Développement en C sous Linux

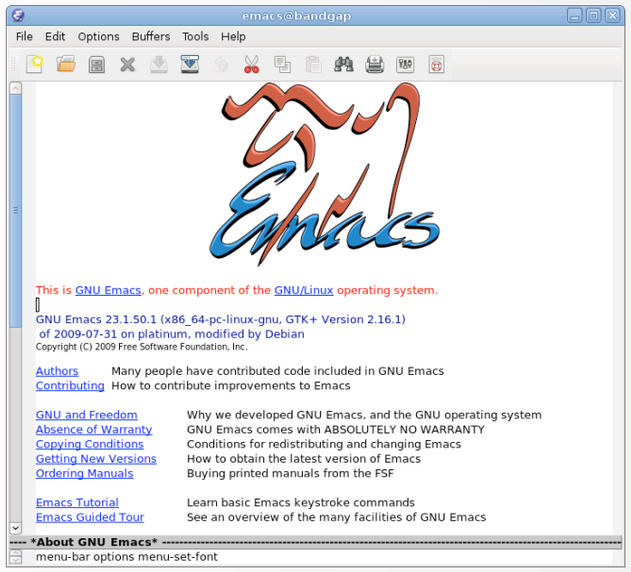
Naïm Qachri, Stéphane Fernandes Medeiros, Eythan Levy, Cédric Meuter

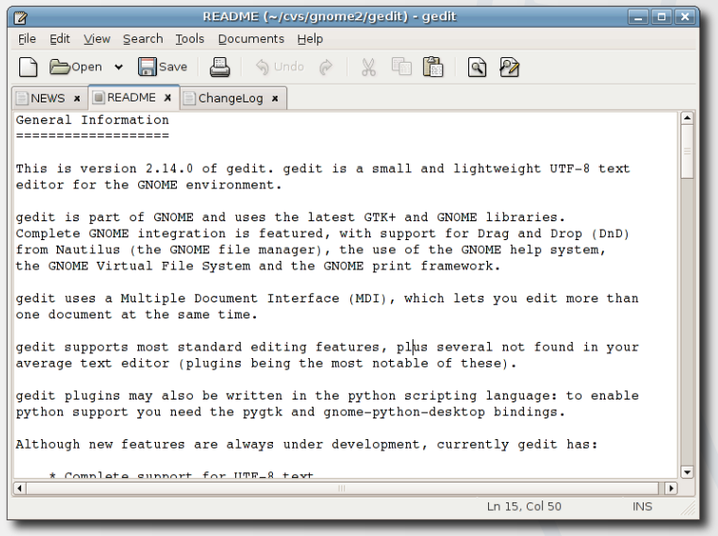
Université Libre de Bruxelles

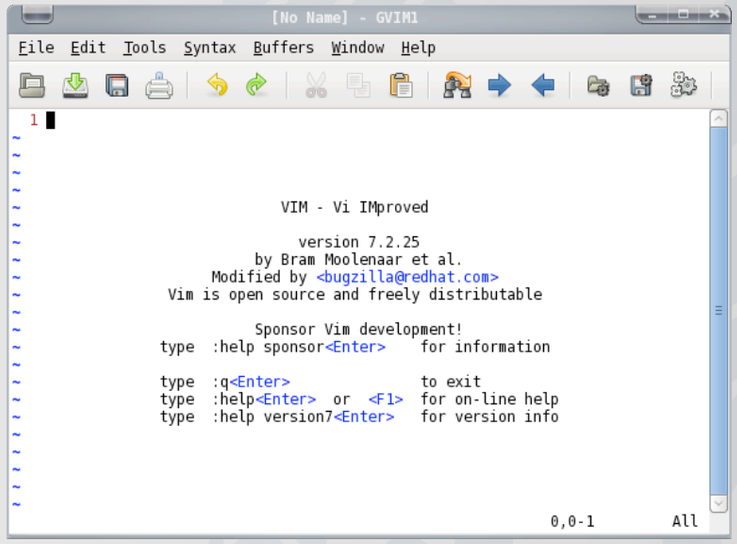
2010/2011

Comment écrire son programme

- Principaux éditeurs:







Emacs ou Xemacs **Gedit** **Vi, Vim ou Gvim**

- Il en existe d'autres: kate, nedit...
- choisissez celui qui vous convient le mieux

