Bonnes pratiques de codage en langage C

par Emmanuel Delahaye (Site d'Emmanuel Delahaye)

Date de publication : 16 septembre 2008

Dernière mise à jour : 2 septembre 2009

Cet article est une synthèse de plusieurs notes autour des bonnnes pratiques de codage à adopter en langage C.

Votre avis et vos suggestions sur cet article nous intéressent !
Alors après votre lecture, n'hésitez pas :

I - Présentation du code source

Il est extrêmement facile d'écrire du code illisible en C, que ce soit par une mauvaise présentation, ou par un usage abusif des formes contractées.

Il est, par contre très utile, de présenter du code clair. Cela aide à la compréhension, à la relecture, au contrôle visuel, à la maintenance.

I-A - Jeu de caractères

Le jeu de caractères normalisé par le langage C est

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
```

plus l'espace, et les caractères de commande représentants :

- le saut de ligne
- le retour chariot
- la tabulation horizontale
- la tabulation verticale
- le saut de page

L'utilisation de caractères supplémentaires (accentués, par exemple) invoque un comportement défini par l'implémentation.

Dans la pratique, la plupart des compilateurs acceptent les extensions courantes comme IBM PC8 et ISO 8859-1 (aussi appelées respectivement OEM et ANSI dans le monde MS-DOS/Windows).

Personnellement, j'évite d'utiliser les caractères autres que les caractères standards dans un code source, y compris dans les commentaires.

I-B - Indentation

I-B-1 - Comment bien présenter le code source

Il existe de nombreuses façons de présenter le code C. Certaines sont plus prisées que d'autres, et je n'entrerai pas dans ce débat. Par contre, il est important, une fois qu'on a décidé de ce qui était "bien", de s'y tenir.

Tout le code présenté sur ce site observe des règles homogènes de présentation. Un moyen simple d'y parvenir est d'utiliser un outil unique pour la mise en page (indentation), tel que **GNU Indent**. Cet outil dispose de nombreux paramètres que personnellement j'ai fixés une fois pour toute, et qui confèrent à mes codes sources une certaine unité.

```
indent -bli0 -npsl -i3 -ts0 -sob
```

I-B-2 - Tabulations contre espaces

Il est fortement recommandé d'utiliser un éditeur qui permet de remplacer les tabulations par des espaces (l'idéal étant que le nombre soit programmable). En effet, l'effet d'une tabulation n'est pas uniforme, alors que celui d'un espace l'est certainement d'avantage (à part en HTML, mais il y a des contournements possibles).

I-C - Commentaires

I-C-1 - Comment bien gérer les commentaires

Le langage C, dans sa version normalisée ISO-1990, ne supporte que les commentaires de type /* comment */. L'imbrication des commentaires n'est pas autorisée.

Il est préférable d'éviter de placer des commentaires en bout de ligne. En effet, certains périphériques ou applications limitent la largeur visible du texte à moins de 80 colonnes. Ce serait dommage de se priver de commentaires qui peuvent être utiles. Il est donc recommandé de placer ses commentaires au-dessus de la portion de code à commenter.

```
/* compteur */
int cpt ;
```

Si un commentaire doit utiliser plusieurs lignes, il est recommandé d'utiliser la présentation 'ouverte', qui facilite la maintenance, tout en préservant la lisibilité. Exemple de forme minimale :

```
/* Ce compteur doit etre initialise
   avant utilisation */
int cpt ;
```

S'il faut isoler provisoirement une portion de code, le mieux est de ne pas utiliser les commentaires (il pourrait y avoir des commentaires imbriqués), mais plutôt les directives du préprocesseur : #ifdef .. #endif ou #if .. #endif

