LES GUIDES DE
GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE
N°70

France METRO : 12,90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ cad — MAR : 130 MAD

LANGAGE C

LE GUIDE POUR MIEUX DÉVELOPPER EN C SOUS LINUX

LES CONSEILS D'EXPERTS POUR MIEUX DÉVELOPPER ET ÉTENDRE VOS CONNAISSANCES DU C

Introduction & Rappels
Ce que vous devez savoir des bases et principes du langage C

La pratique du C
Les techniques avancées pour parfaire votre maîtrise du C : macros, duff's device, co-routines, etc.

Bibliothèques & Toolkits
Utilisation des libs graphiques EFL, du toolkit GTK+ et du langage Python dans vos codes C

Autour du développement
Internationalisation des programmes et interfaces multilingues grâce à GNU Gettext

Édité par Les Éditions Diamond
L 15066 - 70 H - F: 12,90 € - RD
boutique.ed-diamond.com

Retrouvez toutes nos publications

LES ÉDITIONS DIAMOND

sur **boutique.ed-diamond.com**

GNU/Linux Magazine Hors-Série
est édité par **Les Éditions Diamond**

B.P. 20142 / 67603 Sélestat Cedex

Tél. : 03 67 10 00 20 / **Fax :** 03 67 10 00 21

E-mail : cial@ed-diamond.com
lecteurs@gnulinuxmag.com

Service commercial : abo@gnulinuxmag.com

Sites : www.gnulinuxmag.com
boutique.ed-diamond.com

Directeur de publication : Arnaud Metzler

Rédacteur en chef : Denis Bodor

Conception graphique : Kathrin Scali

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France :
(uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04

Service des ventes :
Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : A parution

N° ISSN : 0183-0864

Commission Paritaire : K78 976

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros

Les articles non signés contenus dans ce numéro ont été rédigés par les membres de l'équipe rédactionnelle des Éditions Diamond.

Membre April
www.april.org

PRÉFACE

« The power of assembly language and the convenience of... assembly language »

C'est en ces mots que le regretté *Dennis Ritchie* a un jour décrit le langage qu'il a lui-même créé en 1972 avec, par la suite, la participation de *Ken Thompson* et *Brian Kernighan* que l'on peut aisément qualifier, tous trois, comme étant également les pères d'UNIX. Le C, ce langage de programmation impérative, procédurale et structurée est, ni plus, ni moins, pour le monde de l'informatique, ce qu'est la roue pour le monde moderne.

En créant ce langage, Ritchie, Kernighan et Thompson ont littéralement, et au sens le plus strict du terme, changé le monde. Le C est aujourd'hui massivement utilisé et une quantité incroyable de langages de plus haut niveau se sont inspirés de sa syntaxe ou ont été influencés de manière importante par celle-ci. Nombreux sont ces langages qui sont, eux-mêmes, implémentés en C. De manière assez amusante, le C est ainsi de nos jours vu comme un langage de bas niveau (« bas », car proche du matériel), alors qu'il était considéré de haut niveau pendant longtemps (le bas niveau était l'assembleur).

Mais un langage n'est pas grand-chose sans compilateur associé et le projet GNU n'est certainement pas étranger à la popularité actuelle du C. L'ensemble de compilateurs GCC (pour *GNU Compiler Collection*) a été conçu initialement pour offrir une alternative en logiciel libre aux compilateurs C des systèmes UNIX. Depuis, GCC est devenu disponible pour un nombre incroyable de systèmes d'exploitation et est, pour certains d'entre eux, le compilateur intégré par défaut. La disponibilité de GCC en début des années 90 a également permis à un certain étudiant finlandais, Linus Torvalds, de débuter le développement d'un noyau de système d'exploitation entièrement écrit en C : Linux.

Vous l'aurez compris, le C, plus que tout autre langage est omniprésent, en particulier si vous êtes utilisateur de GNU/Linux. Noyau, environnement graphique X.org, toolkit GTK+, serveur HTTP Apache, SGBD MySQL, éditeur Vim... Lorsque vous utilisez GNU/Linux, vous ne pouvez faire autrement que d'utiliser quelque chose écrit en C. Il en va de même, directement ou indirectement, pour les autres OS, votre smartphone, votre tablette, votre baladeur MP3, votre console de jeu...

En développant en C, vous utilisez non seulement un des plus fantastiques langages de programmation qui soit, mais faites également partie de cette histoire, initiée par une poignée d'hommes. Comme l'a si bien dit Brian Kernighan, paraphrasant Isaac Newton à propos du fait de se tenir sur les épaules des géants, nous nous tenons tous sur celles de Dennis Ritchie...

La rédaction

Sommaire

GNU/Linux Magazine
Hors-Série
N°70

LANGAGE C

1 INTRODUCTION & RAPPELS

2 LA PRATIQUE DU C

3

BIBLIOTHÈQUES & TOOLKITS

4

AUTRES LANGAGES

5

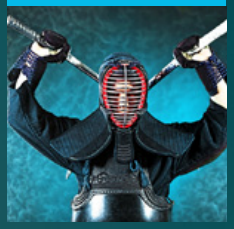
AUTOUR DU DÉVELOPPEMENT

1

1 INTRODUCTION & RAPPELS

À découvrir dans cette partie...

1.1 Introduction au langage C



Le langage C fut considéré comme révolutionnaire dès sa sortie, au début des années 70. Cet article d'introduction revient sur la naissance de ce langage de programmation, majoritairement utilisé aujourd'hui, et rappelle les principes essentiels qui sont à la base de ce langage.

1 INTRODUCTION & RAPPELS

INTRODUCTION AU LANGAGE C

Damien Balima - Ingénieur spécialisé en spécialités spécifiques

Découvrez le langage de référence de la programmation impérative.

1. INTRODUCTION

Le langage C fut un langage révolutionnaire à sa sortie, au début des années 1970. Les programmeurs purent coder dans un langage compréhensible, tout en ayant à disposition des fonctions pour les opérations courantes et la gestion de la mémoire, lequel fut popularisé avec le système Unix, premier système à intégrer le langage C. De nos jours, il reste encore le langage le plus utilisé au monde à la fois pour sa simplicité, sa rapidité et sa robustesse, et cela malgré la pléthore de langages concurrents (on en dénombre pas moins de 600, sans compter les langages spécifiques à un produit). **[1, 2]**

Cependant, malgré son apparente simplicité, le langage C est également subtil et contient quelques pièges à éviter. En effet, l'accès aux zones mémoires physiques de la machine demande une certaine rigueur durant les manipulations pour éviter d'empiéter sur la mémoire du processus voisin. De plus, les pointeurs – ces couteaux suisses du langage C – doivent être utilisés à bon escient : derrière un pointeur se trouve l'adresse d'une zone mémoire, que l'on ait ou non initialisé le pointeur. En appliquant ces quelques précautions, on appréciera de plus en plus, avec l'expérience, ce langage concis, précis et renommé.

Dans la suite de cet article, après s'être intéressé à l'histoire du C, on abordera ce langage par la pratique, avec quelques exemples illustrant les principaux éléments du langage C.

2. BRÈVE D'HISTOIRE

Le langage C remonte au début des années 70. À cette époque, l'informatique était déjà un secteur industriel productif, bien que réservé aux grandes entreprises. Les ordinateurs commercialisés étaient d'imposantes machines de plusieurs tonnes, des *mainframes*, le plus souvent des IBM de la série 360, à lecteur de cartes perforées, stockage sur bandes magnétiques, et systèmes d'exploitation propriétaires OS/MFT, OS/MVT (l'ancêtre du système MVS), ou DOS/360 (à ne pas confondre avec le x86-DOS des années 80). Au niveau des gestionnaires de fichiers, le progiciel Mark IV de la société *Informatics* se vendait à 30 000$ la licence.

Cependant, une nouvelle génération d'ordinateurs, plus petits et moins coûteux, les *mini-ordinateurs*, commençait à prendre essor, surtout dans le milieu universitaire et scientifique, en particulier les mini-ordinateurs System/3 (IBM, série 5400), PDP-7 et PDP-8 (DEC). Les langages de programmation les plus utilisés furent le Fortran (*Formula Translating System*, de 1957), le Cobol (*Common Business Oriented Langage*, de 1960) et les langages assembleurs, qui sont à la limite des langages machine. Il n'y avait pas encore d'ordinateurs personnels à l'époque ; il faudra attendre pour cela la sortie de l'Apple II en 1977.

Le début des années 70, c'est aussi un période de crise économique (Crise américaine du crédit de 1966) et de guerre (Viêt Nam, 1954-1975). L'industrie dans son ensemble est impactée, y compris dans le secteur informatique. Le temps n'est plus aux commandes pharamineuses, mais aux restrictions budgétaires.

Dans les laboratoires Bell, un nouveau système d'exploitation voit le jour : le système UNIX. Ce projet – nommé à l'origine UNICS et débuté en 1969 par deux chercheurs, Ken Thompson et Dennis Ritchie – avait pour but de mettre au point un système d'exploitation plus simple et plus fiable que son prédécesseur issu du consortium Bell labs, General Electric et MIT : le système MULTICS. **[3, 4, 5]**

Écrit en assembleur PDP sur un DEC PDP-7, le système Unix fut également porté sur un PDP-11 disposant de 24 Ko de mémoire vive (16 Ko pour le système et les 8 Ko restant pour les utilisateurs). Multi-tâche, multi-utilisateur et gratuit, le système Unix remporta un grand succès, en particulier dans le milieu universitaire.

Néanmoins, l'assembleur PDP n'était pas le langage le plus adéquat pour maintenir Unix. Ken Thompson tente, en 1969, d'écrire un compilateur Fortran en TMG sur un PDP-7, puis préfère écrire un nouveau langage à partir du BCPL : le langage B. **[15]**

Source : WIKIMEDIA (Creative Commons CC-BY-SA-2.0 licence), http://cm.bell-labs.com/who/dmr/picture.html

► *Ken Thompson (assis) et Dennis Ritchie autour d'un PDP-11.*

Ce langage disposant de quelques lacunes pour maintenir Unix, Dennis Ritchie entreprend en 1971 d'améliorer le langage B dans une nouvelle version, le NB (*New B*), qui fut renommé plus tard en C. Langage procédural et structuré, le langage C possède des structures de contrôle (`if`, `switch`, `while`, `do`, `for`), des types (`int`, `short`, ...), des tableaux, des pointeurs, des structures de données (`struct`) et des fonctions avec des arguments pouvant être passés par valeur ou par adresse. Le succès est au rendez-vous : en trois décennies, il devient le langage de *haut-niveau* (le bas-niveau étant alors le langage machine) le plus utilisé au monde. Le système Unix commença à migrer vers le C dès la version V2 (1972). **[6, 7]**

En 1989, le langage C fut une première fois normalisé par *l'Institut national américain de normalisation* (ANSI), appelé « norme ANSI C », ou **C89**. Il fut de nouveau normalisé en 1995, cette fois par *l'Organisation internationale de normalisation* (ISO), suite à des correctifs et amendements (norme **C94**, aussi appelée **C95**).

C'est surtout en 1999 qu'une évolution importante du langage C eut lieu avec la norme **C99**, encore en vigueur aujourd'hui (les premières lignes d'un fichier de la bibliothèque standard du C, comme `/usr/include/stdlib.h`, devraient vous indiquer *« ISO C99 Standard »)*, qui supporte, entre autres, les tableaux dynamiques. Encore récemment, en 2011, l'ISO normalisa une nouvelle version, la **C11** (ou *C1X*), qui améliore le support du multi-threading et doit, en principe, remplacer la version **C99**. **[8]**

Aujourd'hui le langage C est toujours majoritairement utilisé, en particulier au niveau des systèmes embarqués, des systèmes d'exploitation et des pilotes matériels – pour lesquels on trouve des compilateurs C pour la plupart des micro-contrôleurs et micro-processeurs connus – ainsi que dans le milieu universitaire, où en général il reste le premier langage étudié. C'est également le langage de haut-niveau le plus rapide existant du fait de sa légèreté et de ses compilateurs longuement peaufinés.

3. PRINCIPES DU LANGAGE C

Le langage C est un langage qui permet avant tout de s'abstraire des langages plus proches de la machine, comme le langage assembleur ou le langage binaire du niveau électronique. Cependant, *in fine*, un micro-contrôleur ou un micro-processeur ne peut exécuter qu'une suite d'instructions binaires, comme tout circuit électronique. Aussi, pour qu'un programme en C parvienne à s'exécuter sur un micro-processeur, quelques opérations seront nécessaires.

On parlera souvent de taille et de type dans cet article, aussi voici les principaux types du C99 et leurs tailles, tels que définis dans le fichier **`stdint.h`** (les types **`unsigned`** sont non-signés, c'est-à-dire positifs ou nuls ; ils récupèrent le bit signé (signe '+' ou '-') pour une plus grande plage de valeurs positives) :

TYPE	TAILLE EN BITS
char, unsigned char	8
short int, unsigned short int	16
int, unsigned int	32
long int, unsigned long int	64 (32 sur systèmes 32 bits)
long long int, unsigned long long int	64
float	32
double	64
long double	128
int8_t, uint8_t	8
int16_t, uint16_t	16
int32_t, uint32_t	32
int64_t, uint64_t (sur systèmes 64 bits)	64

3.1 Édition d'un premier programme

Partons d'un cas simple. On écrit le programme C suivant, avec n'importe quel éditeur de texte (comme Vim par exemple), puis on l'enregistre dans un fichier **`prog1.c`**.

Fichier

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

int main(int argc, char *argv[])
{
    char *p = NULL;

    if(argc > 1){
        p = argv[1];
    }else{
        p = "C is the best";
    }
    printf("%s\n", p);

    return EXIT_SUCCESS;
}
```

La fonction **`main()`** est le point d'entrée du programme, fonction unique qui sera exécutée en premier et qui contient deux *arguments* : **`int argc`** et **`char *argv[]`**. Le premier argument **`argc`** est un entier indiquant le nombre de paramètres passés en ligne

de commandes lorsqu'on exécute le programme (nom du programme inclus) ; tandis que l'argument **argv** est un pointeur (notation *****) de tableaux (notation **[]**) de caractères (**char**) qui contient les paramètres de la ligne de commandes. De plus, la fonction **main()** attend un code retour de type **int**, que l'on passe à l'instruction **return**.

La première instruction (**char *p = NULL**) *déclare* un pointeur *p* de caractères (**char *p**), que l'on *initialise* à adresse nulle. Ensuite, si le nombre de paramètres, incluant le programme, est supérieur à 1 (**argc > 1**), on *affecte* le premier paramètre – qui ne soit pas le nom du programme – à *p* (**p = argv[1]**, les tableaux commencent par l'indice 0), sinon on affecte au pointeur *p* la chaîne de caractères *« C is the best »*.

Ensuite, un appel à la fonction **printf** – définie dans le fichier d'en-tête (ou *header*) **stdio.h** que l'on a inclus par la *macro* **#include** – affiche la chaîne de caractères pointée par *p*. **EXIT_SUCCESS** est également une macro qui retourne l'entier 0 (**EXIT_FAILURE** retourne 1). Dans le fichier **stdio.h** (en principe se trouvant dans le dossier **/usr/include/**) nous trouvons le *prototype* de la fonction **printf** qui indique que celle-ci attend un pointeur constant de caractères en argument (**const char ***), suivi d'un nombre quelconque d'arguments (**...**). Cette fonction renvoie également un entier (voir aussi le manuel, commande **man 3 printf**, troisième section des pages du manuel, section développement, après avoir installé le paquet **gcc-X.X-doc**) :

Fichier

```
int printf (const char *__restrict __fmt, ...)  ;
```

Les choses commencent à se compliquer : on a déjà des pointeurs, des constantes, des pointeurs de tableaux, des pointeurs de caractères, des pointeurs de tableaux de caractères... Faisons un rapide point sur les pointeurs avant d'aller plus loin.

3.2 Un point sur les pointeurs

Un *pointeur*, que l'on déclare avec un astérisque (*), est un type de donnée, une variable, de la taille d'une adresse mémoire (32 bits sur un système 32 bits, 64 bits sur un 64 bits, c'est-à-dire la taille d'un *mot* (**__WORDSIZE**), et non la taille d'un *int* qui peut être de 32 bits sur un système 64 bits), et qui **contient une adresse mémoire**. Comme son nom l'indique, il sert à indiquer, à pointer, la première adresse d'une zone mémoire. Cela peut sembler abstrait lorsque l'on est habitué à des langages plus récents dénués de pointeurs (du moins en surface), mais c'est très concret. L'avantage c'est qu'au lieu de recopier toute une zone de mémoire lorsqu'on fait appel à une fonction qui nécessite sa lecture ou son écriture, on utilise juste son adresse de départ que l'on récupère avec un pointeur.

Un *pointeur de caractères* (**char***) c'est donc un pointeur de la taille d'un *mot*, qui contient l'adresse mémoire du début d'une chaîne de caractères. Par commodité, on dit que c'est un pointeur de type *char* (et **int*** un pointeur de type *int*), mais il faut plutôt le voir comme un pointeur qui pointe vers des données de type *char*. Par contre, un pointeur n'indique pas, de lui-même, la fin de la zone mémoire qui contient les données. C'est le point crucial, le point de difficulté du langage C, *the big one*. Comment savoir où s'arrête une zone mémoire lorsqu'on ne connaît que son adresse de départ ?

En principe – bien que l'on pourrait retrouver cette information par des chemins détournés mais non-conventionnels – on ne peut pas connaître directement cette taille, mais il existe des conventions. Par convention, une *chaîne de caractères* se termine par un zéro (un octet 0), que l'on note **'\0'**. Par conséquent, le pointeur de caractères (**char***) pointe sur l'adresse mémoire du début d'une chaîne de caractères et la fonction **printf** devra parcourir toutes les adresses

contiguës jusqu'à trouver un octet à 0. Dans notre exemple, pour revenir au pointeur *p*, celui-ci pointe sur l'adresse mémoire qui contient le premier caractère de la chaîne. Chaque *mot* de mémoire est composé de cellules d'un octet, et chaque cellule possède une adresse différente. Sur un système 64 bits les adresses ont une taille de 64 bits. Il s'agit *d'adresses virtuelles*, la mémoire physique étant gérée par des systèmes de segmentation et de pagination.

Prenons par exemple l'état d'une mémoire 64 bits suivant, lorsque le programme, lancé sans arguments, est arrivé à l'instruction `printf` :

ADRESSES VIRTUELLES 64 BITS	VALEURS HEXADÉCIMALES DES CELLULES EN MÉMOIRE					
...	...					
0x7fffffffe958 (&p)	B4	06	40	00	00	00
...	...					
0x0000004006c0	74 ('t')	00	...			
0x0000004006ba	68 ('h')	65 ('e')	20 (' ')	62 ('b')	65 ('e')	73 ('s')
0x0000004006b4	43 ('C')	20 (' ')	69 ('i')	73 ('s')	20 (' ')	74 ('t')
...	...					

Dans cet exemple, le pointeur *p* est déclaré à l'adresse `0x7fffffffe958` (l'adresse d'une variable peut être obtenue avec *l'opérateur adresse*, noté `&` comme `&p` (à ne pas confondre avec la *référence* & du C++, il n'y a pas de *déclaration de référence* en C). La valeur binaire de cette adresse est `B4` (première cellule du tableau). Cependant, comme un pointeur indique une adresse, et que les adresses tiennent ici sur 64 bits, soit 6 cellules de 8 bits, on trouve comme adresse pointée par *p* la valeur `0x0000004006b4` (les 6 cellules dans l'ordre inverse en format *little-endian* sur processeurs *x86*). On regarde ce qui se trouve à cette adresse, et effectivement, on retrouve le début de notre chaîne de caractères, le caractère `'C'` (`0x43` en hexadécimal). En suivant cellule par cellule la mémoire jusqu'à celle qui contient la valeur `00`, on retrouve bien notre message « *C is the best* ».

Autrement dit, le pointeur *p* contient l'adresse du premier caractère de la chaîne. L'adresse de la cellule suivante – du deuxième caractère (' ') – c'est l'adresse du premier caractère + 1 octet : `0x0000004006b5` ; le troisième ('i') c'est donc `0x0000004006b6`. On peut le vérifier en indiquant à *p* l'index concerné, comme pour un tableau de caractères (*p[0] = 'C', p[1] = ' ', p[2] = 'i', ...*). De plus, on peut incrémenter le pointeur directement en utilisant l'opérateur `++`, ce qui a pour effet d'incrémenter – selon le type de donnée – l'adresse de la mémoire contenue par *p* (et donc de modifier *p*). Par exemple, si l'on ajoute l'instruction `p++` avant la fonction `printf`, celle-ci n'affichera plus que « *is the best* ».

En résumé, la fonction `printf` va parcourir de la première adresse pointée par *p* de valeur 'C' jusqu'à la première cellule de valeur 00, puis afficher le message suivi d'un retour-chariot ('*\n', newline*). Vous pourrez vérifier tout cela en utilisant un *débogueur* comme `gdb` et en scrutant le contenu de la mémoire. [9,10]

3.3 Précompilation, compilation, édition des liens et exécution

Pour passer du code C à un programme exécutable, il suffit de faire appel au compilateur `gcc` (*GNU CC*, version GNU – développée par Richard Stallman – du compilateur CC d'Unix). Par exemple, la commande suivante compile le fichier `prog1.c` en un exécutable `a.out` :

À savoir

On trouve fréquemment des *doubles pointeurs*, comme `char **pp`. Un double pointeur `char**` peut être soit la déclaration d'un pointeur qui pointe vers une suite de pointeurs de caractères (comme un tableau à deux dimensions, on a `pp[0]` un premier pointeur de caractères, `pp[1]` un second, etc.), ou soit, en argument de fonctions, un type *adresse de pointeur* (que l'on utilisera avec un appel à `&pp`, comme par exemple dans la fonction `getline` (cf. `man 3 getline`)). Par ailleurs, dès lors que l'on réutilise la notation * sur un pointeur déjà déclaré, comme `*p`, on n'obtient non pas l'adresse du pointeur (`&p`), ni l'adresse pointée (juste `p`), mais la valeur du premier élément pointé (comme `p[0]`). Par exemple :

```
const char *p = "Test";
printf("%c\n", *p); //
affiche le caractère 'T'
```

Terminal
```
$ gcc prog1.c
$ ./a.out
C is the best
```

On peut bien sûr lui passer le nom du programme avec l'option `-o`.

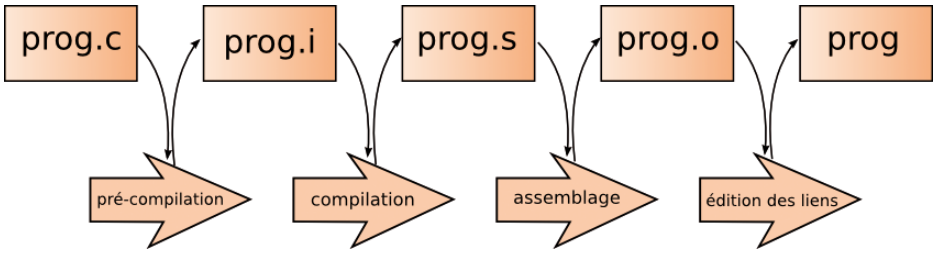
Mais ce qui va nous intéresser, ça ne va pas être de décortiquer la pléthore des options du compilateur, mais de voir ce qui se passe d'un peu plus près.

3.3.1 Un programme pour un micro-processeur et un système d'exploitation

Un programme, c'est une suite d'instructions binaires que le micro-processeur devra exécuter au niveau électronique. Aussi, l'exécutable `a.out` généré précédemment ne contient qu'une suite de bits, c'est du *langage machine*. Ces bits ne peuvent être lus que par un micro-processeur spécifique pour lequel le programme aura été compilé, et le résultat ne pourra être exploité que par un système d'exploitation donné. Pour savoir pour quel micro-processeur sera compilé par défaut un programme, vous pouvez appeler la commande `gcc -v`, qui affiche en particulier le processeur et le système cible (*Target,* ici un processeur de type x86-64 et un système GNU/Linux) :

Terminal
```
$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/4.8/lto-wrapper
Target: x86_64-linux-gnu
...
```

Maintenant, pour passer d'un fichier C à un exécutable binaire, le compilateur devra effectuer plusieurs tâches successives.



3.3.2 Du langage machine au langage assembleur

Il y a fort longtemps, à l'époque des dinosaures ou presque, alors que les ordinateurs ne possédaient pas de transistors mais des tubes à vides et des câbles, comme le *Whirlwind I*, *le Manchester Mark I*, le *SSEC* (1948-1950), les programmeurs, ou plutôt les chercheurs, codaient directement en langage machine, avec des bits (sur certains modèles, comme le *Whirlwind I,* on trouvait tout de même un clavier et un écran cathodique).

Les instructions étaient alors moins complexes et en nombre plus restreint. Une instruction, en binaire, est une suite de bits dont les premiers indiquent l'opération à effectuer (bits *op*, *operation*), et chaque processeur possède son propre jeu d'instructions. On peut observer le *code machine* hexadécimal du programme `a.out`, avec la commande `hexdump` par exemple :

Terminal

```
$ hexdump -C a.out
...
000005d0  8b 7c 24 30 48 83 c4 38  c3 0f 1f 80 00 00 00 00   |.|$0H..8........|
000005e0  f3 c3 00 00 48 83 ec 08  48 83 c4 08 c3 00 00 00   |....H...H.......|
000005f0  01 00 02 00 43 20 69 73  20 74 68 65 20 62 65 73   |....C is the bes|
00000600  74 00 00 00 01 1b 03 3b  30 00 00 00 05 00 00 00   |t......;0.......|
00000610  cc fd ff ff 7c 00 00 00  0c fe ff ff 4c 00 00 00   |....|.......L...|
...
```

Difficile de lire le début et la fin d'une instruction et de savoir ce que fait ce programme, même si l'on voit apparaître la chaîne de caractères. Dès lors, l'utilisation de mnémoniques pour repérer les suites d'instructions à la place de leurs valeurs binaires facilita grandement la programmation : ce fut le *langage assembleur*, apparu dans les années 50. Par exemple, l'instruction machine 1110 1011 0000 1000 (ou EB08 en hexadécimal), est équivalente en assembleur à l'instruction **`jmp label`**.

Aussi pour passer d'un fichier C à un programme binaire, la principale phase de la compilation consiste à traduire le fichier texte C en un fichier texte assembleur. Une fois ce fichier généré, le compilateur **`gcc`** fait ensuite appel au programme assembleur (le programme **`as`**, ou *GNU assembler*), qui va le lire et produire un fichier binaire, appelé aussi *fichier objet* (extension **`.o`**). Bien que cette étape soit cachée par **`gcc`**, on peut néanmoins lui passer des commandes pour l'assembleur. Ainsi, on peut voir le fichier assembleur produit par **`gcc`** avec l'option **`-S`** (ce qui génère un fichier **`.s`**), et le fichier objet généré par l'assembleur avec l'option **`-c`** :

Terminal

```
$ gcc -S prog1.c
$ less prog1.s
        .file   "prog1.c"
        .section        .rodata
.LC0:
        .string "C is the best"
        .text
        .globl  main
        .type   main, @function
main:
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        ...
```

C'est déjà plus compréhensible que le code machine. Au lieu de lire une suite de bits, on lit à présent une suite d'instructions, de même que des *sections* de données : **`.rodata`**, pour les données en lecture seule (la chaîne de caractères s'y trouve) et **`.text`** pour le code exécutable. Le langage assembleur est encore utilisé de nos jours pour effectuer certains traitements spécifiques à un processeur ou coder des routines optimisées. De plus, on peut également utiliser directement le programme assembleur **`as`**, lequel est fourni avec **`gcc`**. [12,13]

3.3.3 Précompilation

Avant de passer à l'étape de compilation, le fichier C est purgé de toutes ses *macros* – directives pour le préprocesseur **`cpp`** (*C preprocessor*) dont les définitions sont reconnaissables par le caractère **`#`** – et cela durant une phase de précompilation (ou *preprocessing*). Le code résultant est un fichier C contenant les valeurs des macros à la place de leurs noms. Dans notre exemple de départ, la macro **`EXIT_SUCCESS`**, qui provient d'une directive **`#define`**, est remplacée par sa valeur 0 et les macros **`#include`** par le

contenu des fichiers à inclure. Pour afficher le fichier obtenu, on peut utiliser `gcc` avec l'option `-E`, et avec `-o` pour indiquer le fichier de sortie (de préférence avec l'extension `.i` pour les fichiers pré-compilés) :

Terminal
```
$ gcc -E -o prog1.i prog1.c
$ less prog1.i
[...]
int main(int argc, char *argv[])
{
    char *p = ((void *)0);
   [...]
    return 0;
}
```

3.3.4 Édition des liens

Enfin, dernière étape effectuée par le compilateur `gcc` : l'édition des liens. Après avoir généré des fichiers objets `.o` avec l'assembleur `as`, `gcc` va utiliser le programme `ld` (*the GNU Linker*), pour lier les symboles des fonctions dont le code se situe dans d'autres fichiers objets, ou dans des bibliothèques externes (statiques ou dynamiques). Dans notre exemple, le code, la définition, *l'implémentation* de la fonction `printf` se situe dans la bibliothèque dynamique `libc.so` (*so* comme *shared object*). Le linker `ld` relie le tout et génère un exécutable final au format ELF (*Executable and Linkable Format*, format des exécutables sous Linux).

Comme l'assembleur `as`, on peut utiliser `ld` à part entière ou lui passer des options via `gcc`, comme `-o` (nom de l'exécutable), `-lm` (utilisation de la bibliothèque mathématique), le nom d'une bibliothèque à linker, etc. On peut également lui indiquer de générer une bibliothèque de fonctions au lieu d'un programme exécutable, avec l'option `-shared` pour une *bibliothèque dynamique* (dynamiquement chargée à l'exécution du programme), ou `-static` pour une *bibliothèque statique* (intégrée au programme durant l'édition des liens).

On peut lister les bibliothèques dynamiques utilisées par un programme avec la commande `ldd`, par exemple :

Terminal
```
$ ldd a.out
 linux-vdso.so.1 (0x00007fffff4bf000)
 libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f04bf050000)
 /lib64/ld-linux-x86-64.so.2 (0x00007f04bf42c000)
```

Cette commande est également utile pour afficher les liens qui font défaut.

Pour lister les fonctions exportées par une bibliothèque dynamique, on peut utiliser les commandes `nm` ou `readelf` :

Terminal
```
$ nm -D --defined-only /lib/x86_64-linux-gnu/libc.so.6
[...]
0000000000050ec0 T printf
[...]
$ readelf -s /lib/x86_64-linux-gnu/libc.so.6
[...]
   594: 0000000000050ec0   161 FUNC    GLOBAL DEFAULT   12
printf@@GLIBC_2.2.5
[...]
```

À savoir

Le programme ne peut inclure deux fois le même fichier, cela multiplierait les prototypes des fonctions et les déclarations. Pour pallier ce problème, on ajoute des macros `#ifndef _LE_FICHIER_H` et `#define _LE_FICHIER_H` en début de fichier d'en-tête, et une macro `#endif` à la fin de fichier, ce qui a pour effet d'inclure une seule fois le contenu du fichier durant la précompilation, à l'instar d'un `#pragma once`, mais standardisé.

En résumé, en passant la commande `gcc prog.c`, le compilateur aura pré-compilé le fichier `prog.c` en un fichier C dépourvu de macros, compilé celui-ci en un fichier texte assembleur, passé ce dernier à l'assembleur `as` pour générer un *fichier objet* binaire, puis relié ce fichier objet avec les *bibliothèques partagées* nécessaires grâce à l'éditeur de liens `ld`, pour produire au final (avec `ld` toujours) un exécutable `a.out` au format ELF. [12]

4. EXEMPLES

4.1 Appels de fonction

Dans cet exemple, nous allons faire quelques appels de fonction, avec quelques erreurs que l'on corrigera par la suite. On reprend l'exemple de départ, qu'on améliore en lui ajoutant une fonction `afficher()`. On passe à cette fonction un argument de type pointeur de caractères constant (`const char*`) qu'elle affichera avec un appel à la fonction `printf`.

Fichier

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

void afficher(const char *message)
{
    printf("%s\n", *message);
}

int main(int argc, char *argv[])
{
    const char *p = NULL;

    if(argc > 1){
        p = argv[1];
    }else{
        p = "C is sexy";
    }
    afficher(p);

    return EXIT_SUCCESS;
}
```

Le programme se compile bien, mais à l'exécution on obtient une *erreur de segmentation* :

Terminal

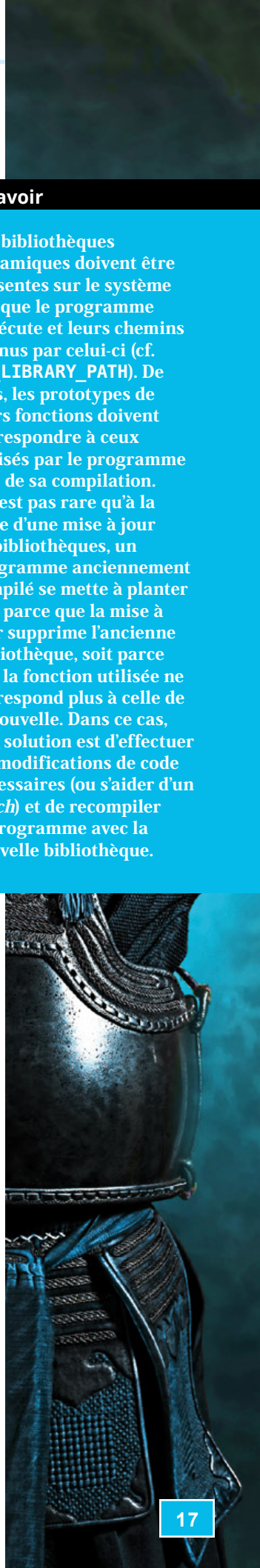
```
$ gcc exemple1.c -o exemple1
$ ./exemple1
Erreur de segmentation
```

Une erreur de segmentation concerne des segments de mémoire, lorsque par exemple le programme tente de lire dans un segment de mémoire dont il n'a pas le droit d'accès, ou lorsqu'il essaye d'écrire dans un segment en lecture seule (dans une constante par exemple).

Dans cet exemple, on passe bien le pointeur *p* à la fonction `afficher`, mais la fonction `printf` attend une chaîne de caractères (`%s`), c'est-à-dire l'adresse du premier caractère pointé par le pointeur. Mais au lieu de lui

À savoir

Les bibliothèques dynamiques doivent être présentes sur le système lorsque le programme s'exécute et leurs chemins connus par celui-ci (cf. `LD_LIBRARY_PATH`). De plus, les prototypes de leurs fonctions doivent correspondre à ceux utilisés par le programme lors de sa compilation. Il n'est pas rare qu'à la suite d'une mise à jour de bibliothèques, un programme anciennement compilé se mette à planter soit parce que la mise à jour supprime l'ancienne bibliothèque, soit parce que la fonction utilisée ne correspond plus à celle de la nouvelle. Dans ce cas, une solution est d'effectuer les modifications de code nécessaires (ou s'aider d'un *patch*) et de recompiler le programme avec la nouvelle bibliothèque.



fournir cette adresse, on lui passe la valeur du premier élément (`*message`), c'est-à-dire le caractère 'C', soit la valeur hexadécimale 0x43. Du coup, au lieu d'aller lire à l'adresse indiquée par le pointeur, le programme va tenter de lire à partir de l'adresse 0x43. Cette adresse ne concernant pas les données du programme, et même pouvant être utilisée par un autre programme, au lieu de poursuivre les dégâts, le système va interrompre le programme et envoyer une erreur de segmentation. Une correction du programme est donc d'indiquer le pointeur (`message`) – et non son premier élément (`*message`) – en paramètre de la fonction `printf`.

Une bonne pratique est d'utiliser des *codes retours* de fonction : on gagne ainsi en temps de débogage et cela facilite également les *tests unitaires*. Ainsi la fonction `printf` renvoie le nombre de caractères affichés si succès, et un nombre négatif en cas d'erreur (cf. `man 3 printf`). Cependant, l'erreur de segmentation intervient avant que la fonction `printf` n'ait pu se terminer : c'est une *interruption de programme*, un arrêt d'urgence en quelque sorte. On ajoute néanmoins des codes retours par sécurité, et également une boucle `for`, laquelle boucle depuis `i = 1` et tant que `i < argc`, en incrémentant `i` de 1 (`i++`).