Comment compiler son programme

- GCC = **G**NU **C** **C**ompiler
- Compilateur C/C++ pour linux
- Interface **en ligne de commande**
- Exemple de compilation :

```c
/* helloworld.c */

int main(int argc, char **argv) {
   printf("hello world!\n");
   return 0;
}
```

```
[cmeuter@litpc34 Archi]$ gcc helloworld.c
[cmeuter@litpc34 Archi]$ ./a.out
hello world!
```

Quelques options bien utiles

- *-o filename* : permet de changer le nom du fichier de sortie.

```
[cmeuter@litpc34 Archi]$ gcc helloworld.c -o hello
[cmeuter@litpc34 Archi]$ ./hello
hello world!
```

- *-Wall* : le compilateur effectue plus de vérification à la compilation et génère des *warnings* en cas de problème.

```
[cmeuter@litpc34 Archi]$ gcc  helloworld.c -Wall -o hello
helloworld.c: In function ‘main’:
helloworld.c:4: warning: implicit declaration of function ‘printf’
[cmeuter@litpc34 Archi]$ ./hello
hello world!
```

Directives de compilations (I)

- *`#include <filename>`* : directive de compilation qui permet d'inclure le contenu d'un fichier. **GCC** va chercher le fichier `filename` dans les répertoires de **librairie standard**

```c
/* helloworld.c */
#include <stdio.h> /* stdio.h contient la déclaration de printf */

int main(int argc, char **argv) {
   printf("hello world!\n");
   return 0;
}
```

```
[cmeuter@litpc34 Archi]$ gcc  helloworld.c -Wall  -o  hello
[cmeuter@litpc34 Archi]$ ./hello
hello world!
```

Directives de compilations (II)

- `#include "filename"` : directive qui permet d'inclure le contenu d'un fichier. **GCC** va chercher le fichier `filename` dans **le répertoire courant**

```
/* helloworld.h */
const char *hw = ‘‘hello world!’’;
```

```
/* helloworld.c */
#include <stdio.h> /* stdio.h contient la déclaration de printf */
#include ‘‘helloworld.h’’

int main(int argc, char **argv) {
printf("%s\n", hw);
return 0;
}
```

```
[cmeuter@litpc34 Archi]$ gcc helloworld.c -Wall -o hello
[cmeuter@litpc34 Archi]$ ./hello
hello world!
```

Directives de compilations (II)

- Un module est un ensemble de fonctions, types, variables qui se concentre sur une partie du programme (*ex: gestion d'un stack, d'un arbre,...*). Un module se compose généralement de deux fichiers : un fichier .h et un fichier .c

- Le fichier ***.h*** contient l'interface du module (ce qui sera visible de l'extérieur du module) :
  - Les prototypes des fonctions
  - Les types, variables et constantes globales

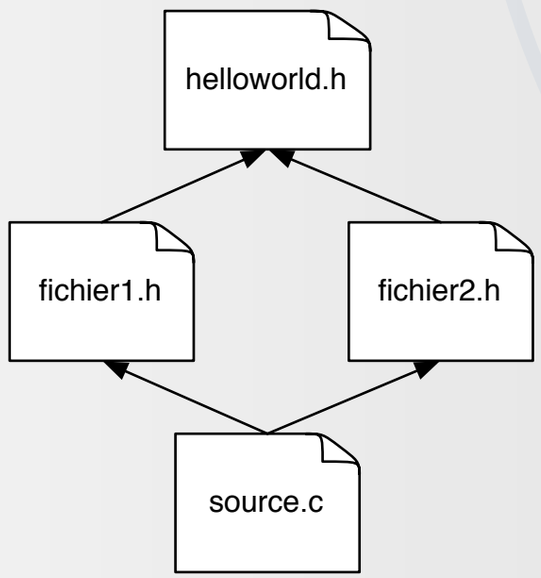
- Le fichier ***.c*** contient l'implantation du module (cette implantation n'est pas visible de l'extérieur) :
  - L'implantation des fonctions
  - Les types, variables et constantes locales au module

Module *helloworld*

```c
/* helloworld.h */
const char *hw = "hello world!";
void print_hello_world();

/* helloworld.c */
#include "helloworld.h"
#include <stdio.h>

void print_hello_world() {
  printf("%s\n", hw);
}

/* main.c */
#include "helloworld.h"

int main(int argc, char **argv) {
  print_hello_world();
  return 0;
}
```

Problème d'inclusions multiples

- Supposons que dans un fichier `source.c`, se trouve inclus deux fichiers : `fichier1.h` et `fichier2.h`, et que par ailleurs, ces fichiers aient besoin de faire appel à notre module `helloworld`, définit dans `helloworld.h`.
- Ce fichier sera inclus ***deux fois*** dans `source.c` (une fois via `fichier1.h` et une fois via `fichier2.h`). Cela peut causer des ***problèmes à la compilation***.



