nombre de caractères en entrée
 */

#include <stdio.h>

int main()
{
    int c, nbchar = 0;


    while ((c = getchar()) != EOF) {
        nbchar++;
    }
    printf("nombre de caractères : %d\n", nbchar);
    return 0;
}
```

```
$ ./getchar < getchar.c
255
```

Cette ligne de commande redirige le fichier `getchar.c` vers l'entrée standard du programme `getchar`, L'avantage d'utiliser les entrées/sorties redirigées est qu'on n'a pas à gérer l'interface avec le système d'exploitation, et donc l'ouverture et la fermeture explicite de fichiers, les permissions, les erreurs, etc.

La fonction `getchar` et la redirection `<` permettent de lire un fichier. Pour réaliser une écriture, on utilise la fonction `putchar` et la redirection `>`

Voir les pages de manuel de `getchar` et `putchar` pour plus de détails.

Exercice 10 *Modifiez le programme du listing 4.9 pour compter le nombre de lignes et le nombre de mots dans le fichier d'entrée.*

Accès sans tampon

Les accès sans tampon se font au travers d'un numéro de fichier attribué par le système d'exploitation. Le principe est relativement simple.

l'appel `open` permet d'ouvrir un fichier et retourne un numéro. Ce numéro est ensuite utilisé par des appels `read` et/ou `write`. Une fois les manipulations sur le fichier terminées, on le ferme par l'appel `close`. Un exemple est donné en listing 4.10.

En passant, on note la façon de gérer les erreurs systèmes en C. Tous les appels systèmes retournent une valeur qui peut être interprétée comme une condition d'erreur (voir la page du manuel pour les spécificités). La fonction `perror` permet d'obtenir la raison éventuelle d'un échec.

Exercice 11 (Condition d'erreur) *Après avoir fait tourné l'exercice du listing 4.10, vérifiez le contenu du fichier produit. Modifiez les permissions sur ce fichier pour qu'une deuxième exécution du programme génère une erreur d'accès. Vérifiez le bon fonctionnement du programme (au sens où l'erreur est bien détectée).*

Les fonctions `read` et `write` ne sont pas limitées au chaînes de caractères, on peut lire et écrire par elle n'importe quelle structure C (mais en général pas les classes C++).

Exercice 12 (Lecture/écriture de nombres) *Modifiez l'exercice du listing 4.10 pour lire et écrire un tableau de 10 nombres entiers, sans utiliser de boucles.*

Accès avec tampon

Les accès avec tampon s'effectuent au travers d'une structure de fichier standard nommée `FILE`. On ouvre un fichier avec la fonction `fopen` et on le ferme avec `fclose`. On peut lire et écrire du texte dans des fichier grâce à une version spéciale des appels console : `fprintf` et `fscanf`. Ces accès permettent de lire et d'écrire dans des fichiers essentiellement comme à la console. Le premier argument de ces fonctions est la structure `FILE` de fichier ouverte. Le listing 4.11 est une illustration.

On peut aussi lire et écrire dans des fichiers avec tampons comme dans les fichiers sans tampon avec les appels `fread` et `fwrite`. Au lieu d'un numéro de fichier, ces appels utilisent une structure `FILE`.

Listing 4.10 – Lecture/écriture en C

```c
/*
 * Lecture/écriture dans des fichier par appels sans tampon
 * On note que les appels système demandent beaucoup de précautions.
 */
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main()
{
    int  file, n, s, l, r, error = 0;
    char fn[] = "monfichier.txt", p[] = "Une chaine de caracteres simples\n";
    char lect[100]; // pour lire

    // pas d'exceptions en C, on utilise un truc...
    do {
        // ouverture du fichier en lecture/écriture, crée si besoin est */
        if ((file = open(fn, O_RDWR | O_CREAT)) < 0)  {
            perror("Open n'a pas marché"); error = 1;
            break; // on ne fait pas la suite
        }
        l = strlen(p); // longueur de la chaîne
        // on écrit dans file la chaîne p sur toute sa longueur
        if ((n = write(file, p, l)) < 0) {
            perror("Write n'a pas marché"); error = 2;
            break;
        }
        if ((s = lseek(file, 0, SEEK_SET)) < 0) { // retourne au début du fichier
            perror("Seek n'a pas marché"); error = 3 ;
            break;
        }
        if ((r = read(file, lect, l)) < 0) { // on lit ce qu'on a écrit
            perror("Read n'a pas marché"); error = 4;
            break;
        }
        lect[r] = '\0'; // pour terminer la chaîne
        printf("On a écrit et lu dans le fichier %s la chaîne : %s\n", fn, lect);
        close(file); // fermeture du fichier
    } while (0); // on ne boucle jamais

    // error = zero si tout s'est bien passé
    return error;
}
```

Listing 4.11 – Lecture/écriture de fichiers en C

```c
/*
 * Ce programme compte le nombre de caractères en entrée
 */

#include <stdio.h>
#include <errno.h>
#include <math.h>

int main()
{
    char f[] = "monfichier.txt";
    FILE *fichier;
    double dval;
    int    nbread;

    fichier = fopen(f, "w+"); // ouverture en lecture/écriture
    if (fichier) {
        fprintf(fichier,
                "On écrit dans le fichier tout comme à la console."
                "Par exemple voici un nombre entier : %d, et une valeur connue %f\n",
                42, atan(1.0)*4.0);

        fseek(fichier, 0, SEEK_SET);
        nbread = fscanf(fichier, " \n %g", &dval);
        if (nbread == 1)
            printf("Nous avons lu la valeur %f dans le fichier %s\n", dval, f);
        fclose(fichier);

    } else {
        perror("Impossible d'ouvrir le fichier");
    }


    return 0;
}
```

Exercice 13 *Les appels avec tampons sont moins rudimentaires que les appels sans tampon. Ils permettent de mettre en œuvre des procédures de traitement de texte simples. Par exemple, soit le texte suivant :*

```
Nom, Informatique, Algorithmique, C++
Jean, 12, 7, 15
Camille, 13, 15, 12
Orie, 8, 5, 7,
Herb, 18, 14, 19
Bjarne, 17, 7, 17
Julie, 12, 11, 13
Marie, 15, 11, 5
Fabrice, 2, 7, 11
```

Il s'agit de notes de contrôle. Copiez ce texte dans un fichier, écrivez un programme qui calculera la moyenne de chaque élève, la moyenne de la classe pour chaque matière et la moyenne générale.

4.3 Fichiers en C++

En plus des appels C comme toujours utilisables en C++, C++ définit des classes de « stream » (flux) qui permettent de lire et d'écrire un peu comme avec `cin` et `cout`.

4.3.1 Fonctions de base

Voici un exemple en listing 4.12. Dans cet exemple on ouvre le fichier `exemple.txt` en écriture, et on y écrit une chaîne de caractères.

Listing 4.12 – Lecture/écriture de fichiers en C++