Fichier

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

#define TRUE    1
#define FALSE   0

int afficher(const char *message)
{
    int erreur = 0;

    erreur = printf("%s\n", message);
    if (erreur < 0) {
        return FALSE;
    }
    return TRUE;
}

int main(int argc, char *argv[])
{
    const char *p = NULL;
    int i = 0;

    if(argc > 1){
      for (i=1; i<argc; i++){
        p = argv[i];
        if (!afficher(p)) {
           perror("erreur de la fonction afficher");
           return EXIT_FAILURE;
        }
      }
    }else{
        p = "C is sexy";
        if (!afficher(p)) {
           perror("erreur de la fonction afficher");
           return EXIT_FAILURE;
        }
    }
    return EXIT_SUCCESS;
}
```

Terminal

```
$ gcc exemple1.c -o exemple1;
$ ./exemple1 "C is nice" "C is incredible"
C is nice
C is incredible
```

4.2 Prototype de fonction

Jusqu'à présent, nous n'avons utilisé qu'un seul fichier et une fonction, mais dans la pratique un programme se compose de nombreuses fonctions. Aussi, il convient de respecter une certaine structure : on place les prototypes des fonctions dans des fichiers d'en-têtes **.h** *(headers)*, et leurs définitions dans des fichiers sources **.c**.

En reprenant le programme précédent, nous allons ajouter le prototype de la fonction **afficher()** dans un fichier **fonctions.h**, et coder sa définition dans un fichier **fonctions.c**. La fonction **main()** est placée dans un fichier **exemple2.c**. Une bonne pratique est de **commenter les prototypes**, en indiquant au moins les codes retours. Dans cet exemple, nous utilisons le format de commentaires du logiciel Doxygen. On obtient les fichiers suivants (notez bien les **#include** : par convention on place les en-têtes locaux entre guillemets et les en-têtes système entre chevrons) :

fonctions.h :

Fichier

```
#ifndef _FONCTIONS_H
#define _FONCTIONS_H

#define TRUE 1
#define FALSE 0

/**
 * affiche un message
 * @param[in] message le message à afficher
 * @return TRUE si succès, sinon FALSE
 **/
int afficher(const char *message);

#endif  /*_FONCTIONS_H*/
```

fonctions.c :

Fichier

```
#include "fonctions.h"
#include <stdio.h>

int afficher(const char *message)
{
    int erreur = 0;

    erreur = printf("%s\n", message);
    if (erreur < 0) {
        return FALSE;
    }
    return TRUE;
}
```

exemple2.c :

Fichier

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

int main(int argc, char *argv[])
{
    const char *p = NULL;
```


Fichier

```
    if(argc > 1){
        p = argv[1];
    }else{
        p = "C is sexy";
    }
    if (!afficher(p))
    {
        perror("afficher");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Gcc va ensuite générer un fichier **exemple2.o** et un fichier **fonctions.o**, puis relier ces deux fichiers objets en un programme exécutable. La compilation du programme s'effectue par la commande suivante :

Terminal

```
$ gcc exemple2.c fonctions.c -o exemple2
$ ./exemple2 "test ok"
test ok
```

4.3 Structures de données

Pour éviter d'avoir à gérer des variables éparpillées, on peut les regrouper dans des structures de données. Les structures se déclarent avec le mot-clé **struct**, suivi du bloc de type de données à inclure, sans oublier le point-virgule final. On peut ensuite initialiser la structure en utilisant des accolades, et accéder aux données membres avec un caractère '**.**'. Par exemple (notez également la fonction **strlen**, qui renvoie la taille d'une chaîne de caractères) :

Fichier

```
#include "fonctions.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct message
{
    char *message;
    int longueur;
};

int main(int argc, char *argv[])
{
    struct message mess = { NULL, 0 };

    if(argc > 1){
        mess.message = argv[1];
    }else{
        mess.message = "C is sexy";
    }
    mess.longueur = strlen(mess.message);
    printf("longueur du message = %i\n", mess.longueur);

    if (!afficher(mess.message))
    {
        perror("afficher");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Une autre façon, plus couramment utilisée, de déclarer une structure est de l'inclure dans un **typedef**, qui sert à redéfinir le nom d'un type. Ainsi, dans l'exemple suivant, l'instruction **typedef** a pour effet de définir le type **struct message** comme un type **s_message** et un pointeur **s_message*** comme un type **p_message**. Par contre, pour accéder aux données membres d'un *pointeur de structure*, on utilise les caractères '**->**'. Par exemple :

Fichier

```
#include "fonctions.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef struct message
{
    char *message;
    int longueur;
} s_message, *p_message;

int main(int argc, char *argv[])
{
    s_message mess = {NULL, 0};
    p_message pmess = &mess;

    if(argc > 1){
        pmess->message = argv[1];
    }else{
        pmess->message = "C is sexy";
    }
    pmess->longueur = strlen(pmess->message);
    printf("longueur du message = %i\n", pmess->longueur);

    if (!afficher(pmess->message))
    {
        perror("afficher");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Terminal

```
$ gcc exemple3.c fonctions.c -o exemple3
$ ./exemple3 "ceci est un test"
longueur du message = 16
ceci est un test
```

4.4 Mémoire dynamique

Jusqu'à présent, nous n'avons pas rencontré beaucoup de problèmes de mémoire, car on se contentait d'utiliser des chaînes de caractères déjà placées en mémoire avant que la fonction **main()** ne commence. Cependant, dès lors que l'on ajoute des données de taille importante durant l'exécution du programme, il devient nécessaire de gérer dynamiquement le stockage en mémoire avec des fonctions comme **malloc**, **calloc**, **realloc** et **free**.

Prenons l'exemple suivant, qui utilise une fonction **memset** pour initialiser le tableau de caractères à zéro, une fonction **strcpy** pour copier une chaîne de caractères vers une autre, et une énumération **enum** qui énumère et incrémente une suite d'entiers à partir du premier, séparés par des virgules (**DEFAUT** vaut 0, **COMMANDE** vaut 1, on aurait pu mettre aussi **DEFAUT=0**). Le programme compile, mais provoque une erreur à l'exécution :

Fichier

```
#include "fonctions.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_CHAR 10000000

typedef enum MESSAGE_TYPE
{
   DEFAUT,
   COMMANDE
} E_TYPE;

typedef struct message
{
    char message[MAX_CHAR];
    int longueur;
    E_TYPE type;
} s_message, *p_message;

int main(int argc, char *argv[])
{
    s_message smess;
    p_message pmess = &smess;

    memset(pmess->message, 0, MAX_CHAR); //initialise la chaîne message à zéro
    pmess->longueur = 0;

    if(argc > 1){
         strcpy(pmess->message, argv[1]);  //copie de argv[1] à pmess->message
         pmess->type = COMMANDE;
    }else{
        strcpy(pmess->message, "C is amazing");
        pmess->type = DEFAUT;
    }

    if (!afficher(&pmess->message[0]))
    {
        perror("afficher");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Terminal

```
$ gcc exemple4.c fonctions.c -o exemple4
$ ./exemple4
Erreur de segmentation
```

Effectivement, on déclare un tableau de caractères de 10 Mo dans la structure **`message`**, puis le programme tente d'allouer la mémoire nécessaire à cette structure de façon statique. Cependant, la *mémoire statique* est d'une taille limitée : elle est allouée avant l'exécution du programme et permet de déclarer des variables de petite taille, comme des **`int`**, **`char`**, etc. Il en va de même pour la mémoire réservée aux fonctions, appelée *mémoire de pile*, ou *mémoire stack*, dont le dépassement engendre un *débordement de pile* (ou *stack overflow*).

D'une taille trop importante pour la mémoire statique, le programme bogue à la première ligne de la fonction **`main()`** : '**`s_message smess;`**'.

Pour allouer de la mémoire de grande taille (>100 Ko), il faut utiliser la *mémoire sur le tas* (ou *mémoire heap*). Pour ce faire, on utilise la fonction **`malloc()`**, qui alloue de l'espace en mémoire

heap, et la fonction `free()` pour libérer celle-ci après utilisation. La libération de mémoire dynamique est très importante, car si on ne le fait pas, et si le programme alloue mémoire sur mémoire, les anciennes allocations n'étant pas libérées, le programme augmentera alors constamment la taille de la mémoire allouée, réduisant d'autant la mémoire libre du système (*fuite de mémoire*), jusqu'à provoquer à la longue un ralentissement ou une panne du système.

La fonction `malloc()` renvoie un pointeur indéfini (`void*`), que l'on peut *caster* en pointeur `p_message` – c'est le *dynamic cast* parenthésé `(p_message)` – et qui pointe vers l'adresse du début de la mémoire allouée (ou `NULL` si erreur). Cette fonction prend en paramètre la taille à allouer en octets (cf. `man 3 malloc`). Comme la structure `message` possède une taille fixe, on peut utiliser l'instruction `sizeof()` pour indiquer la taille à allouer. Cependant, si la chaîne de caractères dépasse la taille allouée, il y a *débordement de tampon (buffer overflow)*, faille fréquemment utilisée par des programmes pirates pour accéder au système et dont le résultat est incertain (cf. `man 3 strcpy`). C'est pourquoi l'on indique spécifiquement la taille d'une zone mémoire en plus de son pointeur, lorsqu'on utilise des données allouées, c'est le cas par exemple de la fonction `main` (taille du tableau, `argc`, suivie du pointeur de données, `*argv[]`).

On corrige et restructure le programme en plaçant les définitions dans le fichier d'en-tête et en modifiant la fonction `afficher` en lui ajoutant une instruction `switch`. Cette instruction `switch` va exécuter un bloc `case` jusqu'au premier `break` ou `return` rencontré selon la valeur de l'étiquette (*label*) du `case`, de type `int` (ici une énumération `E_TYPE`), un peu comme une suite de conditions `if else` mais en plus rapide. Le programme devient le suivant :

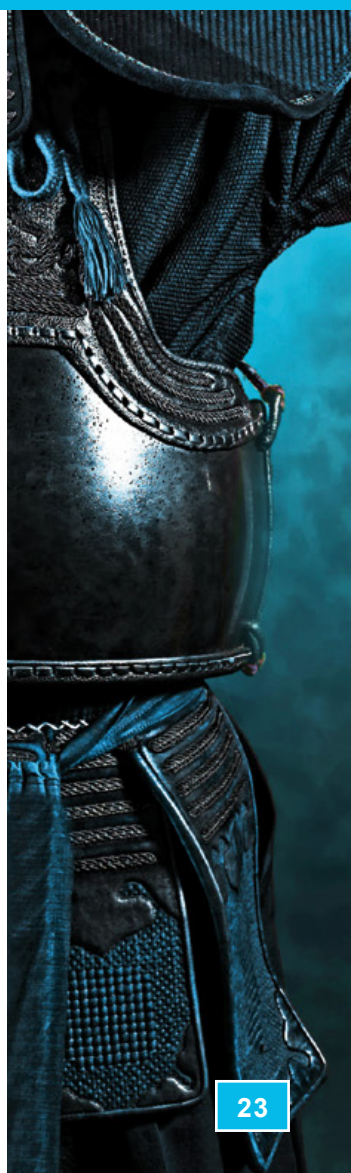
`fonctions.h` :

Fichier

```
#ifndef _FONCTIONS_H
#define _FONCTIONS_H

#define TRUE 1
#define FALSE 0
#define MAX_CHAR (int)10e6

/**
 * Type de message
 */
typedef enum MESSAGE_TYPE
{
    DEFAUT,
    COMMANDE
} E_TYPE;

/**
 * structure d'un message
 */
typedef struct message
{
    char message[MAX_CHAR];
    int longueur;
    E_TYPE type;
} s_message, *p_message;
```

À savoir

Si la fonction `free` n'a pas besoin de la taille de la zone mémoire en paramètre, cela est dû à l'implémentation de la fonction `malloc`, qui ajoute également des informations de taille en zone mémoire. Cependant, il n'est pas recommandé d'utiliser ces informations au sein d'un programme, celles-ci étant dépendantes de l'implémentation de la fonction `malloc`, et servant surtout à éviter les nombreux problèmes mémoire qu'il y aurait s'il fallait également renseigner la taille à la fonction `free`.



Fichier

```
/**
 * affiche un message
 * @param[in] pmessage le p_message à afficher
 * @return TRUE si succès, sinon FALSE
 */
int afficher(const p_message pmessage);

#endif /* _FONCTIONS_H */
```

fonctions.c :

Fichier

```
#include "fonctions.h"
#include <stdio.h>

int afficher(const p_message pmessage)
{
    int erreur = 0;

    if (pmessage == NULL) {
        return FALSE;
    }
    switch (pmessage->type) {
        case DEFAUT:
            erreur = printf("Message par défaut : %s\n", pmessage->message);
            break;
        case COMMANDE:
            erreur = printf("Message commande : %s\n", pmessage->message);
            break;
        default:
            printf("Type de message non reconnu\n");
            erreur = -1;
            break;
    }
    if (erreur < 0) {
        return FALSE;
    }
    return TRUE;
}
```

exemple4.c :

Fichier

```
#include "fonctions.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    p_message pmess = (p_message) malloc(sizeof(s_message));

    if (pmess == NULL)
    {
        fprintf(stderr, "erreur d'allocation\n");
        return EXIT_FAILURE;
    }
    memset(pmess->message, 0, MAX_CHAR);
    pmess->longueur = 0;
```

Fichier

```
    if(argc > 1 && strlen(argv[1]) < MAX_CHAR){
         strcpy(pmess->message, argv[1]);
         pmess->type = COMMANDE;
    }else{
        strcpy(pmess->message, "C is amazing");
        pmess->type = DEFAUT;
    }

    if (!afficher(pmess))
    {
        perror("afficher");
        free(pmess);
        return EXIT_FAILURE;
    }

    free(pmess);
    return EXIT_SUCCESS;
}
```

Terminal

```
$ gcc exemple4.c fonctions.c -o exemple4 ; ./exemple4
Message par défaut : C is amazing
```

Remarquez également l'utilisation de la fonction **fprintf**, qui écrit dans un descripteur de fichier, ici la sortie d'erreur standard **stderr**, que l'on peut par la suite rediriger en console avec les redirections du shell, technique souvent employée pour les journaux (*logs*).

4.5 Lecture et écriture d'un fichier

Enfin, dernier exemple concernant le traitement de fichiers. La fonction **fopen()** ouvre un *pointeur de flux* (**FILE***), lequel pointe vers l'emplacement de lecture du fichier et avance à chaque caractère lu. De plus, pour lire une ligne, on peut utiliser la fonction **getline()**, qui nécessite un *buffer* pour écrire les données lues (cf. **man 3 fopen**, **man 3 getline**).

Cette fois, on prend un tout autre exemple qui consiste à lire les données d'un fichier – des coordonnées de points en 3D – pour en calculer la distance euclidienne *D* (pour deux points *a* et *b*) :

$$D=\sqrt{(Xb-Xa)^2+(Yb-Ya)^2+(Zb-Za)^2}$$

Le fichier de données **data**, construit à partir du nombre pi, est le suivant :

data :

Fichier

```
#Xa:Ya:Za;Xb:Yb:Zb
14.15:92.65:35.89;79.32:38.46:26.43
38.32:79.50:28.84;19.71:69.39:93.75
10.58:20.97:49.44;59.23:07.81:64.06
28.62:08.99:86.28;03.48:25.34:21.17
06.79:82.14:80.86;51.32:82.30:66.47
09.38:44.60:95.50;58.22:31.72:53.59
40.81:28.48:11.17;45.02:84.10:27.01
93.85:21.10:55.59:64.46:22.94:89.54
```

À savoir

On peut détecter les fuites de mémoire avec le logiciel Valgrind, l'un des plus connus sous Linux.

Chaque coordonnée cartésienne X, Y, Z est séparée par deux points '**:**', chaque point de l'espace A, B par un point virgule '**;**'. On écrit le programme suivant :

exemple5.c :

Fichier

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

#define FICHIER_DATA                "data"
#define FICHIER_RESULTATS       "resultats"

typedef struct point
{
    float X;
    float Y;
    float Z;
} s_point, *p_point;

int main(int argc, char *argv[])
{
    FILE *data = NULL;
    FILE *resultats = NULL;
    char *ligne = NULL;
    size_t longueur = 0;
    ssize_t luts = 0;
    char *fin = NULL;
    float distance = 0;
    errno = 0;
    p_point a = NULL;
    p_point b = NULL;

    //allocations
    a = malloc(sizeof(s_point));
    if (a == NULL)
    {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    b = malloc(sizeof(s_point));
    if (b == NULL)
    {
        perror("malloc");
        free(a);
        exit(EXIT_FAILURE);
    }
    //ouverture des fichiers
    data = fopen(FICHIER_DATA, "r");
    if (data == NULL){
        perror("fopen data");
        free(a);
        free(b);
        exit(EXIT_FAILURE);
    }
    resultats = fopen(FICHIER_RESULTATS, "w");
    if (resultats == NULL){
        perror("fopen resultat");
        fclose(data);
        free(a);
        free(b);
        exit(EXIT_FAILURE);
    }
    //lecture des données
    while ((luts = getline(&ligne, &longueur, data)) != -1) {
        if(ligne[0] == '#'){
           continue;
        }
```

Fichier

```
        //analyse la ligne
        a->X = (float)strtod(ligne, &fin);
        a->Y = (float)strtod(++fin, &fin);
        a->Z = (float)strtod(++fin, &fin);
        b->X = (float)strtod(++fin, &fin);
        b->Y = (float)strtod(++fin, &fin);
        b->Z = (float)strtod(++fin, &fin);
        if (errno != 0) {
         perror("strtol");
         break;
        }
        //calcul
       distance = sqrt(pow(b->X - a->X, 2) + pow(b->Y - a->Y, 2) + pow(b->Z - a->Z, 2));
        //écriture du résultat
        fprintf(resultats,
             "A(%.2f;%.2f;%.2f) B(%.2f;%.2f;%.2f) d(A,B)=%.2f\n",
              a->X, a->Y, a->Z, b->X, b->Y, b->Z, distance);
    }
    fclose(data);
    fclose(resultats);
    free(a);
    free(b);
    exit(EXIT_SUCCESS);
}
```

Dans ce programme, on utilise la fonction **fopen** pour ouvrir le fichier **data** en lecture seule (*read* **"r"**) et créer ou écraser le fichier **resultats** en écriture (*write* **"w"**). Ensuite, le programme lit ligne par ligne le fichier **data** avec la fonction **getline** *tant que* (boucle **while**) la fin du fichier n'est pas atteinte, auquel cas la fonction **getline** renvoie **-1** (cf. **man 3 getline**). Si la ligne commence par un caractère '**#**', le programme passe à la ligne suivante (instruction **continue**).

Les valeurs en nombres flottants sont converties avec la fonction **strtod**, qui prend en premier paramètre le début de la chaîne de caractères à convertir et qui indique dans le pointeur **fin** l'adresse du premier caractère non numérique atteint (cf. **man 3 strtod**).

Ainsi, pour la première valeur **a->X**, la fonction **strtod** part du premier caractère de la ligne (**ligne**) et va jusqu'au premier caractère non numérique '**:**', dont l'adresse est placée dans le pointeur **fin**. Pour la seconde valeur **a->Y**, on incrémente le pointeur **fin** (**++fin** pour incrémenter *avant* l'appel de fonction, et non **fin++** qui incrémente *après* celle-ci) pour qu'il pointe vers le début du deuxième nombre, et ainsi de suite. Si une erreur de conversion survient, la fonction **strtod** renseigne la variable d'erreur **errno**, définie dans le fichier **errno.h** (cf. **man 3 strtol**) et le programme sort de la boucle **while** (**break**).

Le calcul de la distance s'effectue avec les fonctions racine carrée (**sqrt**) et puissance (**pow**) de la bibliothèque mathématique, et le résultat est écrit dans un fichier **resultats** avec la fonction **fprintf**. Avant de quitter, les fichiers sont fermés avec la fonction **fclose** et les allocations dynamiques libérées avec **free**. Pour la compilation, comme on utilise la bibliothèque mathématique, on ajoute l'option **-lm** :

Terminal

```
$ gcc exemple5.c -lm -o exemple5
$ ./exemple5
$ cat resultats
A(14.15;92.65;35.89) B(79.32;38.46;26.43) d(A,B)=85.28
A(38.32;79.50;28.84) B(19.71;69.39;93.75) d(A,B)=68.28
A(10.58;20.97;49.44) B(59.23;7.81;64.06) d(A,B)=52.48
A(28.62;8.99;86.28) B(3.48;25.34;21.17) d(A,B)=71.68
A(6.79;82.14;80.86) B(51.32;82.30;66.47) d(A,B)=46.80
A(9.38;44.60;95.50) B(58.22;31.72;53.59) d(A,B)=65.63
A(40.81;28.48;11.17) B(45.02;84.10;27.01) d(A,B)=57.98
A(93.85;21.10;55.59) B(64.46;22.94;89.54) d(A,B)=44.94
```

CONCLUSION

Cette introduction sur le langage C se termine. Après s'être intéressé à l'origine du langage C, on aura vu une bonne partie des fonctions de base et on se sera exercé avec la compilation, les pointeurs et les adresses, notions essentielless pour bien aborder ce langage. Cependant, il ne s'agit ici que d'un aperçu, de nombreux livres, revues et sites Internet, ainsi que la suite de cet ouvrage, approfondissent ces concepts et apportent des compléments de langage.

On aura également apprécié l'importance que prennent les pointeurs, très précis et très délicats à manier, comme vous aurez pu le constater à travers les exemples précédents. On pourra s'aider des pages du manuel Linux (`man`) pour la description des fonctions de l'API (*Application Programming Interface*) et utiliser des outils d'analyse de code et de débogage pour ne pas rester coincer sur un bug.

Il y aurait encore beaucoup à dire, notamment sur les tableaux dynamiques, les fonctions récursives, le débogage, mais cela dépasserait le cadre de cette introduction. Retenez qu'il faut initialiser les variables (un pointeur s'initialise en `NULL`) et qu'il faut libérer les allocations dynamiques (avec la fonction `free`). N'hésitez pas à ajouter des fonctions, à utiliser des fichiers d'en-têtes `.h` et des fichiers sources `.c`, et à factoriser les fonctions : cela rendra le code plus lisible et plus maintenable (en principe, une fonction ne doit pas dépasser 50 lignes de code), sans oublier les commentaires. Bon codage ! ■

RÉFÉRENCES

[1] TIOBE, *The TIOBE Programming Community index,* **http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html**, 2013.

[2] WIKIPEDIA, *List of programming languages*, **http://en.wikipedia.org/wiki/List_of_programming_languages**, 2013.

[3] J-Y BIRRIEN, *Histoire de l'informatique*, Que sais-je, 1990

[4] M. CAMPBELL-KELLY, *Une histoire de l'industrie du logiciel*, Vuibert Informatique, 2003

[5] H. HENDERSON, *Encyclopedia of Computer Science and Technology*, Facts on file science library, 2003

[6] The Unix Heritage Society, The Unix Tree, **http://minnie.tuhs.org/cgi-bin/utree.pl**, 2013

[7] D. M. RITCHIE, K. THOMPSON, *The UNIX Time-Sharing System*, **http://cm.bell-labs.com/who/dmr/cacm.html**, 2013

[8] WIKIPEDIA, *C11 (C standard revision)*, **http://en.wikipedia.org/wiki/C11_(C_standard_revision)**, 2013

[9] Telecom-Paris Tech BCI Informatique, *Gestion de la mémoire*, **http://www.infres.enst.fr/~dupouy/pdf/BCI/6-MemBCI-07.pdf**, 2013

[10] WIKIPEDIA, Endianness, **http://fr.wikipedia.org/wiki/Endianness**, 2013

[11] ISO / IEC, *Programming languages – C,* **http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf, 2007**

[12] P. PRINTZ, T. CRAWFORD, *C in a Nutshell*, O'Reilly, 2006

[13] WIKIPEDIA, Machine code, **http://en.wikipedia.org/wiki/Machine_code**, 2013

[14] Y. METTIER, *C en action*, O'Reilly, 2005

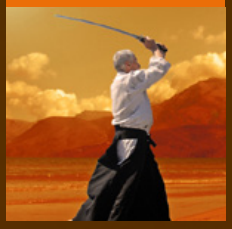
[15] WIKIPEDIA, *TMG (language)*, **http://en.wikipedia.org/wiki/TMG_(language)**, 2013

INDEX

2

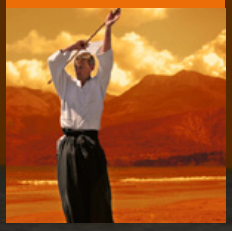
2 LA PRATIQUE DU C

À découvrir dans cette partie...



2.1 Un peu plus loin avec les macros

Définition et bonne utilisation des macros, structures qui s'avèrent très pratiques du fait qu'elles simplifient grandement le code et ce, dans n'importe quel langage de programmation. p. 32



2.2 Static, switch, et cætera

Présentation d'une utilisation particulière de la condition « switch », le Duff'Device, ainsi que d'autres astuces pour simplifier votre code, à savoir l'utilisation de variables « static » et de co-routines. p. 38

2.3 Les règles d'aliasing strictes

Ces règles permettent aux compilateurs d'effectuer certaines optimisations du code. Cet article propose un exemple concret de problèmes d'aliasing et des différentes solutions qui s'imposent. p. 46

2.4 Jouons avec les bits

Exemples d'utilisation des opérateurs de manipulation de bits du langage C et focus sur le cas particulier des puissances de 2, des tests de parité, et des racines carrées inverses. p. 52

UN PEU PLUS LOIN AVEC LES MACROS

David Odin

Même si elles ne font pas vraiment partie du langage C à proprement parler, puisqu'elles appartiennent au préprocesseur, les macros sont des outils qui peuvent s'avérer fort pratiques.

Une macro est l'association entre un symbole (un nom) et une définition. Elle se définit ainsi :

Fichier
```
#define NOM definition
```

où **NOM** est n'importe quel identifiant valide en C (même un mot réservé comme **for**, **return**), mais on préfère souvent se limiter à des majuscules éventuellement séparées par des *underscores*. **definition** peut contenir n'importe quelle séquence (éventuellement vide) de code C valide.

Les macros dont la définition est vide sont surtout utilisées pour permettre l'inclusion optionnelle de code via l'utilisation des primitives **#ifdef** / **#ifndef**.

Lorsque le préprocesseur effectue le remplacement des macros par leur définition, il effectue pratiquement un « chercher / remplacer » du texte à quelques rares exceptions près : les éventuels paramètres de la macro, et les opérateurs spéciaux **#** et **##**.

1. LES MACROS PRÉDÉFINIES

Avant même la première ligne de nos programmes C, un certain nombre de macros sont prédéfinies. On peut en avoir la liste en utilisant le flag **-dM** (pour *Dump Macros*) de **gcc** ainsi :

Terminal
```
gcc -x c /dev/null -dM -E
```

Le **-x c** signifie que l'on veut compiler du C, car **/dev/null** n'a pas d'extension, et le **-E** demande de ne lancer ni la compilation, ni l'édition de lien.

Cette commande donne un résultat ressemblant à ceci :

Fichier
```
#define __DBL_MIN_EXP__ (-1021)
#define __UINT_LEAST16_MAX__ 65535
#define __ATOMIC_ACQUIRE 2
#define __FLT_MIN__ 1.17549435082228750797e-38F
#define __UINT_LEAST8_TYPE__ unsigned char
#define __INTMAX_C(c) c ## L
#define __CHAR_BIT__ 8
[ ... ]
#define __GNUC_GNU_INLINE__ 1
#define __GCC_ATOMIC_SHORT_LOCK_FREE 2
#define __UINT_FAST8_TYPE__ unsigned char
#define __ATOMIC_ACQ_REL 4
#define __ATOMIC_RELEASE 3
```

Vous remarquerez que la plupart de ces macros commencent par deux underscores. Ce genre de symbole est réservé à l'implémentation du compilateur et de la bibliothèque C standard. Il est donc fortement déconseillé d'utiliser des noms de variables, de types ou de fonctions commençant par **__**.

En plus de tout cela, certaines macros prédéfinies sont spéciales et dépendent de l'endroit ou du temps où elles sont utilisées :

→ **__FILE__** est un chaîne de caractères contenant le nom du fichier en cours de compilation (**"fichier.c"** par exemple) ;

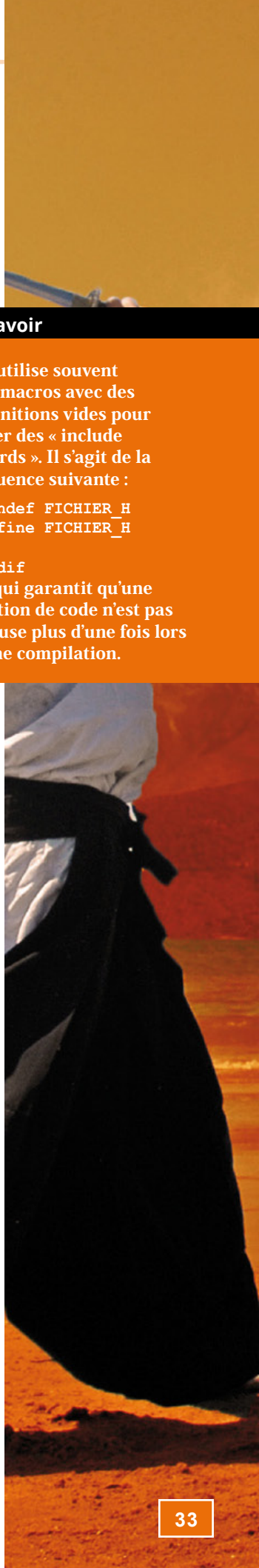
→ **__LINE__** est un entier contenant le numéro de la ligne en cours de compilation dans le fichier courant (pratique pour le debug) ;

À savoir

On utilise souvent des macros avec des définitions vides pour créer des « include guards ». Il s'agit de la séquence suivante :

```
#ifndef FICHIER_H
#define FICHIER_H
...
#endif
```

Ce qui garantit qu'une portion de code n'est pas incluse plus d'une fois lors d'une compilation.



À savoir

Lorsqu'on utilise des macros avec paramètres, on place généralement des parenthèses autour de chaque paramètre dans la partie définition de la macro, afin d'éviter des effets de bord.

→ **__TIME__** est une chaîne de caractères contenant l'heure de la compilation. Par exemple : **"13:37:42"** ;

→ **__DATE__** est une chaîne de caractères contenant la date de la compilation, au format américain. Par exemple : **"Aug 18 2008"**.

2. LES MACROS AVEC PARAMÈTRES

Mais la partie la plus intéressante des macros est qu'elles peuvent prendre des paramètres. Pour cela, il faut impérativement que lors de la déclaration le nom de la macro soit *immédiatement* suivi d'une parenthèse ouvrante.

Chaque paramètre est un identifiant, et chaque occurrence de cet identifiant dans la définition sera remplacée textuellement lors de l'utilisation de la macro. Lors de l'appel de la macro, on peut utiliser n'importe quelle séquence de code C (valide ou non) pour chaque paramètre. Par exemple, si l'on considère la macro suivante :

Fichier
```
#define MACRO(a,b) a+b
```

La séquence qui suit :

Fichier
```
 int a = MACRO(1,2);
 int b = MACRO(1;int c=,2);
```

Le préprocesseur la transformera en ceci :

Fichier
```
int a = 1 +2;
int b = 1;int c=+2;
```

3. # ET

Lorsqu'un **#** précède le nom d'un paramètre dans la définition d'une macro, celui-ci est remplacé par une chaîne de caractères le représentant. Par exemple, le code suivant :

Fichier
```
#define MACRO(a) puts(#a)
MACRO(foo);
```

sera transformé ainsi : **puts("foo");**

Cela peut être très pratique pour créer une macro affichant le contenu d'une variable :

Fichier
```
#define SHOW(a) printf("%s:%d" #a " = %d\n", __FILE__, __LINE__, a);
int foo = 5, bar = 7;
SHOW(foo);
SHOW(bar);
```

qui affichera quelque chose comme ceci :

fichier.c:12 foo = 5 fichier.c:13 bar = 7

Un autre opérateur que l'on ne peut utiliser que dans une définition de macro est **##**. Cet opérateur se place entre deux identifiants pour en créer un autre qui est la concaténation de ces deux identifiants. Ainsi, **a##b** devient **ab**. Cela est évidemment nettement plus intéressant lorsque l'un des deux identifiants est un paramètre. Voyons ça dans un exemple :

Fichier

```
#define MACRO(num) case num: i##num = 0; break;
int i1, i2, i3;
...
switch (foo)
{
  MACRO(1)
  MACRO(2)
  MACRO(3)
}
```

MACRO(1) sera remplacé par **case 1: i1 = 0; break;**. On imagine bien les possibilités de tout cela !

4. LES MACROS MULTI-LIGNES

Nous venons de voir qu'une macro permet de remplacer un morceau de code C par un nom et des paramètres éventuels. Il arrive donc assez souvent que le code que l'on remplace soit composé de plusieurs *statements*.

On peut par exemple définir une telle macro de la façon suivante :

Fichier

```
#define MACRO foo(); bar();
ou :
#define MACRO foo(); \
              bar();
```

Mais on aime pouvoir utiliser une macro un peu comme une fonction, surtout lorsqu'il s'agit d'une macro avec paramètres. Et si on essaie d'utiliser la macro précédente ainsi :

Fichier

```
if (a == 1)
  MACRO;
```

On a un problème. En effet, cela est équivalent à :

Fichier

```
if (a == 1)
  foo();
bar();
```

Une première solution serait d'obliger à ajouter des accolades à chaque utilisation de la MACRO, ce qui n'est pas très satisfaisant et rend l'utilisation de la macro peu sûre. Une autre solution serait d'intégrer les accolades directement dans la définition de la macro :

Fichier

```
#define MACRO { \
                foo(); \
                bar(); \
              }
```


Mais on aurait alors un problème avec l'utilisation suivante :

Fichier

```
if (a == 1)
  MACRO;
else
  baz();
```

Là, le problème est plus subtil. Le point-virgule après **MACRO**, qui semble bien naturel en fin d'expression, provoque une erreur de syntaxe avant le **else**. Il faudrait alors absolument utiliser la MACRO sans mettre de point-virgule, mais ce n'est évidemment pas idéal.

La solution serait donc d'avoir une construction regroupant plusieurs expressions (un bloc), mais qui se comporterait comme une expression unique. Parmi les constructions proposées par le langage C, seule **do { ... } while (condition);** permet cela.

QUICK BONUS : COMPLÉTEZ PRINTF()

Les spécificateurs de format ou *format specifiers* sont ces éléments (%) permettant, comme leur nom l'indique, de spécifier la manière de formater une donnée avec certaines fonctions comme **scanf()** et **printf()**. Cette dernière, rappelons-le, est utilisée pour transférer du texte, des valeurs ou des résultats d'expressions sur la sortie standard **STDOUT**.

Il existe une douzaine de spécificateurs de format permettant ainsi d'afficher entiers, chaînes, caractères, doubles, etc. On peut ainsi, par exemple, afficher une valeur en décimal, hexadécimal et même octal... Mais pas en binaire ! Chose fort déplaisante lorsqu'on développe des programmes accédant à des données très spécifiques ou à du matériel par exemple (activation/désactivation de GPIO entre autres choses). En effet, en binaire, on joue avec les masques, on décale des valeurs, on teste des bits, etc. Et tantôt, on veut aussi afficher les bits. Le copié/collé d'une valeur en hexa ou octal dans un outil comme **bc** (en usant d'un **ibase** ou d'un **obase**) est une solution, mais il y a plus pratique : intégrer directement cela dans **printf()**.

Pour ce faire, nous devons ajouter le spécificateur de format manquant et embarquer tout cela dans notre source. L'exercice est amusant mais, car non couvert par une quelconque norme, il est à double tranchant. Cela vous permettra de vous simplifier la vie, mais rendra votre code moins portable. Il conviendra alors de mettre en place quelques macros bien pensées pour tester l'environnement de développement ou, tout simplement, pour retirer/désactiver tout cela une fois le code terminé et débogué.

Une sympathique et simple fonction de la GNU libc, **register_printf_function()**, existait mais est maintenant obsolète (*deprecated*). Il faut donc se tourner vers **register_printf_specifier()** et procéder ainsi :

Fichier

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <printf.h>

static int print_widget (FILE *stream, const struct printf_info *info,
const void *const *args) {
  int len;
  unsigned char c;
  unsigned char mask=128;

  c = *(unsigned char *) (args[0]);
```

Cette construction est normalement utilisée pour effectuer des boucles, mais si l'on s'arrange pour que le test soit faux (0), le contenu du bloc sera exécuté exactement une fois. Voilà pourquoi on écrit souvent les macros multi-lignes de la manière suivante :

Fichier

```
#define MACRO do { \
                   foo(); \
                   bar(); \
                 } \
              while (0);
```

Cela garantit que l'on pourra utiliser **MACRO** avec un point-virgule à la fin (le compilateur nous indiquera qu'il manque si on l'oublie), que l'on ne sera pas obligé de placer **MACRO** entre accolades et que tout se passera bien en cas de **if (...) MACRO; else** … . ■

Fichier

```
  while(mask>0) {
    len += fprintf (stream, "%d", (c & mask)/mask);
    mask = mask >> 1;
  }

  return len;
}

static int print_widget_arginfo (const struct printf_info *info, size_t n,
int *argtypes, int *size) {
  if (n > 0)
    argtypes[0] = PA_CHAR;
  return 1;
}

int main (int argc, char **argv) {
  register_printf_specifier('b', print_widget, print_widget_arginfo);

  printf("essai: 0b%b / 0x%x /%d \n",'a','a','a');

  return 0;
}
```

La fonction prend en arguments le spécificateur de format souhaité, un pointeur sur la fonction d'affichage et un autre sur la fonction de traitement des arguments. Celle-ci est utile pour déterminer combien d'arguments prend le spécificateur et quels sont leurs types. Ici, nous ne souhaitons traiter qu'un seul argument et il doit être de type **char**. Notre fonction **print_widget()** est plus intéressante, ne serait-ce qu'en raison de **%**args**. Il s'agit, en effet, d'un pointeur sur un tableau de pointeurs désignant, dans notre cas précis, un **char** (c'est là typiquement le genre de choses qui font radicalement aimer ou détester le C). Enfin, notez l'utilisation de **fprintf** et en particulier de **FILE *stream**, puisque nous recevons peut-être un simple **char** mais nous devons retourner 8 (**mask=128**) caractères dans le flux.

