Karnaj

cc BY NC ND

Zeste de savoir

Utiliser une bibliothèque sous Windows avec MinGW ou Code::Blocks en langage C

14 novembre 2020

Table des matières

1. Le fonctionnement des bibliothèques . . . 1
  1.1. Les bibliothèques dans la compilation . . . 1
  1.2. Bibliothèques statiques et dynamiques . . . 2
2. Configuration du projet . . . 3
  2.1. Mise en place du projet . . . 3
  2.2. Avec MinGW . . . 4
  2.3. Avec Code::Blocks . . . 5
  2.4. Erreurs possibles . . . 6
3. Exemple: utiliser la bibliothèque SDL 2 . . . 8
  3.1. Organisation d'une bibliothèque . . . 8
  3.2. Téléchargement de la bibliothèque . . . 8
  3.3. Mise en place du projet . . . 9
  3.4. Code de test . . . 12

L'utilisation d'une bibliothèque tierce sous Windows peut sembler compliquée en C. Si l'on ne sait pas vraiment comment configurer un projet, elle le semble encore plus. Et si l'on est maladroit dans notre utilisation, des erreurs peuvent survenir, ce qui peut être très embêtant. Pourtant, les bibliothèques ne sont pas si méchantes, et une fois leur fonctionnement compris, le processus de programmation est grandement simplifié.

Pour le comprendre il faut s'intéresser au processus de compilation (juste quelques idées), puis faire en sorte que cette compilation se passe justement au mieux. C'est ce que nous allons faire, avant de mettre en place une procédure de configuration des bibliothèques.

Prérequis
Connaître le langage C (un tutoriel est disponible ici).
Prérequis optionnels
Connaissances à propos du compilateur.
Connaissances à propos de la création de *makefiles*.
Objectifs
Apprendre à configurer une bibliothèque pour un projet avec Code::Blocks ou MinGW.

1. Le fonctionnement des bibliothèques

1.1. Les bibliothèques dans la compilation

Pour introduire le principe de la compilation, il faut considérer ses quatre étapes.

— Le **préprocesseur** lit les fichiers sources et traite les **directives de préprocesseur**. Il fait donc en sorte d'exécuter les commandes `include`, `define`… C'est la première étape de la compilation.
— Le **compilateur** traduit alors chaque fichier source en code assembleur. C'est à ce moment que se passe l'analyse du code source.
— L'**assembleur** transforme le code assembleur en fichier objet (`.o`). Il s'agit de fichiers binaires donc directement compréhensibles par notre processeur.
— L'**éditeur de liens** associe les fichiers objets pour en faire un exécutable.

Chacune de ces phases peut conduire à des erreurs variées. Le préprocesseur a, par exemple, besoin d'un chemin valide pour les `include`, pour savoir où trouver les fichiers d'en-tête, l'éditeur de liens a besoin qu'on lui dise où chercher les fichiers des bibliothèques et lesquelles lier, et le compilateur a besoin d'un code source valide. Ainsi, si nous essayons d'inclure un fichier d'en-tête qui n'existe pas, nous aurons une erreur au niveau du préprocesseur. De même, si nous essayons d'utiliser une fonction d'une bibliothèque sans avoir lié cette bibliothèque à notre projet, nous aurons une erreur. En revanche, si nous utilisons des fonctions sans inclure les fichiers d'en-tête dans lesquels elles sont déclarées, nous n'aurons pas d'erreur au niveau du préprocesseur, mais au niveau de la compilation.

1.2. Bibliothèques statiques et dynamiques

Depuis un moment nous parlons des bibliothèques et des fichiers objets des bibliothèques. Mais que sont en fait les bibliothèques?

Les bibliothèques sont en réalité du code déjà compilé, c'est-à-dire des fichiers objets (il y a bien sûr d'autres sortes de fichiers, comme des fichiers d'en-tête). Il en existe deux sortes.

— Les **bibliothèques dynamiques**: les fichiers objets sont chargés lors de l'exécution du programme. Ceux-ci prennent la forme de fichiers `.dll` et doivent en conséquence être présents lors de l'exécution.
— Les **bibliothèques statiques**: les fichiers objets sont inclus dans le programme. Ces bibliothèques permettent de ce fait de s'affranchir des fichiers `.dll` mais, en contrepartie, le programme sera plus lourd.