```
// entree-sorties de base en C++
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{
    // déclaration d'un stream de fichier en écriture
    ofstream myfile;
    // Ouverture du fichier
    myfile.open ("exemple.txt");
    // Ecriture sur le fichier
    myfile << "Ceci est un exemple"<< endl;
    // fermeture du fichier
    myfile.close();
    return 0;
}
```

La fonction-membre `open` admet un argument optionnel `mode`, c-à-d que au lieu d'écrire `myfile.open("toto")` on peut écrire `myfile.open("toto",` *mode*`)`, avec *mode* suivant le tableau suivant :

`ios::in`	Ouverture en lecture
`ios::out`	Ouverture en écriture
`ios::binary`	Ouverture en fichier binaire
`ios::ate`	(at end) Place la position de lecture-écriture en fin de fichier. Par défaut, sans cette option, la position de lecture est en début de fichier
`ios:app`	Ouverture en *append*, c-à-d en ajout. Toute écriture se fera à la fin du fichier s'il existe déjà. Ne fonctionne que sur les fichiers ouverts en écriture seule
`ios::trunc`	Ouverture en *truncate*, c-à-d en tronquement : toute fichier ouvert en écriture existant verra son contenu effacé avant la première écriture.

La signification d'un fichier binaire dépend des systèmes d'exploitation[1].

Ces drapeaux (*flags*) peuvent être précisés de manière multiple en les séparant par des caractères `|`. On aurait pu ouvrir notre fichier-exemple de la manière suivante :

```
ofstream myfile ("exemple.bin", ios::out | ios::app | ios::binary);
```

Dans ce cas le fichier `"exemple.bin"` est ouvert en écriture, en ajout et en binaire.

Le C++ définit trois types standard de *stream* (ou flux) de fichier :

`ifstream`	type de flux ouvert en lecture
`ofstream`	type de flux ouvert en écriture
`fstream`	type de flux ouvert en lecture-écriture

Exercice 14 (Concaténation) *Soit le fichier A suivant*

```
Ceci est un exemple simple
```

et le fichier B suivant

```
Ceci est un exemple
un tout petit peu moins simple
```

Écrire un programme qui lit le contenu des deux fichiers et écrit la concaténation des deux fichiers dans un troisième, appelé C par exemple.

[1]Sous Unix, c'est à dire par exemple Linux et Mac OS/X la notion de fichier binaire est la même que celle de fichier texte ordinaire. Sous Windows il y a une différence au niveau des terminaisons de ligne : en mode texte les terminaisons de ligne `"\n"` sont traduite par les deux caractères `<CR><LF>` dans les deux sens (en lecture et en écriture). En mode binaire il n'y a pas de traduction.

4.3.2 Vérifier qu'un fichier est ouvert

Une tentative d'ouverture infructueuse génère une exception, mais tous les programmes C++ ne gèrent pas les exceptions par défaut. Pour voir si un fichier a été bien ouvert, on peut utiliser la fonction-membre `is_open`, par exemple de la manière suivante :

```
if (myfile.is_open()) {
    /* on peut continuer */
}
```

4.3.3 Fermer un fichier

Avec `myfile.close()`.

4.3.4 Opérations sur fichiers texte

Les opérations C++ sur les fichiers textes correspondent approximativement à celles des fonctions `fprintf` et `fscanf` du C pur.

Écriture sur fichier

Un exemple est donné en figure 4.13. Notez l'ouverture du fichier à partir d'une chaîne de caractères C, l'établissement du flux en sortie, l'écriture proprement dite, la fermeture du fichier et le traitement d'erreur.

Listing 4.13 – Écriture par stream en C++

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{
  ofstream myfile ("exemple.txt");
  if (myfile.is_open())
  {
      myfile << "Une ligne." << endl;
      myfile << "Une autre ligne." << endl;
      myfile.close();
  }
  else cout << "Ouverture du fichier impossible";
  return 0;
}
```

Exercice 15 *Exercice d'écriture*

Lecture sur fichier

4.3.5 États des flux

Les flags d'état comme `eof()`.

4.3.6 Pointeurs de flux

Les fonctions membres `seek` et associées.

4.3.7 Fichiers binaires

Les fonction-membre `read` et `write`.

4.3.8 Synchronisation

Quand est-ce que l'état interne correspond à celui sur le disque ?

4.4 Chaînes de caractères

4.4.1 Chaînes de caractères en C

Les tableaux de char et leurs manipulations.

4.4.2 Chaînes de caractères en C++

La classe container `string`.

4.4.3 Arguments de la ligne de commande

Chapitre 5

Pointeurs et tableaux

Un pointeur est une variable spéciale qui permet d'accéder à l'adresse d'une case mémoire. Contrairement à la plupart des autres langages de programmation, C et C++ font une adéquation complète entre les tableaux et les pointeurs.

5.1 Pointeurs en C

5.1.1 Référence et déréférence

Sauf exception (registres), toute variable en C est associée à une adresse mémoire [1].
Dans le code suivant :

```
int a = 10; // variable entière
int *p = &a; // pointeur vers un entier, référence vers a

printf("Un entier : \%d, son adresse mémoire : %p\n"
       "la déréférence de ce dernier : %d\n", a, p, *p);
```

La variable `a` est un entier habituel. La variable `p` est un `pointeur` vers un entier, c'est à dire une adresse mémoire. L'opérateur *dans une déclaration* indique dans ce cas que `p` doit être compris comme une adresse. L'opérateur `&` permet de prendre l'adresse d'une variable ordinaire, et donc de l'affecter à un pointeur. L'opérateur `dans une expression` est l'opérateur de *déréférence* qui permet de lire la valeur indiquée par le pointeur.

À noter qu'un pointeur est normalement associé à un type, et donc en particulier à une taille. Un pointeur déréférencé de type `char` ne lit qu'un octet en mémoire, alors qu'un pointeur déréférencé de type `double` lit 8 octets d'un coup.

Conversion de type Il arrive assez souvent qu'on est besoin de changer le type d'un pointeur, par exemple pour permettre des changements de type difficiles.

[1] Les registres sont une partie de mémoire extrêmement rapide de l'unité centrale, nécessaire pour réaliser certaines opérations. Les registres sont nominatif (par exemple `AX, BX, ...`) et ne sont pas associés à une adresse mémoire. En revanche, si l'utilisateur exige l'accès à l'adresse d'une variable, alors le compilateur ne pourra pas mettre cette variable en registre, mais uniquement dans la mémoire centrale.

