

OPENGL  
COURSES  
ASM INUNE  
POINTEURS  
ALGORITHMES  
LISTES CHAINÉES  
CODING STYLE  
RAM SOCKETS

POUR LES DÉBUTANTS ET LES INITIÉS. POUR WINDOWS ET LINUX

# À apprendre à programmer en

# C

84 PAGES

n° 2 avril - mai 2005 / 6 €

# THE HACKADEMY PROG

# DEMANDEZ MÉDÉRIC 01 40 21 01 20

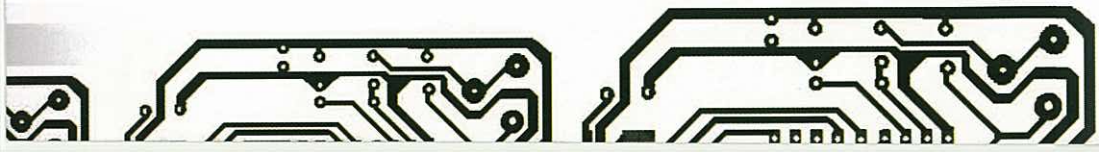
- Initiation Linux / OpenOffice
- Initiation HTML / PHP / Perl / C
- Hacking Newbie, hacking Pro, wifi, architecture sécurisée
- Initiation Internet/sécurité

**Dans toute la France  
et dans plusieurs villes d'Europe**

Cours de sécurité informatique et de  
hacking éthique pour pros et débutants

# THE HACKADEMY SCHOOL

## JOIN



Habitants de la planète Windows, autochtones de l'univers Unix, ce numéro dédié à la programmation en C a été conçu pour un public hétérogène. Cela fait un an que vous essayez de vous mettre au C sans jamais avoir trop compris ni réussi ? Ou peut-être avez-vous saisi l'esprit du C mais aimeriez en découvrir davantage ?

Sans être un guide complet pour apprendre le langage C de A à Z, ce numéro est un magazine de chevet ayant pour vocation de vous initier, tout d'abord, aux subtilités natives de ce langage qui traverse les âges. Découvrez donc les méthodes avancées de travail sur les pointeurs (algorithmes, manipulations), abordez des articles essentiellement orientés pratique pour la création d'outils, accompagnés parfois d'exercices et de conseils méthodologiques afin de vous permettre d'assimiler plus rapidement le dense contenu que nous avons concocté.

### Clad Strife

Post Scriptum : gros greet à Crashtr, ce valeur qui mérite bien que je le salue ici :-)

Création d'un contrôleur d'intégrité, initiation à l'OpenGL, développement d'arbres binaires... Les sujets abordés sont aussi divers que peuvent l'être vos centres d'intérêt. Un article d'exploitation avancée originale sur les buffers overflow constitue la petite surprise que nous a réservée l'équipe de l'ouvrage, mais il ne s'agit là que d'une goutte d'eau dans cet océan d'informations. En espérant qu'elle sera la plus enrichissante possible, je vous souhaite, amis lecteurs, une excellente lecture.

### Sommaire

La chaîne de compilation	p5
Règles de programmation	p9
Écrire un Makefile	p14
Les pointeurs	p18
Les tableaux de pointeurs	p20
Les structures	p23
Les listes chaînées	p26
Récursivité	p29
Assembleur inline	p30
Gdb	p32
Arbres binaires	p36
Coder un contrôleur d'intégrité	p41
Gestion du clavier	p50
Ret into ret	p54
Les bases d'OpenGL	p58
Return into libc	p63
Coder un sniffer TCP	p65
Forger des paquets sous Linux	p71
Initiation à ncurses	p76

## THE HACKADEMY PROG

est une publication de DMP  
26 bis rue Jeanne d'Arc 94160 Saint Mandé  
01 53 66 95 28

Rédacteur en Chef : Clad Strife  
Conception graphique : Weel  
Illustrations : Lechatkitu

Imprimé aux Imprimeries de Champagne  
Directeur de la Publication : Olivier Spinelli

© DMP

# ENTREZ DANS LA DANSE...

Le langage C est à son origine un langage développé dans les laboratoires Bell, à partir de 1972, par un ingénieur talentueux : Dennis M. Ritchie. En 1978, les compères Ritchie et Brian W. Kernighan définissent ensemble ce que devra être le langage C. On connaît alors ce langage sous le nom de K&R C, qui est détaillé dans un livre plus que mythique : « *The C Programming Language* ». En 1983 l'institut national des standards américains (ANSI) se met à l'étude d'une standardisation de ce langage et publie, en 1988, le standard ANSI-C. Le livre de K&R est alors republié en une deuxième édition respectant ce stan-



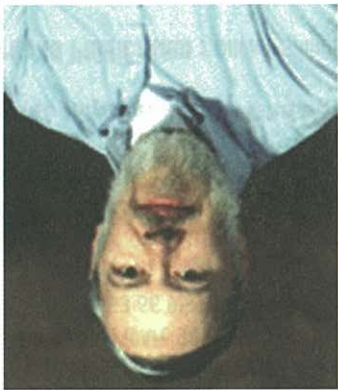
Brian W. Kernighan travaille aujourd'hui dans les laboratoires de Bell.

dard.