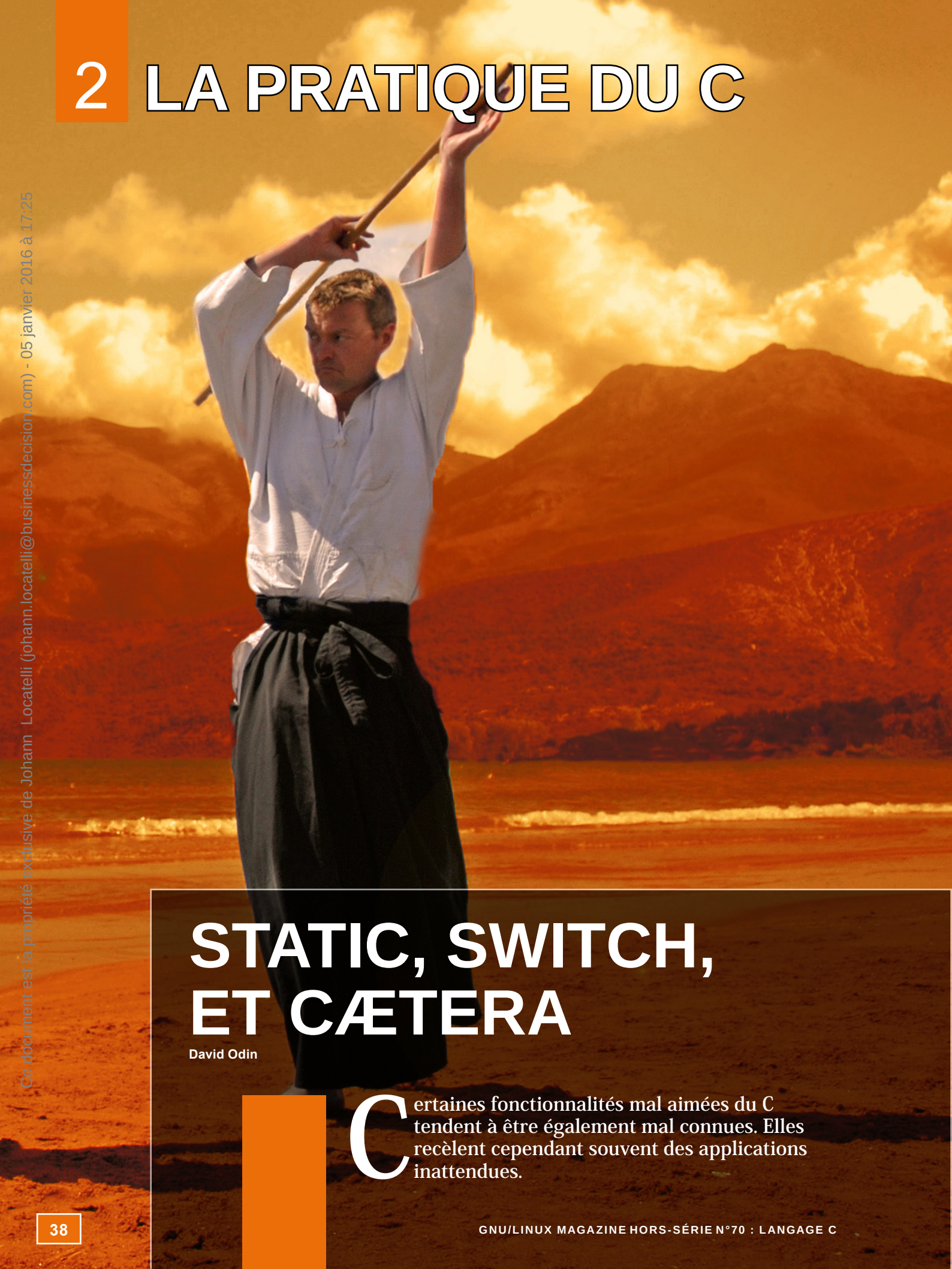
On compile et on teste :

Terminal

```
% make
gcc -O2 -Wall    -c -o monprintf.o monprintf.c
monprintf.c: In function 'main':
monprintf.c:111:3: warning: unknown conversion type character 'b' in format
monprintf.c:111:3: warning: too many arguments for format
gcc   monprintf.o   -o monprintf
% ./monprintf
essai: 0b01100001 / 0x61 / 97
```

Et voilà ! ■

Alex Krycek



STATIC, SWITCH, ET CÆTERA

David Odin

Certaines fonctionnalités mal aimées du C tendent à être également mal connues. Elles recèlent cependant souvent des applications inattendues.

1. UN SWITCH REVISITÉ

La grammaire du C est volontairement la plus simple possible. Ainsi, beaucoup d'éléments lexicaux sont repris dans plusieurs constructions du langage. Par exemple, dans la représentation BNF **[1]** de la grammaire, on retrouve **stat** un grand nombre de fois. Ce symbole représente un *statement* qui est soit une expression, soit un label suivi d'un statement, soit une sélection (**if**, **switch**), soit une itération (**for**, **do**, **while**), soit un bloc (**{ }**), soit un saut (**goto**, **continue**, **return**, ...).

Cela permet ensuite de définir l'une des formes du **if** (celle sans le **else**) ainsi :

Fichier
```
if (expr) stat
```

Ce qui indique que toutes les utilisations suivantes sont correctes grammaticalement parlant (certaines formes ne sont pas particulièrement recommandées pour écrire du code lisible !) :

Fichier
```
if (a == 1) a = 2;
if (a == 1) label: a = 2;
if (a == 1) if (b == 1) a = 0;
if (a == 1) while (a < 20) a = f(a);
if (a == 1) { a = 2; }
if (a == 1) return 5;
```

De la même façon, l'une des huit (!) formes de l'itération **for** est la suivante :

Fichier
```
for (exp ; exp ; exp ) stat
```

Là encore, **stat** peut avoir n'importe laquelle des formes que l'on vient de voir, même si certaines n'ont pas vraiment de sens. On peut évidemment toujours utiliser la forme par bloc, et cela est même quelques fois imposé par des guides de codage. On rencontre toutefois assez fréquemment les autres formes.

En revanche, pour **switch**, il est très rare que l'on s'écarte de la forme classique :

Fichier
```
switch (valeur)
{
  case 1:
    ...
    break;
  case 2:
    ...
    break;
  ...
  default:
    ...
}
```

Pourtant, les **break** n'ont aucune obligation d'être là, et il arrive d'ailleurs de voir deux **case :** à la suite.

De plus, rien n'oblige le programmeur C à utiliser un **stat** de type bloc. La seule restriction est bien évidemment que les **case :** relatifs à ce **switch** soient tous dans le **stat** correspondant. Par exemple, si **a** est une variable entière, l'extrait suivant est parfaitement valide (mais pas très intéressant) :

Fichier

```
switch (a)case 1:return;
```

Bien évidemment, pour qu'un **switch** soit intéressant, il faut avoir plusieurs **case :**, avec du code différent pour chacun. Cela implique d'avoir au moins un bloc dans le **stat**. Mais il n'y a pas vraiment de restriction particulière pour le placement des étiquettes **case :** dans ce bloc.

Voici un exemple (un peu caricatural il est vrai), tiré de la liste de diffusion du développement de **gcc** [2] :

Fichier

```
switch (x)
  default:
    if (prime(x))
      case 2: case 3: case 5: case 7:
        process_prime(x);
    else
      case 4: case 6: case 8: case 9: case 10:
        process_composite(x);
```

Dans cet exemple, si **x** vaut 2, 3, 5 ou 7, la fonction **process_prime(x)** est directement appelée. S'il vaut 4, 6, 8, 9 ou 10, c'est la fonction **process_composite(x)** qui est directement appelée. Sinon, on arrive dans la clause **default:**, et le test **if (prime(x))** est effectué. Le **switch** permet donc de spécialiser 9 cas, sans passer par le **if**. Ce code est fonctionnellement équivalent au suivant :

Fichier

```
switch (x)
{
  case 2: case 3: case 5: case 7:
    process_prime(x);
    break;
  case 4: case 6: case 8: case 9: case 10:
    process_composite(x);
    break;
  default:
    if (prime(x))
      process_prime(x);
    else
      process_composite(x);
}
```

Cet exemple semble suggérer qu'écrire les **switch** n'a pour seul intérêt que de rendre le code plus difficile à lire. Il y a cependant des cas où les constructions curieuses de **switch** ne peuvent pas être réécrites aussi facilement. Nous allons en voir un exemple dans la section suivante.

2. LE DUFF'S DEVICE

Non, il ne s'agit pas d'une machine à fabriquer de la légendaire bière dont il est question dans les Simpsons. C'est une utilisation assez particulière de **switch** proposée la première fois par Tom Duff [3].

Tom Duff était programmeur chez Lucasfilm, et on lui a confié l'optimisation d'un code assez particulier. Il devait recopier dans l'ordre tous les octets d'une zone mémoire à un seul emplacement mémoire (qui était en fait l'adresse d'un périphérique). Le code de départ ressemblait donc à ceci :

Fichier

```
int count;
char *to = device_address;
char *from = memory;
for (count = 0; count < nb_bytes; count++)
  *to = *from++;
```

Source : Wikimedia Commons, Auteur : Tom Duff, Licence : CC BY-SA 3.0

► *Tom Duff, dans son bureau des studios Pixar (2006)*

Le périphérique en question était capable de réagir plus rapidement que le programme ne lui donnait des informations. Tom Duff eut l'idée de déplier la boucle d'un facteur 8, afin d'économiser 87 % des incrémentations et des tests de **count**.

Cependant, la taille de la zone mémoire à copier n'était pas toujours un multiple de 8. Il y avait donc un reste d'au plus 7 octets. Il aurait pu ensuite ajouter une boucle pour ce reste, mais il a préféré une autre voie : « Et s'il était possible de sortir ou d'entrer dans un morceau de code, en plein milieu ? ».

Pour sortir d'un code en cours de route, il faut ajouter un test afin de savoir si l'on a fini et ce, à chaque étape. C'est précisément ce que Tom Duff voulait éviter. Il ne reste donc plus que la possibilité d'entrer dans le code non pas au début, mais exactement où l'on veut. Et cela est justement possible à l'aide d'un **switch** avec la construction suivante :

Fichier

```
switch (count % 8)
  case 0: do { *to = *from++;
  case 7:      *to = *from++;
  case 6:      *to = *from++;
  case 5:      *to = *from++;
```

Fichier

```
    case 4:       *to = *from++;
    case 3:       *to = *from++;
    case 2:       *to = *from++;
    case 1:       *to = *from++;
                } while ((count -= 8) > 0);
```

C'est ce morceau de code (et par extension tout code qui utilise ce genre d'astuce) que l'on appelle maintenant un « Duff's Device ». La valeur de test du **switch** calcule la valeur du reste, puis un saut est fait à l'intérieur de la boucle **do / while**. Par exemple, si le reste vaut 5, on sautera au **case 5:**, puis 5 affectations ***to=*from++;** seront faites avant d'arriver une première fois sur la ligne du **while**. Les autres passages dans cette boucle commenceront évidemment au niveau du **do {** et effectueront bien 8 affectations.

3. LES VARIABLES « STATIC »

Le mot-clef **static** permet d'indiquer que la portée d'une fonction ou d'une variable est limitée à une compilation (et n'est donc pas utilisable par d'autres fichiers C). Une variable **static** est cependant globale au fichier en cours de compilation.

Les variables **static** locales à une fonction ont la restriction supplémentaire de n'être visibles que dans cette fonction, mais le fait d'être dans une fonction ne leur enlève pas leur caractère global. Elles sont initialisées au lancement du programme seulement. Le contenu de ces variables est donc conservé entre deux appels à la fonction.

Cela va nous permettre de créer des fonctions dont l'action continue d'appel en appel. Voyons par exemple la fonction suivante :

Fichier

```
int f(void)
{
  static int i = 0;
  i++;
  return i;
}
```

Cette fonction renvoie une valeur différente à chaque fois qu'on l'appelle. C'est bien pratique dans certains cas, par exemple lorsque l'on veut créer des identifiants uniques. Mais, dès que l'on veut faire autre chose qu'une simple incrémentation sur la variable, l'écriture de la fonction devient assez complexe. On aimerait pouvoir faire quelque chose du genre :

Fichier

```
int f(void)
{
  int tmp;
  static int val1 = 0, val2 = 1;
  static int i;
  for (i = 1; i < 10; i++)
```

Fichier

```
  {
    return val1 + val2;
    tmp = val1;
    val1 = val2;
    val2 = tmp + val2;
  }
}
```

Et que cela nous renvoie successivement les éléments de la suite de Fibonacci (1, 1, 2, 3, 5, 8, 13, etc.), ou n'importe quelle fonction plus complexe.

Malheureusement, cela n'est pas possible puisque le mot-clef **return** met fin à la fonction et qu'elle recommencera du début au prochain appel. Effectivement, on pourrait modifier la fonction pour arriver à nos fins, mais cela la rendrait inutilement complexe.

Pour arriver à avoir le meilleur des deux mondes, nous allons dans un premier temps complexifier notre fonction en utilisant un peu la même idée que pour le Duff's Device :

Fichier

```
int f(void)
{
  int tmp;
  static int val1 = 0, val2 = 1;
  static int i;
  static int state = 0;
  switch(state) {
    case 0:
      for (i = 1; i < 10; i++)
      {
        state = 1; return val1 + val2; case 1:;
        tmp = val1;
        val1 = val2;
        val2 = tmp + val2;
      }
  }
  return -1;
}
```

On a ajouté une variable **static** nommée **state** pour savoir si on est au premier appel de cette fonction ou pas. **state** vaut 0 uniquement au premier appel, et dans ce cas, on commence la fonction normalement. Lors des appels ultérieurs, **state** vaut 1, et on saute directement au cœur de la boucle, juste là où on avait quitté la fonction à l'appel précédent.

En ajoutant quelques macros, on arrive à une version de notre fonction assez lisible et satisfaisante :

Fichier

```
#define BEGIN static int state=0; switch(state) { case 0:
#define RETURN(x) do { state = 1; return x; case 1:; } while (0)
#define FINISH }

int f(void)
{
  int tmp;
```

Fichier

```
  static int val1 = 0, val2 = 1;
  static int i;
  BEGIN;
  for (i = 1; i < 10; i++)
  {
    RETURN(val1 + val2);
    tmp = val1;
    val1 = val2;
    val2 = tmp + val2;
  }
  FINISH;
  return -1;
}
```

4. LES CO-ROUTINES

Une routine est un morceau de code réalisant une tâche donnée. Ce nom assez ancien provient plutôt du monde de l'assembleur. Dans le monde du C, on peut associer cela à la notion de fonction. Cela autorise éventuellement plusieurs points de sortie (avec différents **`return`**), mais toujours un seul point d'entrée (le début de la fonction).

Les co-routines (qui vont souvent par deux, l'une appelant l'autre) admettent également plusieurs points d'entrée. Plus exactement, la fonction appelée est capable de renvoyer une valeur à la fonction appelante, puis de continuer son traitement. Dans un contexte *mono-thread*, cela implique que la fonction appelée sera capable de reprendre là où elle en était lors d'un prochain appel.

L'utilisation des macros décrites permet à la fonction **`f()`** présentée précédemment de s'approcher de cette idée. Ceci dit, **`f()`** n'a que deux points d'entrée : le tout début et l'emplacement de **`RETURN()`**.

Dans une co-routine, on devrait pouvoir placer autant de points de sortie/d'entrée que voulu. Par exemple, dans ce qui suit, on désire que la fonction **`coordinate()`** renvoie l'abscisse d'un point qui décrit un carré, dont les coordonnées des coins opposés sont (5, 5) et (10, 10). Il n'est pas facile de décrire cela avec une seule boucle, puisque l'on a 4 côtés avec 4 comportements différents. La première boucle renvoie successivement 5, 6, 7, 8, 9, et 10, la seconde renvoie 6 fois la valeur 10, etc.

Fichier

```
int coordinate()
{
  static int i;
  BEGIN;
  while (1)
  {
    for (i = 5; i <= 10; i++)
      RETURN(i);
    for (i = 0; i <= 5; i++)
      RETURN(10);
    for (i = 10; i >= 5; i--)
      RETURN(i);
```

Fichier

```
    for (i = 0; i <= 5; i++)
      RETURN(5);
  }
  FINISH;

  return -1;
}
```

Cependant, l'implémentation de la macro `RETURN()` ne permet pas cela, et si l'on essaie tout de même, le compilateur se plaint que le label `case 1:` est défini plusieurs fois.

Il faut donc redéfinir cette macro de sorte qu'elle utilise une valeur différente à chaque fois qu'on l'appelle. Cela est possible à l'aide de la valeur spéciale `__LINE__`, de cette manière :

Fichier

```
#define RETURN(x) do { state=__LINE__; return x; case __LINE__:; } while (0)
```

En utilisant cette définition, la fonction `coordinate()` non seulement compile, mais se comporte comme prévu. En effet, si on l'appelle cinquante fois de suite, elle renvoie bien la séquence suivante : `5, 6, 7, 8, 9, 10, 10, 10, 10, 10, 10, 10, 10, 9, 8, 7, 6, 5, 5, 5, 5, 5, 5, 5, 5, 6, 7, 8, 9, 10, 10, 10, 10, 10, 10, 10, 10, 10, 9, 8, 7, 6, 5, 5, 5, 5, 5, 5, 5, 5, 6`.

Ainsi, en quelques macros et via l'utilisation des variables `static`, on a pu ajouter une fonctionnalité au langage C : la possibilité de continuer l'exécution d'une fonction là où elle était après un `return`.

5. LIMITATIONS

Cette implémentation des co-routines souffre de quelques problèmes, qui ne sont normalement pas très gênants :

→ Il faut impérativement mettre en `static` toutes les variables qui doivent garder leur valeur entre deux appels,

→ Il est impossible d'utiliser la macro `RETURN()` dans un `switch`,

→ Cette implémentation n'est pas *thread-safe*, mais les environnements multi-threads offrent d'autres possibilités pour faire ce genre de choses. ■

RÉFÉRENCES

[1] **http://www.cs.man.ac.uk/~pjj/bnf/c_syntax.bnf**

[2] **http://gcc.gnu.org/lists.html**

[3] **http://www.lysator.liu.se/c/duffs-device.html**

[4] **http://en.wikipedia.org/wiki/Coroutine**

LES RÈGLES D'ALIASING STRICTES

David Odin

Lors d'un développement, vous êtes peut-être déjà tombé sur le message d'erreur incompréhensible suivant : « Dereferencing type-punned pointer will break strict-aliasing rules ». Ce qui pourrait se traduire par « Déréférencer un pointeur pointant sur deux types différents rompt les règles d'aliasing strictes ». Ce qui n'est toujours pas très clair.

1. QU'EST-CE QU'UN « TYPE-PUNNED POINTER » ?

En C, un pointeur contient généralement l'adresse d'un objet (variable ou autre) d'un type donné. Le type `void *` (et `void`) a été introduit dans le standard de 1989 afin de disposer d'un pointeur *générique*. On peut sans problème (et sans *warning*) transformer un pointeur de n'importe quel type vers et depuis un pointeur `void *`. Mais on ne peut pas déréférencer un pointeur `void *`.

À l'aide d'un *casting* (transtypage en français...), on peut également transformer n'importe quel pointeur en un autre. Par exemple :

Fichier

```
int a;
int *p = &a;
short *q = (short*)p;
```

Jusqu'ici, il s'agit de code parfaitement valide. Cependant, `p` et `q` sont deux pointeurs de types différents, pointant sur la même adresse. `q` est un « type-punned pointer », un pointeur qui n'a pas le bon type pour pointer ce sur quoi il pointe. Comme `p` et `q` pointent sur la même adresse, on dit qu'ils sont des alias l'un de l'autre.

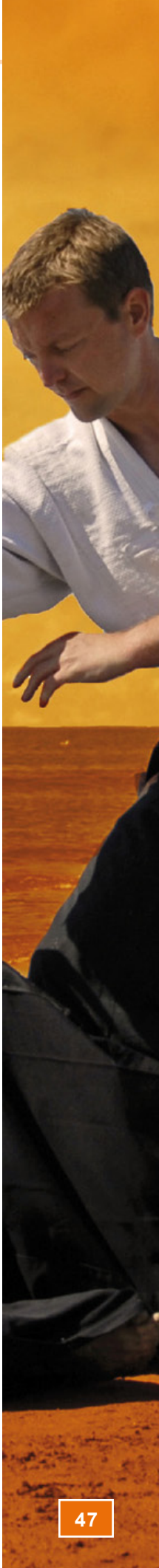
2. QUE SONT CES RÈGLES D'ALIASING STRICTES ?

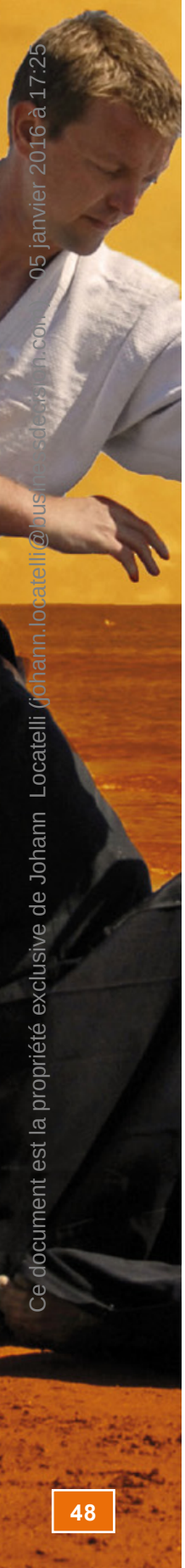
Ces règles sont apparues dans la version C89 du standard C et ont été affinées dans la version C99. Et elles sont là pour permettre aux compilateurs de faire certaines optimisations. C'est pourquoi `gcc` ne les utilise pas par défaut. On peut les activer à l'aide de l'option de compilation `-fstrict-aliasing`. Cette option est automatiquement activée si on utilise un niveau d'optimisation `-O2`, `-O3`, ou `-Os`.

Les règles indiquent que le compilateur est en droit de supposer qu'un pointeur ne peut être l'alias d'un autre que si :

- → les types pointés sont les mêmes,
- → les types pointés ne diffèrent que par l'ajout/la suppression de `const` ou `volatile`,
- → les types pointés ne diffèrent que par l'ajout/la suppression de `signed` ou `unsigned`,
- → on a éventuellement un mélange des deux conditions précédentes,
- → un des types pointés est une union dont l'un des choix est l'autre type pointé, ou une des variations précédentes,
- → un des types pointés est une structure dont le premier champ est l'autre type pointé, ou une des variations précédentes,
- → on a l'application récursive des règles précédentes,
- → au moins l'un des deux types pointés est un type caractère (`char`, `signed char`, etc.).

Cela permet au compilateur d'optimiser du code en se disant qu'une valeur n'a pas pu être changée par des pointeurs qui n'ont pas le droit d'être un alias à l'adresse de cette valeur.





Par exemple, dans le code suivant :

Fichier

```
int a = 2;
short *p;
...
*p = 3;
return a;
```

En imaginant que rien ne change la valeur de **a** dans les **...**, le compilateur peut se dire que l'expression **`*p = 3;`** ne peut en aucun cas changer la valeur de **a** et remplacer la dernière ligne par **`return 2;`**. Évidemment, si à un moment on a fait quelque chose du genre :**`p=(short*)a;`**, on pourrait s'attendre à un autre résultat. Mais cela aurait rompu les règles d'aliasing strictes que l'on vient de présenter.

Dans ce cas-là, on pourrait espérer au moins avoir un *warning*, pour nous dire que l'on enfreint des règles pas toujours bien connues. Par défaut, **`gcc`** ne diagnostique aucun problème ici. Il existe une option, **`-Wstrict-aliasing`** qui est incluse automatiquement si on utilise l'option **`-Wall`**, dont le rôle est précisément de vérifier si l'on respecte bien toutes les règles d'aliasing.

Malheureusement, comme cette option a tendance à faire des faux positifs, il a été décidé de la rendre moins sensible par défaut. Elle a trois niveaux de détection (de 0 à 2) et celui par défaut est 0. Si l'on veut augmenter ce niveau, il faudra utiliser la syntaxe suivante : **`-Wstrict-aliasing=2`**.

3. UN EXEMPLE EN PARTICULIER

L'exemple précédent peut sembler assez artificiel, et l'on peut se dire que l'on ne fera jamais une telle chose. Voyons un exemple un peu plus concret dans le programme suivant :

Fichier

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

void my_copy(uint16_t *dest, const uint16_t *src)
{
  uint64_t *p_d = (uint64_t*)dest;
  const uint64_t *p_s = (uint64_t*)src;
  p_d[0] = p_s[0];
  p_d[1] = p_s[1];
  p_d[2] = p_s[2];
  p_d[3] = p_s[3];
}

int main()
{
  uint16_t a[]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
  uint16_t b[16]={0};
  int i;
 my_copy(b, a);
 printf("a = ");
  for(i = 0; i < 16; i++)
    printf("%d ", a[i]);
  puts("");
```

Fichier

```
    printf("b = ");
    for(i = 0; i < 16; i++)
      printf("%d ", b[i]);
    puts("");
    return 0;
}
```

L'idée de ce programme est de copier le tableau **a** dans le tableau **b**, en utilisant une fonction à nous : **my_copy()**, avant d'afficher le contenu des deux tableaux. Cette fonction essaye d'être optimisée en copiant les éléments 4 par 4, plutôt que 1 par 1.

Si l'on compile ce programme avec l'option **-O0** avec les compilateurs **gcc** de la version 4.4 à la version 4.8, il n'y a aucun warning, et l'on obtient bien le résultat escompté :

Terminal

```
a = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
b = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Le résultat ne change d'ailleurs pas si l'on ajoute les options **-Wall** et **-Wstrict-aliasing=2**.

En revanche, dès que l'on passe en **-O3**, on a des résultats nettement plus bizarres :

→ **gcc-4.4** affiche les warnings attendus : « warning: dereferencing pointer 'p_s' does break strict-aliasing rules » huit fois, et le programme affiche n'importe quoi :

Terminal

```
a = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
b = 2144 64 0 0 4 64 0 0 8 55790 32767 0 12 0 0 0
```

→ **gcc-4.5**, **gcc-4.6** et **gcc-4.7** ne montrent aucun warning et affichent le résultat attendu.

→ **gcc-4.8** n'affiche non plus aucun warning, mais le résultat est décevant :

Terminal

```
a = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
b = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Hormis l'absence d'avertissement, le comportement de **gcc** est pourtant valide quelle que soit la version. Le code proposé utilise des opérations effectuées via des pointeurs qui n'ont pas le droit d'être alias les uns des autres.

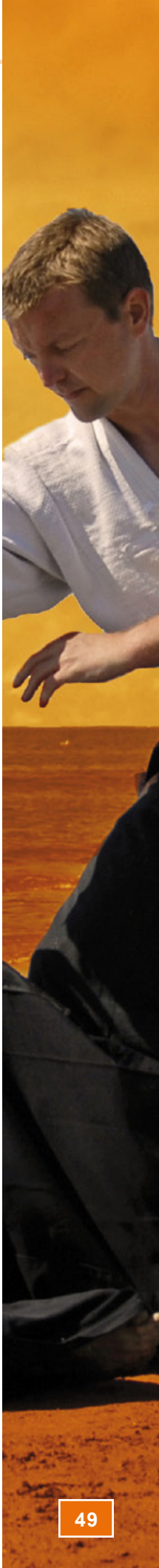
On peut alors se demander comment optimiser ce code.

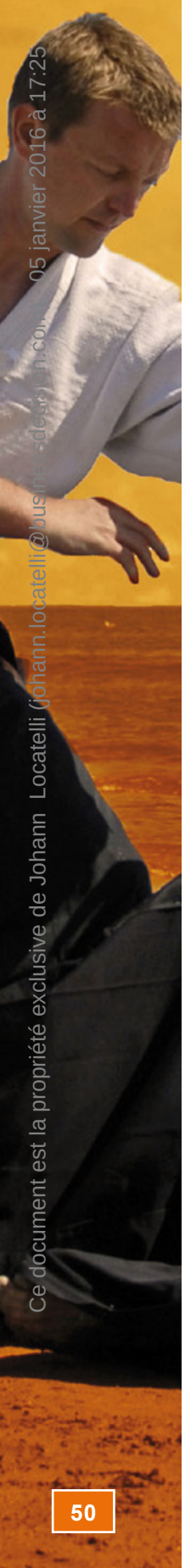
La première solution consiste à faire confiance au compilateur, tout simplement en utilisant la fonction **memcpy()** ainsi :

Fichier

```
void my_copy(uint16_t *dest, const uint16_t *src)
{
  memcpy(dest, src, 32);
}
```

Dans ce cas-là, toutes les versions testées de **gcc** se comportent de la même manière. Aucun warning et le résultat du programme est correct. De plus, le code assembleur généré à chaque fois est le suivant :





Fichier

```
my_copy:
    movq  (%rsi), %rax
    movq  %rax, (%rdi)
    movq  8(%rsi), %rax
    movq  %rax, 8(%rdi)
    movq  16(%rsi), %rax
    movq  %rax, 16(%rdi)
    movq  24(%rsi), %rax
    movq  %rax, 24(%rdi)
    ret
```

Ce qui est précisément ce qu'on essayait de faire avec le code précédent !

D'ailleurs, avec une implémentation à base d'une simple boucle **for** pour copier chaque élément, on obtient un code qui est parfois meilleur :

Fichier

```
    movdqu  (%rsi), %xmm0
    movdqu  %xmm0, (%rdi)
    movdqu  16(%rsi), %xmm0
    movdqu  %xmm0, 16(%rdi)
    ret
```

qui utilise des registres de 128 bits et est donc probablement plus rapide que notre implémentation initiale (qui est en plus buggée).

L'idée générale est donc de faire confiance au compilateur pour tout ce qui est micro-optimisation.

4. SOLUTIONS

Recherchons maintenant comment résoudre les problèmes d'aliasing qui se présentent.

Comme on vient de le voir, la solution la plus simple est bien évidemment de ne pas faire appel à ce genre de pseudo optimisation. Cependant, il peut y avoir des moments où cela semble inévitable. Pour ces cas-là, **gcc** propose des solutions particulières.

4.1 -fno-strict-aliasing

La première est de désactiver spécifiquement les optimisations liées à l'utilisation des règles d'aliasing strictes. Cela se fait à l'aide de l'option **-fno-strict-aliasing**, qu'il faut placer après **-O3** dans la ligne de compilation. Bien évidemment, ce n'est pas une solution idéale, puisque cela empêche le compilateur de trouver certaines solutions qui pourraient bien rendre notre programme plus rapide.

4.2 __attribute__((__may_alias__))

L'utilisation d'attributs permet un contrôle nettement plus fin. L'attribut **__may_alias__** permet d'indiquer que les pointeurs d'un type en particulier sont autorisés à être l'alias de n'importe quel autre. Les règles d'aliasing strictes ne s'appliquent donc pas pour les pointeurs sur ce type et uniquement ceux-là.

On aurait donc pu écrire notre fonction `my_copy()` ainsi :

Fichier

```
typedef uint64_t __attribute__((__may_alias__)) my_uint64_t;
void my_copy(uint16_t *dest, const uint16_t *src)
{
  my_uint64_t *p_d = (my_uint64_t*)dest;
  const my_uint64_t *p_s = (my_uint64_t*)src;
  p_d[0] = p_s[0];
  p_d[1] = p_s[1];
  p_d[2] = p_s[2];
  p_d[3] = p_s[3];
}
```

Il n'y a alors aucun warning, quelles que soient les options et les versions de `gcc` utilisées, le résultat est correct, et le code assembleur généré est semblable à celui généré par un `memcpy()`.

4.3 union

Le standard C laisse aux créateurs de compilateurs certains choix. Ce sont les « implementation defined behaviours » (comportements définis par l'implémentation). L'un de ces comportements est ce qu'il se passe si l'on écrit dans un membre d'une union avant de lire dans un autre. Les concepteurs de `gcc` ont choisi que cela permettrait de contourner les règles d'aliasing strictes. Par exemple, avec le code suivant :

Fichier

```
union
{
  uint16_t a16;
  uint64_t a64;
} a;

a.a64 = 0;
a.a16 = 5;
```

On a la garantie que `a.a64` sera égal à 5.

Cela n'a malheureusement pas beaucoup d'applications réelles.

POUR ALLER PLUS LOIN

Dans le même ordre d'idée que les règles d'aliasing strictes, le standard C99 introduit un nouveau mot-clef : `restrict`. Il s'agit d'un attribut que l'on peut donner à un pointeur, qui correspond à une promesse que l'on fait au compilateur. On indique par là que le pointeur en question est le seul à pointer là où il pointe. Cela permet d'aller au-delà des règles d'aliasing, en interdisant à tout autre pointeur (même s'il a le même type) de pointer au même endroit. Cela ouvre de nouvelles perspectives d'optimisation pour le compilateur [4]. ■

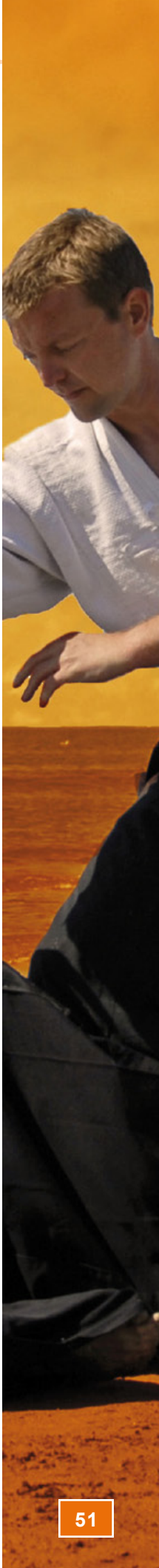
RÉFÉRENCES

[1] **http://gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html**

[2] **http://en.wikipedia.org/wiki/Aliasing_(computing)**

[3] **http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html**

[4] **http://en.wikipedia.org/wiki/Restrict**



JOUONS AVEC LES BITS

David Odin

Le langage C dispose d'opérateurs permettant de tester, modifier et effectuer des opérations au niveau du bit. Ces opérations sont souvent réservées à des utilisations restreintes, alors qu'elles sont ultra-rapides et ont des applications quelques fois insoupçonnées. Une restriction toutefois est que ces opérateurs ne s'appliquent qu'à des entiers (la plupart du temps non signés).

1. COMMENÇONS AVEC DES CHOSES SIMPLES

Les opérateurs de manipulation de bits du langage C sont les suivants :

→ `a & b` renvoie un entier dont les seuls bits à 1 sont ceux qui étaient à 1 à la fois dans `a` et `b`,

→ `a | b` renvoie un entier dont les seuls bits à 0 sont ceux qui étaient à 0 à la fois dans `a` et `b`,

→ `a ^ b` renvoie un entier dont les seuls bits à 1 sont ceux qui étaient différents dans `a` et `b`,

→ `a >> n` renvoie un entier dont la représentation binaire est celle de `a` décalée de `n` bits vers la droite. Cela revient à une division entière par 2^n,

→ `a << n` renvoie un entier dont la représentation binaire est celle de `a` décalée de `n` bits vers la gauche. Cela revient à une multiplication par 2^n,

→ `~a` renvoie un entier où tous les bits à 1 de `a` sont mis à 0 et inversement. On appelle cela le complément à 1, car `a+~a+1=0` (en négligeant la retenue). À ne pas confondre avec `-`, qui réalise un complément à 2. `-a` est donc en fait `~a+1`.

Une application immédiate de l'opérateur `&` est le test d'un bit en particulier. Par exemple, pour tester si un nombre est pair ou impair, il suffit de tester le bit de poids faible : `a & 1`. Si cette opération renvoie 1, c'est que `a` est impair. Plus généralement, on peut tester le `n`ième bit de `a` avec le test suivant : `a & (1<<n)`.

2. ÊTRE OU NE PAS ÊTRE UNE PUISSANCE DE DEUX

Il peut arriver que l'on ait envie de tester si un nombre est ou non une puissance de deux.

Un algorithme que l'on trouve parfois pour cela consiste à comparer ce nombre successivement à chaque puissance de deux : 1, 2, 4, 8, ..., 2^{31}. Il n'y a finalement pas beaucoup (32) de tests à effectuer.

Mais on peut aussi chercher quelles sont les particularités des puissances de deux. En effet, un tel nombre a forcément la représentation suivante en binaire : `000100`... avec un seul 1 et que des zéros. S'il s'agit de 2^N, il y a exactement N 0 à droite du 1. Et si l'on enlève 1 à ce nombre, la représentation binaire est composée de 0, puis exactement N 1. Il n'y a donc aucun 1 à la même place dans les représentations binaires de `a` et de `a - 1` lorsque `a` est une puissance de deux. Pour tous les autres nombres (sauf zéro), le 1 de poids fort est commun entre un nombre et son prédécesseur. L'opération `a & (a - 1)` (qui ne garde que les bits à 1 communs entre `a` et `a - 1`) produit donc un résultat nul que si `a` est une puissance de 2 ou zéro.

On peut donc éviter les 32 tests de la méthode classique et utiliser la fonction suivante :

Fichier