Les fichiers `.a`, `.lib` et `.dll` sont donc tous des fichiers objets. La différence entre les deux réside dans le fait que les bibliothèques statiques sont incorporées à l'exécutable final, alors que les bibliothèques dynamiques sont chargées lors de son exécution.

> *i* La plupart des bibliothèques sont dynamiques. Il en existe même qui sont proposées en dynamique et en statique.

Après avoir compris cela, il paraît logique que, pour utiliser n'importe quelle bibliothèque dans un projet, il faille que le préprocesseur, le compilateur et l'éditeur de liens sachent où trouver les fichiers qui leur seront utiles. Il faut donc leur fournir:

— les fichiers d'en-tête (`.h`);
— les fichiers objets des bibliothèques (`.a`, `.lib`).

2. Configuration du projet

Maintenant que nous avons fait le point sur les bibliothèques, nous savons tout ce qu'il faut pour les configurer.

2.1. Mise en place du projet

Notre but est maintenant de mettre en place un projet organisé de telle sorte qu'il puisse être utilisé sur un autre ordinateur sans rien changer. Ceci permettra de distribuer nos sources et de changer d'ordinateur très facilement, juste avec un copier-coller. Pour cela, nous allons adopter une organisation pour les projets dans laquelle les fichiers objets et les fichiers d'en-tête des bibliothèques seront placés, non pas dans les répertoires du compilateur, mais bien dans des dossiers placés à la racine de notre projet.

Les répertoires du compilateur sont généralement placés dans leur dossier d'installation. Par exemple, dans le dossier d'installation de MinGW, on trouve un dossier `include` et un dossier `lib`. Ces dossiers contiennent tous les fichiers des bibliothèques standards.

Dans le répertoire que nous allons créer, nous aurons donc:

— un dossier `include` qui contiendra nos fichiers d'en-tête et ceux de toutes les bibliothèques que nous voudrons utiliser;
— un dossier `lib` qui contiendra les fichiers objets des bibliothèques (`.a`, `.lib`…);
— un dossier `src` qui contiendra nos fichiers sources.

Nos fichiers `.dll` seront placés dans notre dossier principal, qui contiendra également les fichiers exécutables.

Ainsi, l'arborescence de notre projet sera la suivante.

```
1 /
2         include
3         lib
4         src
```

De cette manière, le projet est indépendant, puisque tout ce dont il a besoin est à sa racine.

Maintenant, il est nécessaire d'indiquer à notre compilateur où trouver les fichiers de bibliothèque et d'en-tête. C'est la dernière étape. Si nous utilisons Code::Blocks par exemple, en sauvegardant notre projet, nous sauvegarderons aussi ses paramètres, et si nous donnons le dossier de notre projet à quelqu'un, en ouvrant le projet, il n'aura plus qu'à le compiler. Tous les fichiers nécessaires seront présents et tous les paramètres seront réglés. Pour faire de même avec MinGW, nous pourrons faire un *makefile*.

2.2. Avec MinGW

Après nous être placés dans le dossier de notre projet, la commande de compilation est très simple. Elle nécessite trois nouvelles options:

— `-I` pour indiquer au préprocesseur des répertoires de fichiers d'en-tête;
— `-L` pour indiquer à l'éditeur de liens des répertoires de fichiers compilés de bibliothèques;
— `-l` pour lier des bibliothèques à notre projet.

Avec une arborescence de ce type…

```
/
        include/
        lib/
                libZDS.a
        src/
                main.c
```

… nous pouvons utiliser cette commande pour créer un exécutable.

```
gcc -Wall -I include -L lib src/main.c -o Programme -lZDS.a
```

La commande se comprend assez bien. Mais faire un *makefile* peut être mieux.

```
CC = gcc                  # Variable représentant le nom du compilateur.
CFLAGS = -Wall -I include # Variable représentant les options de compilation.
LDFLAGS = -L lib -lZDS    # Variable représentant les options d'édition de liens.


Programme : main.o        # Programme dépend de `main.o`.
        $(CC) main.o -o Programme $(LDFLAGS)

main.o : src/main.c       # `main.o` dépend de `src/main.c`.
        $(CC) $(CFLAGS) -c src/main.c -o main.o

# La syntaxe se comprend en remplaçant les variables par leur valeur : $(CC) → gcc.
```

Finalement, nous nous trouvons avec cette arborescence.

```
1  /
2          include/
3          lib/
4                  libZDS.a
5          src/
6                  main.c
7          Makefile
```

Avec MinGW, la commande permettant d'exécuter un *makefile* est `mingw32-make` et non pas `make`.