Le C est un langage proche de la machine. Ce que l'on faisait jusque là en des dizaines d'instructions assembleur tient en deux lignes sous cette forme. Mais

qui dit « *proche de la machine* » dit « *procédural* », soit à la fois une force et une faiblesse par rapport aux langages modernes, plus orientés objet.

Si le langage C résiste si bien au temps, c'est qu'il est très puissant, il permet de tout faire. Une fois passées les premières appréhensions sur la pseudo-complexité dudit langage, on en vient très vite à pouvoir implémenter facilement des routines de code performantes et modulaires. Et, gros avantage que vous avez à le connaître : une fois maîtrisé, il est beaucoup plus facile d'apprendre n'importe lequel... Bon courage ;-)



Dennis M. Ritchie fait également de la recherche dans les laboratoires de Bell.

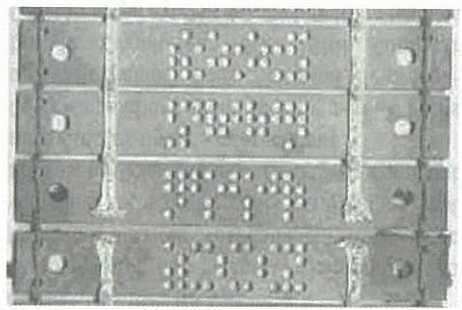
# LA CHAÎNE DE COMPILATION

Pour beaucoup de débutants, la compilation est un processus un peu mystique. Cet article explique en détails la chaîne de compilation des programmes C à travers l'histoire de la compilation.

## Historie de la compilation

Au début de l'histoire des ordinateurs, les opérateurs actionnaient des leviers pour coder les nombres à calculer, la compilation n'existait pas. Par la suite, le concept de programme est né : une suite d'instructions sur un carte perforée indiquant l'ordre des opérations à effectuer ; c'était donc un programme.

On procédait en écrivant un programme sur papier, on traduisait ensuite le programme en instructions sur carte perforée avec une machine à écrire spéciale. Lorsque les programmes fonctionnaient, on les archivait dans des bibliothèques pour ne pas avoir à les réécrire quand on en avait besoin - le concept de bibliothèque de programme était né.



La tâche de traduction, du programme écrit au programme sur carte perforée, était longue et sujette à erreurs. Il fallait des programmes per-

pendant à une routine (comme ceux écrits précédemment sur papier), puis de traduire ce texte en code d'instructions.

Cette étape est aujourd'hui appelée étape d'assemblage, elle permet la traduction de code en langage machine (le code directement exécuté par le processeur).

Les langages d'assemblage sont dépendants du microprocesseur utilisé et demandent du programmeur de faire attention à beaucoup de détails. Beaucoup d'erreurs de programmation surviennent donc (des "bugs").

La première solution à ce problème fut de construire sur ce qui existait déjà, on inventa les macro-processeurs. Ces programmes permettent d'associer un nom à une routine ; dès que ce nom est trouvé dans le fichier source, il est remplacé par la routine en question. Ce processus a évolué pour permettre de paramétrer les remplacements et de contrôler les noms auxquels les programmes ont accès.

Les méthodes de préprocesseurs sont devenues tellement évoluées que l'on avait un langage de programmation permettant d'écrire des programmes. A partir de cette observation, on inventa les langages de haut niveau comme le FORTRAN, le Pascal puis... le C. Ces langages ont une syntaxe qui n'est pas spécifique

ressources mémoire et processeur, les mor-  
ceaux de programmes sont distribués sous  
forme binaire et non sous forme de code.  
Il faut donc un moyen de récupérer les mor-  
ceaux de programmes contenus dans des biblio-  
thèques binaires et les lier au programme qui  
l'utilise. Cette étape est réalisée grâce à un édi-  
teur de liens.