```
#if 0
  /* Compteur */
  int cpt ;
#endif
```

I-C-2 - Comment bien commenter

Le moins possible !

Le principe est de ne commenter que ce qui apporte un supplément d'information. Il est d'usage d'utiliser en priorité l'auto-documentation, c'est à dire un choix judicieux des identificateurs qui fait que le code se lit 'comme un livre'...

I-D - Editeur de texte

I-D-1 - Bien choisir son éditeur de texte

L'outil avec lequel un programmeur passe probablement le plus de temps est son éditeur de texte. Il entre dans une part importante de la productivité du programmeur et de la présentation du code.

Je n'entrerai évidemment pas dans le débat futile Vi/Emacs, car je mets tout le monde d'accord avec UltraEdit32 ! En fait peu importe, pourvu qu'il dispose au moins des fonctions suivantes :

- Police à chasse fixe
- Coloration syntaxique
- La touche TAB produit des espaces (programmable, pour moi, c'est 3)
- Édition en mode colonne
- Annulation
- Copie de sauvegarde
- Patrons avec fonctions de base comme le nom du fichier et la date de création
- La possibilité d'ajouter des outils externes pour compléter ce qui manque

II - Conventions de nommage

Une des façons d'obtenir du code clair est de s'en tenir à une convention de nommage des identificateurs qui soit cohérente et parlante. Voici mes recommandations.

II-A - Utilisation du caractère '_' (Underscore)

Ce caractère peut être utilisé comme séparateur visuel dans les identificateurs. Bien que ce soit techniquement possible, Je recommande de ne pas l'utiliser au début d'un identificateur, pour deux raisons principales :

- La plupart de ces identificateurs sont réservés à l'implémentation du langage
- L'aspect très inesthétique de ces identificateurs

II-B - Macros

Les macros ayant des usages multiples, il est difficile de faire une généralité. Dans les cas où elles représentent une constante, il est recommandé d'utiliser les majuscules. Dans les autres cas, le choix sera le même que pour l'identificateur qu'elles substituent. Cependant, si on cherche à insister sur le fait qu'une soi-disant fonction est en fait une macro, on peut utiliser les majuscules. En fait tout dépend du contexte.

II-C - Constantes

II-C-1 - Conventions typographiques

Il est couramment admis que les constantes 'vraies' (macros, énumérations) doivent être écrites en majuscules. Les variables qualifiées const seront écrites comme des variables.

II-C-2 - Identificateur

L'identificateur dépend du contexte. Il sert souvent à définir les valeurs des propriétés d'un objet (valeurs particulières, états, etc.). Il est recommandé d'utiliser un préfixe qui relie les constantes qui qualifient une même propriété.

```
/*
* Valeurs possibles de l'etat du voyant
*
* OFF   : eteint
* ON    : allume fixe
* BLINK : clignotant
*/
enum
{
  VOYANT_STS_OFF,
  VOYANT_STS_ON,
  VOYANT_STS_BLINK,

  VOYANT_STS_NB
};
```

II-C-3 - Définition

Il y a deux façons de créer des valeurs constantes. Avec une macro ou avec une énumération.

Pour les constantes numériques de type int, il est recommandé d'utiliser les enum, qui présentent des avantages comme le typage, la numérotation automatique, ou l'interprétation textuelle automatique dans les débogueurs.

Pour les constantes numériques des autres types (long, unsigned etc.) et les chaines, il n'y a pas le choix, il faut utiliser les macros. Ne pas oublier les qualificateurs de macros pour les valeurs numériques autres que int : Suffixes de macros numériques u ou U unsigned

l ou L	long
ul ou UL	unsigned long
f ou F	float

Ne pas oublier non plus que pour qu'une constante soit de type double, au moins un des membres doit être de type double, donc comprendre un '.' dans son expression :