Exercice 16 (Types et déréférence) *Le code suivant permet de lire le contenu d'une variable double :*

```
#include <math.h>
#include <stdio.h>

int main()
{
        double d = M_PI; // valeur du nombre Pi
        int    i;
        char   *c;

        c = (char *)&d;
        for (i = 0 ; i < sizeof(double) ; ++i) {
                putchar(c[i]);
        }

        return 0;
}
```

Expliquez son fonctionnement, en particulier la conversion du pointeur de double vers un pointeur de char. Faites tourner l'exemple. Est-il informatif ? Modifiez l'exemple pour imprimer l'encodage du nombre π sous forme de nombre et non de caractère.

5.1.2 Pointeur vide

Il existe un pointeur spécial, le pointeur `void`, qui est non-typé. Un pointeur `void` peut pointer vers tous les types de données, on l'utilise donc volontiers dans les signatures de fonctions, comme par exemple dans la fonction standard `qsort`.

Exercice 17 *Construisez un exemple de programme C comportant un tableau de 10 nombres flottants aléatoires. Triez le tableau en ordre ascendant avec la fonction standard* `qsort`*.*

5.1.3 Pointeurs et tableaux

En C/C++, tableaux et pointeurs sont très similaires. Les pointeurs sont un peu des tableaux non initialisés, par exemple on peut utiliser presque toujours la syntaxe `int[]` au lieu de `int *`.

Un tableau initialisé est son propre pointeur.

A contrario, l'opérateur de déréférence est équivalent au premier élément d'un tableau

```
int tableau[10] = {1, 3, 5, 7, 11, 13, 17, 19, 23, 29};
int *p = tableau; // p pointe vers le tableau

printf("%d\n", *p); // résultat = 1
printf("%d\n", p[0]); // resultat = 1;
printf("%d\n", p[2]); // resultat = 5;
```

De manière similaires, une chaîne de caractère est un tableau de caractère. Le nom de la chaîne est un pointeur :

```
const char *texte = "un texte";

printf("%c\n", texte[2]); // résultat = ' ' (espace)
```

Exercice 18 *Écrivez un programme C qui accepte une chaîne de caractères en entrée et qui compte le nombre de caractères différents qui la composent. Par exemple la chaîne* `"un texte"` *comporte 6 caractères différents. On supposera les caractères codés sur 8 bits.*

5.1.4 Arithmétique de pointeurs

Ce qu'on appelle l'arithmétique de pointeur est simplement la manipulation de pointeurs avec des additions et des soustractions.

Pour accéder au contenu d'un tableau, on peut opérer de la façon suivante :

```
int tableau[10] = {1, 3, 5, 7, 11, 13, 17, 19, 23, 29};
int *p = tableau; // p pointe vers le tableau

printf("%d\n", p[4]); // resultat = 11
printf("%d\n", *(p+4)); // resultat = 11
```

Dans le second cas, on a explicitement déplacé le pointeur de 4 éléments vers la gauche puis déréférencé. On tombe sur le 5ème élément du tableaux (donc `tableau[4]`).

L'arithmétique de pointeur est très courante lors de la manipulation de tableaux pour faire des calculs.

Un exemple est donné en figure 5.1 :

L'exemple peut paraître compliqué mais est assez flexible.

Exercice 19 *Modifier l'exemple précédent pour réaliser une moyenne mobile sur 5 valeurs, puis en remplaçant la moyenne par un maximum mobile (dilatation).*

5.1.5 Pointeurs dans les arguments de fonctions

Comme vu dans l'exemple précédent, on utilise volontiers les pointeurs dans les fonctions pour éviter de passer des tableaux entiers aux fonctions, ce qui imposerait des copies onéreuses.

5.1.6 Allocation de mémoire dynamique

Jusqu'à présent nous n'avons utilisé que des tableaux de taille fixe, c'est-à-dire connue au moment de la compilation. En C/C++ il est possible de définir des tableaux de taille variable, définie au moment de l'exécution. On appelle ces tableaux « dynamiques ».

Pour se faire, on utilise les fonctions standard `malloc`, `calloc` et `free`.

Pour allouer un tableau de 100 entiers, on utilise `malloc` de la façon suivante :

```
int *p; // pointeur vers entiers
p = (int *) malloc(100 * sizeof(int));
```

Ensuite, `p` s'utilise comme un tableau normal.

Pour allouer un tableau rempli de zéros (ce qui est courant), on utilise la fonction `calloc`, de signature un peu différente :

Listing 5.1 – Moyenne mobile en C

```c
#include <stdio.h>

#define DATA_SIZE 100

// exemple de manipulation de pointeurs
// moyenne mobile avec fenêtre de taille variable
int moyenne_mobile(double *vectin, int window_size, double *vectout)
{
    double *p, *q, *r, *end, somme;
    int     i, j, nbpoints, error = 0;

    // filtrage ne marche que pour fenêtre centrée de taille impaire
    if ((window_size % 2) == 1) {
        // filtrage par manipulation de pointeurs
        p = vectin; q = vectout;
        end = p + DATA_SIZE;
        while (p != end) {
            somme = 0; nbpoints = 0;
            for (j = -window_size/2 ; j <= window_size/2 ; ++j) {
                r = p + j;
                // effets de bord
                if ((r >= p) && (r < end)) {
                    somme += *r;
                    nbpoints++;
                }
            }
            *q = (somme / nbpoints);
            ++p; ++q;
        }
    } else {
        error = 1;
    }
    return error;
}

int main()
{
    double values[DATA_SIZE], output[DATA_SIZE];
    int    i, error;

    // initialisation
    for (i = 0 ; i < DATA_SIZE ; ++i) {
        values[i] = i % 10; // données en dent de scie
    }

    // moyenne mobile par une fenêtre de 3 pixels
    if ((error = moyenne_mobile(values, 3, output)) == 0) {
        for (i = 0 ; i < DATA_SIZE ; ++i)
            printf("%.3g ", output[i]);
        printf("\n");
    } else {
        printf("Erreur de taille de fenêtre\n");
    }
    return error;
}
```

```
double *q; // pointeur vers doubles
q = (double *) calloc(100, sizeof(double));
```

Lorsqu'on a fini d'utiliser un tableau dynamique, on doit le *libérer* de façon à restituer la mémoire au système d'exploitation : on utilise pour cela la fonction `free`.

```
free(p); free(q);
```

La mémoire non libérée continue de s'accumuler jusqu'à la mort du processus.

5.2 Pointeurs et références en C++

En C++, on utilise l'opérateur `new` au lieu de `malloc` :
Pour allouer un seul élément :

```
int *p = new int;
```

La raison pour utiliser `new` est que cet opérateur est capable d'initialiser les classes de C++ alors que `malloc` ne le peut pas.
Pour allouer un tableau :

```
int *q = new int[100];
```

Pour libérer un seul élément alloué par `new` on utilise l'opérateur `delete`

```
delete p;
```

Pour libérer un *tableau*, on utilise l'opérateur `delete[]`.

```
delete[] q;
```

Il est très facile de faire des erreurs et de tenter de libérer un tableau avec un simple `delete`. Ça ne marchera pas comme prévu !