Étant donné que beaucoup de programmes utili-  
sent les mêmes bibliothèques, on a trouvé judi-  
cieux de faire en sorte qu'elles ne soient char-  
gées qu'une seule fois en mémoire. Ceci permet  
de gagner beaucoup de place.  
La bibliothèque est liée au programme (étape  
d'édition de liens) au moment du chargement du  
programme, de telle sorte que si elle est déjà  
chargée en mémoire, c'est la copie déjà chargée  
qui est utilisée.  
Cette technique a un deuxième intérêt : les  
bibliothèques sont séparées des programmes,  
c'est-à-dire que l'on peut mettre à jour la biblio-  
thèque sans modifier les programmes qui l'utili-  
sent.

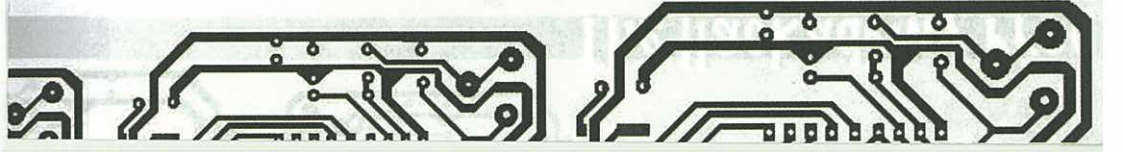
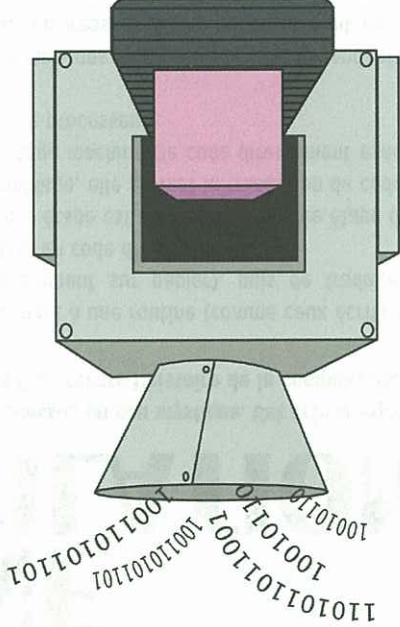
## La compilation du langage C

La compilation du langage C comprend toutes  
les étapes décrites précédemment. Nous allons  
voir dans cette partie quels programmes sont  
utilisés par chacune des étapes et comment  
récupérer les fichiers intermédiaires.  
Ces manipulations étant spécifiques aux outils  
de développement, nous choisissons de présen-  
ter le compilateur GCC (GNU Compiler  
Collection) qui est l'un des compilateurs les plus  
utilisés. Il est présent par défaut sur tous les  
UNIX, Linux, MacOSX. Il peut être installé sur  
Windows en téléchargeant l'environnement de  
développement Dev-C++ ([http://www.bloods-](http://www.bloods-<br/>hed.net/dev/devcpp.html)

à un microprocesseur. On utilise ce que l'on  
appelle aujourd'hui un compilateur pour créer  
des programmes en langage d'assemblage à  
partir du code source dans le langage de haut  
niveau.

Ces langages ont en général des syntaxes plus  
proches des mathématiques et du langage natu-  
rel, de plus, ils laissent au compilateur le soin de  
faire attention aux détails de chaque micropro-  
cesseur ce qui réduit considérablement le nom-  
bre d'erreurs de programmation.

Certains langages, comme le C, ont encore un  
macro-processeur, on appelle ça un préproces-  
seur.  
Lorsque les programmes augmentent en com-  
plexité, il faut pouvoir écrire des morceaux de  
programmes et les réutiliser lorsqu'on en a  
besoin.  
La compilation prenant du temps, beaucoup de





avec des morceaux de programmes.  
Les briques de base de ces bibliothèques sont les fichiers objets. Un fichier objet est la version compilée des fonctions qui se trouvaient dans le fichier source, c'est à dire une sorte de mini-bibliothèque.

```
$ gcc -c fichiers.c
```

il est possible de passer directement le fichier C à gcc :

```
$ gcc -c fichier.c
```

Le programme appelé par derrière est "as", l'assembleur.  
Ceci produit le fichier "fichier.o" contenant le code machine du programme compilé. A partir du fichier objet, on peut créer une bibliothèque en lancer la commande :

```
$ ar cr libfichiers.a fichier1.o \
fichier2.o fichier3.o ...
$ ranlib libfichiers.a
```

Ceci crée le fichier libfichiers.a qui est un bibliothèque.