```
int is_power_of_two(unsigned a)
{
  return a && a & (a - 1);
}
```

3. PARITÉ

Si vous avez manipulé des liaisons séries, il y a des chances que vous ayez entendu parlé de la parité des données. Il s'agit d'une somme de contrôle très basique, puisqu'elle se contente de compter les bits mis à 1 dans une donnée (généralement un octet). Le bit de parité correspondant à la donnée est alors à 1 si le nombre de bits à 1 est impair, et à 0 s'il est pair. Cela permet tout de même de contrôler les erreurs de transmission qui n'affectent pas plus d'un bit.

Là encore, la méthode classique semble être de parcourir la donnée et de tester chacun des bits avec un code du genre :

Fichier

```
int parite(unsigned data)
{
  int par = 0, idx;
  for (idx = 0; idx < 32; idx++)
    if (data & (1 << idx))
      par++;
  return par & 1;
}
```

Brian Kernighan (le K du fameux livre « K&R » sur le C) a proposé une autre façon de faire cela, en utilisant la même astuce que précédemment : si dans le cas d'une puissance de deux, `a & a - 1` vaut zéro, dans le cas général, cette expression met à zéro le bit à 1 de poids le plus faible. Par exemple, si `a` vaut 10, soit b1010, `a - 1` vaut alors 9, soit b1001, et `a & (a - 1)` vaut b1000. Le 1 de poids faible a bien été transformé en un 0. Fort de cela, il a proposé une implémentation ressemblant à celle-ci :

Fichier

```
int parite(unsigned data)
{
  int par = 0;
  while (data != 0)
  {
    par++;
    data &= data - 1;
  }
  return par & 1;
}
```

Non seulement le nombre de passages dans la boucle passe du nombre total de bits de `data`, au nombre de 1 présents dans sa représentation binaire, mais on gagne également un test par itération.

Une autre façon de calculer la parité d'une donnée encore plus rapide consiste à créer une table contenant la parité pour toutes les données possibles. Cela est évidemment hors de propos si la donnée est sur 32 bits, mais cela est couramment utilisé pour les données de 8 bits. Et la nature même de la parité permet de découper une donnée en morceaux, de calculer la parité de chaque morceau et d'ajouter la parité des morceaux. Ainsi, la parité d'une donnée de 32 bits peut être calculée en utilisant 4 fois une table pour données de 8 bits. La méthode de Kernighan est cependant souvent aussi rapide que la méthode par tables.

4. LES RACINES CARRÉES INVERSES

En programmation 3D, on manipule des vecteurs 3D ayant trois composantes : x, y et z. Et pour les calculs d'illumination, on a souvent besoin de normer ces vecteurs (rendre leur taille égale à 1). Pour cela, il faut diviser chaque composante par la taille du vecteur, qui est donnée par **`sqrt(x*x+y*y+z*z)`**. Comme cette opération doit être répétée de nombreuses fois, le calcul de l'inverse d'une racine carrée est une opération que l'on cherche à optimiser au maximum. Aussi, dans le code de Quake 3 Arena, on en trouve l'implémentation suivante, qui est assez surprenante (mais extrêmement rapide) :

Fichier

```
float Q_rsqrt(float number)
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;
  i  = *(long *)&y;                       // evil floating point bit level hacking
  i  = 0x5f3759df - (i >> 1);             // what the fuck?
  y  = *(float *)&i;
  y  = y * (threehalfs - (x2 * y * y)); // 1st iteration
//y  = y * (threehalfs - (x2 * y * y)); // 2nd iteration, this can be removed

  return y;
}
```

À première vue, ce code est assez incompréhensible. En fait, deux techniques sont utilisées ici. La seconde est la méthode de Newton-Raphson.

Il s'agit d'une méthode itérative permettant de trouver rapidement la racine d'une équation du genre f(x)=0. Chaque itération permet de se rapprocher rapidement de la solution exacte : la valeur x pour laquelle f(x)=0, autrement dit la position sur l'axe des x à laquelle la fonction f coupe cet axe. L'idée est que pour une fonction continue, monotone, on peut approximer localement (en x_n) la fonction en une droite (tangente à la fonction). Cette droite coupe l'axe des x en une position x_{n+1}. x_{n+1} est alors plus proche de la solution exacte que x_n.

L'équation de la tangente T à f(x) en x_n est donnée par : $T(x)=f(x_n)+f'(x_n)*(x-x_n)$

À chaque itération, on cherche x_{n+1}, tel que $T(x_{n+A})=0$. Soit $x_{n+1}=x_n-f(x_n)/f'(x_n)$.

Dans notre cas, on cherche x tel que x = 1/sqrt(number), soit tel que $y = 1/number^2$, ou que $1/x^2-number = 0$. La fonction f est donc : $f(x)=1/x^2-number$, et sa dérivée est $f'(x) = -2/x^3$.

On a alors : $x_{n+1}=x_n-(1/x_n^2-number)/(-2/x_n^3)$. Soit $x_{n+1}=x_n+(x_n-number*x_n^3)/2$.

En réarrangeant, on a $x_{n+1}=x_n(1,5*x-number/2*x_n^2)$.

Avec les notations de la fonction, **`threehalfs`** vaut 1.5, **`x2`** vaut number/2. Et donc la ligne **`y = y * (threehalfs - (x2 * y * y));`** permet à **`y`** de se rapprocher de la solution (soit 1/sqrt(number)) à chaque étape. On peut remarquer que John Carmack (l'auteur de ce code et du reste de Quake 3 d'ailleurs) avait prévu de faire deux itérations, mais il a remarqué qu'une seule était vraiment nécessaire.

La méthode de Newton-Raphson est bien pratique, mais elle demande de partir d'une « bonne » première approximation. Cette première approximation est obtenue par la ligne très très bizarre : **`i = 0x5f3759df - (i >> 1);`**. L'explication complète de l'opération, et en

particulier de la constante `0x5f3759df`, prend une dizaine de pages [2]. On peut cependant commencer à comprendre le comportement de cela.

Pour commencer, il faut comprendre comment sont codés les `float`. Il s'agit d'une donnée sur 32 bits (ce qui explique que l'on peut la caster en entier long).

|s|e7|e6|e5|e4|e3|e2|e1|e0|m22|...|m0|

Le bit de poids fort (s) représente le signe, les 8 bits suivants (e7 à e0) sont un exposant E (à 127 près) et les 23 bits de poids faibles (m22 à m0) sont la partie fractionnaire de la mantisse M. La partie entière de cette mantisse est toujours 1.

Dans notre cas, étant donné que l'on calcule des racines carrées, le signe est toujours positif et le bit de poids faible est donc toujours 0. Un nombre `float` (positif) est représenté par $1.M*2^{E-127}$, M étant le nombre binaire |m22|..|m0|.

Par exemple, le nombre 16 a un signe +, une mantisse de 1.0 et un exposant de 4. Il est donc codé par un 0 pour le signe, 4+127=131 pour l'exposant et 0 pour la mantisse.

Par ailleurs, 1/sqrt(16) = 1/4, = $1.0*2^{-2}$ est codé par un 0 pour le signe, 0 pour la mantisse et -2+127=125 pour l'exposant. Plus généralement, lorsqu'un nombre est une puissance (positive ou négative) de 2, il s'écrit $1.0*2^n$, et la mantisse est alors nulle. En revanche, un nombre comme 1,5 (1 + 1/2) aura une mantisse dont m22 est à 1 (les autres chiffres restant à 0). On supposera par la suite que la mantisse n'a pas un rôle très important. Consultez [3] pour plus d'informations.

Prendre l'inverse d'un nombre revient à le mettre à la puissance -1 ($1/x = x^{-1}$). Prendre la racine carrée d'un nombre revient à le mettre à la puissance 1/2 ($sqrt(x) = x^{1/2}$). L'inverse de la racine carrée est donc une mise à l'exposant -1/2. Si un nombre est une puissance de 2 (avec une mantisse nulle dans notre cas), prendre l'inverse de la racine carrée revient à transformer 2^n en $(2^n)^{-1/2}$, ce qui est égal à $2^{-n/2}$. Donc, lorsque l'on prend l'inverse de la racine carrée d'un nombre, sa partie « exposant » est multipliée par -1/2.

Dans l'expression `0x5f3759df - (i >> 1)`, essayons de voir ce qui se passe au niveau de l'exposant. Nous avons vu qu'il s'agit des huit bits suivant le bit de poids fort dans l'entier `i` (e7 à e0). Le nombre dans ces bits (que je note `e`) est en fait l'exposant E, auquel on a ajouté 127. L'opération `i >> 1` décale tous les bits de `i` vers la droite. Les bits de la mantisse sont décalés (mais on a choisi de ne pas s'intéresser à eux). Le bit e0 se déplace dans m22, e1 dans e0, e2 dans e1, etc., et finalement, le bit de signe (qui vaut zéro) est copié dans e7. Cette opération effectue donc bien une division par 2 de l'exposant.

La fameuse constante `0x5f3759df` s'écrit en binaire ainsi : `0101 1111 0011 0111`.... Les bits qui nous intéressent (qui correspondent à ceux de l'exposant) sont les huit après le premier, soit `101 1111 0`. Ce qui vaut 190 en décimal. Donc, si on se restreint à l'exposant, l'opération `i = 0x5f3759df - (i >> 1)` est équivalente à `e = 190 - e/2`. Mais e est en fait E+127.

L'opération effectue en fait : `E+127 = 190 - (E+127)/2`, soit `E = 190 - E/2 - 127/2 - 127`, soit `E = 190 - 127 - 63 - E/2` et donc `E = -E/2`.

Cette ligne de code assez obscure permet donc de calculer la partie exposant de l'inverse de la racine carrée, en jouant directement avec sa représentation binaire !

Le reste des bits de la constante `0x5f3759df` est là pour minimiser l'erreur au niveau de la mantisse, qui ne vaut pas forcément 0. Cette partie-là est nettement plus complexe à justifier, je vous renvoie à [2] pour les détails. ▮

RÉFÉRENCES

[1] **http://fr.wikipedia.org/wiki/Méthode_de_Newton**

[2] **http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf**

[3] **http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html**

QUICK BONUS : PRÉ- ET POST-EXÉCUTION AUTOUR DE MAIN()

Le point d'entrée d'un programme en C est la fonction **main()**. Il existe cependant des situations où l'on souhaiterait procéder à un ensemble d'actions, soit avant, soit après **main()**. Ceci peut être utile, par exemple, lorsqu'il y a systématiquement des opérations à appliquer pour bien finir un programme. Attention, ce qui est décrit ici est à utiliser avec précaution, ce n'est EN AUCUN CAS, une solution pour sauvagement libérer des structures, de la mémoire ou toute autre chose du même genre. Ces solutions ne remplacent pas le fait de structurer correctement un programme sous prétexte qu'on dispose d'un « code de nettoyage » en sortie de programme.

Une fonction spécifique permet de faire ce type de choses simplement en fin d'exécution, ou plus exactement après la sortie de la fonction **main()**. C'est **atexit()** :

Fichier

```
void bye(void) {
  printf("Game Over\n");
}
[...]
/* dans main() */
  int i;
  i = atexit(bye);
  if (i != 0) {
    fprintf(stderr, "cannot set exit function\n");
    exit(EXIT_FAILURE);
  }
```

Une fois **atexit(bye)** utilisé avec succès, l'arrêt normal du programme (via **exit()** ou un **return** dans **main()**) déclenchera l'exécution de notre fonction **bye()**. Notez qu'il est possible de multiplier les appels à **atexit()** pour définir jusqu'à **ATEXIT_MAX** fonctions à lancer en fin d'exécution. On consultera **sysconf(_SC_ATEXIT_MAX)** pour obtenir la valeur actuellement définie. Certaines documentations parlent de **on_exit()**. Cette fonction est un héritage provenant de SunOS 4 et conservée (et doit être considérée) en tant que tel à titre historique.

Il n'existe pas de fonction **atstart()** ou **atbegin()**. Pour tout dire, il n'existe étrangement pas de fonction standard permettant d'enregistrer une fonction ou une pile de fonctions à exécuter au démarrage d'un programme (et pour cause, comment feriez-vous pour les « enregistrer » et surtout où, dans **main()** (sic) ?). Il existe plusieurs manières d'obtenir un effet similaire. L'une d'entre elles consiste à utiliser les attributs de fonctions. Ainsi, il nous suffit de prototyper nos fonctions ainsi :

Fichier

```
void f_start(void) __attribute__((constructor(101)));
void f_exit(void) __attribute__((destructor(101)));
```

Puis d'utiliser ensuite :

Fichier

```
void f_start(void) {
  printf("on start\n");
}
void f_exit(void) {
  printf("on exit\n");
}
```

Les attributs de fonction aident initialement le compilateur à optimiser les appels aux fonctions et éventuellement à aider à la mise au point de programmes. ATTENTION, ceci est une spécificité du C GNU et ne sera donc portable que dans une certaine mesure ! Il existe des attributs pour les fonctions, les variables et les types.

Procéder ainsi à l'exécution d'actions en début ou en fin de programme peut paraître très séduisant. L'un des problèmes qui se pose cependant, tient dans la manière de déterminer ce que vos fonctions devront ou non faire. La plupart du temps, ceci passe par l'utilisation d'une variable globale stockant un code d'état par exemple, ce qui sous-entend le besoin de définir le code avant chaque opération provoquant la fin du programme. C'est clairement non idéal. Évitez donc des choses comme **atexit(SDL_Quit)** qui, à terme, vous joueront plus de mauvais tours que vous n'en tirerez de bénéfices, en particulier si votre code doit fonctionner aussi bien sous GNU/Linux, Mac OS X et Windows, comme c'est bien souvent le cas lorsqu'on utilise la lib SDL (*Simple Directmedia Layer*). ■

Alex Krycek

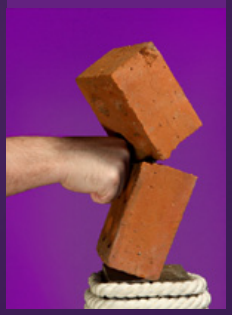
3

3 BIBLIOTHÈQUES ET TOOLKITS

À découvrir dans cette partie...

3.1 Initiation à la programmation graphique en C avec GTK+

GTK+ (pour GIMP ToolKit) est une « boîte à outils » multi-plateforme permettant de créer des interfaces graphiques utilisateur (GUI). Initialement conçu pour le logiciel GIMP, il supporte plusieurs langages, dont le C. Retrouvez ici la construction pas à pas d'une petite fenêtre à boutons. p. 60



3.2 Programmer avec les Enlightenment Foundation Libraries

Les EFL sont à la base du gestionnaire de fenêtres e17 et de Tizen, un système pour smartphones et tablettes. Elles sont très appréciées car elles permettent de créer des interfaces du plus bel effet. Découvrez dans cet article quelques-unes de leurs possibilités... p. 80

3 BIBLIOTHÈQUES ET TOOLKITS

INITIATION À LA PROGRAMMATION GRAPHIQUE EN C AVEC GTK+

Yann Morère (Daily Linux User since 1996, LaTeX fetishist ;-) and Lightsaber fan)

Dans une série d'articles dédiés à l'IDE Eclipse [GLMF n°159], je vous ai présenté la programmation graphique en C++ avec la librairie graphique GTK+. Nous allons ici l'utiliser avec le langage C, langage natif de ce toolkit.

1. INTRODUCTION

Pour ceux qui ne le sauraient pas encore, GTK+ est un *toolkit* multiplate-forme pour créer des interfaces graphiques utilisateur (GUI). Il fut développé initialement pour le logiciel de traitement d'image GIMP (GTK = *GIMP Tool Kit*). GTK+ a été pensé dès sa conception pour supporter plusieurs langages dont C/C++, Perl et Python. Bien sûr, GTK+ est un logiciel libre et fait partie du projet GNU. Ce toolkit est aussi la brique de base du bureau GNOME.

Cette bibliothèque vient de subir une mise à jour majeure en passant de la version 2.24 à la version 3.x. Ceci s'accompagne de nombreux changements et rompt la compatibilité avec la version antérieure. La page **[1]** explique, par l'intermédiaire d'un guide de migration, les principales différences entre ces versions. On retrouve d'ailleurs deux branches distinctes de documentation pour les versions 2 **[2]** et 3 **[3]**. De la même manière, les exemples de programmes seront différenciés pour les versions 2 et 3.

Les changements majeurs entre les deux versions sont les suivants :

→ Cairo remplace dorénavant GDK pour les zones de dessin et l'affichage des images ;

→ La prise en compte de périphériques d'entrée modernes (tablette, multitouch, etc.) ;

→ La configuration des thèmes utilise la syntaxe CSS ;

→ La gestion de la géométrie des composants graphiques (*widgets*) est simplifiée ;

→ L'apparition de nouveaux composants, comme des boutons et sélecteurs d'application ;

→ L'intégration de la bibliothèque gérant l'accessibilité GAIL (*GNOME Accessibility Implementation Library*) ;

→ L'intégration d'une gestion des événements tactiles et le support du défilement doux.

Cependant, tous les outils libres et applications n'ont pas encore migré vers la version 3 et les deux versions cohabitent sans problème dans votre système d'exploitation préféré.

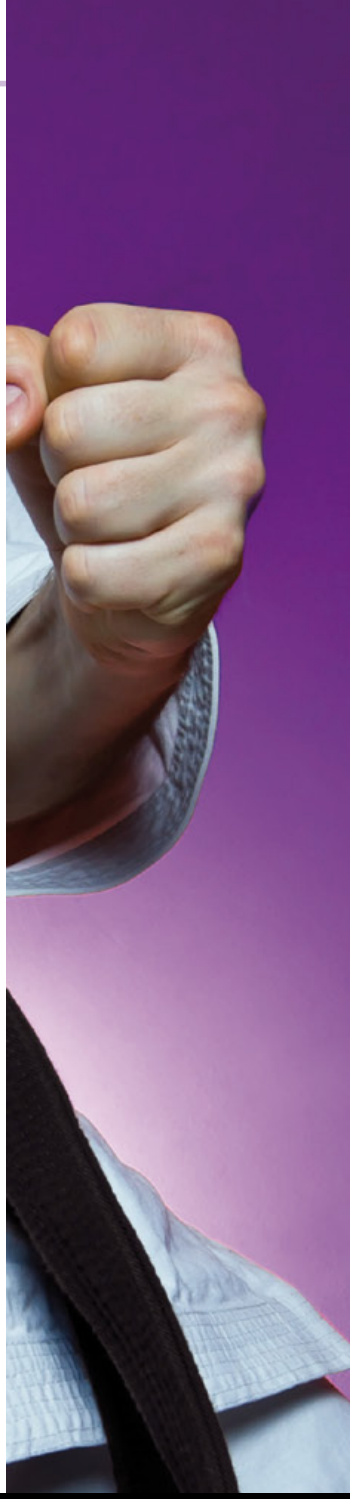
2. LES PRÉREQUIS

Dans la suite, nous utiliserons la version 3 de GTK+. Avant de pouvoir programmer, il vous faudra installer les outils nécessaires : un compilateur (et ses outils), le toolkit GTK+ avec les outils de développement et un éditeur de texte orienté programmation.

Terminal

```
$ sudo apt-get install libgtk-3-dev autoconf autogen
automake build-essential geany
```

Dans cette première partie, nous n'allons pas utiliser d'EDI/IDE (Environnement de Développement Intégré). Ceci nous obligera à concevoir notre interface graphique directement dans le code source.



Note

Ah oui, j'oubliais, il faut aussi connaître les bases du langage C :-D. À ce stade, il n'est pas nécessaire de consulter la globalité de l'API (*Application Programming Interface*) de GTK+. Mais cela deviendra nécessaire lorsque pour dépasserez le stade de l'initiation.

3. PREMIER PROGRAMME

Ce premier programme est destiné à comprendre comment créer votre première application GTK+. Il s'agit juste d'une fenêtre graphique vide. Ce sera aussi l'occasion de vérifier que tous les outils installés fonctionnent.

Voici le code source tiré de **[4]** :

Fichier

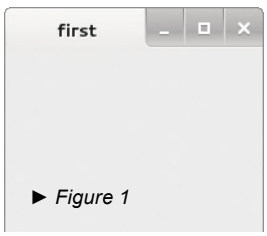
```
#include <gtk/gtk.h>

int main (int   argc,char *argv[])
{
  GtkWidget *window;

  gtk_init (&argc, &argv);
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

  g_signal_connect (window, "destroy", G_CALLBACK (gtk_main_quit), NULL);

  gtk_widget_show (window);
  gtk_main ();

  return 0;
}
```

La compilation est réalisée par la ligne de commandes suivante :

Terminal

```
$ gcc `pkg-config --cflags gtk+-3.0` -o first first.c `pkg-config --libs gtk+-3.0`
$ ./first
```

Ceci donne la figure 1, l'affichage d'une fenêtre vide.



► *Figure 1*

Avant de poursuivre, expliquons le rôle du programme **`pkg-config`**. Afin de pouvoir créer une simple application utilisant GTK+, on doit utiliser une série d'inclusions d'entête et de bibliothèques dans le projet. Heureusement, il existe **`pkg-config`** qui permet de configurer convenablement le projet de manière quasi-automatique.

Celui-ci renvoie des métadonnées concernant les bibliothèques installées sur votre système. Son utilisation, en passant en paramètre une bibliothèque, nous permet de connaître les bibliothèques dont elle dépend et les fichiers d'entête nécessaires à son utilisation. Le programme retrouve ces informations à partir de fichiers de métadonnées spécifiques. Ils sont stockés dans le répertoire **`/usr/lib/pkgconfig directory`** et possèdent l'extension **`.pc`**.

Par exemple, pour les fichiers concernant GTK+ nous avons :

Terminal

```
$ ls /usr/lib/x86_64-linux-gnu/pkgconfig/g[td]*
/usr/lib/x86_64-linux-gnu/pkgconfig/gdk-2.0.pc
/usr/lib/x86_64-linux-gnu/pkgconfig/gdk-3.0.pc
/usr/lib/x86_64-linux-gnu/pkgconfig/gdk-pixbuf-2.0.pc
/usr/lib/x86_64-linux-gnu/pkgconfig/gdk-pixbuf-xlib-2.0.pc
/usr/lib/x86_64-linux-gnu/pkgconfig/gdk-x11-2.0.pc
/usr/lib/x86_64-linux-gnu/pkgconfig/gdk-x11-3.0.pc
/usr/lib/x86_64-linux-gnu/pkgconfig/gthread-2.0.pc
/usr/lib/x86_64-linux-gnu/pkgconfig/gtk+-2.0.pc
/usr/lib/x86_64-linux-gnu/pkgconfig/gtk+-3.0.pc
```

Terminal

```
/usr/lib/x86_64-linux-gnu/pkgconfig/gtk+-unix-print-2.0.pc
/usr/lib/x86_64-linux-gnu/pkgconfig/gtk+-unix-print-3.0.pc
/usr/lib/x86_64-linux-gnu/pkgconfig/gtk+-x11-2.0.pc
/usr/lib/x86_64-linux-gnu/pkgconfig/gtk+-x11-3.0.pc
```

Et voici le fichier de configuration **gtk+-3.0.pc** pour la bibliothèque GTK+ 3.0 sur une machine 64 bits :

Fichier

```
prefix=/usr
exec_prefix=${prefix}
libdir=/usr/lib/x86_64-linux-gnu
includedir=${prefix}/include
targets=x11

gtk_binary_version=3.0.0
gtk_host=x86_64-pc-linux-gnu

Name: GTK+
Description: GTK+ Graphical UI Library
Version: 3.4.2
Requires: gdk-3.0 atk cairo cairo-gobject gdk-pixbuf-2.0 gio-2.0
Requires.private:  pangoft2 gio-unix-2.0
Libs: -L${libdir} -lgtk-3
Cflags: -I${includedir}/gtk-3.0
```

Ce fichier contient les dépendances d'utilisation de la bibliothèque GTK+ 3.0 qui sont **atk**, **cairo**, **gio** et **pangoft2**.

L'utilisation de **pkg-config** permet de récupérer la bonne configuration pour le compilateur. En fournissant l'option **--cflags**, nous récupérons tous les fichiers d'entête nécessaires à la compilation d'une application GTK+ 3.0 :

Terminal

```
$ pkg-config --cflags gtk+-3.0
-pthread -I/usr/include/gtk-3.0 -I/usr/include/pango-1.0 -I/usr/include/
gio-unix-2.0/ -I/usr/include/atk-1.0 -I/usr/include/cairo -I/usr/include/
gdk-pixbuf-2.0 -I/usr/include/freetype2 -I/usr/include/glib-2.0 -I/usr/
lib/x86_64-linux-gnu/glib-2.0/include -I/usr/include/pixman-1 -I/usr/
include/libpng12
```

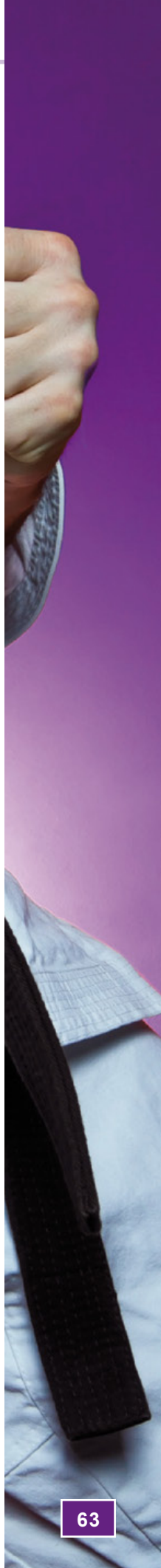
De manière similaire, on récupère toutes les bibliothèques nécessaires à l'édition de liens par l'intermédiaire de l'option **--libs**. On remarque que toutes les bibliothèques ne sont pas forcément nécessaires :

Terminal

```
$ pkg-config --libs gtk+-3.0
-lgtk-3 -lgdk-3 -latk-1.0 -lgio-2.0 -lpangocairo-1.0 -lgdk_pixbuf-2.0
-lcairo-gobject -lpango-1.0 -lcairo -lgobject-2.0 -lglib-2.0
```

Détaillons maintenant le code source précédent. Nous n'avons ici qu'une fonction **main**.

L'inclusion de départ est nécessaire et permet d'utiliser les fonctions relatives à la librairie GTK+. La fonction **gtk_init** a pour but d'initialiser la bibliothèque GTK+ et de se connecter à l'environnement fenêtré. Elle prend en arguments les paramètres que l'on peut passer à notre programme pour en contrôler son comportement et le comportement de GTK+. La page **[5]** présente les options de ligne de commandes que l'on peut utiliser.



Par exemple :

Terminal
```
$ ./first --name "Mon programme"
```

permet de renommer le titre de la fenêtre graphique générée par notre programme.

La fonction **gtk_window_new()** va créer un nouvel objet **GtkWindow** et renvoyer son pointeur dans la variable **window** précédemment déclarée. La fenêtre créée est de type **GTK_WINDOW_TOPLEVEL**, ce qui signifie qu'elle sera gérée par le système de fenêtrage de votre système d'exploitation.

Afin de pouvoir terminer l'application lorsque la fenêtre est fermée, le signal **destroy** est attaché à la fonction **gtk_main_quit()** par l'intermédiaire de la fonction **g_signal_connect**. Ceci permet de quitter la boucle principale de GTK+ (en attente d'action utilisateur) lancée par la fonction **gtk_main()**. Le signal **destroy** est émis dès qu'un widget (*Window Gadget* : composant de l'interface graphique) est détruit, soit par l'appel explicite à la fonction **gtk_widget_destroy()**, soit lorsque le composant est orphelin (la fenêtre dans laquelle il était n'existe plus). Dans le cas d'une fenêtre, ce signal est émis lorsque l'on clique sur le bouton « fermer ».

Tous les composants de l'interface (ou **GTKWidgets**) sont cachés par défaut. La fonction **gtk_widget_show()** permet d'afficher le widget passé en paramètre. On termine par le lancement de la boucle principale de GTK+, **gtk_main()**. Ceci a pour effet de bloquer l'exécution de la fonction **main** jusqu'à ce que la fonction **gtk_main_quit()** soit appelée.

Pendant que le programme est en exécution, GTK+ va recevoir des événements provoqués par l'interaction de l'utilisateur avec l'interface (clic bouton, entrée de texte, etc.). Ces événements peuvent aussi être envoyés par le gestionnaire de fenêtres ou d'autres applications. La connexion des signaux aux gestionnaires de signaux (*signal handlers*) est la manière générale de traiter les actions en réponse aux entrées utilisateur.

4. UN PEU PLUS DE SIGNAUX

Voyons cela sur l'exemple suivant, toujours tiré de **[4]** :

Fichier
```
#include <gtk/gtk.h>
//gcc `pkg-config --cflags gtk+-3.0` -o hello hello.c
//`pkg-config --libs gtk+-3.0`
static void print_hello (GtkWidget *widget, gpointer   data)
{
  g_print ("Hello World\n");
}

static gboolean on_delete_event (GtkWidget *widget, GdkEvent  *event,
gpointer   data)
{
  g_print ("delete event occurred\n");
  return TRUE;
}

int main (int   argc, char *argv[])
{
  GtkWidget *window;
  GtkWidget *button;
```

Fichier

```
    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Hello");

    g_signal_connect (window, "delete-event", G_CALLBACK (on_delete_event), NULL);
    g_signal_connect (window, "destroy", G_CALLBACK (gtk_main_quit), NULL);

    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    button = gtk_button_new_with_label ("Hello World");

    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);
    g_signal_connect_swapped (button, "clicked", G_CALLBACK (gtk_widget_destroy),
    window);

    gtk_container_add (GTK_CONTAINER (window), button);
    gtk_widget_show (button);
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

La compilation, puis l'exécution donne la figure 2 :

Terminal

```
$ gcc `pkg-config --cflags gtk+-3.0` -o hello hello.c `pkg-config --libs gtk+-3.0`
$ ./hello
```

► *Figure 2*

Voici quelques explications sur le code source. La fonction **print_hello** permet d'afficher le texte « Hello World » dans la console à l'aide de la fonction **g_printf**. Cette fonction est appelée lorsque le bouton est cliqué. C'est ce que l'on appelle une fonction « callback ». Dans notre cas, les paramètres sont simplement ignorés car non utilisés.

La fonction **on_delete_event** est appelée lorsque le signal **delete** est envoyé. On affiche dans la console le signal qui est survenu. Ce type de fonction peut être utile si l'on désire afficher une fenêtre de confirmation avant de fermer une fenêtre. Comme la fonction renvoie **true**, la fenêtre ne sera pas fermée. Si l'on avait renvoyé **false**, GTK aurait émis le signal **destroy** et la fenêtre aurait été fermée.

La fonction **gtk_window_set_title** permet de définir le titre de la fenêtre.

Dans cette fonction, on remarquera que l'objet **window** est « transtypé » (« casté ») en **GtkWindow**. En effet, c'est ce type qu'attend la fonction, alors que notre variable a été définie en **GtkWidget**.

La ligne :

Fichier

```
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
```

permet de définir la distance entre les bords de la fenêtre et les composants qu'elle va contenir. La ligne :

Fichier

```
button = gtk_button_new_with_label ("Hello World");
```

crée un nouveau bouton avec l'intitulé « Hello World ». Ensuite, on attache le signal **clicked** à la fonction callback **print_hello**. Le code suivant :

Fichier

```
g_signal_connect_swapped (button, "clicked",
G_CALLBACK (gtk_widget_destroy), window);
```

permet de fermer la fenêtre en cliquant sur le bouton. En effet, l'objet sur lequel le signal est émis (ici **button**) et le contenu du champ **data** de la fonction (ici **window**), sont échangés lors de l'appel au gestionnaire. Ceci aura pour effet d'appliquer la fonction **gtk_widget_destroy** à **window** et non à **button**.

Finalement, la ligne :

Fichier

```
gtk_container_add (GTK_CONTAINER (window), button);
```

ajoute la bouton dans la fenêtre. On remarquera ici encore le transtypage en **GTK_CONTAINER** pour respecter la définition de la fonction.

5. PLUS D'OBJETS DANS LA FENÊTRE

Afin de pouvoir placer plusieurs composants dans la fenêtre de notre interface graphique et de contrôler précisément leurs tailles et positions, nous allons utiliser différents conteneurs (*layout containers*) pour disposer nos composants et remplir l'interface. On nomme cela le « packing ».

Dans l'exemple suivant, nous allons voir comment utiliser le conteneur **GtkGrid** pour disposer les 3 boutons de notre interface. La compilation, puis l'exécution du code source ci-après donne la figure 3 :

Terminal

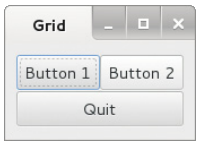
```
$ gcc `pkg-config --cflags gtk+-3.0` -o packing packing.c `pkg-config
--libs gtk+-3.0`
$ ./packing
```

Fichier

```
#include <gtk/gtk.h>
//gcc `pkg-config --cflags gtk+-3.0`
//-o packing packing.c `pkg-config --libs gtk+-3.0`
static void print_hello (GtkWidget *widget, gpointer   data)
{
  g_print ("Hello World\n");
}

int main (int   argc, char *argv[])
{
  GtkWidget *window;
  GtkWidget *grid;
  GtkWidget *button;

  gtk_init (&argc, &argv);
```

Fichier

```
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Grid");
    g_signal_connect (window, "destroy", G_CALLBACK (gtk_main_quit), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    grid = gtk_grid_new ();

    gtk_container_add (GTK_CONTAINER (window), grid);

    button = gtk_button_new_with_label ("Button 1");
    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

    gtk_grid_attach (GTK_GRID (grid), button, 0, 0, 1, 1);

    button = gtk_button_new_with_label ("Button 2");
    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

    gtk_grid_attach (GTK_GRID (grid), button, 1, 0, 1, 1);

    button = gtk_button_new_with_label ("Quit");
    g_signal_connect (button, "clicked", G_CALLBACK (gtk_main_quit), NULL);

    gtk_grid_attach (GTK_GRID (grid), button, 0, 1, 2, 1);

    gtk_widget_show_all (window);

    gtk_main ();

    return 0;
}
```



► *Figure 3*

Donnons quelques explications sur la construction de l'interface. Le code ci-dessous crée un nouvel objet **GtkGrid** (grille) et l'ajoute à la fenêtre :

Fichier

```
grid = gtk_grid_new ();
gtk_container_add (GTK_CONTAINER (window), grid);
```

Ensuite, le premier bouton est créé avec son gestionnaire de signal. Le code :

Fichier

```
gtk_grid_attach (GTK_GRID (grid), button, 0, 0, 1, 1);
```

place le bouton dans le première cellule de la grille (coordonnées (0,0)) ; il remplit une et une seule cellule (1,1) et ne s'étend pas sur d'autres cellules (« spanning »).

On crée ensuite le second bouton et de la même manière, on vient le placer dans la seconde cellule de la première ligne de la grille :

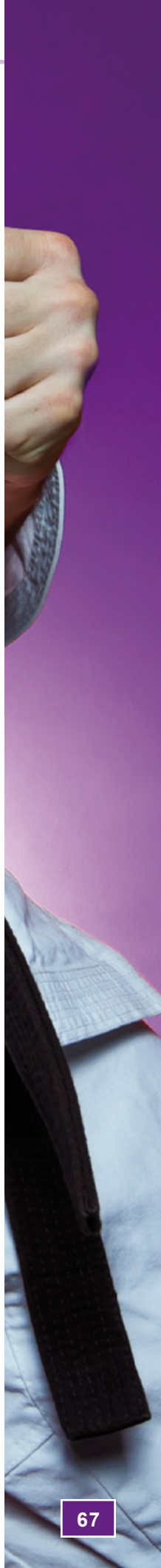
Fichier

```
gtk_grid_attach (GTK_GRID (grid), button, 1, 0, 1, 1);
```

Finalement, on crée le bouton « Quitter » et on le place dans la première cellule de la seconde ligne (0,1), puis on indique qu'il va occuper 2 cellules en largeur et 1 cellule en hauteur (2,1) :

Fichier

```
gtk_grid_attach (GTK_GRID (grid), button, 0, 1, 2, 1);
```



Finalement, on affiche tous les composants de la fenêtre à l'aide de la fonction :

Fichier

```
gtk_widget_show_all (window);
```

Notre interface est maintenant terminée !
Pour finir, voici la liste exhaustive des conteneurs disponibles dans GTK+ 3 :

→ **GtkGrid** : dispose les composants en lignes et en colonnes (grille) ;

→ **GtkAlignment** : composant qui permet de contrôler l'alignement et la taille de son composant enfant ;

→ **GtkAspectFrame** : cadre qui contraint son enfant suivant un rapport d'aspect ;

→ **GtkBox** : classe de base pour les conteneurs « box » (boîte) ;

→ **GtkHBox** : conteneur box horizontal ;

→ **GtkVBox** : conteneur box vertical ;

→ **GtkButtonBox** : classe de base pour les conteneurs **GtkHButtonBox** et **GtkVButtonBox** ;

→ **GtkHButtonBox** : conteneur pour disposer les boutons horizontalement ;

→ **GtkVButtonBox** : conteneur pour disposer les boutons verticalement ;

→ **GtkFixed** : conteneur qui permet de disposer les composants à des coordonnées fixes ;

→ **GtkPaned** : classe de base pour 2 volets ajustables ;

→ **GtkHPaned** : conteneur avec 2 volets ajustables disposés horizontalement ;

→ **GtkVPaned** : conteneur avec 2 volets ajustables disposés verticalement ;

→ **GtkLayout** : zone déroulante infinie, qui peut contenir des composants ou une zone de dessin ;

→ **GtkNotebook** : conteneur de type classeur à onglets ;

→ **GtkTable** : dispose les composants suivant des modèles réguliers ;

→ **GtkExpander** : conteneur qui peut cacher ses composants enfants ;

→ **GtkOrientable** : interface qui permet de modifier les composants (retournement, miroir).