Exercice 20 (Tableaux de taille variable) *Recodez l'exemple du listing 5.1 en C++ en utilisant des tableaux de taille variable.*

5.3 Tableaux dynamiques à plus d'une dimension

On ne peut pas allouer directement de tableaux dynamiques à plus d'une dimension, ni en C, ni en C++. On peut cependant le faire indirectement de plusieurs manières, par exemple :

Listing 5.2 – pseudo-matrice en C

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    // simulation d'une matrice dimx x dimy
    int *matrix, dimx=3, dimy=4, x, y, i;

    matrix = (int *)malloc(dimx * dimy * sizeof(int));
    for (i = 0 , y = 0 ; y < dimy ; ++y)
        for (x = 0 ; y < dimx ; ++x)
            matrix[dimx * y + x] = i++;

    // affichage à faire

    free(matrix);
    return 0;
}
```

5.3.1 Simulation des dimensions avec un vecteur

Dans cette solution, on alloue un vecteur à une dimension, et on simule l'usage des dimensions supérieures, par exemple comme dans la figure 5.2. Dans celle-ci, on alloue un vecteur et on simule l'existence de dimensions supérieures par des manipulations d'indices, ici en 2-D. L'avantage de cette méthode est la simplicité d'allocation et de libération de la mémoire, mais la manipulation d'indice est quelque peu complexe et nécessite qu'on s'y habitue.

Exercice 21 (Affichage de pseudo-matrice) *Complétez le programme de la figure 5.2 pour afficher le contenu de la matrice sous la forme d'une matrice, colonne par colonne, c'est à dire de la façon suivante, dans le cas d'une matrice* 4×3 *:*

```
0  4  8
1  5  9
2  6 10
3  7 11
```

Exercice 22 (3-D) *Étendez l'exemple de l'exercice précédent pour un exemple de données 3-D, avec affichage plan par plan, c'est à dire, pour un parallélépipède de* $4 \times 3 \times 2$ *:*

```
 0  4  8
 1  5  9
 2  6 10
 3  7 11

12 16 20
13 17 21
14 18 22
15 19 23
```

Exercice 23 (Ajout d'une constante) *On veut ajouter une constante à tous les membres de la matrice (ou du parallélépipède). Quelle est la meilleure façon de s'y prendre et pourquoi ?*

5.3.2 Allocation par vecteur de vecteur

Avec cette méthode, on alloue des vecteurs de vecteurs comme dans l'exemple de la figure 5.3.

Listing 5.3 – vecteur de vecteurs en C

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    // simulation d'une matrice dimx x dimy
    int **matrix, dimx=3, dimy=4, x, y, i;

    // allocation
    matrix = (int **)malloc(dimy * sizeof(int*));
    for (y = 0 ; y < dimy ; ++y)
        matrix[y] = (int *)malloc(dimx * sizeof(int));

    // utilisation
    for (i = 0, y = 0 ; y < dimy ; ++y)
        for (x = 0 ; x < dimx ; ++x)
            matrix[y][x] = i++;

    // affichage à faire

    // libération
    for (y = 0 ; y < dimy ; ++y)
        free(matrix[y]);
    free(matrix);

    return 0;
}
```

Avec cette méthode, l'utilisation du tableau 2-D est très simple et intuitive mais l'allocation et la libération de la matrice n'est pas simple, et les données ne sont pas consécutives, ce qui peut poser des problèmes.

Exercice 24 (Affichage, 3-D et ajout d'une constante) *Reprenez les exercices de la section précédente et adaptez-les pour les vecteurs de vecteurs. Quelles sont vos conclusions ?*

Exercice 25 (Gestion d'erreurs) *Normalement la fonction* `malloc` *retourne* `NULL` *si elle ne peut plus allouer de mémoire dynamique (cas où elle est épuisée). Cela arrive en pratique lorsque l'on alloue des structures de grandes dimensions. Dans le cas de l'allocation de vecteur de vecteurs, on peut donc avoir un appel* `malloc` *qui échoue à un endroit arbitraire. Comment gérer cette condition ?*

5.3.3 Imposition de la structure n-dimensionnelle

Il existe une troisième voie qui résout la plupart des problèmes, mais en crée de nouveaux. Elle est illustrée en 5.4. Avec cette méthode, on alloue la matrice en deux temps, d'une part en allouant l'espace de données, puis une structure. On manipule la structure par arithmétique de pointeurs, ce qui impose une structure de tableau multi-dimensionel.

Avec cette méthode, les données sont consécutives, ce qui simplifie certaines opérations, l'allocation est plus simple que celle du vecteur de vecteurs et la manipulation du tableau est possible avec les opérations du langage. C'est donc un bon compromis, mais il y a un petit problème, illustré par les exercices suivants :

Exercice 26 (Reprise) *Reprendre tous les exercices de la section précédente : affichage, 3-D, ajout d'une constante et gestion d'erreur. Quelles sont vos conclusions ?*

Exercice 27 (Libération) *Comment doit-on libérer la mémoire allouée dans ce cas ?*

Exercice 28 (Performances) *Cet exercice est valide pour toutes les structures proposées, mais on l'exécute sur la dernière. Comparez le temps nécessaire pour remplir une matrice* 10×10*,* 100×100*,* 1000×1000 *puis* 10000×10000 *par des nombres consécutifs (0, 1, 2, etc). Remplissez la matrice d'abord par les colonnes, puis par les lignes. Constatez vous une différence ? Pouvez vous l'expliquer ?*

Listing 5.4 – Imposition de la structure de matrice en C

```
#include <stdlib.h>
#include <stdio.h>

void* allocation_matrix(int ncol, int nlign)
{
        int **matrix, *data;
        int   i, j;

        data = (int *)malloc(ncol * nlign * sizeof(int));

        if (data != NULL) {
                matrix = (int **)malloc(nlign * sizeof(int*));
                if (matrix != NULL) {
                        for (i = 0 ; i < nlign ; ++i) {
                                // arithmetique sur pointeur
                                matrix[i] = data + i * ncol;
                        }
                }
        }

        return matrix;
}


int main(int argc, char *argv[])
{
    int **matrix, dimx=3, dimy=4, i, x, y;

    // allocation
    matrix=allocation_matrix(dimx, dimy);

    // utilisation
    for (i = 0, y = 0 ; y < dimy ; ++y)
        for (x = 0 ; x < dimx ; ++x)
            matrix[y][x] = i++;

    // affichage ?

    // libÃ©ration ?

    return 0;
}
```

Chapitre 6

Classes

On voit bien dans la d'exemple assez simples somme toutes sur les tableaux multi-dimensionnels, qu'il est souvent délicat de manipuler une structure de données sans une série de fonctions qui y sont associées. En C++, on associe structures de données et opérations dans une *classe*. Avant d'introduire ce concept, on présente les structures de données en C qui sont plus primitives, mais qui sont le point de départ des classes.