A partir du fichier objet, on peut aussi créer un exécutable, via la commande :

```
$ gcc fichier1.o fichier2.o \
-o myexec
```

Ici, on crée le fichier "myexec" qui est exécutable. Si l'un des fichiers objet utilise une fonction d'une bibliothèque, il faut l'indiquer sur la ligne de commande avec l'option "<nom-lib>" et "l<répertoire-lib>", exemple :

```
$ gcc fichier1.o fichier2.o \
-fichier -L. -o myexec
```

GCC en réalité ne fait rien tout seul. Il appelle des programmes qui font le travail à sa place. Chaque étape de la compilation est prise en charge par un programme spécial. On peut demander à GCC de limiter son action : seulement l'un de ces programme doit être exécuté en passant une option spéciale.

La première étape est de passer le préprocesseur sur les fichiers sources, de transformer les .c (fichiers source en langage C) en .i (fichiers source en langage C qui n'ont pas besoin d'être passés au préprocesseur). Ceci se fait avec la commande :

```
$ gcc -E fichier.c -o fichier.i
```

Le programme appelé par derrière est "cpp", le préprocesseur.

Après cette étape, chaque fichier .c contient tout le contenu des fichiers d'en-tête (include files), toutes les macros sont étendues, le code n'est plus que du C pur, ce qui permet au compilateur de le compiler.

L'étape suivante est la génération du code assembleur à partir du source C. Pour faire ceci, entrez la commande :

```
$ gcc -S fichier.c
```

Le programme appelé par derrière est "cc1", le compilateur.

Ceci produit le fichier "fichier.s" contenant le code assembleur pour votre micro-processeur.

La prochaine étape est un peu complexe. Il faut générer un fichier objet à partir du fichier assembleur. Comme expliqué dans la première partie, on veut pouvoir créer des bibliothèques

## Un petit résumé

Le langage C est un langage compilé. Le fichier source ne peut être directement exécuté par le microprocesseur. Il doit subir des modifications d'un programme annexe (le compilateur) afin d'être traduit dans un langage de plus bas niveau directement compréhensible par le microprocesseur, ce qui permettra à notre fichier final de s'exécuter sur l'ordinateur de manière autonome.

Notre compilateur va donc devoir lire puis transformer notre fichier source. La compilation d'un programme en C se décompose en quatre parties.

La première étape est la lecture du fichier source par le préprocesseur. Celui-ci va donc se charger, après lecture du code, d'inclure les headers (`#include`) dans le fichier source. La seconde étape est la compilation. Le fichier source précédent est transformé en assembleur, une suite d'instructions associées à des fonctions du processeur. Puis se produit l'assemblage, notre fichier subit encore une modification. Le code assembleur est alors converti en langage machine dans un fichier binaire. Ce fichier est aussi appelé fichier objet. Vous pensiez que c'était terminé et pourtant non, le compilateur appelle alors le linker qui va se charger d'intégrer tous les éléments extérieurs comme les bibliothèques et les fonctions pour créer notre dernier fichier exécutable, et ce de façon autonome.

Ceci assemble le fichier exécutable "myexec" à partir des fichiers objets "fichier1.o" et "fichier2.o", dont l'un utilise la bibliothèque "libfichier.a" contenue dans le répertoire "libfichier.a" (répertoire courant). Notez que si le fichier contenant la bibliothèque s'appelle libfichier.a, il faut passer -lfichier au compilateur et non pas -libfichier.a.

Lorsque les étapes intermédiaires ne vous intéressent pas, vous pouvez très bien passer directement les fichiers C en argument à gcc :

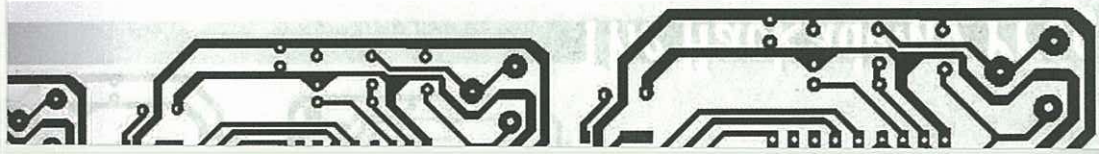
```
$ gcc fichier1.c fichier2.c -o myexec -lfichier
```

## Philippe Amanceco

La compilation de programmes C est donc un processus long et complexe, même si ça n'en a pas l'air lorsqu'on ne se préoccupe pas des étapes et fichiers intermédiaires.

Certaines étapes intermédiaires sont cependant importantes à retenir, en particulier lorsque l'on doit gérer les fichiers source de plus gros projets.

Pour ceux qui ne lancent pas leurs commandes de compilation dans un shell (je pense aux utilisateurs de Windows ou autre environnement de développement graphique), toutes les options présentées restent disponibles à travers les menus de configuration.