```
#define PI (22/7)
```

est un entier qui vaut 3

```
#define PI (22/7.0)
```

est un double qui vaut environ 3.14

II-D - Types

Le langage C permet de créer un *alias* sur un type existant plus ou moins complexe ou un autre *alias* existant à l'aide du mot clé typedef.

II-D-1 - Conventions typographiques

Il est d'usage d'utiliser les minuscules. Il est recommandé d'utiliser des suffixes tels que :

_e	typedef enum
_u	typedef union
_s	typedef struct
_a	tableau (*array*)
_f	fonction

Il est recommandé de ne pas utiliser le suffixe '_t', car il est réservé par la norme POSIX pour définir des alias de types.

II-E - Objets (Variables)

II-E-1 - Conventions typographiques

Il est d'usage d'utiliser les minuscules.

II-E-2 - Identificateur

Il est un principe général de bonne conception qui veut que la longueur des identificateurs des objets et des fonctions soit proportionnelle à leur portée.

Les variables indroduisent en plus la notion de durée de vie (duration). Il est bon qu'au premier coup d'oeil, on sache exactement à quoi on a affaire. Il est bon de pouvoir aussi identifier rapidement les variables en fonction de leurs propriétés. Les plus remarquables sont :

- simple (*plain*)
- pointeur
- tableau statique
- tableau dynamique
- chaîne de caractères

Il est donc recommandé d'utiliser les préfixes suivants :

	simple
p_	pointeur
a_ ou sa_	tableau statique(*static array*)
da_	tableau (*dynamic array*)
s_	chaîne de caractères (*string*)

et en ce qui concerne la durée de vie et la portée:

	Portée	**Durée de vie**
	bloc	bloc
g_	bloc	programme
S_	module	programme
G_	programme	programme

II-F - Fonctions

II-F-1 - Conventions typographiques

Il est d'usage d'utiliser les minuscules. Il existe des cas où l'on a besoin de 2 mots pour nommer une fonction. Il n'est évidemment pas question d'accoler ces deux mots en minuscule, car le code serait vite illisible. Il existe 2 pratiques répandues :

- Coller les mots en mettant une majuscule au début de chaque mot :

```
OuvrirFichier()
```

ou (variante 'à-la-Java') à partir du 2ème mot :

```
ouvrirFichier()
```

- Séparer les mots en mettant un souligné (underscore) entre chaque mot.

```
ouvrir_fichier()
```

C'est une question de goût, le principal est d'être clair et cohérent. La seconde méthode a ma préférence.

II-F-2 - Identificateur

Pour une fonction, on choisit plutôt un identificateur qui exprime une action (verbe, verbe + substantif).

III - Organisation du code source

Il est pratique d'adopter une disposition logique et cohérente dans les fichiers sources et d'en-têtes. Il est d'usage d'utiliser un principe simple, qui consiste à définir ce que l'on doit utiliser. C'est pourquoi la disposition suivante est recommandée :

III-A - Fichiers sources (*.c)

III-A-1 - Liste ordonnée des éléments pouvant être contenus dans un fichier source

- Inclusion des en-têtes (.h) nécessaires
- Définition des macros privées
- Définition des constantes privées
- Définition des types privés
- Définition des structures privées
- Définition des variables globales privées
- Définition des fonctions privées
- Définition des fonctions publiques
- Définition des variables globales publiques

Voici par exemple, une liste de séparateurs utilisés pour structurer des fichiers sources :

```
/* macros ================================================================ */
/* constants ============================================================= */
/* types ================================================================= */
/* structures ============================================================ */
/* private variables ===================================================== */
/* private functions ===================================================== */
/* internal public functions ============================================= */
/* entry points ========================================================== */
/* public variables ====================================================== */
```

III-B - Fichiers d'en-tête (*.h)

III-B-1 - Règles d'or régissant la définition des fichiers d'en-tête

- Un fichier d'en-tête doit être protégé contre les inclusions multiples dans la même unité de compilation
- Un fichier d'en-tête doit être autonome[1]
- Un fichier d'en-tête ne doit inclure que le strict nécessaire à son autonomie

[1] Pour s'en assurer, lors des tests de l'implémentation correspondante, le placer en 1ère position et ajouter ce qu'il manque dedans.

III-B-2 - Liste ordonnée des éléments pouvant être contenus dans un fichier d'en-tête

- Inclusion des headers nécessaires et suffisants
- Définition des macros publiques
- Définition des constantes publiques
- Définition des types publiques
- Définition des structures publiques
- Déclaration des fonctions publiques
- Déclaration des variables globales publiques
- [C99] Définition des fonctions publiques inline

Voici par exemple, une liste de séparateurs utilisés pour structurer des fichiers d'en-tête :