6.1 Structures en C

En C pur, une structure `struct` est simplement la mise en commun de plusieurs éléments de données simples, afin de pouvoir les manipuler ensemble. Par exemple, on pourrait définir une matrice de la manière suivante :

```
struct matrix {
int **data;
int   dimx, dimy;
};
```

Cette structure contient un pointeur vers des données (non encore allouées !) et les dimensions associées. On est pas encore capable de manipuler ces éléments, faute

6.1.1 Définition de type par typedef

Dans la très grande majorité des cas on associe une structure avec la définition d'un nouveau type, qui se fait avec l'opérateur du langage `typedef`.

```
// définition du type « byte »
typedef unsigned char byte;
```

On définira presque toujours une structure avec un nouveau type :

```
// définition d'un type de structure
typedef struct matrix MyMatrix;
```

Souvent on définit un type et sa structure d'un seul coup :

```
// définition d'un type de structure
typedef struct matrix {
int **data;
int   dimx, dimy;
} MyMatrix;
```

Il y a plusieurs subtilité liées à la définition des structures, en particulier quand une structure utilise son propre type ou le type d'une autre structure. C'est extrèmement courant dans les structures de listes chaînées par exemple.

```
// définition d'un type de structure
typedef struct intliste {
int  data;
struct intlist *next;
} MyMatrix;
```

Ici on ne pourrait pas taper `MyMatrix *next`, car le type n'est pas encore défini quand on en a besoin. On doit donc utiliser le nom complet de la structure et non son type associé.

Lorsque l'on utilise deux structures (ou plus) associées, il y a un problème d'œuf et de poule :

```
// deux structures liées
typedef struct A {
      struct B data;
} MyA;

typedef struct B {
      struct A data2;
} MyB;
```

Pour s'en tirer, il faut définir la structure B d'abord de manière incomplète :

```
// type incomplet, voir plus loin.
struct B;

typedef struct A {
      struct B data;
} MyA;

typedef struct B {
      struct A data2;
} MyB;
```

6.1.2 Utilisation des structures en C

Pour être utile on doit pouvoir accéder aux membres d'une structure.

Syntaxe ordinaire

Les structures en C contiennent uniquement des données, qu'on accède au moyen de la syntaxe « . », par exemple :

```
// définition d'un type de structure
typedef struct matrix {
int **data;
int   dimx, dimy;
} MyMatrix;

MyMatrix m;

m.dimx = m.dimy = 3; // on peut faire ça, initialiser deux variables d'un coup
m.data = (int **)malloc(mdimy * sizeof(int*));
...
```

Les variables accédées de cette façon, par exemple `m.dimx` ont le même statut qu'une variable ordinaire (ici un entier).

exercice

Syntaxe pour pointeurs

Dans beaucoup de cas, on associe structures et pointeurs.

6.2 Classes en C++

Royaume exclusif du C++

6.2.1 Définition

6.2.2 Méthodes

6.2.3 Héritage

6.2.4 Constructeurs et destructeurs

Chapitre 7

C++ avancé

Dans ce chapitre, on va passer vite sur la plupart des sujets, car on pourrait y passer des mois. Si vous en avez besoin il vous faudra consulter une référence du C++ adéquate.

7.1 Les formes d'héritage étranges

Multiple, private ou protected.
Dans le cadre du cours, on ne considère que l'héritage par défaut : simple et public.

7.2 Exceptions

Le langage C++ a des exceptions comme en Java. Cela permet d'écrire des programmes plus lisibles plutôt que de tester sans arrêt des conditions d'erreurs, genre `if (malloc(...) == NULL)`. La library standard C++ utilise les exceptions de manière libérale, mais on ne les utilisera pas dans le cadre de ce cours.

La raison principale étant l'interaction délicate avec les constructeurs et destructeurs.

7.3 RTTI

Cette caractéristique du langage permet d'obtenir des informations limitées sur les classes définies dans un programme. On peut la plupart du temps s'en passer.

7.4 Opérateurs

Le C++ permet de redéfinir les opérateurs de base dans certains contextes. Par exemple on peut parfaitement avoir la syntaxe suivante :

```
MyMatrix A, B, C;

C = A + B;
```

Avec `MyMatrix` une classe définie par nous-même, et `+` l'opérateur d'addition ! Cela peut rendre un programme très lisible. On peut aussi s'en passer facilement, par exemple par

```
MyMatrix A, B, C;

C = MatrixSum(A,B);
```

ce qui n'est pas beaucoup moins lisible. On ne peut pas redéfinir les opérateurs des types de base, heureusement.

7.5 Friends

Les classes et les fonctions `friend` permettent de contourner les protections liées au classes. On en a surtout besoin dans le cadre de la redéfinition des opérateurs pour ses propres classes.

7.6 Templates

Y compris la STL.

7.7 Lien entre C et C++

Au niveau de l'éditeur de liens.

Chapitre 8

Idiomes du C++

C++ est un langage difficile à maîtriser. Pour éviter les erreurs les plus difficiles à déboguer, on utilise le plus souvent des « idiomes », ce sont des constructions qui fonctionnent bien et que tout développeur sérieux doit connaître.

8.1 Ressource allocation is initialisation

Abbréviée en RAII.

Chapitre 9

Systèmes de développement

9.1 Système UNIX/GNU

éditeur (vi, emacs, etc), gcc, make, shell, gdb, valgrind, etc.

9.2 Le débogueur GDB

Tout le monde, même Linus Thorvalds, écrit des programmes avec des bogues (bugs en anglais). Beaucoup de gens confrontés avec le résultat de ces bogues cherchent à en trouver la cause en réfléchissant, en regardant leur code fixement, en ajoutant des directive `printf` ou `cout`, mais il existe des moyens plus puissants et plus sophistiqués.

La première ligne de défense contre les bogues est de ne pas en écrire, certes, mais c'est une partie perdue d'avance dès que le moindre programme dépasse la dizaine de lignes. La deuxième ligne consiste à utiliser un débogueur symbolique, c'est à dire un programme qui permet l'exécution pas-à-pas, l'arrêt sur condition, l'examen des variables, etc.

Le programme libre le plus répandu pour déboguer s'appelle `GDB`, c'est tout simplement le GNU débogueur. La suite de cette section est une introduction à ce programme. `GDB` est complexe et puissant mais relativement facile d'accès. Les fonctions avancées peuvent être introduites au rythme de l'utilisateur et en fonction de ses besoins.