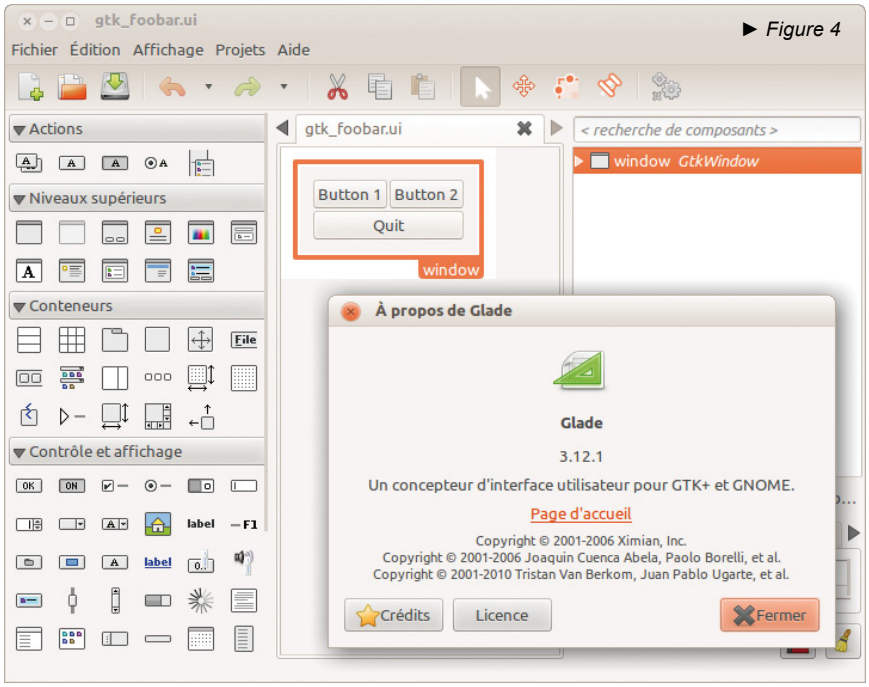
Vous avez maintenant les outils nécessaires et compris le principe de la génération d'une interface. Cependant, lorsque vous construisez une interface utilisateur plus complexe, avec plusieurs dizaines de composants, le codage dans le fichier source de l'interface peut rapidement devenir lourd et fastidieux.

GTK+ vous permet de séparer la définition de votre interface graphique de votre code source, qui décrit le fonctionnement logique de votre programme. Pour cela, on utilise une description de l'interface dans le format XML. Ensuite, ce fichier est analysé par une classe dédiée, **GtkBuilder**, qui permet de reconstruire l'interface. Voyons comment cela fonctionne.

6. GLADE ET GTKBUILDER

Glade **[5]** est un outil de développement rapide d'application (RAD, *Rapid Application Development*). Il permet de créer des interfaces destinées au toolkit GTK+ et à l'environnement de bureau GNOME. L'interface produite est codée sous la forme d'un fichier XML. Ce dernier est utilisé par la classe **GtkBuilder** **[6]** pour permettre le chargement dynamique d'interfaces par les applications. La classe analyse le fichier XML et construit l'interface correspondante.

À l'aide de cet outil graphique, on va reconstruire l'interface de l'exemple précédent (Fig. 4). Son utilisation est assez intuitive, je vous laisse donc le découvrir par vous-même.



► *Figure 4*

J'ai nommé le fichier **`gtk_foobar.ui`**. L'extension importe peu pour l'instant.

Ensuite, on peut utiliser le fichier produit dans notre programme à l'aide du code source suivant :

Fichier

```
#include <gtk/gtk.h>
//gcc `pkg-config --cflags gtk+-3.0`
//-o rad rad.c `pkg-config --libs gtk+-3.0`
static void print_hello (GtkWidget *widget, gpointer   data)
{
  g_print ("Hello World\n");
}

int main (int   argc, char *argv[])
{
  GtkBuilder *builder;
  GObject *window;
  GObject *button;

  gtk_init (&argc, &argv);

  builder = gtk_builder_new ();
  gtk_builder_add_from_file (builder, "gtk_foobar.ui", NULL);

  window = gtk_builder_get_object (builder, "window");
  g_signal_connect (window, "destroy", G_CALLBACK (gtk_main_
  quit), NULL);

  button = gtk_builder_get_object (builder, "button1");
  g_signal_connect (button, "clicked", G_CALLBACK (print_
  hello), NULL);
```

Note

Le fichier XML produit est bien sûr éditable manuellement, mais je vous conseille fortement de réaliser vos modifications à l'aide de Glade !

Fichier

```
    button = gtk_builder_get_object (builder, "button2");
    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

    button = gtk_builder_get_object (builder, "quit");
    g_signal_connect (button, "clicked", G_CALLBACK (gtk_main_quit), NULL);

    gtk_main ();

    return 0;
}
```

Afin que le programme précédent fonctionne, il est nécessaire que le fichier source C et le fichier de l'interface soient dans le même répertoire. La compilation, puis l'exécution du code source, fournit la même figure 4 :

Terminal

```
$ gcc `pkg-config --cflags gtk+-3.0` -o rad rad.c `pkg-config --libs gtk+-3.0`
$ ./rad
```

Voyons plus en détails le code source. Le ligne suivante permet de créer un nouvel objet **GtkBuilder** :

Fichier

```
builder = gtk_builder_new();
```

Ensuite, la fonction **gtk_builder_add_from_file** analyse le fichier de description de l'interface et ajoute les composants à l'objet **builder** que l'on vient de créer. Modifiez le chemin vers votre fichier d'interface s'il n'est pas au même endroit que votre code source.

Fichier

```
gtk_builder_add_from_file(builder, "gtk_foobar.ui", NULL);
```

La fonction **gtk_builder_get_object** permet de récupérer un objet de l'interface par l'intermédiaire de son nom. Il est donc important de nommer intelligemment les composants de vos fenêtres, afin de pouvoir les récupérer facilement.

Fichier

```
window = gtk_builder_get_object (builder, "window");
```

On peut alors ensuite leur attacher un gestionnaire d'événements, comme on l'a vu précédemment. Le reste du programme est identique au cas précédent. On remarque que notre code source est bien plus lisible, car la définition de l'interface ne vient pas polluer la logique de votre programme.

Voyons maintenant l'utilisation d'un IDE prévu pour GTK+, Anjuta, pour le développement d'un petit programme de test des tables de multiplication ;-).

7. PROGRAMMER AVEC ANJUTA

Anjuta **[7]** est un EDI/IDE (Environnement de Développement Intégré). C'est l'outil dédié à la programmation C/C++, Python, avec le toolkit GTK+. Il est aussi capable de gérer d'autres langages et d'autres toolkits graphiques (comme wxWidgets). Son développement est très actif et il possède maintenant de nombreux outils supplémentaires par l'intermédiaire de plugins :

→ Gestion de projets ;

→ Générateur de projets ;

→ Outil de déverminage interactif ;

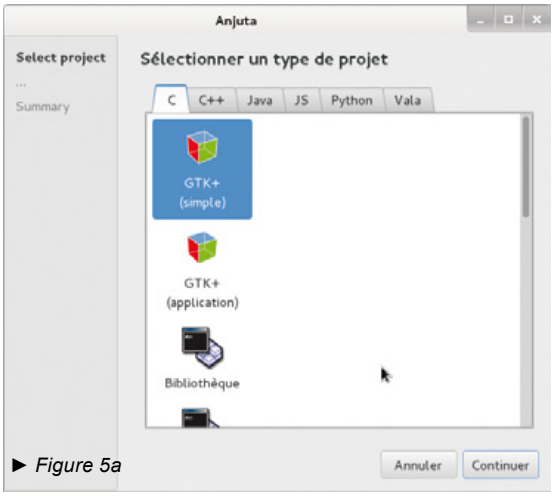
→ Gestion de révisions ;

→ Générateur graphique d'interfaces Glade intégré dans l'IDE ;

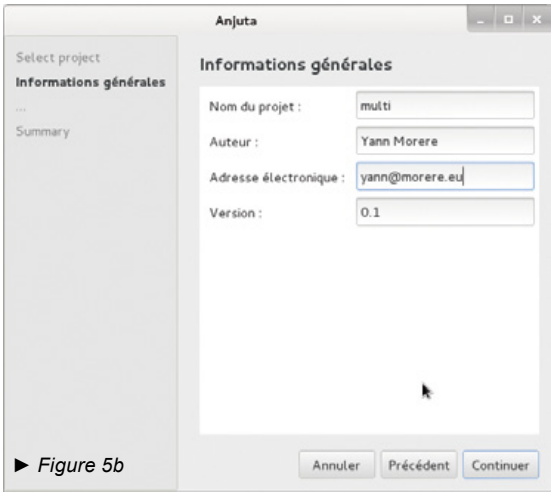
→ Navigateur d'aide DevHelp intégré dans l'IDE.

L'outil est assez léger et peut être utilisé sur des machines peu puissantes. De plus, il utilise les autotools **[8]** en standard pour la génération de projet. Ceci permet donc très simplement de générer une archive .tar.gz, qui permettra d'installer votre projet à l'aide d'un simple `./configure && make && sudo make install`.

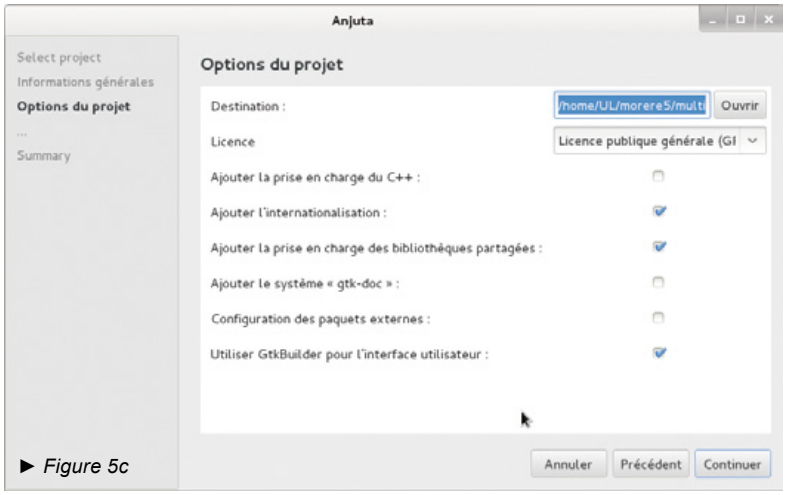
Après l'avoir installé à partir des dépôts, il suffit de l'exécuter et de créer un nouveau projet (Fig. 5).



► *Figure 5a*



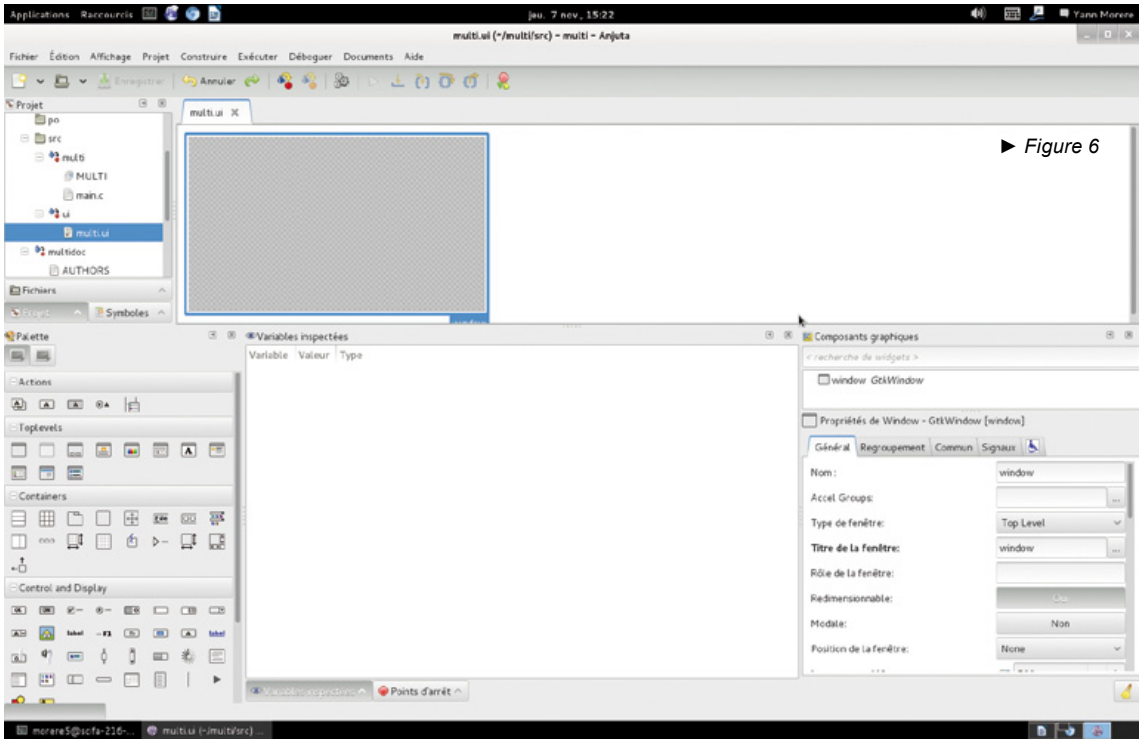
► *Figure 5b*



► *Figure 5c*

Anjuta génère alors tous les fichiers nécessaires à la gestion du projet et vous obtenez la fenêtre de la figure 6, page suivante.

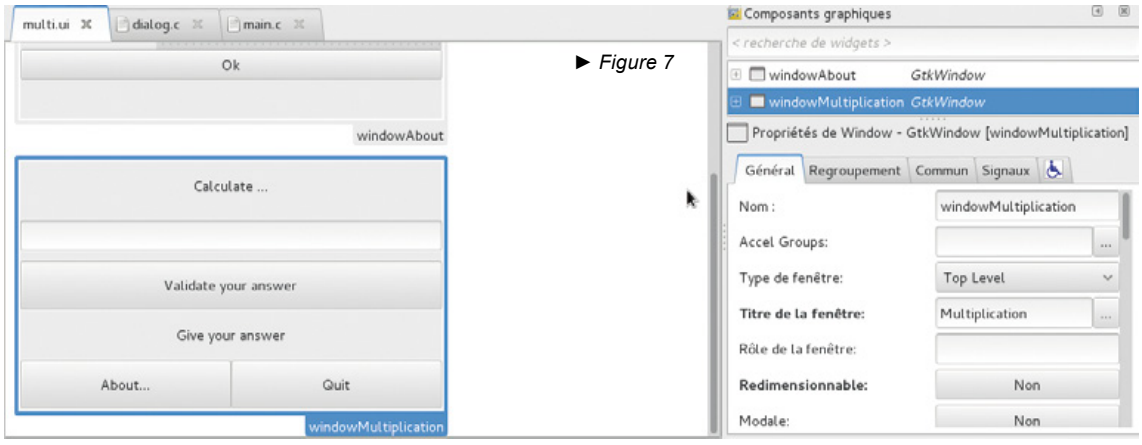
On remarque que notre IDE possède déjà un générateur d'interface graphique ; il s'agit en fait de Glade intégré sous la forme d'un greffon.



► *Figure 6*

Passons maintenant à notre petit programme. Le but est de tester la connaissance des tables de multiplication de 0 à 10. En tant que parent, il faut savoir mélanger l'utile à l'agréable... Je vous laisse choisir auquel, la programmation ou les devoirs des enfants, va votre préférence ;-).

L'interface sera très simple et est décrite en figure 7. On remarque que, pour l'instant, on réalise tout dans la langue de Shakespeare.



► *Figure 7*

L'arborescence des composants graphiques est décrite en figure 8.

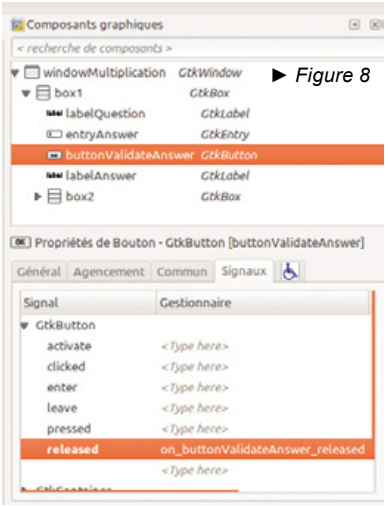
Toujours dans le module de création d'interface, nous allons ajouter les signaux nécessaires aux différents composants, ainsi que le nom de la fonction callback qu'ils vont appeler :

→ Sur **`entryAnswer`**, on ajoute le signal **`activate`** qui va appeler la fonction **`on_entryAnswer_activate`** ;

→ Sur **buttonValidateAnswer**, on ajoute le signal **released** qui va appeler la fonction **on_buttonValidateAnswer_released** ;

→ Sur **buttonAbout**, on ajoute le signal **released** qui va appeler la fonction **on_buttonAbout_released** ;

→ Sur **buttonQuit**, on ajoute le signal **released** qui va appeler la fonction **on_buttonQuit_released**.



► *Figure 8*

Pour cela, il faut ouvrir l'onglet ***Signaux*** des propriétés des composants (Fig. 8).

Décortiquons le code source de notre application :

Fichier

```
#include <config.h>
#include <gtk/gtk.h>
#include <glib/gi18n.h>
#include <stdlib.h>
#include <time.h>

typedef struct _Private Private;
static struct _Private
{
 /* ANJUTA: Widgets declaration for multi.ui - DO NOT REMOVE */
};
static struct Private* priv = NULL;

/* For testing propose use the local (not installed) ui file */
/* #define UI_FILE PACKAGE_DATA_DIR"/ui/multi.ui" */
#define UI_FILE "src/multi.ui"
#define TOP_WINDOW "windowMultiplication"
```

Toute cette première partie du code est générée automatiquement par l'assistant de création de projet. Il faut juste penser à modifier la variable **TOP_WINDOW** si vous avez renommé la fenêtre principale. J'ai juste ajouté les inclusions nécessaires aux fonctions de génération de nombre aléatoire **rand** et **srand** (**time.h**) et la conversion **atoi** (**stdlib.h**).

Fichier

```
/* Some global variables used in callback function*/
GtkBuilder *builder;
gint a,b,c;
```

Pour plus de simplicité, nous allons utiliser des variables globales pour le stockage des entiers nécessaires à notre multiplication et l'objet qui contient la définition de l'interface graphique.

Fichier

```
void prepare_new_calculus(int *a, int *b, int *c)
{
 srand(time(NULL));
 *a = rand() % 11;
 *b = rand() % 11;
 *c = (*a)*(*b);
}
```

Note

On remarque l'inclusion du fichier gi18n.h, qui n'a plus de secret pour vous. En effet, il s'agit bien du support de l'internationalisation pour GTK+.

La fonction précédente permet de choisir aléatoirement des nombres entiers entre 0 et 10. La fonction **srand** permet d'initialiser le moteur de nombres pseudo-aléatoires à chaque tirage (ce qui n'est pas vraiment nécessaire, une seule fois suffirait dans le **main**). La fonction **rand()** utilisée avec le reste de la division entière (% modulo) avec 11 permet de donner un entier entre 0 et 10, que l'on stockera dans chaque opérande (**a** et **b**). Ensuite, on effectue la multiplication que l'on stocke dans la dernière variable (**c**).

Fichier

```
/* Signal handlers */
/* Note: These may not be declared static because signal autoconnection
 * only works with non-static methods
 */

void on_buttonValidateAnswer_released (GtkWidget *widget, gpointer data)
{
const gchar *get_result;
gint result;
GtkEntry *entryResult = GTK_ENTRY(gtk_builder_get_object (builder,
"entryAnswer"));
get_result = gtk_entry_get_text(entryResult);
result=atoi(get_result);
GtkLabel *label_info = GTK_LABEL (gtk_builder_get_object (builder,
"labelAnswer"));

if(result==c)
 gtk_label_set_text(GTK_LABEL(label_info),"Bonne réponse ! Recommencez...");
 else
 gtk_label_set_text(GTK_LABEL(label_info),"Mauvaise réponse ! Ré-essayez...");

prepare_new_calculus(&a, &b, &c);

GtkLabel *label = GTK_LABEL (gtk_builder_get_object (builder,
"labelQuestion"));
gtk_label_set_text(GTK_LABEL(label), g_strdup_printf ("Calculez %d x %d =",
a, b) );
gtk_entry_set_text(GTK_ENTRY(entryResult),"");
}
```

Il s'agit de la fonction qui est au cœur de notre programme. Elle est appelée par l'événement **released** du bouton de validation (**buttonValidateAnswer**). Cet événement est produit lorsque l'on appuie sur [Entrée] pour valider la saisie. Cette fonction récupère la valeur entrée, puis la compare avec le résultat réel. Si les deux valeurs sont égales, on affiche un message de félicitations ; sinon, on demande de faire une nouvelle tentative.

Les fonctions **gtk_builder_get_object** permettent de récupérer les pointeurs sur les objets de notre interface graphique. Ceci va nous permettre de les manipuler par la suite.

La fonction **gtk_entry_get_text** permet de récupérer le contenu de la zone de saisie. La chaîne de caractères est alors convertie en entier avec la fonction **atoi**.

Une fois le test effectué, on modifie l'intitulé de la zone de texte (**label**) grâce à la fonction **gtk_label_set_text**. On relance ensuite la préparation d'un nouveau calcul, puis on compose la nouvelle question au format texte à l'aide des fonctions **gtk_label_set_text** et **g_strdup_printf**. Finalement, on efface le message de résultat.

Fichier

```
void on_entryAnswer_activate (GtkWidget *widget, gpointer data)
{
 on_buttonValidateAnswer_released( widget, data);
}
```

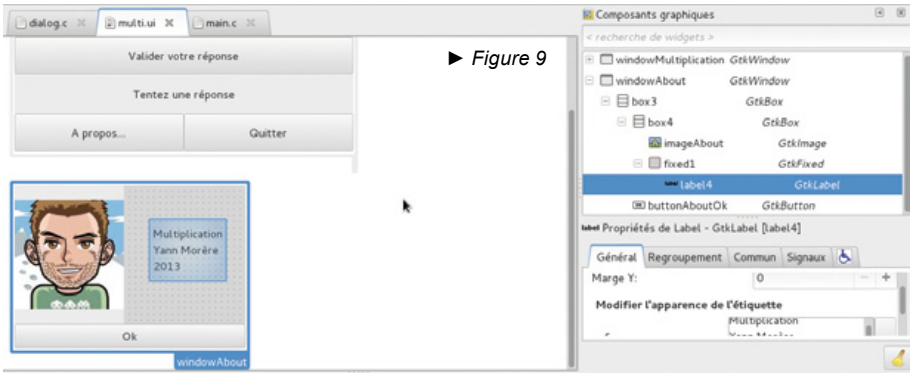
Elle est appelée par l'événement **activate** de la zone de saisie (**entryAnswer**). Cet événement est produit lorsque l'on appuie sur [Entrée] pour valider la saisie. Comme les opérations à effectuer sont identiques à celles réalisées par l'appui sur le bouton, on appelle la fonction callback définie précédemment.

Fichier

```
void on_buttonAbout_released (GtkWidget *widget, gpointer data)
{
 GtkWidget *dialog, *label, *content_area;
 gchar *message = "Yann Morère - 2013";
 GtkWindow *main_window = GTK_WINDOW (gtk_builder_get_object (builder, TOP_WINDOW));
   dialog = gtk_dialog_new_with_buttons ("Message",
  main_window,
  GTK_DIALOG_DESTROY_WITH_PARENT,
  _("_OK"),
  GTK_RESPONSE_NONE,
  NULL);
   content_area = gtk_dialog_get_content_area (GTK_DIALOG (dialog));
   label = gtk_label_new (message);

  g_signal_connect_swapped (dialog, "response", G_CALLBACK (gtk_widget_destroy),
dialog);
  gtk_container_add (GTK_CONTAINER (content_area), label);
   gtk_widget_show_all (dialog);
}
```

La fonction précédente, tirée de [9], permet de créer une boîte de dialogue affichant un message lorsque l'on clique sur le bouton **buttonAbout**. Ici, la boîte de dialogue est composée de toutes pièces, sans utiliser le générateur d'interface.

La seconde solution consiste à créer une nouvelle fenêtre à l'aide du générateur d'interface, comme dans la figure 9. On peut alors ajouter des images et composer la fenêtre comme bon nous semble.



► *Figure 9*

La fonction **on_buttonAbout_released** devient alors la suivante :

Fichier

```
void on_buttonAbout_released (GtkWidget *widget, gpointer data)
{
 GtkWidget*windowAbout=GTK_WIDGET(gtk_builder_get_object(builder,"windowAbout"));
        if (!windowAbout)
        {
                g_critical ("Widget \"%s\" is missing in file %s.",
                            "windowAbout",
                            UI_FILE);
        }
 gtk_widget_show (windowAbout);

}
```

On ajoute le signal **released** sur le bouton **buttonAboutOk** créé, puis on ajoute la fonction **on_buttonAboutOk_released** qui permet de cacher le fenêtre :

Fichier

```
void on_buttonAboutOk_released(GtkWidget *widget, gpointer data)
{
 GtkWidget * windowAbout = GTK_WIDGET (gtk_builder_get_object (builder,
"windowAbout"));
        if (!windowAbout)
        {
                g_critical ("Widget \"%s\" is missing in file %s.",
                            "windowAbout",
                            UI_FILE);
        }
 gtk_widget_hide (windowAbout);
}
```

Voyons maintenant comment terminer notre programme :

Fichier

```
/* Called when the window is closed */
void destroy (GtkWidget *widget, gpointer data)
{
 g_object_unref (builder);
 gtk_main_quit ();
}
```

Cette fonction, appelée à la fermeture de la fenêtre du programme, permet de quitter la boucle principale GTK+. On n'oublie pas de libérer la mémoire utilisée par notre objet **builder** à l'aide de la fonction **g_object_unref**.

Fichier

```
void on_buttonQuit_released (GtkWidget *widget, gpointer data)
{
 destroy(widget,data);
}
```

Cette fonction permet de quitter proprement le programme à l'aide du bouton idoine.

Fichier

```
static GtkWidget* create_window (void)
{
 GtkWidget *window;
 GError* error = NULL;
 /* Load UI from file */
 builder = gtk_builder_new ();
 if (!gtk_builder_add_from_file (builder, UI_FILE, &error))
 {
  g_critical ("Couldn't load builder file: %s", error->message);
  g_error_free (error);
 }
 /* Auto-connect signal handlers */
 gtk_builder_connect_signals (builder, NULL);
 /* Get the window object from the ui file */
 window = GTK_WIDGET (gtk_builder_get_object (builder, TOP_WINDOW));
        if (!window)
        {
                g_critical ("Widget \"%s\" is missing in file %s.",
                            TOP_WINDOW,
                            UI_FILE);
        }
```

Fichier

```
    priv = g_malloc (sizeof (struct _Private));
    /* ANJUTA: Widgets initialization for multi.ui - DO NOT REMOVE */
    /*Comment the following to keep builder alive for others functions*/
    /*g_object_unref (builder); */

    return window;
}
```

La fonction **`create_window`** a été créée par l'assistant de projet et n'a pas été modifiée.

On remarque ici qu'aucun gestionnaire de signaux n'a été défini. Cette opération est effectuée automatiquement par la fonction **`gtk_builder_connect_signals`**. Elle permet de définir les gestionnaires de signaux correspondant aux signaux définis lors de la génération de votre interface dans Glade.

Terminons par la fonction **`main`**.

Fichier

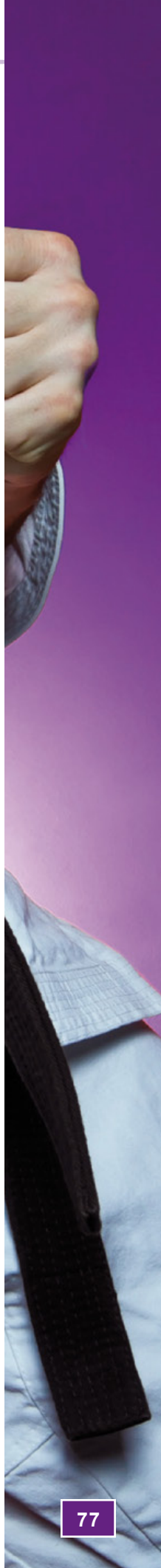
```
int main (int argc, char *argv[])
{
 GtkWidget *window;

#ifdef ENABLE_NLS
 bindtextdomain (GETTEXT_PACKAGE, PACKAGE_LOCALE_DIR);
 bind_textdomain_codeset (GETTEXT_PACKAGE, "UTF-8");
 textdomain (GETTEXT_PACKAGE);
#endif

 gtk_init (&argc, &argv);
 prepare_new_calculus(&a, &b, &c);
 window = create_window ();
 GtkLabel *label=GTK_LABEL(gtk_builder_get_object(builder,"labelQuestion"));
 gtk_label_set_text(GTK_LABEL(label), g_strdup_printf ("Calculez %d x %d =",
 a, b) );
 gtk_widget_show (window);
 gtk_main ();
 g_free (priv);
 return 0;
}
```

Celle-ci est aussi générée automatiquement. Il suffit d'ajouter la génération d'un premier calcul et son affichage dans le label approprié, afin que le programme commence avec une demande de réponse.

8. INTERNATIONALISER LE PROJET ANJUTA

Avant de poursuivre, il est impératif de lire l'article sur l'internationalisation des programmes en C, dans ce même numéro. La macro **`_()`** est déjà définie par l'inclusion **`glib/gi18n.h`**. Il suffit de reprendre le code source et d'englober chaque chaîne de caractères par **`_( string )`**.

Pour le fichier d'interface, les intitulés des composants sont repérables dans le code XML par le mot-clé « translatable ». Par exemple, pour le bouton de validation de réponse on a le code suivant :



Fichier

```
<object class="GtkButton" id="buttonValidateAnswer">
  <property name="label" translatable="yes">Validate your answer</property>
```

Toutes les exécutions suivantes se déroulent dans le répertoire **po** du projet.

Cependant, le programme **xgettext** ne reconnaît ni les fichiers **.ui**, ni **.xml**. Il n'accepte que l'extension **.glade**. Les fichiers ont pourtant exactement le même format. On renomme le fichier pour le fournir à l'utilitaire, en n'oubliant pas d'ajouter le nouveau mot-clé :

Terminal

```
$ xgettext --sort-output --keyword=translatable --keyword=_ \
-o multi.pot ../src/multi.ui ../src/main.c
xgettext: AVERTISSEMENT : pour le fichier " ../src/multi.ui ", l'extension
" ui " est inconnue. On suppose que c'est du C
$ cp ../src/multi.ui ../src/multi.xml
$ xgettext --sort-output --keyword=translatable --keyword=_ -o multi.pot
../src/multi.xml ../src/main.c
xgettext: AVERTISSEMENT : pour le fichier " ../src/multi.xml ",
l'extension " xml " est inconnue. On suppose que c'est du C
$ cp ../src/multi.ui ../src/multi.glade
$ xgettext --sort-output --keyword=translatable --keyword=_ -o multi.pot
../src/multi.glade ../src/main.c
```

On vérifie rapidement que les chaînes issues du code et de l'interface sont présentes et on dérive les fichiers de traduction :

Terminal

```
$ msginit --locale=fr_FR.utf8 -input=multi.pot
fr.po a été créé.
$ msginit --locale=pt_BR.utf8 -input=multi.pot
pt_BR.po a été créé.
$msgfmt fr_FR.po -o fr_FR.mo
$msgfmt pt_BR.po -o pt_BR.mo
```

Après avoir réalisé les traductions, on crée les binaires **.po** et on les copie dans les répertoires appropriés :

Terminal

```
$ cp fr.mo fr_FR/LC_MESSAGES/multi.mo
$ cp pt_BR.mo pt_BR/LC_MESSAGES/multi.mo
```

Afin de faire des tests sans installation, on ajoute le code suivant à la suite afin de remplacer la définition générique :

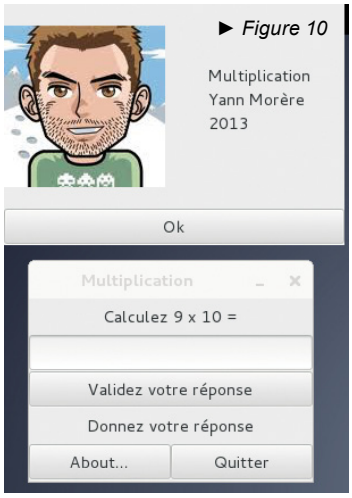
Fichier

```
#ifdef ENABLE_NLS
 bindtextdomain (GETTEXT_PACKAGE, PACKAGE_LOCALE_DIR);
 bind_textdomain_codeset (GETTEXT_PACKAGE, "UTF-8");
 textdomain (GETTEXT_PACKAGE);
#endif
 bindtextdomain( "multi", "./po" );
 bind_textdomain_codeset ("multi", "UTF-8");
 textdomain( "multi" );
```

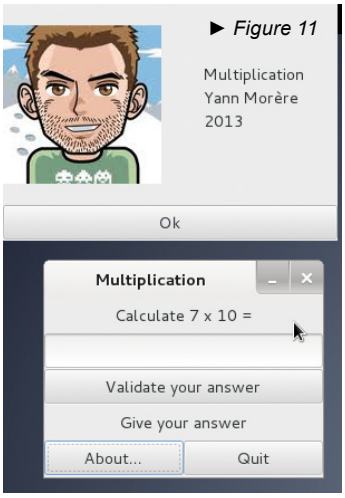
Après compilation, on teste notre programme en l'exécutant à partir de la racine du projet.

Terminal

```
$ Debug/src/multi
```



► *Figure 10*



► *Figure 11*

On obtient la figure 10. Notre programme est bien francisé !

Ensuite, on teste avec une langue non installée sur le système d'exploitation :

Terminal

```
$ LANG=de_DE.utf8 Debug/src/multi
(process:14759): Gtk-WARNING **: Locale not supported by C library.
 Using the fallback 'C' locale.
```

On obtient notre affichage en anglais, codé en dur dans le programme (Fig. 11).

Tous les codes sources sont disponibles à l'adresse **[10]**.

CONCLUSION

Nous voici au terme de cet article. Nous avons vu comment s'initier à la programmation C à l'aide de la bibliothèque GTK+. Nous avons utilisé différents outils et différents modes de programmation des interfaces. Cependant, afin de maîtriser ce toolkit, il faudra consulter régulièrement l'API (*Application Programming Interface*). Pour ceux qui veulent poursuivre cette initiation, je leur conseille vivement la lecture de la page **[11]**. Bon GTK+ ! ■

RÉFÉRENCES

[1] **http://developer.gnome.org/gtk3/stable/gtk-migrating-2-to-3.html**

[2] **http://developer.gnome.org/gtk2**

[3] **http://developer.gnome.org/gtk3**

[4] **https://developer.gnome.org/gtk3/3.0/gtk-getting-started.html**

[5] **https://glade.gnome.org/**

[6] **https://developer.gnome.org/gtk3/3.0/GtkBuilder.html**

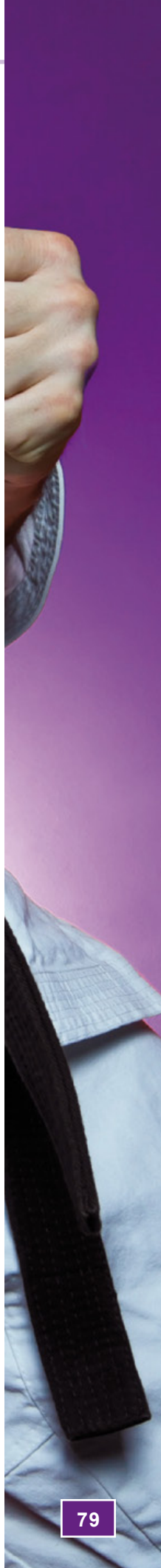
[7] **http://anjuta.org/**

[8] **http://fr.wikipedia.org/wiki/Autotools**

[9] **https://developer.gnome.org/gtk3/stable/GtkDialog.html**

[10] **http://www.morere.eu/spip.php?article173**

[11] **https://developer.gnome.org/gnome-devel-demos/unstable/beginner.c.html.fr**



PROGRAMMER AVEC LES ENLIGHTENMENT FOUNDATION LIBRARIES

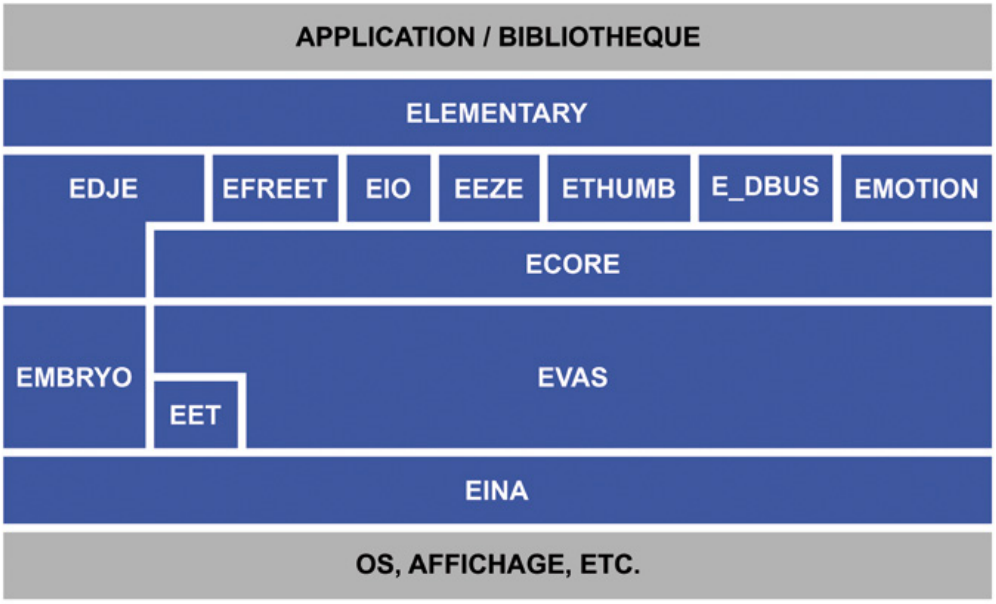
Denis Bodor

Vous avez besoin d'une bibliothèque C graphique, capable d'utiliser plusieurs types de systèmes d'affichage (X, framebuffer, OpenGL, etc.), fonctionnant aussi bien sur PC que sur système embarqué, incluant des fonctionnalités avancées comme un système de description d'interfaces, plusieurs langages de scripts intégrés, le tout reposant sur des fondations solides et des bibliothèques utilitaires adaptées ? Ne cherchez plus ! C'est des EFL qu'il s'agit !