# RÈGLES DE PROGRAMMATION EN C

Programmer en C, c'est bien, programmer proprement en C, c'est mieux. Les règles suivantes vous aideront à créer un code lisible et compréhensible par tous, et à éviter quelques erreurs bêtes.

## Instructions et commentaires

Les instructions composent l'essentiel du programme. Il est donc important qu'elles soient lisibles. Aussi, la règle est : une seule instruction par ligne (si possible, suivi d'un commentaire explicite sur son rôle).

Les commentaires sont très importants : ils permettent aux programmeurs (auteurs du code ou non) de ne pas avoir à se casser la tête pour savoir ce qu'un bout de code fait. Que vous travailliez en équipe ou non, commentez toujours votre code, aussi court soit-il.

Il existe deux sortes de commentaire en C :

- les commentaires multi-lignes, qui sont placés entre /\* et \*/
- les commentaires sur une ligne, qui sont placés entre // et la fin de la ligne.

Le mieux consiste à placer les commentaires généraux portant sur un ensemble d'instructions sur des lignes isolées, avant l'instruction concernée, tandis que les commentaires précisant un détail seront placés à droite de l'instruction en question.

## Exemple

```
/* fonction isZero :
 * retourne 1 si l'entier passe en
 * argument est nul, 0 sinon
 */
int isZero(int n)
{
    if (n==0) //si l'entier est nul
        return 1;
}
```

Autre chose, évitez les goto. Ils rendent en effet plus difficile la compréhension du code par d'autres personnes, qui doivent à chaque fois retrouver l'étiquette correspondante et la logique sous-jacente. Remplacez-les par des boucles, plus claires et plus faciles à lire.

Enfin, évitez également le « C condensé » : remplacer cinq lignes par une ne sert à rien d'autre qu'à compliquer la relecture, y compris la vôtre. Les deux codes suivants sont équivalents, mais le second est nettement plus compréhensible :

```
//en clair :
if (a==4)
{
    i++;
    x = i;
}
else {
    x = tab[j];
    j++;
}
x = x * c;
c--;
//en clair :
x=((a==4)?++i:tab[j++])*c--;
if (a==4)
    i++;
else {
    x = tab[j];
    j++;
}
x = x * c;
c--;
if (a==4)
    i++;
else {
    x = tab[j];
    j++;
}
x = x * c;
c--;
if (a==4)
    i++;
else {
    x = tab[j];
    j++;
}
x = x * c;
c--;
```

Si votre code est correctement indenté, il sera beaucoup plus facile à relire, pour vous comme pour les autres.

**Remarque :** il est bien question d'espaces, et non de tabulations. En effet, tous les éditeurs n'affichant pas les tabulations de la même façon (plus ou moins grandes), il est déconseillé de les utiliser. La plupart des éditeurs proposent d'insérer un nombre défini d'espaces à la place des tabulations. De même, utilisez toujours une police à pas fixe (tous les caractères ont la même taille), par exemple Courier New.

## Accolades

Les accolades marquent le début et la fin d'un bloc d'instructions. Il est conseillé de les placer seules sur une ligne, et de les indenter comme ce qui les précède. Ainsi, vous risquez moins de vous tromper dans le nombre d'accolades et retrouverez rapidement une accolade manquante.

### Exemple

```
if (test)
{
    instruction1;
    instruction2;
}
if (test) {
    précède :
    instruction1;
    instruction2;
}
Certains personnes préfèrent mettre l'accolade ouvrante sur la même ligne que ce qui la précède :
```

Il vaut mieux n'utiliser cette forme que dans les courts exemples où aux endroits où la place est

## Indentation

L'indentation sert à rendre le code plus lisible par l'insertion d'espaces avant les instructions. Chaque instruction d'un bloc d'instructions (boucle, fonction, définition) doit être précédée par le même nombre d'espaces. Lorsqu'on a des blocs imbriqués (les uns dans les autres), on augmente de manière régulière le nombre d'espaces (par exemple deux espaces à chaque fois).

```
else
    x = tab[j++];
    x = x * c--;
```

```
Exemple
//definition
struct mastructure
{
    int var1;
    char var3[10];
}
//fonction
int test_fonction (void)
```

```
instruction1;
instruction2;
instruction3;
//bloc if dans la fonction
if (test)
{
    instruction4;
    instruction5;
}
```

Lorsqu'un bloc d'instructions n'en contient qu'une seule, elle est quand même indentée :

```
if(a==4)
    printf("a = 4 i");
```