`GDB` peut être utilisé pour déboguer tous les langages supportés par `GCC`, donc le C et le C++ en font partie mais ce ne sont pas les seuls, en particulier les version GNU de Objective-C, FORTRAN et ADA sont compatibles avec `GDB`.

9.2.1 Utiliser GDB

Dans cette section nous explorons les premières fonctions de `GDB` parmi les plus importantes. Nous partons du programme de la figure 9.1

Nous appellerons cet exemple `gdb1.c`.

Avant de démarrer

Il faut d'abord compiler votre programme. Afin que `GDB` puisse retrouver les numéros de lignes, les classes, fonctions et les variables dans votre programme il faut y adjoindre les informations de débogage.

Listing 9.1 – Exemple de programme à déboguer

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

double afunction(double arg1)
{
  double ret;

  ret = M_PI * arg1 * arg1;
  return ret;
}

int main(int argc, char *argv[])
{
  int i;
  double rad;

  printf("Hello, %s, thanks for using %s\n", getenv("USER"), argv[0]);

  for (i = 1 ; i < argc ; i++) {
    rad = atof(argv[i]);
    printf("area of disk of radius %5.3g is: %5.3g\n", rad, afunction(rad));
  }
  return 0;
}
```

Avec le compilateur `gcc` ou `c++`, il faut ajouter l'option `-g` au moment de la compilation et de l'édition de lien. Pour compiler le programme précédent, par exemple, on peut utiliser la commande suivante :

```
% gcc -g -o gdb1 gdb1.c
```

Notez que le `makefile` livré avec l'archive de tous les programmes de ce poly ajoute par défaut l'option `-g`. C'est un bon réflexe. Comme nous le verrons plus tard, ces informations de débogage ne ralentissent pas le programme, et en plus de permettre le débogage standard, elles permettent d'effectuer un *post-mortem* même après une erreur fatale.

Note : Avec beaucoup de compilateurs, les options d'optimisation et de débogage ne sont pas compatibles, mais avec `gcc` et `g++` c'est tout de même possible. Cependant, si on débogue un programme optimisé, le flot du programme peut ne pas être linéaire, et certaines variables peuvent ne pas être accessibles (si elles sont en registres par exemple). En phase de développement il ne vaut mieux pas utiliser les deux options en même temps.

9.2.2 Faire tourner `gdb1` sous GDB

Pour démarrer notre exemple sous le contrôle de `GDB`, tapez

```
% gdb gdb1
```

Vous devriez voir quelque chose comme[1].

```
147 talbot <notes> % gdb gdb1
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.15.1 (i586-unknown-linux),
Copyright 1995 Free Software Foundation, Inc...
(gdb)
```

Vous êtes maintenant à l'invite de `GDB`.

Obtenir de l'aide

Et maintenant ? A l'aide !

```
(gdb) help
List of classes of commands:

running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain points
```

[1] Eh oui la première version de cette partie du poly date d'environ 1996

```
files -- Specifying and examining files
status -- Status inquiries
support -- Support facilities
user-defined -- User-defined commands
aliases -- Aliases of other commands
obscure -- Obscure features
internals -- Maintenance commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

En spécifiant des commandes plus précises on a aussi de l'aide plus précise, exemple :

```
  (gdb) help running
... (long page)...
```

Et ainsi de suite

Se déplacer dans le code

Viewing your code En premier, vous pouvez voir où vous êtes dans le code avec la commande suivante :

```
  (gdb) list
```

Vous obtenez :

```
  (gdb) list
9         ret = PI * arg1 * arg1;
10        return ret;
11      }
12
13      int main(int argc, char *argv[])
14      {
15        int i;
16        double rad;
17
18        printf("Hello, %s, thanks for using %s\n", getenv("USER"), argv[0]);
```

Note :

- La plupart des commandes acceptent des arguments. On peut lister la ligne 20 en tapant `list 20` (essayez !).
- Si vous tapez sur `<Enter>` (retour chariot) sans entrer de commande, la dernière commande est répétée. C'est utile en conjonction avec la commande `list` qui reprend toujours de là où elle s'était arrêtée, et non du début. On peut donc faire défiler le code facilement en tapant `list` une fois, puis plusieurs fois la touche `<Enter>`.

– Vous pouvez rappeler l'historique des commandes que vous avez tapé avec les flèches de défilement, et même éditer vos commandes antérieures. Vous pouvez même faire une recherche dans les commandes antérieures avec la combinaison de clés `<Command> <R>` et bien d'autre choses[2]
– Vous pouvez abréger vos commandes, par exemple pour lister il suffit de taper `l` et non forcément `list` tout entier (essayez !).

Running the program Tapez simplement `run` ou encore simplement `r`. Si une bogue est présente dans le code qui va arrêter le programme avec un message d'erreur (par exemple bus error, segmentation fault, etc), l'execution dans `GDB` s'arrête au moment du bug et vous pouvez voir où l'erreur a eu lieu.

On peut spécifier les arguments du programme en les entrant après la commande `run`, comme ceci :

```
  (gdb) r 1 3 4
Starting program: /opt/home/talbot/text/TeX/notes/gdb1 1 2 3

Hello, Hugues Talbot, thanks for using exmp
area of disk of radius     1 is:  3.14
area of disk of radius     2 is:  12.6
area of disk of radius     3 is:  28.3

Program exited normally.
```

Ici le programme n'a pas de bogue visible, et s'arrête normalement.

Spécifier un point d'arrêt Pour l'instant rien d'extraordinaire. Les choses sont plus intéressantes dès lors qu'on peut arrêter un programme avant la fin. Pour cela il faut spécifier un *point d'arrêt* (breakpoint en anglais).

Ici, on spécifie un point d'arrêt en une ligne particulière :

```
 (gdb) break 18
Breakpoint 1 at 0x8000506: file gdb1.c, line 18.
 (gdb) r
Starting program: /opt/home/talbot/text/TeX/notes/gdb1

Breakpoint 1, main (argc=1, argv=0xbffff6a4) at gdb1.c:18
18        printf("Hello, %d, thanks for using %s\n", getenv("USER"), argv[1]);
(gdb)
```

Le programme s'est arrêté *juste avant* l'exécution de la ligne 18 et vous attend !