2.3. Avec Code::Blocks

Créons d'abord notre projet, puis dans le dossier de notre projet, créons les dossiers `lib`, `src` et `include`. Déplaçons alors notre fichier `main.c` dans notre dossier `src` (le mieux pour cela est de supprimer le fichier de notre projet, de le déplacer dans le dossier `src` et de l'ajouter au projet). Nous nous retrouvons donc avec une arborescence de ce type.

```
1  /
2          include/
3          lib/
4                  libZDS.a
5          src/
6                  main.c
7          autres_fichiers_de_code_blocks
```

Sous Code::Blocks, nous devons aller dans le menu «Project», puis dans «Build options», pour accéder aux options de compilation.

Il ne faut choisir ni «Debug» ni «Release», mais rester dans la configuration de l'intégralité du projet. Nos réglages seront alors appliqués à l'intégralité du projet et seront donc valables en changeant de cible.

Notons que quand nous choisissons le projet en général et non le mode «Debug» ou «Release», nous ne pouvons rien choisir pour «Policy» qui est alors grisé.

— Dans le menu «Search directories» → «Compiler», ajoutons le dossier `include` que nous avons créé. À la question «*Keep this as a relative path ?*», répondons oui, pour que son

chemin soit par rapport à notre projet (il sera donc `/include`). Le compilateur saura alors qu'il faut rajouter ce dossier aux dossiers dans lesquels chercher les fichiers d'en-tête.

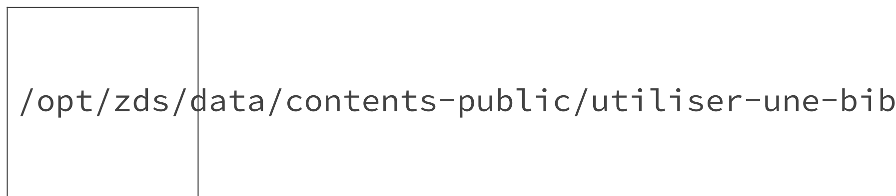

FIGURE 2.1. – On renseigne un chemin de fichiers d'en-tête.

— Dans le menu «Search directories» → «Linker», ajoutons le dossier `lib` que nous avons créé. À la question «*Keep this as a relative path ?*», répondons oui, pour la même raison que précédemment. L'éditeur de lien saura alors où chercher les fichiers compilés des bibliothèques.

FIGURE 2.2. – On renseigne un chemin de fichiers de bibliothèques.

— Dans le menu «Linker Settings», il faut alors renseigner les bibliothèques à lier au projet. On le fait, soit en les choisissant dans le répertoire avec le menu de gauche, soit en saisissant les options de *linkage* à rajouter (dans ce cas, on écrit `-lZDS` pour lier le fichier `libZDS.a` au projet).

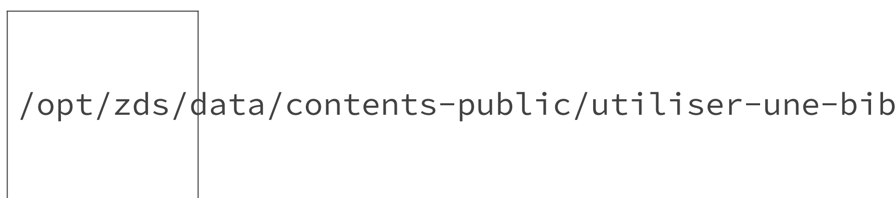

FIGURE 2.3. – On lie les bibliothèques.

Notre projet est fin prêt pour être compilé.

2.4. Erreurs possibles

Nous avons dit que la compilation pouvait mener à des erreurs. Voyons quelques erreurs fréquentes.

2.4.1. *Undefined reference to*

L'erreur *undefined reference to* intervient lorsque l'éditeur de liens ne trouve pas une fonction. C'est donc un problème de fichiers objets (et la plupart du temps de bibliothèques). Dans ce cas, il nous faut vérifier que:

— le fichier objet de la bibliothèque pour cette fonction existe;
— ce fichier est bien lié à la bibliothèque;
— ce fichier est compatible avec notre compilateur (s'il ne l'est pas, il n'est pas lu);
— il ne manque aucune bibliothèque dont dépendrait la première bibliothèque.