```
/* macros ================================================================ */
/* constants ============================================================= */
/* types ================================================================= */
/* structures ============================================================ */
/* internal public functions ============================================= */
/* entry points ========================================================== */
/* public variables ====================================================== */
/* inline public functions  ============================================== */
```

III-B-3 - Comment bien utiliser les séparateurs

A l'origine, on a un fichier avec main() (par exemple main.c, comme ça, on voit tout de suite de quoi on parle). main() est le point d'entrée (unique) de ce module. On peut donc écrire :

```
/* main.c */
#include

/* entry points ========================================================== */

int main (void)
{
   puts ("hello world");

   return 0;
}
```

Le programme évoluant, on a maintenant une fonction privée (static) :

```
/* main.c */
#include

/* private functions ===================================================== */

static void hello(void)
{
   puts ("hello world");
}

/* entry points ========================================================== */

int main (void)
{
   hello();

   return 0;
}
```

Nouvelle évolution, on crée un bloc fonctionnel (BF) hello composé pour le moment d'une seule fonction. On retrouve donc la structure habituelle avec 3 fichiers :

```
#ifndef H_HELLO
#define H_HELLO
/* hello.h */

/* entry points ========================================================== */

void hello(void);
```

```
/* hello.c */
#include "hello.h"
#include

/* entry points ========================================================== */
```

```
void hello(void)
{
   puts ("hello world");
}
```

```
/* main.c */
#include "hello.h"

/* entry points ============================================================ */

int main (void)
{
   hello();

   return 0;
}
```

On fait dans le luxe et on décompose la fonction hello() en plusieurs petites fonctions (un peu théorique, mais c'est pour montrer le principe).

```
/* hello.c */
#include "hello.h"
#include

/* private functions ======================================================= */

static void h(void)
{
   printf ("hello");
}

static void w(void)
{
   printf ("world");
}

static void spc(void)
{
   putchar (' ');
}

static void eol(void)
{
   putchar ('\n');
}

/* entry points ============================================================ */

void hello(void)
{
   h();
   spc();
   w();
   eol();
}
```

Le reste (interface, allocation) est inchangé.

Attention, étape ultime et subtile.

Le fichier d'implémentation de hello est devenu tellement gros qu'il nécessite lui-même un découpage. Il va donc falloir distinguer ce qui est accessible à l'utilisateur de hello (les points d'entrées) et les fonctions 'internes', c'est à dire connues de hello mais pas de l'application, d'où cette distinction, qui se traduit aussi par des headers distincts :

```
/* hellotools.c */
#include "hellotools.h"
```

```
#include

/* internal public functions =================================================== */
void hello_spc(void)
{
   putchar (' ');
}

void hello_eol(void)
{
   putchar ('\n');
}
```

```
#ifndef H_HELLOTOOLS
#define H_HELLOTOOLS
/* hellotools.h */

/* internal public functions =================================================== */
void hello_spc(void);
void hello_eol(void);

#endif
```

```
/* hello.c */
#include "hellotools.h"
#include "hello.h"

#include

/* private functions =========================================================== */
static void h(void)
{
   printf ("hello");
}

static void w(void)
{
   printf ("world");
}

/* entry points ================================================================ */
void hello(void)
{
   h();
   hello_spc();
   w();
   hello_eol();
}
```

Cela revient à créer plusieurs niveaux de visibilité...

III-B-4 - Protection contre les inclusions multiples

Les fichiers d'en-tête pouvant être inclus dans des fichiers sources, comme dans des fichiers d'en-tête, et ce, dans un ordre non spécifié, il est indispensable de se prémunir contre les risques d'inclusions multiples dans la même unité de compilation.