Notes :

[2]Il s'agit des commandes de l'éditeur GNU EMACS par défaut

- Il y a plusieurs types de points d'arrêt, regardez `help breakpoints`. Ici on ne décrit que les principes les plus basiques. Il est utile de savoir tout de même qu'on peut demander à `GDB` de ne s'arrêter que sous certaines conditions : par exemple lorsqu'une variable atteint une certaine valeur. C'est utile pour déboguer un programme pour lequel la bogue n'apparaît qu'à la 10 000^{e} itération d'une boucle !
- Vous pouvez lister les points d'arrêt avec `info breakpoints`, exemple

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00001ec6 in main at gdb1.c:18
        breakpoint already hit 1 time
```

- Vous pouvez temporairement activer ou désactiver un point d'arrêt avec `enable` et `disable` suivi du numéro de point d'arrêt. Exemple :

```
(gdb) disable 1
(gdb) r
Starting program: /Users/talbot/Teaching/ISBS/CXX/Textbook/isbs_cxx/gdb1
Hello, talbot, thanks for using /Users/talbot/Teaching/ISBS/CXX/Textbook/isbs_cxx/gdb1

Program exited normally.
(gdb) enable 1
(gdb) r
Starting program: /Users/talbot/Teaching/ISBS/CXX/Textbook/isbs_cxx/gdb1

Breakpoint 1, main (argc=1, argv=0xbfffe2f0) at gdb1.c:18
18        printf("Hello, %s, thanks for using %s\n", getenv("USER"), argv[0]);
```

  Ici nous avons successivement désactivé puis réactivé le point d'arrêt précédent.
- Vous pouvez effacer un point d'arrêt de manière permanente avec `delete`.
- Finalement vous pouvez mettre des points d'arrêt non seulement sur des numéros de lignes, mais sur des noms de fonctions (ou de méthode en C++). `GDB` s'arrêtera à l'entrée de la fonction :

```
(gdb) disable 1
(gdb) break afunction
Breakpoint 2 at 0x1e8d: file gdb1.c, line 9.
(gdb) r 1
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /Users/talbot/Teaching/ISBS/CXX/Textbook/isbs_cxx/gdb1 1
Hello, talbot, thanks for using /Users/talbot/Teaching/ISBS/CXX/Textbook/isbs_cxx/gdb1

Breakpoint 2, afunction (arg1=1) at gdb1.c:9
9         ret = M_PI * arg1 * arg1;
(gdb)
```

  Ici nous désactivons le point (1), puis nous mettons un point (2) à l'entrée de la fonction `afunction`, puis nous démarrons le programme avec l'argument "1". `GDB` arrête le programme à l'entrée de `afunction`.

Exercice 29 (Manipulation de `GDB`) *Reprenez le listing au début, désactivez le 2^{e} point d'arrêt, puis redémarrez le programme avec les arguments* `1 2 3`.

Exécution ligne à ligne Pour exécuter un programme pas à pas, tapez `next` (ou `n`).

```
 (gdb) n
Hello, talbot, thanks for using gdb1
20        for (i = 1 ; i < argc ; i++) {
```

Continuons jusqu'à la ligne suivante et ainsi de suite jusqu'à la ligne 22. Cette fois utilisons une commande différente :

```
20        for (i = 1 ; i < argc ; i++) {
(gdb) n
21          rad = atof(argv[i]);
(gdb) n
22          printf("area of disk of radius %5.3g is: %5.3g\n", rad, afunction(rad));
(gdb) step
afunction (arg1=12) at gdb1:9
9         ret = PI * arg1 * arg1;
```

En tapant `step` (ou `s`), au lieu de passer à la ligne 23, nous sommes allé à la fonction `afunction`. La commande `step` analyse si la ligne suivante contient un appel de fonction, et si oui, elle s'arrête au début de cette fonction. Sinon elle se comporte comme `next`.

Il est à noter que cette commande ne s'arrête dans une fonction que si `GDB` en connaît les sources. Ainsi, à la ligne 21 il y a aussi eu un appel de fonction (`atoi`), mais comme nous ne disposons pas des sources de cette fonction, sur cette ligne `step` se comporterait comme `next`.

Nous pouvons continuer à avancer dans le programme par `step` ou `next`. A l'intérieur d'une fonction, on peut revenir à l'appelant par la commande `finish`, qui termine l'appel et retourne à l'endroit du `step` ayant causé l'appel de fonction.

Inspecter des variables Un point important est de pouvoir lire le contenu des variables en cours d'exécution. `GDB` permet cette opération. En combinaison avec les points d'arrêt cette fonctionnalité permet de déboguer la plupart des programmes.

La commande pour lire une variable est la commande `print` (ou `p`).

En ligne 9, dans `afunction`, on peut taper :

```
(gdb) s
afunction (arg1=1) at gdb1.c:9
9         ret = PI * arg1 * arg1;
(gdb) p arg1
$4 = 1
```

Ici le résultat est que la variable `arg1` contient la valeur 1.

Notes :

– `print` permet de connaître la valeur des variables, mais bien plus encore, on peut ainsi connaître le résultat d'appels de fonctions ou d'expressions. Par exemple :

```
(gdb) print (char *)getenv("USER")
$14 = 0xbffff91f "talbot"
(gdb) print 2+2
$15 = 4
(gdb) print (double)sqrt((double)2)
$6 = 1.4142135623730951
```

- Si on souhaite suivre la valeur d'une variable au cours d'une execution on peut aussi utiliser la fonction `display`. Cette fonction causera l'affichage de la valeur d'une variable ou expression après chaque commande.
- On peut modifier le contenu d'une variable par la commande `set variable`, exemple :

```
(gdb) set variable i = 10
(gdb) p i
$17 = 10
```

Exercice 30 (Suivi d'une variable) *Mettez un point d'arrêt à la ligne 20. Recommencez l'exécution du programme avec les arguments "1 2 3 4", et inspectez la valeur de la variable* `i` *à l'aide de la fonction* `display`*. Commentez les valeurs de* `i` *au fur et a mesure de l'exécution jusqu'à la sortie de la boucle.*

Recommencer l'exécution Pour reprendre l'exécution jusqu'au prochain point d'arrêt ou à la fin du programme, utilisez la commande `continue`.

Autres aspects

Voici un condensé d'appels de fonctions utiles.

La pile d'appel Si `GDB` s'arrête après une erreur, il est difficile de savoir *a priori* où nous sommes. La fonction `backtrace` (ou `bt`) permet d'afficher la pile des appels de fonctions (qui a appelé qui, etc depuis la fonction `main` pour nous retrouver où nous sommes). Les fonction `up` et `down` permettent de monter et descendre dans la pile d'appels.

Cette fonctionnalité est indispensable en cas d'appels récursifs.

Exemple