2.4.2. *No such file or directory*

L'erreur *no such file or directory* intervient lorsque le préprocesseur ne trouve pas un fichier. Dans ce cas, il nous faut vérifier que:

— le fichier existe;
— le nom du fichier est correct;
— nous avons indiqué à notre compilateur le répertoire qui lui permet d'avoir accès à ce fichier.

2.4.3. *Implicit declaration of function*

L'erreur *implicit declaration of function* est un problème de fichier d'en-tête, mais il n'est repéré qu'à la compilation. Il intervient lorsque le compilateur ne connaît pas une fonction (elle n'est pas encore déclarée), donc lorsque le fichier d'en-tête dans lequel elle est déclarée n'est pas inclus. Il nous faut donc vérifier que:

— le fichier d'en-tête dans lequel est déclaré la fonction est bien inclus;
— son nom est correct;
— la fonction est bien déclarée dans ce fichier.

Généralement, les erreurs obtenues sont dues à un de ces points et il faut donc songer à les vérifier avant de demander de l'aide.

Une erreur qui existe également, mais cette fois au niveau de l'exécution, est «impossible de démarrer le programme car il manque **xxx**.dll sur notre ordinateur» qui signifie tout simplement que le fichier `xxx.dll` n'est pas présent dans le même répertoire que notre exécutable.

3. Exemple : utiliser la bibliothèque SDL 2

La Simple DirectMedia Layer est une bibliothèque multimédia qui permet un accès de bas-niveau à l'audio, au clavier, à la souris, au joystick, aux graphismes… Elle permet donc de créer des applications multimédias, comme des jeux vidéo 2D, des émulateurs ou encore des lecteurs de médias, et elle peut même s'utiliser conjointement avec une bibliothèque 3D comme OpenGL. C'est cette bibliothèque que nous allons configurer dans cette partie. Cela nous permettra de mettre en pratique ce que nous avons appris, mais aussi de voir concrètement comment peut s'organiser le site d'une bibliothèque.

3.1. Organisation d'une bibliothèque

L'organisation d'une bibliothèque commence sur sa page de téléchargement avec le choix de la plate-forme, et même du compilateur (pour nous, Windows avec MinGW). Nous pouvons également avoir, outre le choix de l'architecture, des spécificités de compilation (elles peuvent également se présenter sous la forme de plusieurs dossiers dans l'archive de la bibliothèque):

- `x86` ou `i686` pour la bibliothèque en 32 bits;
- `x86_64` pour la bibliothèque en 64 bits;
- `static` pour la bibliothèque en statique;
- `debug` ou `d` pour la bibliothèque avec les informations de débogage.

Bien sûr, il en existe d'autres, et ils ne sont pas toujours tous présents.

Les bibliothèques ont ensuite une disposition commune et très pratique. Ainsi, on trouve généralement trois dossiers, qui correspondent tout simplement aux fichiers nécessaires:

- un dossier `bin` contenant les fichiers nécessaires à l'exécution (`.dll`);
- un dossier `include` contenant les fichiers d'en-tête de la bibliothèque;
- un dossier `lib` contenant les fichiers nécessaires à l'éditeur de liens.

Le dossier `bin` n'est pas toujours présent. Ses fichiers peuvent tout simplement être à la racine, ou encore ne pas exister (cas d'une bibliothèque statique). D'autres fichiers peuvent être présents (crédits, *readme*, documentation…).

3.2. Téléchargement de la bibliothèque

1. Commençons par nous rendre sur le site de la SDL. Dans le menu à droite, nous avons une section «*Download*». Nous voulons configurer un projet utilisant la SDL 2, c'est donc cette version que nous choisissons.
2. La page de téléchargement est séparée en plusieurs parties. Celle qui nous intéresse est bien évidemment la section «*Development Libraries*». Comme nous l'avons dit, elle est elle-même composée de plusieurs sous-parties correspondant à divers systèmes d'exploitation.
3. Dans la sous-section «Windows», nous avons deux liens qui correspondent aux compilateurs MinGW et Visual C++. C'est le compilateur MinGW qui nous intéresse. Téléchargeons-le. Nous remarquons que les fichiers pour la compilation en 32 bits et ceux pour la compilation en 64 bits sont tous les deux présents dans l'archive.