```
   #ifndef IDENTIFICATEUR
   #define IDENTIFICATEUR

   /* zone protegee contre les inclusions multiples */

   #endif /* guard */
```

Le principe de protection étant basé sur la définition d'une macro, cette macro doit être unique afin d'éviter les protections abusives.

Personnellement, j'utilise le format suivant :

```
H_<initials>_<file>_<date>
initials ::= En majuscule : initiales du développeur, nom de la société etc.
file     ::= En majuscule : nom du fichier (sans l'extension)
date     ::= <year><month><day><hour><minute><second>
year    ::= année (0000-9999)
month   ::= mois (01-12)
day     ::= jour (01-31)
hour    ::= heure (00-23)
minute  ::= minute (00-59)
second  ::= seconde (00-59)
```

Par exemple :

```
H_ED_EXEMPLE_20040529115235
```

Le choix de mettre le H_ en tête est justifié. En effet, il évite de briser une règle du langage C qui dit qu'un identificateur commençant par E suivi d'une majuscule est réservé pour l'implémentation. En fait ils sont utilisés pour implémenter les valeurs symboliques de errno.

IV - Comment bien organiser son développement

Le langage C autorise la compilation séparée. Cette technique permet de créer des unités de compilations (*compile units*) séparées. Une bonne maîtrise de la programmation permet de réaliser du code indépendant ou tout au moins suffisamment indépendant pour être testable individuellement.

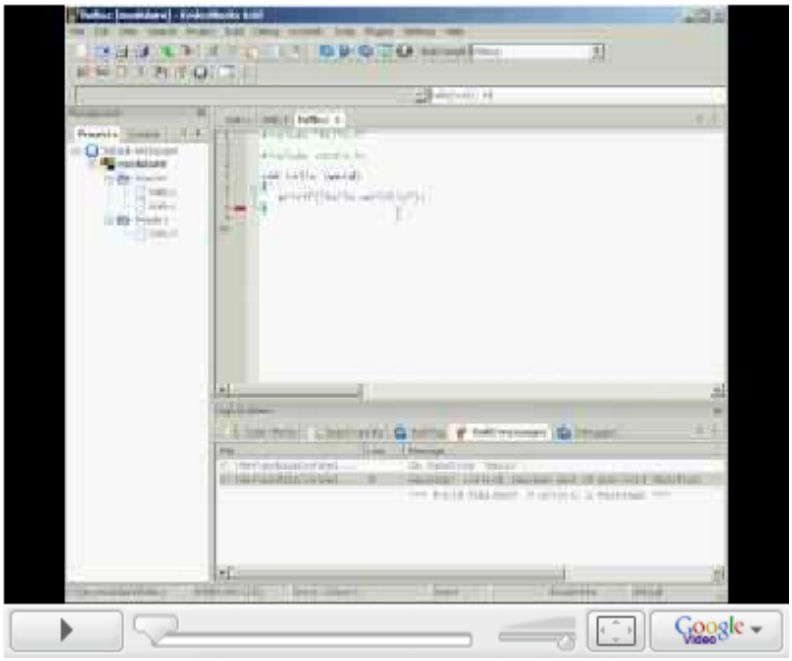
Voici un exemple élémentaire de compilation séparée reprenant le célèbre "Hello world!". Le code est divisé en 3 sources. Un fichier d'interface (hello.h), un fichier d'implémentation (hello.c) et un fichier d'application (main.c) :

main.c
```
/* main.c */

#include "hello.h"

int main (void)
{
  hello ();
  return 0;
}
```

hello.h
```
#ifndef H_HELLO
#define H_HELLO

/* hello.h */

void hello (void);

#endif /* guard */
```

hello.c
```
/* hello.c */

#include "hello.h"
#include <stdio.h>

void hello (void)
{
  puts ("Hello world!");
}
```

Vidéo sonore (flash) expliquant comment faire sous **Code::Blocks** (malheureusement, la compression par Google rend l'image peu lisible)



Cliquer sur l'image ci-dessus pour lancer l'animation flash

Afin de bien organiser les fichiers, je conseille de créer des répertoires pour les sources et pour les fichiers de sortie:

Type	Répertoire	Fichiers	nom.ext
Génération	./	*make*	Makefile
Source	inc/	inclus	*.h
Source	src/	source	*.c
Sortie	obj/	objets intermédiaires	*.obj
Sortie	exe/	exécutable	*.exe

et par conséquent, d'adopter l'organisation suivante pour les répertoires :