```
(gdb) break afunction
(gdb) r 1 2 3 4
afunction (arg1=1) at gdb1.c:9
9         ret = PI * arg1 * arg1;
(gdb) bt
#0  afunction (arg1=1) at gdb1.c:9
#1  0x00001f1d in main (argc=6, argv=0xbfffe2cc) at gdb1.c:22
(gdb) up
#1  0x8000569 in main (argc=6, argv=0xbfffe2cc) at gdb1.c:22
22          printf("area of disk of radius %5.3g is: %5.3g\n", rad, afunction(rad));
(gdb) down
#0  afunction (arg1=1) at gdb1.c:9
9         ret = PI * arg1 * arg1;
(gdb)
```

Cycle Compilation/débogage

En général, on utilise le débogueur en cycles exécution/débogage/édition/compilation. Il est possible de recompiler (avec `make`) sans quitter `GDB`. Vous pouvez taper `make` avec ses arguments comme une commande de `GDB` ou plutôt comme à la console, puis reprendre l'exécution sous débogueur directement. `GDB` se rendra compte qu'un changement a eu lieu et rechargera les symboles, les points d'arrêts, etc.

Listing 9.2 – Exemple de programme à boucle infinie

```
#include <stdio.h>
#include <ctype.h>

int main(int argc, char **argv)
{
    char c;

    c = fgetc(stdin);
    while(c != EOF){

        if(isalnum(c))
            printf("%c", c);
        else
            c = fgetc(stdin);
    }

    return 0;
}
```

Exercice 31 (Débogage d'un programme avec boucle infinie) *le programme de la figure 9.2 a une bogue évidente. Compilez le et exécutez le :*

```
% make gdb2
% ./gdb2
toto
ttttttttttttttttttttttttttttttttttttttttt

^C
```

Interrompez avec `<Control><C>`*. Bon, il y a un souci. Faites tourner le programme sous* `GDB` *et trouvez la nature de la bogue (*`<Control><C>` *fonctionne sous* `GDB` *aussi !).*

Exercice 32 (Débogage d'un programme avec un segfault) *Le programme de la figure 9.3 se termine avec une faute de segmentation. Trouvez en la raison avec* `GDB`*.*

9.2.3 GDB et le débogage mémoire

Les erreurs mémoires sont parmi les plus difficiles à déboguer. On doit souvent se servir de programmes ou bibliothèques auxiliaires.

Listing 9.3 – Exemple de programme qui segfault

```
#include <stdio.h>
#include <stdlib.h>

// programme pour lire l'entrÃ©e standard et l'afficher a
// l'ecran. Mais il y a un probleme...


#define BEAUCOUP 10000 // Une assez grande quantitÃ© de mÃ©moire
char *buff;

int main(int argc, char **argv)
{
    char *buf = (char *)malloc(BEAUCOUP * sizeof(char));

    fgets(buff, 1024, stdin); // on lit sur la ligne de commande
    printf("%s\n", buf); // on ecrit sur la sortie standard

    return 1;
}
```

9.2.4 Debugging libraries

GDB can perfectly debug programs with incomplete debugging information. If you are trying to debug a dynamically loaded library in Splus, for example, you should be able to run Splus under gdb, start Splus with the `r` command, load your function (compiled with debugging on). Then break into GDB by typing Control-C. You can then access your function, set breakpoints, etc. You may have to fiddle with the `dir` command.

9.2.5 Front-ends to gdb

There are at least two good front-ends to gdb : `xxgdb` which is X11-based, and... Emacs.

- `xxgdb` is started in the same way as GDB. You get a prettier display of the source code, buttons for the major functions, plus a few whistles. A neat GUI.
- `GDB-mode` in Emacs is started with `M-x gdb`. It then asks you for a program to debug. There is a quite reasonable interface with the mouse, etc.

9.2.6 More information

- On the web :
  `http://www-oss.fnal.gov:8000/mans/gdb/gdb_toc.html`
  Has the gdb documentation.
- On at least 'fire' :
  `info gdb`
  Has the same information in info format. Run `info` preferably within Emacs with `M-x info`.
- Within GDB :
  Don't forget the help system !

9.2.7 Summary of commands

Run [args]	Run the program being debugged
List [line, function name, ...]	List the source code
Break [line, function name, ...]	Breakpoiht set at the given location
Next	Next line of code. Skip over function calls
Step	Next line of code. Enter functions
Finish	Finish a function's execution and stops at caller
Continue	Continue execution
Up	Go to the caller function
Down	Go back to the function called
Info [breakpoints,...]	All sorts of information (on current breakpoints...)
Disable [breakpoint number]	Disable the given breakpoint
Enable [breakpoint number]	Re-enable the given breakpoint
Delete [breakpoint number]	Delete (permanently) the given breakpoint
Directory [Path]	Add the given path to GDB's path for looking for sources if empty, reset the Path (confirmation required
Set variable [X]=[VALUE]	Set variable X at value VALUE
Print [Variable, expression]	Print variable, execute function and show the result
Help [Topic]	Get some help

9.3 Système Windows

IDE dont visual studio.

Chapitre 10

Conseils et conclusion

10.1 Sites web

10.2 Livres

10.3 Bibliothèques

10.4 C ou C++ ?

Vaut-il mieux programmer en C ou en C++ ? Impossible de répondre en toute généralité, mais les conseils suivants s'appliquent :

10.4.1 Complexité

Le langage C peut paraître compliqué et trompeur, mais peut être maîtrisé assez rapidement, avec quelques mois de pratique et la lecture assidue d'un seul livre en français. Le C++ demande des années de pratique pour maîtriser raisonnablement, contient des pièges redoutables et subtils et exige la lecture de plusieurs ouvrages (au moins 3) non tous traduits.

Ceci dit, le C++ peut s'étudier petit à petit et sa pratique est très formatrice, car elle permet d'aborder presque toutes les grandes façons de programmer de manière cohérente :

- Programmation impérative ;
- orientée-objet (classes et héritage) ;
- générique (templates) ;
- fonctionnelle (similaire à LISP avec récursion) ;
- par contrat ;
- défensive/offensive ;
- par « patrons » (design patterns) ;
- etc...

De plus en C++ toutes ces façons de programmer fonctionnent naturellement ensemble. Du fait de son énorme bibliothèque et de son extension du C, C++ est sans conteste **le** langage le plus puissant au monde, même s'il n'est pas le plus élégant.

Finalement, il est possible de programmer de façon plus sûre en C++ grâce aux classes spéciales de la librarie ISO (containers, etc). On peut programmer efficacement en C++ sans utiliser de pointeurs.

10.5 Références web

Sur le site suivant, on trouve de bonnes références en-ligne sur le C++
`http://docs.mandragor.org/files/Programming_languages/Cpp/`
Voir en particulier le cours complet en français de Christian Casteyde et le cours plus avancé en anglais de Bruce Eckel (thinking in C++).

Annexe A

Solution des exercices

A.1 Dans chapitre Introduction

Solution 1 (Erreur logique) *Quelle est l'erreur dans le code du listing 2.1 ?*

L'indentation est incorrecte.

A.2 Dans chapitre Bases