En décompressant le fichier téléchargé, nous obtenons plusieurs dossiers. Le dossier `i686-w64-mingw32` pour les compilateurs 32 bits, celui `x86_64-w64-mingw32` pour les 64 bits… Ici, nous considérerons que nous sommes avec un compilateur 32 bits, mais il faut bien faire attention à bien choisir le dossier correspondant à notre compilateur. Voici son arborescence.

```
/
        bin     /* Contient les fichiers `.dll`. */
        include /* Contient les fichiers d'en-tête. */
        lib     /* Contient les fichiers de bibliothèque. */
        share
```

3.3. Mise en place du projet

> Peu importe ce que nous utilisons, il faut lire la partie sur MinGW, car c'est lui qui est utilisé comme compilateur avec Code::Blocks. De plus, certaines informations ne seront pas reprises.

En nous baladant dans les fichiers, nous pouvons remarquer qu'il existe un fichier `libSDL2.dll.a`, mais aussi `libSDL2.a`. Ces deux fichiers correspondent à la même chose, mais l'un permet de lier la bibliothèque statique, et l'autre la bibliothèque dynamique.

> Mais où obtenir cette information? Si c'est comme ça pour toutes les bibliothèques on risque de se perdre, non?

En fait, les bibliothèques ont dans leur documentation, mais aussi dans l'archive téléchargée, des informations de configuration. Et si nous ouvrons les fichiers `lib/pkgconfig/sdl2.pc` ou `bin/sdl2-config` avec un éditeur de texte, nous pouvons voir les lignes suivantes.

```
Libs: -L${libdir}  -lmingw32 -lSDL2main -lSDL2  -mwindows
Libs.private: -lmingw32 -lSDL2main -lSDL2  -mwindows
    -Wl,--no-undefined -lm -ldinput8 -ldxguid -ldxerr8 -luser32
    -lgdi32 -lwinmm -limm32 -lole32 -loleaut32 -lshell32 -lversion
    -luuid -XCClinker -static-libgcc
```

```
--libs)
      echo -L${exec_prefix}/lib  -lmingw32 -lSDL2main -lSDL2
          -mwindows
      ;;
    --static-libs)
```

```
5 #    --libs|--static-libs)
6       echo -L${exec_prefix}/lib  -lmingw32 -lSDL2main -lSDL2
            -mwindows  -Wl,--no-undefined -lm -ldinput8 -ldxguid
            -ldxerr8 -luser32 -lgdi32 -lwinmm -limm32 -lole32
            -loleaut32 -lshell32 -lversion -luuid -XCClinker
            -static-libgcc
```

Nous avons bien là les bibliothèques à lier. En fait, pour la SDL sous Windows, les bibliothèques à lier sont :

— en dynamique, `-lmingw32 -lSDL2main -lSDL2 -mwindows` ;
— en statique, `-lmingw32 -lSDL2main -lSDL2 -mwindows -lm -ldinput8 -ldxguid -ldxerr8 -luser32 -lgdi32 -lwinmm -limm32 -lole32 -loleaut32 -lshell32 -lversion -luuid -static-libgcc`.

Nous voyons donc que seul `libSDL.DLL.a` ou `libSDL.a` est utile. Le premier est à utiliser pour compiler en dynamique et le second pour compiler en statique. Le fichier `mingw32.a` est un fichier présent dans les dossiers de MinGW. Nous n'avons pas à nous en soucier. L'éditeur de liens saura où le trouver, de même que les autres bibliothèques que nous avons liées (hormis la SDL bien sûr).

L'option `-mwindows` n'est pas obligatoire. En fait, elle permet de supprimer la console. Sans cette option, la console apparaîtra toujours lors de l'exécution de notre programme, et nous pourrions l'utiliser (ce qui peut être un comportement souhaité).

En dynamique, le fichier `SDL2.dll` du dossier `bin` est indispensable.

Le fichier `sdl2.pc` est un fichier utilisé par le programme `pkg-config`. Le programme `pkg-config` a pour but de faciliter l'utilisation des bibliothèques. Nous n'en traiterons toutefois pas ici.

3.3.1. Avec MinGW (en dynamique)

Nous allons créer le dossier de notre projet—pour nous `test`—et, dans ce dossier, nous allons créer nos différents sous-dossiers. Maintenant, il nous faut placer les fichiers de la SDL comme il le faut.