```
hello/Makefile
hello/inc/hello.h
hello/src/hello.c
hello/src/main.c
hello/obj/hello.obj
hello/obj/main.obj
hello/exe/hello.exe
```

Afin de garantir un comportement correct du code et des outils de développement, je recommande que les noms de répertoires et de fichiers soient tous écrits en minuscule. Je rappelle que si on utilise un outil de gestion de version comme ***CVS****, il est extrêmement difficile de modifier la casse des répertoires et même des fichiers après coup. C'est donc une démarche à adopter dès la première ligne de code.*

Pour compiler et exécuter ce code sous gcc avec cette organisation, on peut utiliser ce Makefile (**DJGPP**). (Remplacer <tab> par une véritable tabulation)

```
# Makefile HELLO

# Paths
DSRC = src
DINC = inc
DOBJ = obj
DEXE = exe
```

```
DLIB = d:/djgpp/lib

# Compiler flags
CFLAGS = -I$(DINC)

# Commands
CC = gcc -c $(CFLAGS)
LK = ld
RM = del

# Project list
OBJS =      \
$(DOBJ)/hello.obj  \
$(DOBJ)/main.obj   \

#Rebuild all target
all: $(DEXE)/hello.exe

# Clean all target
clean:
<tab>cd $(DOBJ)
<tab>$(RM) *.obj
<tab>cd ../$(DEXE)
<tab>$(RM) *.exe
<tab>cd ..

# main target (OBJS + init code + library)
$(DEXE)/hello.exe : $(OBJS)
<tab>$(LK) -o $(DEXE)/hello.exe $(OBJS) $(DLIB)/crt0.o -lc

# objects

# The hello.c file
$(DOBJ)/hello.obj : $(DSRC)/hello.c          \
    $(DINC)/hello.h           \
    Makefile
<tab>$(CC) $(DSRC)/hello.c -o$(DOBJ)/hello.obj

# The main.c file
$(DOBJ)/main.obj : $(DSRC)/main.c            \
    $(DINC)/hello.h           \
    Makefile
<tab>$(CC) $(DSRC)/main.c -o$(DOBJ)/main.obj
```

Avec une utilisation comme suit (MS-DOS):

- Génération ou mise à jour :

```
D:\HELLO>make
gcc -c -Iinc src/hello.c -oobj/hello.obj
gcc -c -Iinc src/main.c -oobj/main.obj
ld -o exe/hello.exe obj/hello.obj obj/main.obj  d:/djgpp/lib/crt0.o -lc
```

- Effacement complet :