Les utilisateurs GNU/Linux connaissent bien Enlightenment, un gestionnaire de fenêtres qui dès ses premières versions était très orienté « personnalisation graphique ». Le passage de la version 0.16 à la 0.17, renommé depuis e17, a fait d'Enlightenment un mythe pour certains, un *vaporware* pour d'autres, ou encore une monstrueuse avancée en termes de création d'interfaces graphiques pour les plus confiants. Il est vrai que e17 s'est fait attendre, mais Carsten Haitzler, alias Rasterman, créateur d'Enlightenment, a immédiatement vu dans le passage à la nouvelle version, l'occasion de modulariser le code. Ainsi, avant d'envisager la finalisation d'e17, il a décidé de créer un ensemble de bibliothèques se chargeant de tâches bien spécifiques, qu'il suffirait alors d'utiliser dans le gestionnaire de fenêtres (Ceci ne vous rappelle rien ? Des composants qui ne font qu'une chose, mais qui le font très bien ? Vraiment ?). Le projet des EFL, *Enlightenment Foundation Libraries*, était né et leur développement prit certes un temps conséquent, mais pour un résultat à la hauteur des attentes de bien des développeurs, sinon au-delà.

Aujourd'hui, e17 est disponible et des versions stables de chaque composant peuvent être utilisées en toute sérénité. Mieux encore, Rasterman faisant maintenant partie des employés du géant sud-coréen Samsung, tout comme d'autres développeurs Enlightenment, les EFL forment également la base graphique du système pour smartphones et tablettes Tizen.

Les EFL se structurent ainsi, de la couche la plus basse à la plus haute :



→ **Eina** fournit les structures de données de base et les fonctions utilitaires sur lesquelles reposent toutes les EFL.

→ **Evas** est le moteur graphique chargé du *rendering* et de la composition graphique. Cette bibliothèque est modulaire et dispose de greffons permettant l'affichage via diverses méthodes accélérées matériellement ou non. Evas rend votre application indépendante de la sortie graphique utilisée. Le même code, sans aucune modification fonctionnera de la même manière dans X qu'avec un affichage en *framebuffer*, par exemple. Sans même recompiler !

→ **Eet**, reposant sur Eina, apporte les fonctionnalités d'encodage/décodage des données et les fonctions liées au stockage des informations.

→ **Embryo** est une machine virtuelle et un compilateur Pawn permettant d'embarquer des scripts dans vos développements. Il s'agit là de l'une des solutions pour ce faire. Lua et JavaScript sont également de la partie.

→ Au-dessus d'Evas se trouve **Ecore**, un autre élément majeur des EFL. Sa tâche est de fournir une couche d'abstraction pour l'affichage des éléments graphiques, quelques fonctions utilitaires, mais aussi et surtout la gestion des événements et de la boucle principale.

→ Sur Ecore reposent **Efreet**, **eIO**, **Eeze**, **Ethumb**, **EDBus** et **Emotion** fournissant respectivement, une couche d'abstraction pour le standard FreeDesktop, les E/S asynchrones, une gestion de périphériques, un générateur de vignettes (*thumbnails*), une interface pour DBus et une API de gestion vidéo/audio.

→ **Edje** est la troisième pièce maîtresse de l'ensemble et tire sa puissance d'Embryo et Evas, via Ecore. C'est la couche d'abstraction pour la GUI. Ce qualificatif est bien modeste face à ce que Edje offre dans les faits. En effet, Edje permet de séparer la présentation et le code, en regroupant ces éléments sous la forme d'une description utilisée par votre programme. Plus qu'un moteur de thèmes, Edje permet d'automatiser le comportement de l'interface que vous composez en incluant une partie de la logique (programmes, signaux, messages, événements) et en communiquant avec le cœur de votre application. Il est ainsi possible de changer entièrement l'aspect d'une application reposant sur les EFL sans la moindre compilation, en travaillant simplement sur la description Edje.

→ Au-dessus de cet ensemble, qui permettrait déjà de créer e17, s'ajoute une dernière couche, **Elementary**. C'est la bibliothèque de Widgets, le *toolkit* utilisé par e17 contenant boutons, menus, listes déroulantes, *sliders*, zones de saisie, etc. Bref, tout ce qu'il faut pour créer une interface utilisateur, à la manière de ce que propose GTK+, mais en reposant sur la puissance des briques sous-jacentes et donc sur la souplesse, la richesse et les performances d'Evas, Ecore et Edje.

Comme vous pouvez le constater, les EFL forment un ensemble complet et homogène. Dans cet article d'introduction, nous nous contenterons d'utiliser celles de plus haut niveau, souvent suffisantes pour fournir à vos programmes une interface du plus bel effet. Au programme donc Evas/Ecore et Edje...

1. PREMIER PROGRAMME ECORE/EVAS

Notre premier vrai code sera ici précédé d'une petit entrée en matière simpliste. Avant de réellement entrer dans le vif du sujet, découvrons une partie de ce qui fait la force des EFL. Considérons tout d'abord le code suivant :

Fichier

```
#include <stdlib.h>
#include <Eina.h>
#include <Evas.h>
#include <Ecore.h>
#include <Ecore_Evas.h>

int
main(int argc, char *argv[])
{
  Eina_List *engine_list, *l;
  char *engine_name;

  ecore_evas_init();

  engine_list = ecore_evas_engines_get();
  if (!engine_list) {
    fprintf(stderr, "ERROR: Evas supports no engine ! Bye.\n");
   exit(EXIT_FAILURE);
  }
```

Fichier

```
    printf("Available Evas Engines:\n");

    EINA_LIST_FOREACH(engine_list, l, engine_name)
      printf("  %s\n", engine_name);

    ecore_evas_engines_free(engine_list);
    ecore_evas_shutdown();
    return EXIT_SUCCESS;
}
```

Ces quelques lignes devraient être présentes dans n'importe quel programme souhaitant utiliser les EFL, ou du moins les éléments reposant sur Evas (ceci concerne, bien entendu, le test **`if (!engine_list)`**). Comme dit précédemment, Evas est un moteur générique rendant vos codes indépendants du *backend* graphique existant, on parle alors d'*engines*. Nous utilisons ici Ecore-evas, un module Ecore permettant de facilement gérer et manipuler les objets Evas. Evas est une bibliothèque de gestion de *canvas* n'intégrant que des éléments propres à son fonctionnement. Ecore, pour sa part, est une sorte de couteau suisse des EFL, offrant au développeur une interface de haut niveau, étendant ainsi les fonctionnalités disponibles à des domaines qui dépassent Evas. C'est le cas, par exemple, de la gestion d'événements ou de celle des *timers*.

Nous utilisons donc ici Evas au travers d'Ecore et donc, initialisons le système avec **`ecore_evas_init()`**. L'initialisation du *wrapper* initialise implicitement les bibliothèques Evas et Ecore. Nous pouvons ensuite utiliser **`ecore_evas_engines_get()`** pour récupérer la liste des *moteurs* disponibles sur le système. Notez que cette fonction retourne une **`Eina_List`**, l'un des nombreux types définis et utilisés par la couche Eina et les autres briques des EFL.

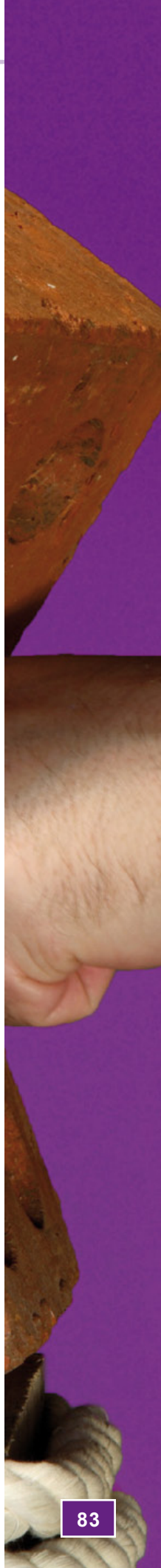
Mais Eina ne définit pas que des types ; cette bibliothèque met également à notre disposition un certain nombre de macros dont **`EINA_LIST_FOREACH`**, nous permettant une élégante itération sur la liste des moteurs de rendu, dont nous affichons le nom sur la console.

Avant de quitter le programme, nous n'oublions pas de libérer la mémoire occupée par la **`Eina_List`**, allouée par **`ecore_evas_engines_get()`**, avec **`ecore_evas_engines_free()`**. Ceci est une habitude à prendre, même si le système s'en occupera d'une manière ou d'une autre pour vous à la terminaison du programme.

Pour compiler notre toute petite création, il nous suffit de composer un **`Makefile`** se basant sur le très pratique **`pkg-config`** :

Fichier

```
TARGET  := base
WARN    := -Wall
CFLAGS  := -O2 ${WARN} `pkg-config --cflags eina evas ecore ecore-evas`
LDFLAGS := `pkg-config --libs eina evas ecore ecore-evas`
CC      := gcc

C_SRCS    = $(wildcard *.c)
OBJ_FILES = $(C_SRCS:.c=.o)

all: ${TARGET}

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

${TARGET}: ${OBJ_FILES}
        ${CC} ${WARN} ${LDFLAGS} -o $@ $(OBJ_FILES)

clean:
        rm -rf *.o ${TARGET}
```



Après compilation, et si toutes les dépendances de compilation sont satisfaites (paquet **-dev** sous Debian), l'exécution du programme nous affiche :

Terminal

```
Available Evas Engines:
  software_x11
  opengl_x11
  fb
  buffer
  ews
```

Ainsi, sur la plateforme en cours d'utilisation, nous avons à notre disposition pas moins de 5 backends de rendering pour Evas : rendu X non accéléré, idem via OpenGL, en framebuffer, dans un buffer (mémoire) et avec EWS, le *Evas Single Process Windowing System*, un système de fenêtrage pseudo mono-fenêtre. Lors du lancement du programme, ou plus exactement en créant le canevas, Evas va automatiquement chercher à utiliser le moteur de rendu le plus adapté. Il est cependant possible dans votre code, ou via une variable d'environnement (**ECORE_EVAS_ENGINE**), de définir arbitrairement ce moteur. Il n'est pas rare ainsi de voir un code fonctionner parfaitement et avec une fluidité exemplaire, avant de se rendre compte que le rendu logiciel est utilisé tant les performances d'Evas sont importantes. L'accélération matérielle n'est ainsi souvent pas indispensable.

Il est temps maintenant de pousser plus loin nos expérimentations en attaquant justement la partie graphique. Avant toute chose, nous devons créer un objet Evas. Ici, un objet **Ecore_Evas** que nous appellerons **windows** :

Fichier

```
#define WIDTH 640
#define HEIGHT 640

[...]
Ecore_Evas *window;
[...]

  window = ecore_evas_new(NULL, 0, 0, WIDTH, HEIGHT, NULL);
  if (!window) {
    EINA_LOG_CRIT("could not create window.");
    return -1;
  }
```

L'utilisation des macros **WIDTH** et **HEIGHT** est une habitude qu'il est bon de prendre, pour ensuite baser vos calculs et positionnements d'objets en fonction de ces valeurs. La fonction **ecore_evas_new**, comme son nom l'indique, provient du module Evas d'Ecore. Nous aurions également pu créer directement un objet équivalent avec Evas, mais il n'y a pas ici d'intérêt à utiliser des fonctions de plus bas niveau.

Notre **window** n'est pas le canevas Evas avec lequel nous allons travailler, c'est un objet **Ecore_Evas**. Nous pouvons récupérer le canevas, créé implicitement, avec :

Fichier

```
Evas *canvas;
[...]
canvas = ecore_evas_get(window);
```

Le canevas Evas est ainsi représenté par l'objet **canvas** et nous pouvons alors le manipuler avec les différentes fonctions mises à disposition par cette bibliothèque. Nous pouvons alors ajouter un objet, un simple rectangle blanc, qui fera office de fond pour notre fenêtre. Pour ce faire, il nous suffit d'ajouter un objet Evas (**Evas_Object**) ainsi :

Fichier
```
Evas_Object *bg;
[...]
bg = evas_object_rectangle_add(canvas);
```

Nous pouvons ensuite définir quelques propriétés de **bg** :

Fichier
```
// sa couleur
evas_object_color_set(bg, 255, 255, 255, 255);
// sa position
evas_object_move(bg, 0, 0);
// sa taille
evas_object_resize(bg, WIDTH, HEIGHT);
// sa visibilité
evas_object_show(bg);
// et son focus (pour les événements pointeurs)
evas_object_focus_set(bg, EINA_TRUE)
```

À ce stade, rien n'est visible, bien que nous ayons rendu visible **bg** la fenêtre Evas **window**, elle ne l'est pas. Nous procédons alors dans ce sens avant d'entrer dans la boucle événementielle d'Ecore :

Fichier
```
ecore_evas_show(window);

ecore_main_loop_begin();
```

Comme avec n'importe quel toolkit ou bibliothèque graphique comme SDL, la construction d'une interface graphique repose sur une gestion d'événements : saisie clavier, opération de pointeur (souris, *touchscreen*, etc.). Le *moteur* de l'interface utilisée repose sur une boucle infinie permettant la capture de ces événements et leur traitement. Ici **ecore_main_loop_begin()** est équivalent au **gtk_main()** de GTK+, ou un **while(true)** tel qu'on en retrouve avec d'autres solutions.

Nous n'avons pas encore prévu de système permettant de quitter notre programme de manière élégante. La boucle événementielle ne va donc s'arrêter qu'avec l'arrêt brutal du programme lui-même (**kill**, **SIGINT**, etc.). Cependant, après l'appel à la fonction, nous prévoyons de suite un brin de nettoyage avec le marquage pour suppression de **bg** et la libération des ressources de **window** avant l'appel à **ecore_evas_shutdown()** :

Fichier
```
evas_object_del(bg);
ecore_evas_free(window);
```

L'arrêt violent n'est pas très intéressant, d'autant que nous pouvons très simplement ajouter quelques lignes de code pour faire propre. Ainsi, nous commençons par déclarer une fonction de callback :

Fichier
```
void
key_down(void *data, Evas *e,
        Evas_Object *obj, void *event_info)
{
  Evas_Event_Key_Down *ev;
  ev = (Evas_Event_Key_Down *)event_info;
  printf("You hit key: %s\n", ev->keyname);

  if (!strcmp(ev->keyname, "q"))
    ecore_main_loop_quit();
}
```

Cette fonction prend en argument respectivement, un pointeur sur des données *personnelles*, un pointeur sur le canevas concerné, un pointeur vers l'objet Evas ayant déclenché l'événement et enfin, un pointeur sur l'information concernant l'événement (**`event_info`**). Notre fonction récupère l'information dans **`ev`** qu'on sait être de type **`Evas_Event_Key_Down`** (cf. ci-après) et l'utilise avec **`strcmp()`** pour déterminer le comportement à adopter. S'il s'agit bien de la touche « q », il nous suffit d'appeler **`ecore_main_loop_quit()`** pour provoquer la sortie de la boucle événementielle Ecore.

Notre callback prêt, nous pouvons lier son appel à un événement juste avant d'appeler **`ecore_evas_show(window)`** :

Fichier

```
evas_object_event_callback_add(bg,
        EVAS_CALLBACK_KEY_DOWN, key_down, NULL);
```

On passe en paramètre à la fonction, l'objet auquel attacher le callback, le type d'événement le déclenchant, le nom de notre fonction callback et éventuellement un pointeur vers des données à passer à la fonction en question (ou **`NULL`**). Le type d'événement peut être vu comme un système de filtrage, mais c'est également l'élément qui détermine le type d'information (**`event_info`**) passée et donc, la manière de *caster*. Un coup d'oeil à la documentation (**http://docs.enlightenment.org/auto/evas/group__Evas__Object__Group__Events.html**) des EFL nous apprend ainsi que l'événement de type **`EVAS_CALLBACK_KEY_DOWN`** nous passe un pointeur sur une structure **`Evas_Event_Key_Down`**.

La compilation de notre code se fera exactement comme le précédent et son exécution nous présentera une magnifique fenêtre blanche dénuée de tout intérêt… qu'il sera cependant possible de fermer en appuyant simplement sur la touche **`q`**. Vous pouvez éventuellement basculer sur une console Getty (Ctrl+Alt+F1), afin de lancer à nouveau ce programme et constater qu'il fonctionnera tout aussi bien avec le backend en framebuffer (si vous disposez d'un tel périphérique et avez les permissions adéquates). En déclarant la variable d'environnement **`EINA_LOG_LEVEL=3`**, vous aurez tout le loisir de voir Evas déterminer le backend adapté…

C'est bien gentil, mais il est temps maintenant de passer aux choses sérieuses…

2. BASCULONS SUR EDJE !

Avec Evas seul, si nous souhaitons concevoir une interface graphique complète, nous nous retrouvons dans une situation assez similaire à celle de bon nombre de toolkits connus tels qu'ils existaient dans leur première version : nous devons, dans notre code, placer les objets et composer l'interface. Ceci n'est ni pratique, ni agréable, mais fort heureusement, les EFL nous proposent une magnifique solution : Edje !

Edje est une bibliothèque permettant de créer des designs graphiques structurés. Derrière cette désignation un peu absconse se cache une machinerie monstrueusement puissante. Pour appréhender Edje comme il se doit, imaginez simplement que ce système offre, non pas des widgets comme le ferait un toolkit (menus, boutons, curseur, etc.), mais tout ce qu'il faut pour créer un tel toolkit avec, en plus, la capacité d'animer, superposer, grouper et contrôler les éléments graphiques.

Un fichier Edje (**`.edj`**) regroupe les éléments graphiques (formes, images, polices) et la logique qui s'y rattache, tout en incluant des programmes pouvant décrire le comportement des éléments et les relations avec votre code écrit en C. Il est ainsi possible d'envoyer des signaux aux éléments et déclencher des comportements planifiés, ainsi que de recevoir des signaux en provenance de ces objets afin de pouvoir réagir en conséquence dans votre

code. En d'autres termes, Edje permet de séparer tout le comportement graphique de votre application, de la mécanique « métier » de votre programme. Votre code C n'a plus besoin de se charger de choses relevant purement de l'aspect GUI, comme l'animation d'un bouton ou l'affichage/masquage d'un menu. Ceci permet, par effet de bord, de décliner votre interface graphique en autant de versions qu'il vous plaira, sans avoir à toucher ou même recompiler votre code, allant ainsi bien au-delà d'un simple moteur de thèmes.

2.1 Évolution du code

En guise d'exemple, nous pouvons revoir le code de notre dernier exemple, de manière à ne pas créer et manipuler directement un objet Evas **bg**, mais en reposant sur Edje. Éliminons donc tout ce qui se rapporte à **bg** et commençons par créer un objet Edje (qui est, bien entendu, un objet Evas comme le précédent) :

Fichier

```
  Evas_Object *edje;

[...]

  edje = edje_object_add(canvas);
  if (!edje) {
    EINA_LOG_CRIT("could not create edje object!");
  }
```

Notre objet créé, nous n'avons pas besoin d'en définir les caractéristiques comme précédemment. Il nous suffit de préciser le fichier Edje à utiliser et de le charger :

Fichier

```
  if (!edje_object_file_set(edje, "base.edj", "my_group")) {
    int err = edje_object_load_error_get(edje);
    const char *errmsg = edje_load_error_str(err);
    EINA_LOG_ERR("could not load 'my_group' from base.edj: %s", errmsg);
    evas_object_del(edje);
  }
```

Notez que la fonction **edje_object_file_set()** prend en argument non seulement le nom de fichier **base.edj**, mais également le *groupe* qui constitue l'objet que nous initialisons (**edje**). Un fichier peut, en effet, contenir plusieurs groupes et donc stocker bien des objets, voire plusieurs versions d'une même interface.

On utilise ensuite l'objet **edje** comme nous l'avons fait avec **bg** en associant le callback et en l'intégrant au canevas :

Fichier

```
evas_object_focus_set(edje, EINA_TRUE);
evas_object_event_callback_add(edje, EVAS_CALLBACK_KEY_DOWN, key_down, NULL);

evas_object_move(edje, 0, 0);
evas_object_resize(edje, WIDTH, HEIGHT);
evas_object_show(edje);
```

Bien entendu, en sortie de boucle principale Ecore, nous n'oublions pas de faire un brin de ménage tout comme nous l'avions fait pour **bg** :

Fichier

```
evas_object_del(edje);
```

2.2 Notre fichier Edje

Il ne nous reste plus qu'à nous pencher sur le fichier Edje. Celui-ci est obtenu par compilation, avec l'outil **`edje_cc`**, d'un fichier source qu'on aura pour habitude de suffixer **`.edc`**. Un source Edje est un simple fichier texte structuré :

Fichier

```
collections {
  group {
    images {
      [...]
    }
    parts {
      part {
        description {
          [...]
        }
      }
    }
    programs {
      program {
        [...]
      }
    }
  }
}
```

Nous n'avons pas présenté ici l'ensemble de la syntaxe, mais uniquement les éléments dont nous allons faire usage. Edje permet de faire ÉNORMÉMENT de choses, comme inclure des éléments audio, des scripts Pawn ou Lua, des composants texte (polices, etc.), une gestion multi-résolution pour les images, et bien d'autres choses. Décrire et détailler Edje par l'exemple nécessiterait tout un magazine (sinon deux) et nous nous en tiendrons aux fonctions basiques (qui sont déjà impressionnantes).

Un fichier Edje contient une liste (une collection) de groupes qui composent un « thème ». Notre programme charge par exemple **`my_group`**. Un groupe est constitué de **`parts`** et de **`programs`**. Une **`part`** est l'élément le plus basique d'un ensemble/thème Edje. Elle possède un type, un ou plusieurs état(s), ainsi qu'un ensemble de caractéristiques, groupés en **`description`**. Enfin, un **`program`** est un composant actif permettant de définir comment votre interface va réagir à certains événements en changeant l'état des **`parts`** ou en générant d'autres événements.

Tout ceci est très dense. Sans plus attendre, voyons notre fichier source Edje pour asseoir toutes ces considérations très abstraites :

Fichier

```
collections {
  group {
    name: "my_group";
    parts {
      part {
        name: "background";
        type: RECT;
        mouse_events: 0;
        description {
           state: "default" 0.0;
           color: 255 255 255 255;
           rel1 {
              relative: 0.0 0.0;
              offset: 0 0;
           }
           rel2 {
```

Fichier

```
                relative: 1.0 1.0;
                offset: -1 -1;
            }
        }
    }
  }
}
```

Nous avons ici un groupe défini par le nom **my_group**. Celui-ci est composé d'une **part** nommée **background**, de type rectangulaire (**RECT**) et n'acceptant pas les événements du périphérique de pointage. L'état par défaut de notre **background** est celui qui sera automatiquement utilisé. Il décrit des paramètres qui sont propres au type d'objet, mais qui peuvent également être communs à plusieurs types (comme la couleur et la transparence).

Parmi les éléments décrivant un état, nous trouvons sa taille et sa position, définis par **rel1** et **rel2**. Le système de positionnement est un point important puisqu'il s'agit là de quelque chose que vous utiliserez souvent sinon systématiquement. **rel1** et **rel2** définissent respectivement le coin supérieur gauche et le coin inférieur droit d'un élément relativement à un autre élément. Dans le cas présent, ce positionnement se fait par rapport au positionnement du groupe lui-même (comportement par défaut). Dans d'autres situations, vous pourrez être amené à positionner un élément par rapport à un autre, ce qui aura pour effet de lier la position de l'un à l'autre et ainsi de les déplacer en même temps.

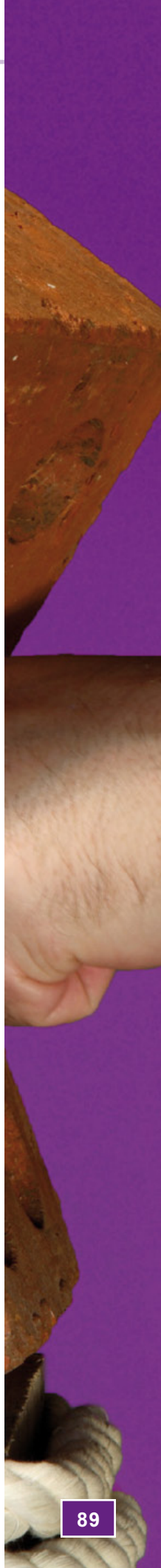
Le système de coordonnées utilisé se base sur des valeurs données en pixels. Cependant, dans bien des cas, ceci ne constitue pas une solution très pratique. On utilise alors la directive **relative** afin de spécifier des coordonnées sous forme de proportions, soit relativement à un autre élément (via **to:** suivi du nom de l'élément), soit, par défaut, relativement au groupe lui-même. Ici, le coin supérieur gauche de notre rectangle blanc correspond à la position de **my_group**, lui-même défini dans le code C. Il en va de même pour **rel2**, le coin inférieur droit, à **1.0 1.0**, correspondant à « la fin » du groupe conteneur. Vous pouvez voir ce système de coordonnées comme utilisant des pourcentages. **0.0** correspondant à 0%, **0.5** à 50% et **1.0** à 100% d'une position.

offset représente le décalage souhaité avec l'élément de référence. Les deux lignes spécifiées ici sont « classiques ». La valeur pour **rel1** est parfaitement inutile et celle pour **rel2** est une correction. En effet, notre **background** a **exactement** la même taille que **my_group**, mais les pixels sont comptés à partir de zéro (**rel[1|2]** indique une **position** en pixels, mais **relative** utilise la **taille** d'un élément). Il est nécessaire, pour obtenir un parfait alignement, de retrancher un pixel à la position du coin inférieur droit (**rel2**). Ceci est tellement classique que ces deux lignes n'ont parfaitement aucun autre intérêt que pédagogique car, en leur absence, la valeur de décalage par défaut est justement **0 0** et **-1 -1**.

À présent, pour obtenir notre **base.edj**, il nous suffit d'invoquer **edje_cc**. L'option **-v** nous apportera quelques informations statistiques qui peuvent s'avérer intéressantes :

Terminal

```
% edje_cc -v base.edc
edje_cc: Wrote       106 bytes (   0Kb) for "edje_file" header
edje_cc: header: 0.00032
edje_cc: groups: 0.00047
edje_cc: scripts: 0.00000
edje_cc: lua scripts: 0.00000
edje_cc: source: 0.00018
edje_cc: fontmap: 0.00008
edje_cc: images: 0.00000
edje_cc: fonts: 0.00000
```



Terminal

```
edje_cc: sounds: 0.00000
edje_cc: THREADS: 0.00000
Summary:
  Wrote 1 collections
  Wrote 0 images
  Wrote 0 sounds
  Wrote 0 fonts
```

Bien entendu, la solution idéale est de, tout simplement, modifier notre **Makefile** :

Fichier

```
TARGET  := base
WARN    := -Wall
CFLAGS  := -O2 ${WARN} `pkg-config --cflags eina evas ecore ecore-evas edje`
LDFLAGS := `pkg-config --libs eina evas ecore ecore-evas edje`
CC      := gcc

C_SRCS    = $(wildcard *.c)
OBJ_FILES = $(C_SRCS:.c=.o)

all: ${TARGET} ${TARGET}.edj

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

${TARGET}: ${OBJ_FILES} ${TARGET}.edj
        ${CC} ${WARN} ${LDFLAGS} -o $@  $(OBJ_FILES)

${TARGET}.edj: ${TARGET}.edc
        edje_cc -id images -sd images -fd images ${TARGET}.edc

clean:
        rm -rf *.o ${TARGET}  ${TARGET}.edj
```

Notez les options **-id**, **-sd** et **-fd** permettant respectivement de spécifier le répertoire où chercher d'éventuels images, fichiers audio et polices de caractères (cf. plus loin dans l'article). Un simple **make** produira à la fois le binaire **base** et le fichier Edje **base.edj**. Lancer le binaire produira le même affichage d'un carré blanc que notre code précédent... mais cette fois avec l'aide de Edje !

3. UN PEU PLUS LOIN AVEC EDJE

Inutile de se le cacher, cette première réalisation n'est pas ce qu'on pourrait appeler une démonstration technologique. En revanche, il s'agit d'une bonne base pour en faire une. Nous ne toucherons plus, pour l'instant, au code C, mais nous pencherons exclusivement sur le source Edje, car c'est bien là que réside toute la magie. Que diriez-vous à présent d'ajouter un peu d'animation ?

Mais avant cela, voyons une particularité du langage de description utilisé par Edje en prenant, par exemple, ce morceau de code :

Fichier

```
rel1 {
  relative: 0.0 0.0;
  offset: 0 0;
}
```

Cette façon de décrire cette caractéristique n'est que l'une des deux options qui se présentent à vous. Ainsi, vous risquez de trouver tantôt dans les fichiers **.edc** (il est possible de décompiler un **.edj** à l'aide de l'outil **edje_decc**), la syntaxe suivante :

Fichier
```
rel1.relative: 0.0 0.0;
rel1.offset: 0 0;
```

Ceci permet, pour les **part** nécessitant peu de description, d'avoir une syntaxe plus compacte. Mais revenons à nos moutons et ajoutons maintenant une image en plus du fond. Intégrer des fichiers dans un fichier Edje est relativement simple. Il suffit de les spécifier ainsi dans le source :

Fichier
```
images {
  image: "fond.png" COMP;
  image: "aquaburn.png" COMP;
}
```

Il seront ainsi embarqués dans le **.edj**, ici avec une compression sans perte (**COMP**). Le chemin de recherche de ces fichiers est spécifié par l'option **-id** (*image directory*) passée à **edje_cc**. Il est également possible de spécifier une compression avec perte (**LOSSY** suivi, en option, d'un taux de compression), aucune compression (**RAW**), ou encore un référencement externe pour ne pas embarquer le fichier (**USER**). **images** peut être placé dans la portée d'une **collection**, d'un **group** ou d'une **part** afin de faciliter l'organisation de vos données (en particulier si vous souhaitez ensuite diviser le source en plusieurs fichiers). Notez qu'une directive **set** existe, permettant de décliner un élément graphique en un *set* composé d'images identiques mais dans des résolutions/tailles différentes. Ceci cependant dépasse le cadre de ce simple article d'introduction.

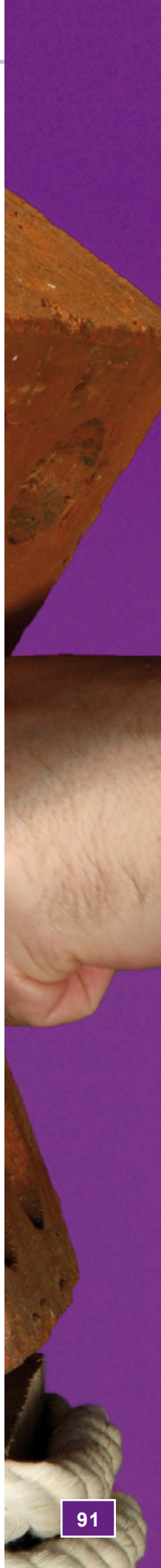
Avec ces deux images, nous pouvons travailler sur quelque chose de plus respectable. Commençons par changer le fond en commentant (avec **//** ou **/*** [...] ***/**) le type de **part** pour **background**, ou en le changeant de **RECT** en **IMAGE** (type par défaut en l'absence de **type:**). Dans la description, nous pouvons ensuite spécifier l'image à utiliser, telle que déclarée dans le bloc que nous venons de présenter :

Fichier
```
image {
  normal: "fond.png";
}
Ou encore :
image.normal: "fond.png";
<file>
```

Toujours dans la portée de la description, il nous faut ensuite définir la manière dont la zone sera remplie par l'image :

Fichier
```
<file>
  fill {
    size.relative: 0.0 0.0;
    size.offset: 64 64;
  }
```

size.relative (ou **relative** dans la portée d'un **size** donc. Je sais, j'insiste), indique le point de départ, relativement à l'objet conteneur, à partir d'où il faut commencer à remplir. **size.offset** détermine la largeur et la hauteur en pixels sur laquelle l'image couvrira l'objet. Si **size.offset** est plus petit que l'objet, l'image sera répétée pour couvrir la zone (*tilling*). C'est ici précisément ce que nous souhaitons, puisque notre **background** hérite de la taille du canevas Evas et fait donc 640×640 pixels. Pour ce type de remplissage de fond, on a pour habitude de spécifier pour **size.offset** la taille effective du fichier source. Je vous recommande de tester avec différentes valeurs pour vous imprégner du fonctionnement de ces directives.



En résumé, notre **part** ressemble donc maintenant à ceci :

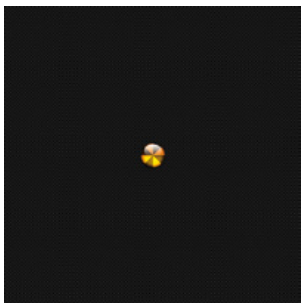
Fichier

```
part {
  name: "background";
  mouse_events: 0;
  description {
     image.normal: "fond.png";
     fill {
       size.relative: 0.0 0.0;
       size.offset: 64 64;
     }
     state: "default" 0.0;
     rel1.relative: 0.0 0.0;
     rel1.offset: 0 0;
     rel2.relative: 1.0 1.0;
     rel2.offset: -1 -1;
   }
}
```

Après génération du fichier **base.edj** et lancement de notre programme, nous constatons que le fond est pavé avec l'image, comme souhaité. Nous pouvons directement utiliser ces nouvelles connaissances pour ajouter un nouvel élément (**part**) à notre fichier Edje :

Fichier

```
part {
  name: "B1";
  description {
    state: "default" 0.0;
    image.normal: "aquaburn.png";
    fill {
      size.relative: 0.0 0.0;
      size.offset: 64 64;
    }
    rel1.relative: 0.45 0.45;
    rel2.relative: 0.55 0.55;
  }
}
```



Rien de nouveau ici. Nous ne faisons qu'ajouter un élément **B1** rempli avec une image de 64 pixels de côté, que nous plaçons au centre de **my_group** (notez les valeurs de **relative** correspondant à une taille de 1/10 de celle du canevas et donc à une zone de 64 pixels de côté). Après cette intégration, l'exécution de notre programme nous donne ceci :

Il est temps maintenant de créer un peu de mouvement. Pour ce faire, nous allons produire un autre état pour notre objet **B1** en ajoutant une nouvelle description :

Fichier

```
description {
  state: "up" 0.0;
  inherit: "default" 0.0;
  rel1.relative: 0.10 0.10;
  rel2.relative: 0.20 0.20;
}
```

Assez similaire à la précédente, la seule différence réside dans la position de notre élément qui se retrouve proche du coin supérieur gauche. Cette état est nommé **up** et la directive **inherit:** nous permet d'hériter de toutes les autres valeurs provenant de l'état **default**. Nous n'avons pas encore parlé de la valeur suivant le nom spécifié pour **state:** et **inherit:**. Il s'agit d'un index (entre

0.0 et **1.0**) permettant de décliner un état. Cependant, bon nombre des documentations précisent que cette valeur n'est pas utilisée et tous les fichiers sources que nous avons eu l'occasion d'étudier utilisent systématiquement la valeur **0.0** (la valeur d'index est, par ailleurs, optionnelle depuis Edje 1.7.0, mais la 1.2.0, considérée comme une version majeure, est encore largement installée sur bon nombre de systèmes).

Ce second état n'est, pour l'instant, pas utilisé. Il nous faut quelque chose pour signifier le passage de **default** à **up**. Nous pourrions le faire depuis notre code C, mais il est plus simple d'écrire directement un programme dans notre source Edje. Le scénario est le suivant : au chargement et à l'affichage de la fenêtre, l'élément **B1** est montré au centre, mais cet affichage devra également déclencher le passage de **B1** à l'état **up**.