Nous copions le fichier `SDL2.dll` du dossier `bin` dans le dossier de notre projet, le dossier `SDL2` du dossier `include` dans le dossier `include` de notre projet et les fichiers `libSDL2.dll.a` et `libSDL2main.a` du dossier `lib` dans notre dossier `lib`.

Créons maintenant un fichier `main.c` qui nous permettra de vérifier que notre configuration marche.

Finalement, l'arborescence du projet est celle-ci.

```
/
        include/
                SDL2/
                        /* Tous les fichiers d'en-tête de la SDL.
                           */
        lib/
                libSDL2.dll.a
                libSDL2main.a
        src/
                main.c
        SDL2.dll
```

Avec toutes ces données, nous devons être capable de trouver la commande permettant de compiler.

```
gcc -L lib -I include main.c -o Programme -lmingw32 -lSDL2main
    -lSDL2  -mwindows
```

Et même de faire un *makefile*.

```
CC = gcc
CFLAGS = -Wall -I include
LDFLAGS = -L lib -lmingw32 -lSDL2main -lSDL2 -mwindows

Programme : main.o
        $(CC) main.o -o Programme $(LDFLAGS)

main.o : src/main.c
        $(CC) $(CFLAGS) -c src/main.c -o main.o

# Suppression des fichiers temporaires.
clean :
        del -rf *.o

# Suppression de tous les fichiers, sauf les sources,
# en vue d'une reconstruction complète.
mrproper : clean
        del Programme
```

3.3.2. Avec Code::Blocks (en dynamique)

Pour commencer, créons un projet en C. Nous allons configurer le projet nous-mêmes, donc nous choisissons un projet en console et n'utilisons pas le *template* de Code::Blocks pour la SDL.

Allons dans le dossier de notre projet, et créons tous les dossiers dont nous aurons besoin: les dossiers `src`, `include` et `lib`. Bien sûr, déplaçons le fichier `main.c` créé en même temps que le projet dans le dossier `src` que nous venons de créer. Maintenant, plaçons les fichiers de la SDL correctement.

Nous copions le fichier `SDL2.dll` du dossier `bin` dans le dossier de notre projet, le dossier `SDL2` du dossier `include` dans le dossier `include` de notre projet et les fichiers `libSDL2.dll.a` et `libSDL2main.a` du dossier `lib` dans notre dossier `lib`.

Configurons maintenant notre compilateur. Allons donc dans le menu «Project» → «Build options». Là, commençons par indiquer au compilateur les dossiers `lib` et `include`, dans les onglets «Search directories» → «Linker» et «Search directories» → «Compiler». Et finalement, dans l'onglet «Linker settings», écrivons `-lmingw32 -lSDL2main -lSDL2 -mwindows` dans la partie «Other linker options» pour lier ces fichiers.

3.4. Code de test

Testons maintenant cette configuration. Voici un code de test à placer dans le fichier `src/main.c`.

```c
#include <stdio.h>
#include <SDL2/SDL.h>

int main(int argc, char* argv[])
{
        SDL_Window *window = NULL;
        if(0 != SDL_Init(SDL_INIT_VIDEO))
        {
                fprintf(stderr,
                    "Erreur d'initialisation de la SDL : %s\n",
                    SDL_GetError());
                return -1;
        }
        window = SDL_CreateWindow("Test", SDL_WINDOWPOS_CENTERED,
            SDL_WINDOWPOS_CENTERED,
                                      500, 500, SDL_WINDOW_SHOWN);
        if(NULL == window)
                fprintf(stderr,
                    "Erreur de creation de la fenetre : %s\n",
                    SDL_GetError());
        else
        {
                SDL_Delay(500);
                SDL_DestroyWindow(window);
        }
        SDL_Quit();
        return 0;
}
```

Nous sommes censés obtenir une fenêtre vide, de largeur et de hauteur 500 pixels, et de position centrée. Elle reste affichée 500 ms et se ferme ensuite.

Pour faire un test, nous pouvons aussi compiler ce code en statique et comparer la taille des deux exécutables.

Voilà, nous avons fini ce tutoriel. Nous pouvons maintenant partir vers de nouveaux horizons et approfondir notre connaissance du langage C.

De grands remerciements vont à @**Dominus Carnufex** pour son aide dans la correction orthotypographique et à @Taurre pour ses très nombreuses remarques et pour avoir validé ce tutoriel.

N'hésitez pas à demander de l'aide sur le forum .