```
D:\HELLO>make clean
cd obj
del *.obj
cd ../exe
del *.exe
cd ..
```

Pour en savoir plus, consultez l'article ***l'outil MAKE***

V - Comment construire sa bibliothèque

Le langage C utilisant lui même la notion de bibliothèque de fonctions, il est logique que cette possibilité soit offerte aux développeurs. Les avantages sont bien connus :

- Identification et bonne gestion du code réutilisable
- Interface bien défini (headers)
- Gain de productivité (réutilisation,moins de code à compiler)

Les détails de réalisation dépendent des outils de développement, mais le principe d'organisation du code est le même. La compilation d'une bibliothèque est évidemment une application directe du principe de la **compilation séparée**.

S'ajoutent quelques règles de bon sens telles que l'indépendance du code vis à vis de l'application. Cela concerne particulièrement les sorties qui, si il le faut, seront implémentées par des 'callbacks' (détails dans cet article : **Concevoir et réaliser un composant logiciel en C**). Plus que jamais, les globales seront évitées.

Pour réaliser la bibliothèque, il faut créer un projet rassemblant les fichiers d'interface (headers : .h) et les fichiers d'implémentation (sources : .c). Evidemment, il ne doit pas y avoir de fonction main().

Ensuite, après compilation classique, et sans faire d'édition de lien, bien évidemment, on utilise un *'librarian'* qui est un outil particulier faisant partie des outils de développements courants, et qui sert à créer la bibliothèque. Le résultat est un fichier dont l'extension dépend de l'outil. (par exemple, .lib ou .a)

V-A - Outil de développement gcc

Tout d'abord, avec gcc, il y a des règles de nommage à respecter. En effet, le nom du fichier produit doit impérativement commencer par **lib** et son extension doit être **.a** (*comme archive*).

Ensuite, l'outil s'appelle **ar** (ar.exe sous DOS/Windows) comme ... *archiver*. Les détails d'appels en mode commande sont à lire dans la documentation de gcc. Comme toujours en mode ligne de commande, un Makefile simplifie et automatise la tâche de production du code.

Les exemples suivants utilisent le fichier d'interface **hello.h** et le fichier d'implémentation **hello.c** décrits dans l'article sur la **compilation séparée**. La bibliothèque à créer s'appelle **libhello.a**. Elle ne sert évidemment qu'à illustrer le propos.

V-B - Création d'une bibliothèque sous Code::Blocks 8.02

- File > New > Project
- Sélectionner 'static library'
- Sélectionner un répertoire <racine> et taper le nom de la bibliothèque : libhello **/!\ IMPORTANT /!**
- Le répertoire <racine>/libhello est créé automatiquement ; c'est le répertoire de travail
- Debug et Release sont cochées, ne rien changer
- Cliquer sur [finish]

Dans le 'workspace', une ligne en gras a été ajoutée : 'libhello'. Un fichier main.c a été créé. On peut le retirer du projet

- Clic droit > Remove from project

Il faut maintenant ajouter les 2 fichiers source (.c et .h). Si il n'y sont pas déjà, les déplacer (ou les créer) dans le répertoire du projet (<racine>/libhello), puis :

- Sur la ligne 'Biblio. Hello' : click droit
- Add files
- Vérifier qu'on est bien dans le répertoire <racine>/libhello
- (On en profite pour supprimer main.c)
- Selectionner hello.h et hello.c
- Valider
- Compiler

La fenêtre de compilation doit donner quelque chose comme ceci :

```
-------------- Build: Debug in libhello ---------------

Compiling: hello.c
Linking static library: libhello.a
ar.exe: creating libhello.a
Output size is 3.63 KB
Process terminated with status 0 (0 minutes, 0 seconds)
0 errors, 0 warnings
```

On voit que hello.c a été compilé, que ar.exe a été invoqué et que libhello.a a été crée. On peut maintenant créer un petit programme de test avec le main.c et la bibliothèque. Le plus simple pour le moment est de la créer dans le même répertoire.

```
Project    : Test hello
Compiler   : GNU GCC Compiler (called directly)
Directory : C:\dev\hello\
--------------------------------------------------------------------------------
Switching to target: default
Compiling: main.c
Linking console executable: C:\dev\hello\test.exe
.objs\main.o: In function `main':
C:/dev/hello/main.c:7: undefined reference to `hello'
collect2: ld returned 1 exit status
Process terminated with status 1 (0 minutes, 2 seconds)
1 errors, 0 warnings
```

Bien sûr, le main.c tout seul ne suffit pas, il faut ajouter la bibliothèque au projet :

- Sur la ligne 'Test biblio. Hello' : clic droit
- Build options > Linker > Add > [...] > libhello.a > Ouvrir > NON > OK > OK
- Compiler

On obtient alors :

```
Project    : Test hello
Compiler   : GNU GCC Compiler (called directly)
Directory : C:\dev\hello\
--------------------------------------------------------------------------------
Switching to target: default
Linking console executable: C:\dev\hello\test.exe
Process terminated with status 0 (0 minutes, 0 seconds)
0 errors, 0 warnings
```

et le fameux

```
Hello world!