Dans la portée de notre **group**, nous allons donc inclure un programme ainsi :

Fichier

```
programs {
  program {
    name: "to_up";
    signal: "show";
    action: STATE_SET "up" 0.0;
    target: "B1";
    transition: LINEAR 0.50;
  }
}
```

Les programmes peuvent être déclenchés par des signaux, ou à la suite d'autres programmes. Ici, **signal: "show"** correspond à l'événement découlant de l'affichage de l'objet Evas (le groupe) au sein de notre canevas. Notre programme **to_up** a pour effet (**action:**) de changer l'état (**STATE_SET**) de **B1** (**target**) en **up**. Comme nous souhaitons que ceci ne se fasse pas de manière instantanée, nous précisons une transition de type linéaire (**LINEAR**) sur une durée d'une demi-seconde (**0.50**). Nos modifications s'arrêtent là ! Dès le lancement du programme, la fenêtre apparaît et nous voyons se déplacer notre élément du centre vers le coin de la fenêtre, puis stopper à cet endroit.

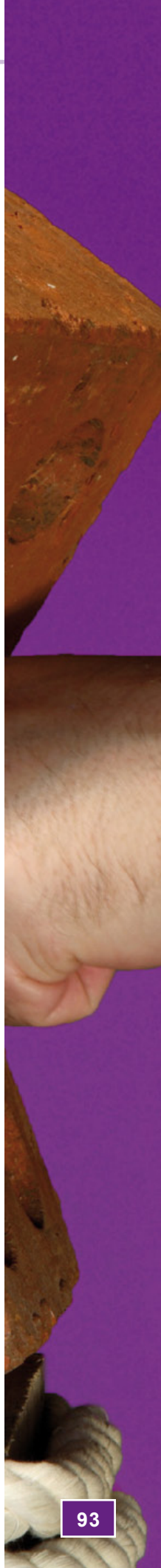
Envie d'une petite boucle ? Rien de plus simple, ajoutons un nouveau programme :

Fichier

```
program {
  name: "to_default";
  in: 0.25 0;
  action: STATE_SET "default" 0.0;
  target: "B1";
  transition: SPRING 0.5 4.5 6;
  after: "to_up";
}
```

Celui-ci aura pour effet de repasser **B1** à l'état **default**, mais son exécution est retardée d'un quart de seconde (**in:** prend en argument un nombre de secondes plus une valeur entre zéro et x secondes aléatoires, ici **0.0** = pas d'aléa). La transition choisie est un effet « élastique » d'une durée de 0,5 seconde, avec une diminution non linéaire (**0.0** = linéaire, >0 = diminution plus rapide) s'étalant sur 6 mouvements (essayez/observez, c'est plus simple à comprendre). **after:** nous permet d'enchaîner sur un autre programme et une ligne équivalente sera également ajoutée en fin du programme **to_up** pointant sur **to_default**. En quelques lignes, nous provoquons un mouvement perpétuel, indépendant de notre code C. L'élément **B1** se déplace vers le coin, y reste un quart de seconde et retourne au centre comme tenu par un élastique. Puis le mouvement reprend...

En utilisant les états, l'héritage des positions et propriétés, les signaux et les programmes, il est possible de composer des interfaces dont le comportement est majoritairement non seulement autonome, mais complètement révisable sans avoir à toucher au code qui fait le cœur de votre application.



4. LIEN AVEC LE CODE C

À ce stade, nous avons d'une part le code qui ne fait que charger le fichier Edje et afficher la fenêtre du canevas, et de l'autre, nos éléments graphiques avec un fonctionnement autonome. Il est temps de jeter un pont entre les deux mondes : nous allons échanger des signaux.

Révisons tout d'abord nos programmes Edje :

Fichier

```
programs {
  program {
    name: "to_up";
    signal: "byebye";
    source: "dacode";
    action: STATE_SET "up" 0.0;
    target: "B1";
    transition: LINEAR 0.50;
    after: "to_default";
  }
  program {
    name: "to_default";
    in: 0.25 0;
    action: STATE_SET "default" 0.0;
    target: "B1";
    transition: SPRING 0.5 4.5 6;
    after: "quit";
  }
  program {
    name: "quit";
    action: SIGNAL_EMIT "quit" "last";
  }
}
```

to_up ne réagit plus au signal **`show`** émis automatiquement à l'affichage de l'objet, mais à un signal personnalisé **`"byebye"`**. L'effet de ce signal reste inchangé et nous passons ensuite le relais à **`to_default`** comme précédemment. Cependant, nous supprimons la boucle, car à l'issue de l'exécution de ce programme, nous appelons **`quit`** qui, enfin, a pour seule tâche d'émettre un signal **`"quit"`** provenant de l'émetteur auto-déclaré **`"last"`**. Les signaux sont des chaînes de caractères arbitrairement choisies par le développeur. Il en va de même pour la source. Notez que le programme **`to_up`** filtre les messages, car nous avons spécifié **`source: "dacode"`**. Il ne s'exécutera donc qu'à réception du signal, si celui-ci est envoyé par l'émetteur en question. Ceci permet d'affiner la gestion des signaux et de déclencher des actions sur certains éléments spécifiques uniquement ou, inversement, de globaliser les signaux au besoin. Le scénario est le suivant : réception de **`"byebye"`**, animation, pause, retour, émission de **`"quit"`**.

Côté code C, nous modifions le callback déclenché par l'appui d'une touche. Nous ne quittons plus la boucle principale Ecore, mais émettons le signal **`"byebye"`** à destination de notre objet Evas **`edje`** :

Fichier

```
void
key_down(void *data, Evas *e, Evas_Object *obj, void *event_info)
{
  Evas_Event_Key_Down *ev;
  ev = (Evas_Event_Key_Down *)event_info;

  if (!strcmp(ev->keyname, "q"))
    edje_object_signal_emit(obj, "byebye", "dacode");
}
```

Après animation, le signal **"quit"** est émis et nous devons alors le traiter dans notre code C afin de procéder à l'arrêt propre du programme. Nous commençons donc par créer la fonction *handler* qui traitera le signal. Notre sortie de boucle Ecore se retrouve ici :

Fichier

```
static void
quitHandler (void *data, Evas_Object *obj,
    const char  *emission, const char  *source)
{
  ecore_main_loop_quit();
}
```

Enfin, dans le **main()**, avant affichage de notre objet **edje**, non loin de la définition du callback pour l'évènement **EVAS_CALLBACK_KEY_DOWN**, il nous suffit d'ajouter le callback vers la fonction de gestion du signal :

Fichier

```
edje_object_signal_callback_add(
    edje,
    "quit",
    "last",
    quitHandler, NULL);
```

C'est tout ! Recompilez et lancez le programme. Par l'appui de la touche **q** l'élément central va faire son petit manège puis revenir en place, ce qui aura pour effet de quitter l'application. Le tour est joué !

CONCLUSION

Ce que nous venons de voir n'est qu'un petit aperçu de ce que permettent de faire Edje et les EFL en général. Les fonctionnalités de Edje ne sont pas sans rappeler les mécanismes de séparation entre le fond et la forme que l'on retrouve, par exemple, dans LaTeX. Ceci tout en conservant une forte interactivité entre l'interface et le code. Les signaux ne sont pas la seule manière d'échanger des données. Il est, par exemple, possible depuis le code C de modifier des caractéristiques des éléments graphiques. Les éléments textuels sont un parfait exemple de ce type de besoins. Produire un bot graphique se connectant à un serveur XMPP afin d'afficher les messages lui parvenant, à grands recours d'effets graphiques, est l'affaire de quelques heures. Mieux encore, la légèreté et la polyvalence d'Evas, rendent cela possible non seulement sur une machine puissante avec une carte 3D, mais également sur un système embarqué. Et ce, avec la même simplicité et la même souplesse.

J'espère que cet article vous aura convaincu d'investir un peu de votre temps et de votre énergie dans l'étude et l'essai des EFL. Notez que ces bibliothèques forment également la base graphique du système pour smartphone (et autres *appliances* multimédias) Tizen de Samsung (**https://www.tizen.org/fr**). Bien que l'environnement de développement soit radicalement différent (même pour le développement d'applications dites « native »), il n'en reste pas moins qu'il s'agit là d'une parfaite démonstration de l'adaptabilité des EFL. Un autre bel exemple est le système domotique open source Calaos, qui utilise les EFL pour construire son interface graphique (**https://calaos.fr/**). Et n'oublions pas Enlightenment 0.17 qui est, tout simplement, la raison d'être initiale des EFL. Autant de sources d'inspiration et de documentation pour vos interfaces et vos codes... ▮

4

4 AUTRES LANGAGES

À découvrir dans cette partie...

4.1 Utilisez Python dans vos applications C

Python et C sont deux langages complémentaires, qu'il est tout à fait possible d'utiliser conjointement. On peut par exemple envisager de réutiliser des modules Python qui n'ont pas d'équivalent en C.
Un exemple précis ici, avec le module « pickle » du langage Python.

UTILISEZ PYTHON DANS VOS APPLICATIONS C

Sébastien Chazallet - Ingénieur logiciels libres

Lors du dernier numéro hors-série sur le langage C, nous vous avions proposé d'étendre Python avec des modules C. Nous avions alors introduit quelques bases. Aujourd'hui, nous vous proposons le contraire : utiliser des modules Python au sein de vos applications en C.

1. PRÉSENTATION

Comme on le sait, Python et C sont deux langages parfaitement complémentaires. Ils ne sont pas faits pour s'opposer, mais pour travailler ensemble. En effet, en caricaturant légèrement, on peut dire que Python montre ses qualités là où C montre ses limites et inversement. De plus, les deux langages sont parfaitement utilisables conjointement.

La question que l'on peut se poser est : à quoi pourrait bien servir le fait d'intégrer du code Python à mon application C ? La réponse peut être multiple : intégrer un module prototype qui sera plus rapidement réalisable en Python (ce qui est courant dans la finance), réutiliser des modules Python qui n'ont pas d'équivalent en C, ...

Pour cet article, nous allons prendre un exemple concret. En effet, Python dispose d'un module nommé **pickle** qui permet de sérialiser des données dans un format brut, qui n'est pas fait pour être lu par d'autres technologies, mais pour être performant au regard de son utilisation par le langage Python, et simplement pour cela.

On va donc proposer de créer une fonctionnalité permettant d'aller lire et afficher le contenu d'un fichier écrit avec le module **pickle**, le tout sans réinventer la roue, en réutilisant la méthode **load** proposée par Python.

2. UN PEU DE PYTHON

2.1 Introduction à Python

On se propose, pour ceux qui ne connaissent pas Python, de découvrir rapidement quelques tuyaux permettant de se débrouiller seul pour découvrir rapidement le langage et ce module **pickle**. Pour commencer, ouvrons une console, dans un répertoire de travail que l'on aura préalablement créé :

Terminal
```
$ python
```

En Python, pour utiliser un module, il faut l'importer :

Terminal
```
>>> import pickle
```

Le fait d'importer un module crée en réalité une variable, portant le nom du module et pointant vers ce module. Il est possible de lister tout ce que le module contient à l'aide de la fonction **dir** :

Terminal
```
>>> dir(pickle)
['APPEND', 'APPENDS', 'BINFLOAT', 'BINGET', 'BININT', 'BININT1',
'BININT2', 'BINPERSID', 'BINPUT', 'BINSTRING', 'BINUNICODE', 'BUILD',
'BooleanType', 'BufferType', 'BuiltinFunctionType', 'BuiltinMethodType',
'ClassType', 'CodeType', 'ComplexType', 'DICT', 'DUP', 'DictProxyType',
'DictType', 'DictionaryType', 'EMPTY_DICT', 'EMPTY_LIST', 'EMPTY_TUPLE',
'EXT1', 'EXT2', 'EXT4', 'EllipsisType', 'FALSE', 'FLOAT', 'FileType',
'FloatType', 'FrameType', 'FunctionType', 'GET', 'GLOBAL', 'GeneratorType',
'GetSetDescriptorType', 'HIGHEST_PROTOCOL', 'INST', 'INT', 'InstanceType',
'IntType', 'LIST', 'LONG', 'LONG1', 'LONG4', 'LONG_BINGET', 'LONG_BINPUT',
```

Terminal

```
'LambdaType', 'ListType', 'LongType', 'MARK', 'MemberDescriptorType',
'MethodType', 'ModuleType', 'NEWFALSE', 'NEWOBJ', 'NEWTRUE', 'NONE',
'NoneType', 'NotImplementedType', 'OBJ', 'ObjectType', 'PERSID', 'POP',
'POP_MARK', 'PROTO', 'PUT', 'PickleError', 'Pickler', 'PicklingError',
'PyStringMap', 'REDUCE', 'SETITEM', 'SETITEMS', 'SHORT_BINSTRING', 'STOP',
'STRING', 'SliceType', 'StringIO', 'StringType', 'StringTypes', 'TRUE',
'TUPLE', 'TUPLE1', 'TUPLE2', 'TUPLE3', 'TracebackType', 'TupleType', 'TypeType',
'UNICOD', 'UnboundMethodType', 'UnicodeType', 'Unpickler', 'UnpicklingError',
'XRangeType', '_EmptyClass', '_Stop', '__all__', '__builtins__', '__doc__',
'__file__', '__name__', '__package__', '__version__', '_binascii',
'_extension_cache', '_extension_registry', '_inverted_registry', '_keep_alive',
'_test', '_tuplesize2code', 'classmap', 'compatible_formats', 'decode_long',
'dispatch_table', 'dump', 'dumps', 'encode_long', 'format_version', 'load',
'loads', 'marshal', 'mloads', 're', 'struct', 'sys', 'whichmodule']
```

Il faut savoir qu'en Python, tout est objet ; on aura l'occasion d'y revenir. Les règles de nommage permettent de savoir à quoi l'on a affaire : les constantes en majuscules, les classes en CamelCase, les variables et fonctions en minuscules. Lorsque l'on commence par un caractère souligné, il s'agit d'un **objet privé**, tandis que lorsque l'on commence et termine par deux caractères soulignés, il s'agit d'un **objet spécial**, c'est-à-dire qui est susceptible d'être utilisé par des fonctionnalités de base du langage.

Il est possible d'obtenir une aide plus ou moins développée sur chaque objet :

Terminal

```
>>> help(pickle.load)
Help on function load in module pickle:

load(file)
>>> help(pickle.dump)
Help on function dump in module pickle:

dump(obj, file, protocol=None)
```

Ici, l'aide n'est pas très fournie, mais on a au moins la signature des méthodes.

Comme on peut le supposer, la méthode `load` permet de charger les données depuis un fichier et la méthode `dump` d'y enregistrer des données, ce que leurs signatures respectives confirment.

2.2 Création de fichiers de test

Voici un scénario complet pour créer deux fichiers de test :

Terminal

```
>>> import pickle
>>> with open('test1.pkl', 'w') as f:
...     print pickle.dump("Hello World", f)
...
None
>>> with open('test2.pkl', 'w') as f:
...     print pickle.dump({"test": 42, "say": "Hello World", "func":
<function load at 0xcf6488>}, f)
...
None
```

Voici comment lire un de ces fichiers :

Terminal
```
>>> with open('test2.pkl', 'r') as f:
...     print pickle.load(f)
...
{"test": 42, "say": "Hello World", "func": pickle.load}
```

Ce qu'il faut comprendre, est qu'à la lecture, on obtient directement un objet Python, ici un dictionnaire dont les clés comme les valeurs peuvent être des objets quelconques, mais que ce qui est affiché à l'écran est la représentation textuelle de ces objets, ce qui n'a pas toujours de sens.

3. DANS LE VIF DU SUJET

3.1 Principes généraux

Pour commencer, lorsque vous appelez un programme Python depuis C, il est nécessaire de commencer par lancer la machine virtuelle Python, au moment où vous en avez besoin, puis de l'éteindre au moment où elle n'est plus utile. Ceci se fait ainsi :

Fichier
```
Py_Initialize();
Py_Finalize();
```

Ensuite, le point important est le suivant : **en Python, tout est objet**. Un objet Python est représenté sous la forme d'une structure **`PyObject`**. Ce fait n'est pas une simple marotte, c'est un fait, et ce fait a de multiples implications. Entre autres, en Python, les chaînes de caractères, entiers, flottants, mais aussi les listes, les dictionnaires ou encore les classes, fonctions, méthodes ou modules, sont des objets.

Chacun de ces objets est particulier et dispose de ses propres méthodes, lesquelles sont déclinées de manière à être utilisées en C. Il est donc nécessaire de savoir passer d'un type C à son équivalent Python, mais aussi d'aller chercher une fonction par exemple, ou importer un module.

Pour commencer, Python dispose de deux objets uniques permettant de représenter les deux booléens. Les voici :

Fichier
```
PyObject* Py_False
PyObject* Py_True
```

Voici les signatures des fonctions permettant d'obtenir un objet Python de type **`long`** :

Fichier
```
PyObject* PyLong_FromLong(long v)
PyObject* PyLong_FromUnsignedLong(unsigned long v)
PyObject* PyLong_FromDouble(double v)
PyObject* PyLong_FromString(char *str, char **pend, int base)
```

Voici comment faire l'opération inverse :

Fichier
```
long PyLong_AsLong(PyObject *pylong)
unsigned long PyLong_AsUnsignedLong(PyObject *pylong)
double PyLong_AsDouble(PyObject *pylong)
```

Voici les signatures des fonctions permettant d'obtenir un objet Python de type **int** :

Fichier
```
PyObject* PyInt_FromString(char *str, char **pend, int base)
PyObject* PyInt_FromLong(long ival)
long PyInt_AsLong(PyObject *io)
```

Voici comment faire l'opération inverse :

Fichier
```
PyObject* PyFloat_FromString(PyObject *str, char **pend)
PyObject* PyFloat_FromDouble(double v)
double PyFloat_AsDouble(PyObject *pyfloat)
```

Voici comment passer d'une chaîne C à une chaîne Python et réciproquement :

Fichier
```
PyObject* PyString_FromString(const char *v)
PyObject* PyString_FromFormat(const char *format, ...)  # printf style
char* PyString_AsString(PyObject *string)
```

Voici comment créer un tuple, lire et écrire ses composantes (collection ordonnée invariable) :

Fichier
```
PyObject* PyTuple_New(Py_ssize_t len)
PyObject* PyTuple_GetItem(PyObject *p, Py_ssize_t pos)
PyObject* PyTuple_GetSlice( PyObject *p, Py_ssize_t low, Py_ssize_t high)
int PyTuple_SetItem( PyObject *p, Py_ssize_t pos, PyObject *o)
```

Voici ce qu'il en est pour les listes (collection ordonnée variable) :

Fichier
```
PyObject* PyList_New( Py_ssize_t len)
PyObject* PyList_GetItem(PyObject *list, Py_ssize_t index)
PyObject* PyList_GetSlice(PyObject *list, Py_ssize_t low, Py_ssize_t high)
int PyList_SetItem(PyObject *list, Py_ssize_t index, PyObject *item)
int PyList_SetSlice(PyObject *list, Py_ssize_t low, Py_ssize_t high,
PyObject *itemlist)
```

On peut ajouter de nouveaux éléments dans une liste :

Fichier
```
int PyList_Insert(PyObject *list, Py_ssize_t index, PyObject *item)
int PyList_Append(PyObject *list, PyObject *item)
```

On peut également réaliser d'autres opérations :

Fichier
```
int PyList_Sort(PyObject *list)
int PyList_Reverse(PyObject *list)
```

Ceux qui connaissent un peu Python verront rapidement à quelles méthodes des objets cela correspond. Ces méthodes respectent parfaitement ce pourquoi les objets ont été conçus.

On peut terminer par exposer les méthodes pour un dictionnaire Python, nommé dans d'autres langages « table de hachage », qui permet de mettre en relation une clé

avec une valeur. Il faudra faire attention, car en Python, les clés ne sont pas forcément des chaînes de caractères, mais des facilités sont permises lorsque c'est le cas :

Fichier
```
PyObject* PyDict_New()
void PyDict_Clear(PyObject *p)
int PyDict_Contains(PyObject *p, PyObject *key)
PyObject* PyDict_Copy(PyObject *p)
```

On peut mettre à jour la valeur associée à une clé, ou accéder ou supprimer un enregistrement par sa clé :

Fichier
```
int PyDict_SetItem(PyObject *p, PyObject *key, PyObject *val)
int PyDict_SetItemString(PyObject *p, const char *key, PyObject *val)
int PyDict_DelItem(PyObject *p, PyObject *key)
int PyDict_DelItemString(PyObject *p, char *key)
PyObject* PyDict_GetItem(PyObject *p, PyObject *key)
PyObject* PyDict_GetItemString(PyObject *p, const char *key)
```

On peut également mettre à jour le dictionnaire par deux méthodes :

Fichier
```
int PyDict_Merge(PyObject *a, PyObject *b, int override)
int PyDict_Update(PyObject *a, PyObject *b)
```

On peut accéder à l'ensemble des clés, valeurs ou items (tuple de clés, valeurs), comme le langage le permet :

Fichier
```
PyObject* PyDict_Items(PyObject *p)
PyObject* PyDict_Keys(PyObject *p)
PyObject* PyDict_Values(PyObject *p)
```

Il ne reste plus qu'à voir comment importer un module et aller chercher une fonction, ce qui est proposé dans l'exemple final, présenté immédiatement après ces dernières précisions.

Pour terminer, le dernier point important consiste à gérer pour chaque objet Python un compteur de référence, ce qui est particulièrement important lorsque l'on utilise plusieurs fonctions. C'est un point critique et fondamental : **http://docs.python.org/2/extending/extending.html#reference-counting-in-python**.

Pour cela, on utilise **PyINCREF** pour incrémenter le compteur et **PyDECREF** pour le décrémenter. Il faut savoir que ce compteur peut être incrémenté automatiquement lors de l'utilisation de fonctions telles que celles présentées auparavant, ou décrémenté automatiquement, lors d'un retour de fonction C, par exemple.

En effet, lorsque l'on retourne **Py_None**, équivalent au **None** de Python ou au **null** de C, et qui est lui aussi un objet Python comme les autres, il faut penser à incrémenter le compteur de référence auparavant, sans quoi on se retrouve tout simplement avec une erreur de segmentation :

Fichier
```
PyINCR(Py_None)
return PyNone
```


3.2 Exemple complet

Voici sans plus attendre l'exemple complet :

Fichier

```
#include <Python.h>

int
main(int argc, char *argv[])
{
    PyObject *pModuleName, *pModule, *pFunc, *pFilename, *pFile, *pArgs,
    *pValue, *pRepr;
    const char* result;
    int i;

    /* Optionnel, mais conseillé */
    Py_SetProgramName(argv[0]);

    /* On vérifie les arguments passés en paramètre */
    if (argc != 2) {
        fprintf(stderr,"Usage: call filename\n");
        return 1;
    }

    /* On initialise la machine virtuelle Python */
    Py_Initialize();

    /* On importe le module pickle */
    pModuleName = PyString_FromString("pickle");
    pModule = PyImport_Import(pModuleName);
    Py_DECREF(pModuleName);

    /* On vérifie que l'import s'est bien passé */
    if (pModule == NULL) {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"pickle\"\n");
        return 1;
    }

    /* On récupère la fonction dump ou load depuis pickle */
    pFunc = PyObject_GetAttrString(pModule,"load");

    /* on vérifie que la fonction est bien disponible */
    if (pFunc == NULL) {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"load\" from \"pickle\"\n");
        Py_DECREF(pModule);
        return 1;
    }

    /* on vérifie que ce qui a été chargé est bien une fonction */
    if (!PyCallable_Check(pFunc)) {
        if (PyErr_Occurred())
            PyErr_Print();
        fprintf(stderr, "Cannot find function \"load\"\n");
        Py_XDECREF(pFunc);
        Py_DECREF(pModule);
        return 1;
    }

    /* On récupère le paramètre, le nom du fichier pickle */
    pFilename = PyString_FromString(argv[1]);
```

Fichier

```
    /* On ouvre le fichier en lecture */
    pFile = PyFile_FromString(argv[1], "r");

    /* on vérifie que le fichier est bien chargé */
    if (pFile == NULL) {
        PyErr_Print();
        fprintf(stderr, "Failed to load file \"%s\"\n", argv[1]);
        Py_XDECREF(pFunc);
        Py_DECREF(pModule);
        Py_DECREF(pFilename);
        return 1;
    }

    /* constitution de la liste des arguments pour la fonction dump */
    pArgs = PyTuple_New(1);
    PyTuple_SetItem(pArgs, 0, pFile);

    /* appel de la fonction */
    pValue = PyObject_CallObject(pFunc, pArgs);
    Py_DECREF(pArgs);
    Py_DECREF(pFunc);
    Py_DECREF(pModule);
    Py_DECREF(pFile);
    Py_DECREF(pFilename);

    /* On vérifie si l'appel a fonctionné */
    if (PyErr_Occurred()) {
        PyErr_Print();
        fprintf(stderr, "Failed to load data from file \"%s\"\n", argv[1]);
        Py_XDECREF(pValue);
        Py_DECREF(pModule);
        return 1;
    }

    /* Affichage du résultat */
    if (pValue != NULL) {

        /* Représentation des données sous forme d'une chaîne Python */
     pRepr = PyObject_Repr(pValue);

        /* conversion du résultat PyObject en chaîne de caractères */
        result = PyString_AsString(pRepr);

        /* affichage du résultat */
        printf("Result of call: %s\n", result);
        Py_DECREF(result);
        Py_DECREF(pRepr);
        Py_DECREF(pValue);
    }

    /* Extinction de la machine virtuelle */
    Py_Finalize();
    return 0;
}
```

Les étapes importantes sont les suivantes :

→ Inclusion du fichier **Python.h** :

Fichier
```
#include <Python.h>
```

→ Vérifications d'usage, en C :

Fichier
```
    if (argc != 2) {
        fprintf(stderr,"Usage: call filename\n");
        return 1;
    }
```

→ Initialisation de la machine virtuelle Python :

Fichier
```
    Py_Initialize();
```

→ Import de la fonction **load** du module Pickle :

Fichier
```
    pModuleName = PyString_FromString("pickle");
    pModule = PyImport_Import(pModuleName);
    pFunc = PyObject_GetAttrString(pModule,"load");
```

→ Appel de la fonction :

Fichier
```
    pFilename = PyString_FromString(argv[1]);
    pFile = PyFile_FromString(argv[1], "r");
    pArgs = PyTuple_New(1);
    PyTuple_SetItem(pArgs, 0, pFile);
    pValue = PyObject_CallObject(pFunc, pArgs);
```

→ Transformation du résultat sous sa représentation Python et affichage :

Fichier
```
    pRepr = PyObject_Repr(pValue);
    result = PyString_AsString(pRepr);
    printf("Result of call: %s\n", result);
```

→ Extinction de la machine virtuelle :

Fichier
```
    Py_Finalize();
```

Outre ces étapes essentielles, le reste du programme permet de faire des vérifications de cohérence et de gérer les compteurs des variables.

3.3 Tester le programme

Pour tester le projet, il faut commencer par la phase de compilation :

Terminal
```
$ gcc -I/usr/include/python2.7 -lpython2.7 -fPIC -g -o test test.c
```

On obtient ainsi un exécutable que l'on peut tester. Sans arguments :

Terminal

```
$ ./test
Usage: call filename
```

Avec un fichier qui n'existe pas :

Terminal

```
$ ./test existe_pas
IOError: [Errno 2] No such file or directory: 'existe_pas'
Failed to load file "existe_pas"
```

Avec un fichier de format incorrect :

Terminal

```
$ ./test test.c
Traceback (most recent call last):
  File "/usr/lib/python2.7/pickle.py", line 1378, in load
    return Unpickler(file).load()
  File "/usr/lib/python2.7/pickle.py", line 858, in load
    dispatch[key](self)
KeyError: '#'
Failed to load data from file "test.c"
```

Avec un fichier de test (fonctionnel) :

Terminal

```
$ ./test test.pkl
Result of call: 'Hello World'
$ ./test test2.pkl
Result of call: {'test': 42, 'say': 'Hello World', 'func': <function load
at 0x133b488>}
```

CONCLUSION

Embarquer une machine virtuelle Python n'est certainement pas quelque chose que vous ferez pour des raisons de performance, de consommation mémoire ou autre, mais il faut savoir que le langage de programmation Python est vraiment complémentaire de C, propose de très nombreux avantages et permet de développer très vite et très bien.

En ce sens, il trouve à l'heure actuelle de très nombreuses applications dans l'industrie, dans l'embarqué, ou encore dans des applications temps réel, que ce soit pour du prototypage, ou pour permettre de faire de la glue.

Outre ces aspects, pour une personne connaissant bien Python, cet exercice consistant à mêler les deux langages est très enrichissant et permet de mieux connaître le langage en ayant un éclairage différent. Cela permet également à une personne connaissant bien le C de découvrir Python en tant que programme écrit en C, et ayant fait des choix intéressants à appréhender.

Si vous connaissez les deux, vous ne pourrez donc pas résister à mettre votre nez dans tout cela, sans compter que la documentation officielle est relativement complète, même si des tutoriels ou de plus nombreux exemples auraient été appréciables. ▮

5

5 AUTOUR DU DÉVELOPPEMENT

À découvrir dans cette partie...

5.1 Internationaliser/régionaliser vos programmes en C

Vous développez aujourd'hui pour un public français. Mais que deviendra votre programme si vous êtes amené à étendre votre cible à l'international ? Retrouvez ici tous les outils nécessaires et les bonnes pratiques pour régionaliser proprement vos programmes, afin de le proposer à d'autres pays.

5 AUTOUR DU DÉVELOPPEMENT

INTERNATIONALISER/ RÉGIONALISER VOS PROGRAMMES C

Yann Morère

Si votre projet prend de l'ampleur, qu'il risque d'être utilisé dans d'autres pays que le vôtre, il convient alors de présenter une interface dans la langue de l'utilisateur. Pour cela, il faut, entre autres, traduire tous les intitulés textes de votre programme. On dit que l'on « régionalise » l'application. Voyons quels sont les outils qui vont nous aider dans cette tâche...

1. INTRODUCTION

Vous avez sans doute tous remarqué dans votre distribution GNU/Linux des paquets contenant les sous-chaînes « i18n » et « l10n ». Ils représentent des paquets relatifs aux supports des langues. En effet, « i18n » signifie : un mot commençant par « i » suivi de 18 lettres et se terminant par « n », soit « internationalisation » **[1]** et « l10n » signifie « localisation » avec la même méthode. Je sais, ils sont vraiment fainéants ces informaticiens !

Mais que signifient ces termes pour la gestion d'un programme/projet en C ?

2. TRADUCTION, INTERNATIONALISATION, RÉGIONALISATION ET GLOBALISATION

Pour régionaliser votre programme, vous pourriez simplement en créer une nouvelle version en traduisant dans votre code source toutes les chaînes de caractères. Cependant, cette méthode possède des défauts majeurs :

→ Création d'un programme par langue (coût de maintenance important) ;

→ Coût de traduction important, il faut parcourir à chaque fois l'ensemble du code source ;

→ Il est nécessaire que quelqu'un de l'équipe de développement connaisse la langue de destination ;

→ Impossibilité de changer simplement la langue de l'interface en cours d'utilisation ;

→ Intégration de nouvelles langues très fastidieuse.

Disons-le tout de suite, cette méthode ne doit pas être utilisée. Les outils de programmation et les systèmes d'exploitation possèdent des mécanismes et outils pour le support des différentes langues. La traduction se résumera donc à l'action du linguiste/traducteur : écrire une expression dans une autre langue. En aucun cas il ne doit réaliser une modification du code source (travail du programmeur).

La régionalisation de logiciel concerne le processus de traduction de l'interface utilisateur d'un logiciel d'une langue vers une autre et en l'adaptant à la culture locale. On utilise parfois l'anglicisme « localisation ».

L'internationalisation est un prérequis de la localisation **[2]**. L'internationalisation d'un logiciel consiste à « préparer » son adaptation à des langues et des cultures différentes. C'est un travail technique réalisé par les programmeurs (moi, vous, nous en somme ;-)). Le but est de produire un programme qui puisse être immédiatement déployé dans différentes langues, en ajoutant simplement un nouveau fichier de traduction. Cela consiste, la plupart du temps, à séparer dans le code source d'un programme ce qui est indépendant de la langue et de la culture de ce qui en est dépendant (généralement les intitulés des menus, des boutons, etc.).

On se placera dans un cas simple dans le cadre de cet article : seuls les intitulés textes doivent être traités pour la régionalisation. Cependant, l'internationalisation peut concerner d'autres paramètres de l'interface comme les codes de couleurs, les formats des dates, la direction de l'écriture (langues arabes par exemple).

Pour permettre ce type de régionalisation, la passage à l'encodage des caractères UTF-8/16 est parfois nécessaire pour le support des caractères spéciaux des langues utilisant la calligraphie (arabe), ou les idéogrammes (asiatique). Cette étape est appelée « multilingualisation ».

Contrairement à l'internationalisation, la régionalisation nécessite des compétences en langues (il s'agit de traduire), et non en programmation.

Au-delà de la simple traduction, la régionalisation concerne l'adaptation d'un logiciel à une culture. Ces paramètres régionaux (le type de virgule, la représentation des chiffres, le format de la date et de l'heure, les unités monétaires, etc.) sont généralement partagés par les autres applications.

Cette démarche de régionalisation doit être prévue dès la conception et les développeurs doivent la prendre en compte dans leur méthode de développement.

La globalisation [3], quant à elle, doit permettre de réaliser chaque localisation avec un minimum d'effort. Pour cela, il faut définir une stratégie de régionalisation qui permettra une intégration aisée de la nouvelle langue dans l'architecture existante.

Sous GNU/Linux, c'est la bibliothèque logicielle **gettext** [4] qui sert à l'internationalisation des programmes. Nous détaillerons plus loin son utilisation en programmation C.

3. « LOCALES » ET VARIABLES D'ENVIRONNEMENT

Il est possible de modifier simplement la régionalisation d'un système Linux. On peut ajouter (ou supprimer) des langues que le système et les applications peuvent utiliser : on parle alors d'installation de « locales ». Le paquet **locales** peut alors être reconfiguré par l'intermédiaire des utilitaires **locale-gen** et **dpkg-reconfigure** pour ajouter et supprimer des langues :

Terminal

```
# less /usr/share/i18n/SUPPORTED
# more /var/lib/locales/supported.d/local
# locale-gen fr_FR@euro ISO-8859-15
# dpkg-reconfigure locales
# locale -a
```

Les bureaux fournissent aussi leurs outils de régionalisation à partir de leur centre de contrôle. En mode console, les informations sur la langue, la monnaie utilisée, sont récupérées grâce à des variables d'environnement du shell. La principale se nomme **LANG** :

Terminal

```
$ echo $LANG
fr_FR.UTF-8
```

La langue peut être configurée en renseignant la variable **LANG** dans le fichier de configuration global **/etc/environment** [5] :

Fichier

```
LANG=fr_FR.UTF-8
```

Son formalisme est le suivant : **langue_Pays[.norme][@variante]**

→ **langue** est la langue utilisée (ici « fr » pour français) ;

→ **Pays** est le pays, en majuscules (ici « FR ») ;

→ **norme** indique la norme utilisée pour le codage des caractères (ici en UTF-8) ;

→ **@variante** précise une variante de la langue. Sur certains systèmes qui utilisent le codage ISO-8859-15, on peut préciser **@euro** pour indiquer une variante supportant le caractère « euro ».

Il existe d'autres variables de régionalisation que **LANG**. Cependant, cette dernière remplace toutes les autres lorsqu'elles ne sont pas d
éfinies. La liste de ces variables est donnée par la commande **locale** :

Terminal

```
$ locale
LANG=fr_FR.UTF-8
LANGUAGE=
LC_CTYPE="fr_FR.UTF-8"
LC_NUMERIC="fr_FR.UTF-8"
LC_TIME="fr_FR.UTF-8"
LC_COLLATE="fr_FR.UTF-8"
LC_MONETARY="fr_FR.UTF-8"
LC_MESSAGES="fr_FR.UTF-8"
LC_PAPER="fr_FR.UTF-8"
LC_NAME="fr_FR.UTF-8"
LC_ADDRESS="fr_FR.UTF-8"
LC_TELEPHONE="fr_FR.UTF-8"
LC_MEASUREMENT="fr_FR.UTF-8"
LC_IDENTIFICATION="fr_FR.UTF-8"
LC_ALL=
```

Chacune des variables **LC_** peut être modifiée et adaptée. Le tableau ci-dessous résume leur signification.

VARIABLE	RÔLE
LANG	Le paramètre de régionalisation de base utilisé par les applications du système, tant qu'il n'est pas outrepassé par une autre variable.
LC_CTYPE	Le jeu de caractères utilisé pour saisir et afficher du texte
LC_NUMERIC	Mise en forme des valeurs numériques non-monétaires
LC_TIME	Format de la date et de l'heure
LC_COLLATE	Comment trier diverses informations ; définit par exemple l'ordre alphabétique afin que les éléments puissent être triés alphabétiquement en utilisant la commande sort.
LC_MONETARY	Format des valeurs numériques monétaires
LC_MESSAGES	Langue utilisée pour afficher les messages à l'utilisateur
LC_PAPER	Définition des formats de papier standards
LC_NAME	Format des noms
LC_ADDRESS	Format des adresses
LC_TELEPHONE	Structure des numéros de téléphone
LC_MEASUREMENT	Unités de mesure à utiliser
LC_IDENTIFICATION	Peut être utilisée pour outrepasser la configuration de régionalisation au démarrage.
LC_ALL	Cette variable a un rôle puissant pour écraser les autres paramètres régionaux. Lorsqu'une valeur lui est affectée, les applications utiliseront cette valeur quelles que soient les valeurs des autres variables.

Passons maintenant à l'utilisation de la bibliothèque logicielle GNU **gettext**.