Press ENTER to continue.
```

tant espéré !

VI - Comment bien configurer son compilateur

Le compilateur dispose de moyens d'analyse du code qui peuvent être mis à profit pour détecter toutes sortes d'erreurs. Il est donc important de savoir configurer son compilateur pour qu'il signale au mieux les anomalies pouvant se produire dans le code (Warnings). C'est ensuite au programmeur de réagir et, soit de justifier ou d'expliquer l'avertissement, soit de corriger le code.

La première des vérifications à faire est de s'assurer que l'on compile avec le bon compilateur. Certaines extensions ou réglages par défaut font que c'est parfois le compilateur C++ qui est invoqué au lieu du compilateur C. Ces quelques lignes placées au début de chaque fichier source (.c) permettent de détecter cette erreur :

```
#ifdef __cplusplus
#error This source file is not C++ but rather C. Please use a C-compiler
#endif
```

Si l'erreur se produit, vérifier les réglages et l'extension. Celle-ci doit impérativement être .c (minuscule), et non .cpp, ni .C (majuscule)

VI-A - gcc

gcc est le 'gnu c compiler', version **GNU** du cc Unix pour GNU/Linux. Il a été porté sur de nombreuses plateformes dont Windows (les plus connues sont **MinGW** et **CygWin**)

Par défaut, le niveau d'avertissement (*Warnings*) est très laxiste. Il est fortement recommandé d'utiliser la configuration minimale suivante (C90) :

```
-Wall -Wextra -ansi -O -Wwrite-strings -Wstrict-prototypes -Wuninitialized
-Wunreachable-code
```

Sur les anciennes version de gcc (< 3.4.x), remplacer -Wextra par -W.

Si la version de gcc le permet, il est possible d'ajouter ces options :

```
-Wno-missing-braces  -Wno-missing-field-initializers
```

Experts uniquement

Il est aussi possible de procéder à un réglage 'fin'. Voici comment je configure gcc en mode 'paranoïaque' (C90)

```
-O2 -Wchar-subscripts -Wcomment -Wformat=2 -Wimplicit-int
-Werror-implicit-function-declaration -Wmain -Wparentheses
-Wsequence-point -Wreturn-type -Wswitch -Wtrigraphs -Wunused
-Wuninitialized -Wunknown-pragmas -Wfloat-equal -Wundef
-Wshadow -Wpointer-arith -Wbad-function-cast -Wwrite-strings
-Wconversion -Wsign-compare -Waggregate-return -Wstrict-prototypes
-Wmissing-prototypes -Wmissing-declarations -Wmissing-noreturn
-Wformat -Wmissing-format-attribute -Wno-deprecated-declarations
-Wpacked -Wredundant-decls -Wnested-externs -Winline -Wlong-long
-Wunreachable-code
```

*Attention ce réglage révèle certains défauts dans les headers de **MinGW**. La correction est possible, mais uniquement si on sait ce qu'on fait. Je n'en dirai donc pas plus ici.*

Le détail de chaque paramètre est expliqué dans la **documentation de gcc**

VI-B - Réglages dans Code::Blocks

- Settings > Compiler > Onglet 'Compiler' > Onglet 'Other options'
- Copier/coller les options
- OK
- Régénérer (Ctrl-F11)

S'assurer qu'une coche ne traine pas dans l'onglet de configuration du compilateur

Plus d'informations sur Code::Blocks

VI-C - Réglages dans wxDev-C++ (remplace Dev-C++ devenu obsolète)

- Outils > Options du compilateur
- Dans la zone de saisie "Ajouter les commandes suivantes lors de l'appel du compilateur", copier/coller les options que je recommande
- Valider
- Régénérer (Ctrl-F11)

VI-D - Microsoft Visual C++ (6, 2005, 2008 etc.)

- Propriétés du projet -> C/C++ -> général
- Mettre le niveau (level) de warning à 4

Plus d'informations sur Microsoft Visual C++

VI-E - Borland C / Turbo C

- Option -> Compiler -> Messages -> Display
- (*) ALL

VII - Ressources

- **Recommended C Style and Coding Standards**
- **MISRA**
- **Macros prédéfinies**