4. GNU GETTEXT

GNU gettext permet de mener à bien l'internationalisation d'un projet **[6]**. Elle est composée d'un ensemble d'outils qui comprend :

→ Une collection de conventions qui définissent comment les programmes doivent être écrits pour supporter les catalogues de messages (ensemble des traductions pour un programme/projet) ;

→ Une convention de nommage pour les catalogues eux-mêmes ;

→ Une bibliothèque qui permet de retrouver les messages traduits ;

→ Quelques programmes pour manipuler les ensembles des chaînes de caractères à traduire, ou déjà traduites ;

→ Une bibliothèque supportant l'analyse et la création de fichier contenant les messages traduits.

GNU gettext a été conçue pour minimiser l'impact de l'internationalisation sur le code source.

Voyons comment le mettre en œuvre sur le fameux **`hello.c`**.

4.1 Mise en place et première traduction

Voici le code source de notre programme :

Fichier

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

#define _(STRING) gettext(STRING)

int main(void)
{
 int year = 2013;
 setlocale( LC_ALL, "" );
 bindtextdomain( "hello", "." );
 textdomain( "hello" );
 printf( _("Hello, world!\n") );
 printf( gettext("My name is Yann\n") );
 // TRANSLATORS: Please let %i as it is, because it is exchanged by the program.
 // TRANSLATORS: Thank you for contributing to this project.
 printf( gettext("We are in year %i\n"),year );
 return EXIT_SUCCESS;
}
```

Voici quelques explications sur le code source.

L'inclusion de l'entête **`libintl.h`** permet de fournir le support de l'internationalisation au programme. C'est une partie de GNU gettext. Ce fichier fournit notamment la fonction **`gettext`**, qui permet de récupérer les chaînes de caractères à traduire.

L'inclusion de l'entête **`locale.h`** permet de fournir le support de régionalisation à un programme. Cette librairie contient notamment la fonction **`setlocale`**, qui permet de fixer la langue d'un programme.

La directive

Fichier

```
setlocale( LC_ALL, "" );
```

permet de ne fixer aucune langue en dur dans notre programme. Comme cela, le programme utilisera les variables d'environnement pour faire la sélection.

La fonction **bindtextdomain** fixe le répertoire de base de l'arborescence contenant les catalogues de messages pour un domaine donné. Un « domain » de messages est un ensemble de traductions (ensemble de msgid). Généralement, chaque projet (programme) possède son propre « domain ». Cet appel à **bindtextdomain** est nécessaire, car les programmes ne sont pas toujours installés au même endroit (avec le même « prefix »). Dans notre cas, pour des raisons de simplicité, nous avons nommé notre « domain » du nom du programme, « hello », et le chemin est le répertoire courant **.** (pour éviter d'utiliser les droits administrateur pour une installation dans l'arborescence système **/usr/share/locales**).

Il faudra donc enregistrer les catalogues de messages dans les répertoires **./locale/LC_MESSAGES/hello.mo**. **locale** prendra les valeurs des langues supportées : « fr », « en_EN », « pt_BR », etc.

La fonction **textdomain** permet ensuite de définir notre ensemble de messages de traduction : « hello ».

Ensuite, le code source a été modifié pour utiliser les appels à la fonction GNU gettext. Pour la plupart des langages, ceci se fait en insérant les chaînes destinées à l'utilisateur dans la fonction **gettext**. Pour gagner du temps de saisie et clarifier le code, on peut utiliser l'alias de substitution **_**. Pour cela, il faut le définir alias dans l'entête du code source :

Fichier

```
#define _(STRING) gettext(STRING)
```

GNU gettext utilise alors la chaîne fournie comme clef de recherche de traduction et renvoie la chaîne d'origine si aucune traduction n'est disponible. On prendra alors comme convention de toujours écrire le programme de base en langue anglaise, puisque c'est maintenant la langue quasi-universelle et aussi la langue par défaut des systèmes d'exploitation. On se basera sur cette langue pour réaliser la traduction.

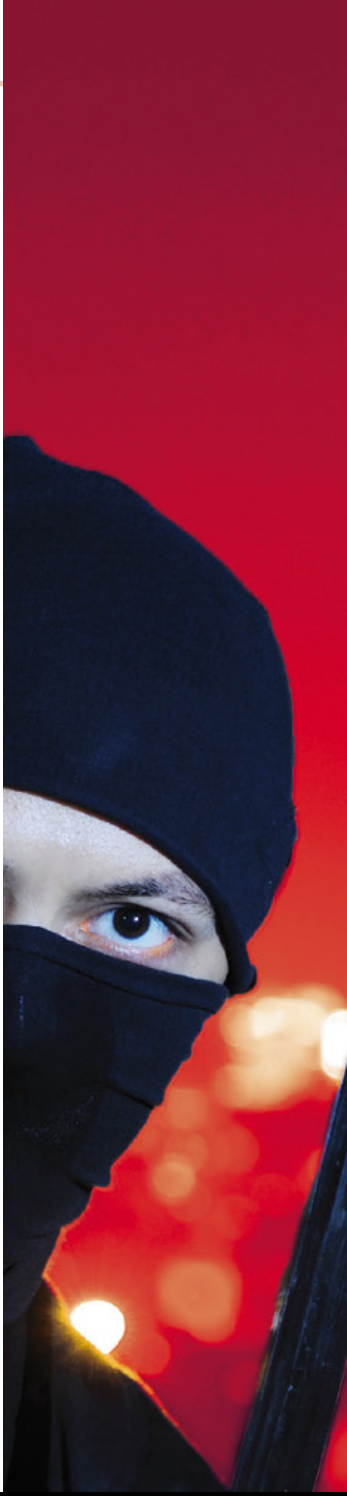
La fonction **gettext** est aussi disponible dans de nombreux autres langages : C++, bash script, Python, Java, Pascal, Perl, PHP, etc.

Afin de lister toutes les chaînes de caractères incluses dans le processus d'internationalisation, le programme **xgettext** est appliqué aux sources pour produire un fichier **.pot**, ou modèle. Ce dernier contiendra une liste de toutes les chaînes traduisibles extraites du code.

La commande suivante permet de réaliser l'opération :

Terminal

```
$ xgettext --sort-output --keyword=_ -o hello.pot hello.c
```



Note

Dans le cadre de cet article, nous n'avons qu'un seul fichier source. Il est bien sûr possible de collecter l'ensemble des chaînes à traduire d'un projet par l'intermédiaire de la commande suivante :

```
$ xgettext --sort-
output --keyword=_ -o
mon_projet.pot src/*.c
```

L'option **`--sort-output`** permet de trier les chaînes par ordre alphabétique. L'option **`--keyword`** permet d'ajouter le marqueur « _ » pour la récupération des chaînes de caractères. S'il est oublié, seules les chaînes marquées par **`gettext`** seront prises en compte.

Voici le fichier **`hello.pot`** produit pour notre exemple :

Fichier

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2013-10-29 09:20+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: hello.c:12
#, c-format
msgid "Hello, world!\n"
msgstr ""

#: hello.c:13
#, c-format
msgid "My name is Yann\n"
msgstr ""

#: hello.c:18
#, c-format
msgid "We are in year %i\n"
msgstr ""
```

Si le développeur désire donner au traducteur une indication pour une chaîne spécifique, il peut le faire dans le code source à l'aide d'une étiquette (TAG) destinée à **`xgettext`**. Cela permet à ce dernier de filtrer ces indications et de les mettre dans le fichier **`.pot`**. Ces indications seront affichées par les logiciels d'aide à la traduction, comme Gtranslator et Poedit par exemple (cf. ci-après).

Dans notre code, nous utiliserons l'étiquette « TRANSLATORS: » pour définir les commentaires destinés aux traducteurs. Ensuite, la commande suivante permet d'intégrer ces commentaires dans le fichier **`.pot`** :

Terminal

```
xgettext --sort-output --add-comments=TRANSLATORS:
--keyword=_ -o hello.pot hello.c
```

Ce qui donne :

Fichier

```
#. TRANSLATORS: Please let %i as it is, because it is
exchanged by the program.
#. TRANSLATORS: Thank you for contributing to this project.
#: hello.c:18
#, c-format
msgid "We are in year %i\n"
msgstr ""
```

Une fois ce fichier « modèle » généré, nous allons dériver un fichier **.po** de la langue de destination en utilisant le programme **msginit**.

Ainsi, pour la dérivation en langue française utilisant l'encodage UTF-8, on utilise la commande suivante :

Terminal

```
$ msginit --locale=fr_FR.utf8 --input=hello.pot
...
fr.po a été créé.
$ msginit --locale=pt_BR.utf8 --input=hello.pot
```

On réalise ensuite la traduction, ce qui donne le fichier **fr.po** suivant :

Fichier

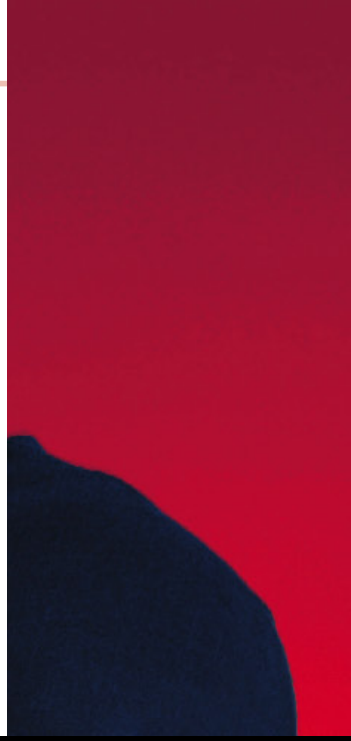
```
# French translations for PACKAGE package.
# Copyright (C) 2013 THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the
PACKAGE package.
# yann <yann@geonosis>, 2013.
#
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2013-10-29 09:20+0100\n"
"PO-Revision-Date: 2013-10-29 09:23+0100\n"
"Last-Translator: yann <yann@geonosis>\n"
"Language-Team: French\n"
"Language: fr\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=ASCII\n"
"Content-Transfer-Encoding: 8bit\n"
"Plural-Forms: nplurals=2; plural=(n > 1);\n"

#: hello.c:12
#, c-format
msgid "Hello, world!\n"
msgstr "Bonjour, le monde!\"

#: hello.c:13
#, c-format
msgid "My name is Yann\n"
msgstr "Mon nom est Yann\n"

#. TRANSLATORS: Please let %i as it is, because it is
exchanged by the program.
#. TRANSLATORS: Thank you for contributing to this project.
#: hello.c:18
#, c-format
msgid "We are in year %i\n"
msgstr "Nous sommes dans l'année %i\n"
```

Finalement, les fichiers **.po** sont compilés en fichiers binaires **.mo** à l'aide de l'utilitaire **msgfmt** (cf. les commandes suivantes) :

Terminal

```
$ msgfmt fr.po -o fr.mo
$ msgfmt pt_BR.po -o pt_BR.mo
```

Après avoir créé les répertoires **fr_FR/LC_MESSAGES** et **pt_BR/LC_MESSAGES** dans le répertoire courant, on y copie les fichiers **.mo** correspondants en les renommant du nom du « domain », ici « hello » :



À savoir

Les plus attentifs auront noté l'utilisation de l'encodage UTF-8. Ce dernier est spécifié lors de la dérivation du modèle. Si vous avez omis l'option, vous pouvez toujours modifier la ligne suivante dans le fichier .po généré :

```
"Content-Type: text/
plain; charset=UTF-8\n"
```

Le cas échéant, la commande msgfmt chargée de transformer les fichiers .po en fichiers binaires .mo donnera ce type d'erreur :

```
$ msgfmt fr.po -o fr.mo
fr.po:33:31: Séquence
d'octets multiples
invalide
fr.po:33:32: Séquence
d'octets multiples
invalide
msgfmt: 2 erreurs
fatales trouvées
```

Terminal
```
$ cp fr.mo fr_FR/LC_MESSAGES/hello.mo
$ cp pt_BR.mo pt_BR/LC_MESSAGES/hello.mo
```

Finalement, il ne nous reste plus qu'à tester la régionalisation de notre petit programme :

Terminal
```
$ LC_ALL="pt_BR.utf8" ./hello
Bom dia, o mundo!
Meu nome é Yann
Estamos em ano 2013
$ LANG="en_EN.utf8" ./hello
Hello, world!
My name is Yann
We are in year 2013
$ ./hello
Bonjour, le monde!
Mon nom est Yann
Nous sommes en l'année 2013
$
```

Comme nous n'avons pas fixé en dur dans le programme la langue (fonction **setlocale**), on peut basculer simplement d'une langue à l'autre en modifiant la variable d'environnement **LANG**, ou **LC_ALL/LC_MESSAGES** si la langue doit être différente de **LANG**. Il faut aussi que les locales de la langue en question soient installées.

Note

Certains auront noté que je n'ai pas créé de répertoire `en_EN/LC_MESSAGES`, mais que pourtant l'affichage est bien en anglais. C'est un effet de bord, de la chance… enfin presque. En fait, comme le programme ne trouve pas le bon fichier de traduction (`en_EN.mo` en UTF-8), il utilise la langue par défaut qui est l'anglais. Voici, pour vous en convaincre, un exemple qui tente de régionaliser le programme en langue allemande. Cette langue n'étant pas gérée, on récupère la traduction anglaise :

```
$$ LC_ALL="de_DE.utf8"
./hello
Hello, world!
My name is Yann
We are in year 2013
```

4.2 Un cas particulier de chaînes à traduire

Cependant, un problème subsiste. En effet, il n'est pas toujours possible de marquer les chaînes de caractères à traduire avec la fonction **gettext**. C'est notamment le cas lors de l'utilisation d'un tableau initialisé de chaînes **[7]** :

Fichier
```
char *messages[] = {
    "some very meaningful message",
    "and another one.. less meaningful"
  };
char *string;
int index = 0;
string = index > 1 ? gettext("a default message") :
gettext(messages[index]);
printf("%s\n",string);
```

Alors que le marquage de la chaîne « a default message » ne pose pas de souci, il n'est pas possible de marquer les chaînes d'initialisation du tableau « messages ». Pour réaliser la régionalisation, nous devons faire deux choses :

→ Premièrement, marquer les chaînes de caractères afin que le programme **xgettext** les retrouve ;

→ Deuxièmement, traduire la chaîne de caractères en temps réel avant l'affichage à l'écran (**printf**).

La première tâche est réalisée en créant un nouveau mot-clé, qui nomme une non-opération (no-op, on ne fait rien, c'est juste un marquage pour la récupération par **xgettext**) :

Fichier

```
#define gettext_noop(String) (String)
```

Pour la seconde tâche, il faut marquer tous les accès au tableau de chaînes « messages ». Pour notre petit programme la solution est la suivante :

Fichier

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

#define _(String) gettext(String)
#define gettext_noop(String) (String)

int main(void)
{
 int year = 2013;
 char *messages[] = {
    gettext_noop("some very meaningful message"),
    gettext_noop("and another one.. less meaningful")
  };
 char *string;
 int index = 0;
 setlocale( LC_ALL, "" );
 bindtextdomain( "hello", "." );
 textdomain( "hello" );
 printf( _("Hello, world!\n") );
 printf( gettext("My name is Yann\n") );
 // TRANSLATORS: Please let %i as it is,
 // because it is exchanged by the program.
 // TRANSLATORS: Thank you for contributing to this project.
 printf( gettext("We are in year %i\n"),year );
 string = index > 1 ? gettext("a default message") :
gettext(messages[index]);
 printf("%s\n",string);
 index=1;
 string = index > 1 ? gettext("a default message") :
gettext(messages[index]);
 printf("%s\n",string);
 index=2;
 string = index > 1 ? gettext("a default message") :
gettext(messages[index]);
 printf("%s\n",string);
 return EXIT_SUCCESS;
}
```

Ensuite, on régénère le fichier modèle en ajoutant le nouveau mot-clé :

Terminal

```
$ xgettext --sort-output --add-comments=TRANSLATORS: \
--keyword=gettext_noop --keyword=_ -o hello.pot hello.c
```

Cela ajoute le code suivant au fichier **.pot** :

Fichier

```
#: hello.c:30 hello.c:33 hello.c:36
msgid "a default message"
msgstr ""

#: hello.c:15
msgid "and another one.. less meaningful"
msgstr ""

#: hello.c:14
msgid "some very meaningful message"
msgstr ""
```

Après traduction, transformation en fichiers binaires **.mo**, puis copie dans le répertoire adéquat, on obtient :

Terminal

```
$ LANG="pt_BR.utf8" ./hello
Bom dia, o mundo!
Meu nome é Yann
Estamos em ano 2013
alguma mensagem muito significativa
e outra .. menos significativa
uma mensagem padrão
$ ./hello
Bonjour, le monde!
Mon nom est Yann
Nous sommes en l'année 2013
un message très significatif
et un autre... moins significatif
un message par defaut
$ LC_ALL="de_DE.UTF-8"  ./hello
Hello, world!
My name is Yann
We are in year 2013
some very meaningful message
and another one.. less meaningful
a default message
```

Vous êtes maintenant prêt pour l'internationalisation de vos programmes.

À savoir

N'oubliez pas de recompiler votre fichier source après modifications des sources et traductions. Cela vous évitera des erreurs bizarres de chaînes traduites et d'autres non traduites :-D.

4.3 Un petit mot sur les entrées « fuzzy »

Chaque entrée du fichier .po peut avoir des attributs qui donnent des informations sur la qualité d'une traduction en utilisant un système de commentaire particulier (« #. », « #. », etc.). Un de ces attributs se nomme « fuzzy » (flou, approximatif en français) [9]. Les entrées possédant cet attribut possède donc une traduction approximative.

Ces entrées fuzzy, bien que prises en compte dans la régionalisation, indiquent aux traducteurs le besoin de réviser l'entrée. Elles peuvent être produites après application du programme de fusion **msgmerge** pour

mettre à jour un ancien fichier **.po** à partir d'un nouveau modèle **.pot**. Mais elles peuvent être aussi produites par les traducteurs eux-mêmes s'ils ne sont pas sûrs de leur traduction et indiquent que leur travail doit être révisé.

4.4 En cas de mise à jour du code source

Bien sûr, votre programme/projet n'est pas figé après la phase de régionalisation ; vous le faites évoluer et de nouvelles chaînes de caractères vont devoir être traduites. Cependant, la régénération du fichier modèle ne doit pas signifier une nouvelle traduction complète pour les traducteurs. Voyons comment gérer cela.

Ajoutons le code suivant dans notre fichier **hello.c** :

Dans la partie déclaration de variables :

Fichier

```
char brand[10] = "Apple"; //replace with the brand you dislike ;-)
```

Et, en fin de fichier :

Fichier

```
// TRANSLATORS: Please let %s as it is, because it is exchanged by the program.
// TRANSLATORS: Thank you for contributing to this project.
printf( gettext("I don't like the %s brand ;-)\n"),brand );
```

Il y a donc une nouvelle chaîne de caractères à traduire. On relance donc la commande suivante pour régénérer le fichier modèle :

Terminal

```
xgettext --sort-output --add-comments=TRANSLATORS: --keyword=gettext_noop
--keyword=_ -o hello.pot hello.c
```

Le fichier **hello.pot** contient la nouvelle chaîne de caractères à traduire :

Fichier

```
#. TRANSLATORS: Please let %s as it is, because it is exchanged by the program.
#. TRANSLATORS: Thank you for contributing to this project.
#: hello.c:22
#, c-format
msgid "I don't like the %s brand ;-)\n"
msgstr ""
```

Au lieu de dériver un nouveau fichier de traduction .po, depuis ce nouveau modèle, on va fusionner l'ancien avec les nouvelles chaînes à traduire issues du modèle. Pour cela, on utilise le programme **msgmerge**. L'option **-v** donne le mode verbeux et l'option **-U** indique de faire une mise à jour du fichier .po passé en paramètre :

Terminal

```
$ msgmerge -v -U fr.po hello.pot
.....
 lues 1 anciennes + 1 références, 7 fusionnées, 0 approximatives,
1 manquantes, 0 périmées.
```

msgmerge nous indique le nombre de chaînes non traduites, par l'intermédiaire du nombre de chaînes manquantes.

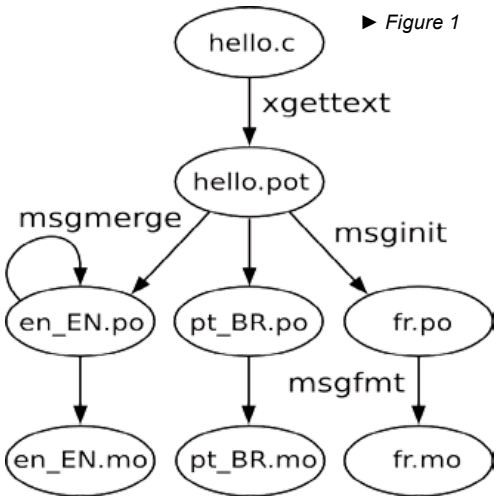
Notre fichier **fr.po** est modifié, et il ne nous reste plus qu'à traduire les parties du fichier qui ont été mises à jour. On fait de même pour les autres fichiers **.po**, puis on compile le tout en fichiers binaires **.mo** :

Terminal
```
for i in `ls *.po`; do msgfmt $i -o `basename $i  .po`.mo ; done
```

On copie les nouveaux fichiers **.mo** dans l'arborescence adéquate et on teste tout cela :

Terminal
```
$ LANG="pt_BR.utf8" ./hello
Bom dia, o mundo!
Meu nome é Yann
Estamos em ano 2013
Eu não gosto da marca Apple ;-)
alguma mensagem muito significativa
e outra .. menos significativa
uma mensagem padrão
$ ./hello
Bonjour, le monde!
Mon nom est Yann
Nous sommes en l'année 2013
Je n'aime pas la marque Apple ;-)
un message très significatif
et un autre... moins significatif
un message par defaut
$ LC_ALL="de_DE.UTF-8"  ./hello
Hello, world!
My name is Yann
We are in year 2013
I don't like the Apple brand ;-)
some very meaningful message
and another one.. less meaningful
a default message
```

On peut alors résumer tout notre travail par la figure ci-contre.



► *Figure 1*

4.5 Et si cela ne fonctionne pas comme prévu ?

La plupart du temps, si la traduction ne se fait pas, c'est que le programme ne trouve pas le bon fichier **.mo**. On peut suivre les chargements des fichiers utilisés par un programme à l'aide de la commande **strace**. C'est un outil puissant pour le diagnostic et le débogage. On pourra l'utiliser comme suit sur notre programme **hello.c** :

Terminal

```
$ strace -o loading.txt -e open ./hello
```

Ceci permet de générer le fichier **loading.txt** suivant :

Fichier

```
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
open("/usr/share/locale/locale.alias", O_RDONLY|O_CLOEXEC) = 3
open("./fr_FR.UTF-8/LC_MESSAGES/hello.mo", O_RDONLY) = -1 ENOENT (No such
file or directory)
open("./fr_FR.utf8/LC_MESSAGES/hello.mo", O_RDONLY) = -1 ENOENT (No such file
or directory)
open("./fr_FR/LC_MESSAGES/hello.mo", O_RDONLY) = 3
open("/usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache", O_RDONLY) = 3
```

On voit alors toutes les actions de recherche concernant la régionalisation de notre programme. On remarque que certaines commandes se terminent sur des erreurs (**-1**), mais que finalement le fichier **./fr_FR/LC_MESSAGES/hello.mo** est chargé convenablement (**3**). On peut bien sûr tester d'autres langues :

Terminal

```
$ LC_ALL="pt_BR.UTF-8" strace -o loading.txt -e open ./hello
```

Voici maintenant un exemple où la régionalisation est un échec et le programme bascule sur la langue anglaise par défaut :

Terminal

```
$ LC_ALL="de_DE.UTF-8" strace -o loading.txt -e open ./hello
Hello, world!
My name is Yann
We are in year 2013
I don't like the Apple brand ;-)
```

Et le contenu du fichier **loading.txt**, qui montre clairement qu'aucun fichier de traduction n'a été trouvé :

Fichier

```
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
open("/usr/share/locale/locale.alias", O_RDONLY|O_CLOEXEC) = 3open("./
de_DE.UTF-8/LC_MESSAGES/hello.mo", O_RDONLY) = -1 ENOENT (No such file or
directory)
open("./de_DE.utf8/LC_MESSAGES/hello.mo", O_RDONLY) = -1 ENOENT (No such
file or directory)
open("./de_DE/LC_MESSAGES/hello.mo", O_RDONLY) = -1 ENOENT (No such file
or directory)
open("./de.UTF-8/LC_MESSAGES/hello.mo", O_RDONLY) = -1 ENOENT (No such
file or directory)
open("./de.utf8/LC_MESSAGES/hello.mo", O_RDONLY) = -1 ENOENT (No such file
or directory)
open("./de/LC_MESSAGES/hello.mo", O_RDONLY) = -1 ENOENT (No such file or
directory)
```

Fichier

```
open("/usr/share/locale-langpack/de_DE.UTF-8/LC_MESSAGES/hello.mo",
O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/share/locale-langpack/de_DE.utf8/LC_MESSAGES/hello.mo",
O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/share/locale-langpack/de_DE/LC_MESSAGES/hello.mo",
O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/share/locale-langpack/de.UTF-8/LC_MESSAGES/hello.mo",
O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/share/locale-langpack/de.utf8/LC_MESSAGES/hello.mo",
O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/share/locale-langpack/de/LC_MESSAGES/hello.mo",
O_RDONLY) = -1 ENOENT (No such file or directory)
```

5. LES OUTILS GRAPHIQUES D'AIDE À LA TRADUCTION

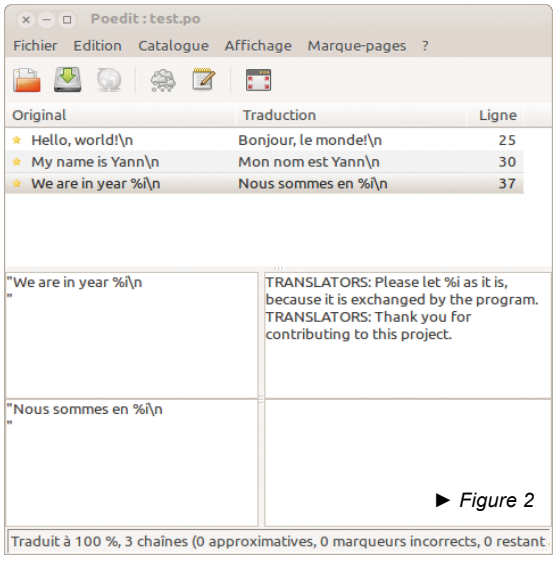
Bien que toutes les manipulations d'internationalisation/régionalisation soient faisables à partir de la console, des outils graphiques d'aide à la traduction sont disponibles. Leur plus grand intérêt réside sans doute dans la disponibilité d'une mémoire de traduction, qui permet d'accélérer/automatiser le travail.

5.1 Poedit

Poedit est un éditeur de catalogues de messages **gettext** (les fichiers **.po**) multiplateforme. Il a été développé à l'aide de la librairie graphique wxWidgets. Il a été créé pour rendre l'édition des catalogues plus simple et plus pratique. On l'installe simplement à l'aide de la commande :

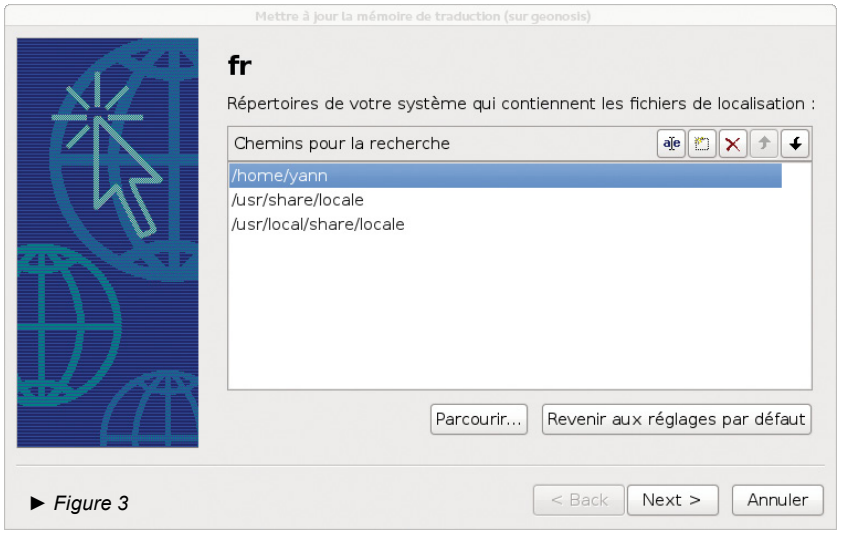
Terminal

```
sudo apt-get install poedit
```



► *Figure 2*

Au premier lancement, on vous demandera votre nom et votre e-mail pour ajouter ces détails de contributeur dans l'entête du fichier **.po** produit. Vous pouvez ensuite ouvrir le fichier **.po** à traduire (il contient déjà la langue de traduction). En haut de la fenêtre apparaissent les chaînes de caractères à traduire. Il suffit de cliquer sur l'une d'elles pour accéder à sa modification. Vous pouvez maintenant réaliser la traduction dans la zone d'édition correspondante (Fig. 2).



► *Figure 3*

Si vous n'êtes pas sûr de votre traduction, marquez l'entrée comme approximative « fuzzy » (bouton avec le petit nuage) ; les autres traducteurs pourront alors vous corriger. Vous pourrez aussi ajouter des commentaires à destination des autres contributeurs. Pour cela, il suffit de cliquer sur le bouton « commentaire » (icône calepin).

Poedit possède la fonctionnalité de mémoire de traduction ou « Translation Memory » (TM). Ces mémoires de traduction, ou bases de données de traduction, sont des collections d'entrées où le texte source est associé à sa traduction en plusieurs langues. Ainsi, Poedit permet d'automatiser la traduction des chaînes de caractères fréquentes en réutilisant ce que vous avez déjà traduit, ou encore en récupérant les données de traduction des fichiers `.po`, `.mo` déjà présents sur votre système.

Pour cela, il suffit de générer la base de données à partir du menu ***Édition > Préférences***, puis onglet ***Mémoire de traduction***. On ajoute les langues que l'on veut inclure dans la base de données, puis les répertoires contenant les fichiers `.po` (Fig. 3). Finalement, on lance la construction de la base à l'aide du bouton idoine. Attention, cette génération peut prendre un certain temps.

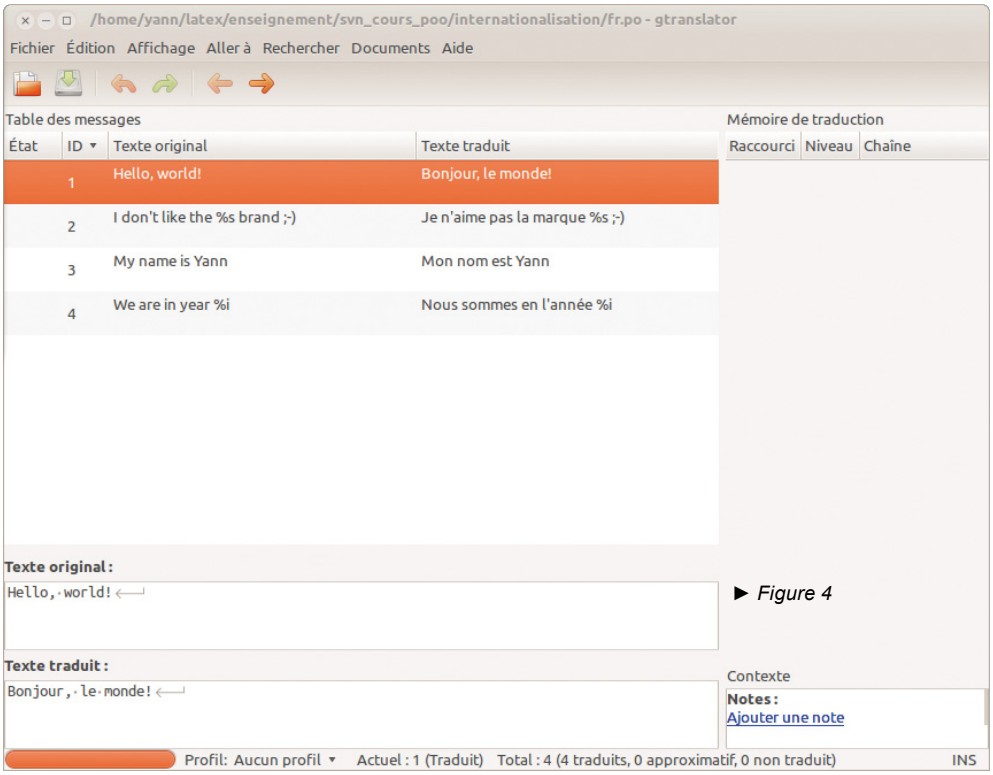
Ensuite, lors de vos prochaines traductions, il vous sera possible d'utiliser la traduction automatique avec la mémoire de traduction : ***Catalogue > Traduire automatiquement avec TM***.

5.2 Gtranslator

Gtranslator **[10]**, à l'instar de Poedit, propose une interface graphique pour la modification des fichiers `.po gettext` (Fig. 4, page suivante). Il est développé pour l'environnement de bureau GNOME. Il permet l'ouverture et la modification de plusieurs fichiers .po en même temps. Il gère des « profils » qui vous permettent l'utilisation de « comptes » différents, comportant les informations nécessaires pour remplir l'entête. Il possède aussi un système de mémoire de traduction.

Gtranslator intègre également un système de greffons, qui permet d'étendre ses fonctionnalités (svn, affichage code source, plein écran, table de caractères, ...).

Son fonctionnement est sensiblement identique à celui de Poedit, je vous laisse donc le découvrir par vous-même.



► *Figure 4*

CONCLUSION

Nous voici au terme de cet article qui a présenté les outils pour la régionalisation/internationalisation de vos programmes en C. Nous avons traité ce sujet à l'aide d'un seul fichier source. La page **[11]** vous donne la recette rapide pour la gestion de la régionalisation dans la cadre d'un projet plus important, géré à l'aide des outils **`autotools`** et **`make`**. Bonne traduction ! ■

RÉFÉRENCES

[1] **http://en.wiktionary.org/wiki/i18n**

[2] **http://fr.wikipedia.org/wiki/Internationalisation_%28informatique%29**

[3] **http://en.wikipedia.org/wiki/Language_localisation**

[4] **https://www.gnu.org/software/gettext/**

[5] **https://help.ubuntu.com/community/EnvironmentVariables**

[6] **https://www.gnu.org/software/gettext/manual/gettext.html#Introduction**

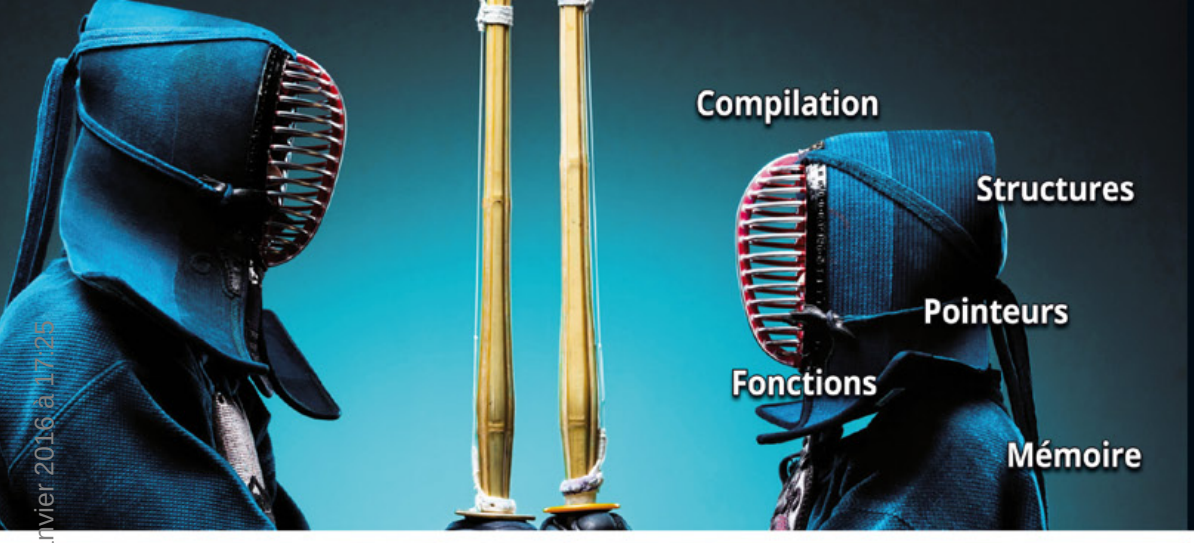
[7] **https://www.gnu.org/software/gettext/manual/gettext.html#Special-cases**

[8] **http://www.gnu.org/software/gettext/manual/html_node/Fuzzy-Entries.html**

[9] **http://www.poedit.net/**

[10] **https://projects.gnome.org/gtranslator/**

[11] **http://www.gnu.org/software/gettext/FAQ.html**



Introduction & Rappels

Ce que vous devez savoir des bases et principes du langage C

La pratique du C

Les techniques avancées pour parfaire votre maîtrise du C : macros, duff's device, co-routines, etc.

Bibliothèques & Toolkits

Utilisation des libs graphiques EFL, du toolkit GTK+ et du langage Python dans vos codes C

Autour du développement

Internationalisation des programmes et interfaces multilingues grâce à GNU Gettext