D'autres directives à la rescousse!

- Il faut donc protéger les fichiers `.h` contre les inclusions multiples. Ceci est réalisé en utilisant les directives de compilation `#define` et `#ifndef` ... `#endif`.

- `#define` permet de définir un symbole qui sera remplacé à la compilation

```
#define PI 3.141592654
```

- `#ifndef` ... `#endif` permet d'inclure du code selon qu'un symbole a été défini précédemment ou pas

```
#ifndef SYMBOLE

/* Code à inclure (include) si SYMBOLE
n'est pas défini */

#endif
```

Protection contre l'inclusion multiple

- Lors de la première inclusion, on définit un symbole unique pour le fichier `.h`.

  À chaque inclusion, on teste l'existence du symbole.

- Le module *`helloworld`* revisité :

```
/* helloworld.h */
#ifndef __HELLOWORLD_H
#define __HELLOWORLD_H

const char *hw = "hello world!";

void print_hello_world();

#endif
```

Compilation séparée (I)

- Pour compiler un programme composé de plusieurs modules, on peut passer tout les fichiers `.c` en options à ***gcc***

```
[cmeuter@litpc34 Archi]$ gcc -Wall  helloworld.c  main.c  -o  hello
[cmeuter@litpc34 Archi]$ ./hello
hello world!
```

- En utilisant cette méthode, si on modifie un module, on doit **tout recompiler !**
- Heureusement, **gcc** offre une alternative : la compilation ***séparée*** avec l'option *-c*
- Chaque fichier `.c` est compilé en un fichier `.o`

```
[cmeuter@litpc34 Archi]$ gcc -Wall -c helloworld.c
[cmeuter@litpc34 Archi]$ gcc -Wall -c main.c
```

Compilation séparée (II)

- Pour créer l'exécutable, il ne reste plus qu'à faire l'édition des liens (*linkage*)

```
[cmeuter@litpc34 Archi]$ gcc helloworld.o main.o -o hello
[cmeuter@litpc34 Archi]$ ./hello
hello world!
```

- De cette manière, lorsqu'un module est modifié, il suffit de recompiler le module modifié uniquement, et ensuite d'effectuer l'édition des liens

Compilation séparée (II)

- Le problème de la compilation séparée, c'est qu'il faut recompiler ***à la main*** chaque fichier modifié : ça peut vite devenir long et fastidieux !

- On peut **automatiser** le processus en utilisant l'outil `make`

- Pour cela, il faut décrire dans un fichier appelé `Makefile` ou `makefile`, les règles de compilation.

- Chaque règle est de la forme :

```
<fichier voulu> : <fichiers dont il dépend>
                        <commande pour créer le fichier>
```

Compilation séparée (II)

- Reprenons l'exemple du module *helloworld*:

```
# Makefile pour le module helloworld
# (le charactère '#' dénotes un commentaire)

hello: main.o helloworld.o
	gcc main.o helloworld.o -o hello

helloworld.o: helloworld.c helloworld.h
	gcc -Wall -c helloworld.c

main.o: main.c helloworld.h
	gcc -Wall -c main.c
```

La gestion des erreurs en C (I)

- Les appels systèmes, ainsi que les fonctions des librairies standards C sont susceptibles de ne pas fonctionner comme attendu :
  - ouverture (`fopen()` ou `open()`) d'un fichier alors que l'utilisateur n'en a pas les permissions
  - allocation (`malloc`) de mémoire alors qu'il n'y a plus assez de mémoire disponible
  - ...
- Le C offre toute une série de **fonctions/variables** permettant une gestion d'erreur propre :
  - `errno`
  - `strerror`
  - `perror`
  - `assert`

La gestion des erreurs en C (II)

- ```
  #include <errno.h>
  extern int errno;
  ```

  La variable entière `errno` est **mise à jour par les appels systèmes** et par certaines fonctions des librairies standard C. En cas d'erreur, elle contient un code d'erreur **détaillant** ce qui s'est mal passé

- ```
  #include <string.h>
  char *strerror(int errnum);
  ```

  La fonction `strerror()` permet d'obtenir à partir d'un code d'erreur (venant de `errno`) **un message plus compréhensible**

La gestion des erreurs en C (III)

- ```
  #include <stdio.h>
  void perror(const char *msg);
  ```

  La fonction `perror` permet d'afficher u**n message d'erreur (`msg`) sur le standard error** suivit du détail de la dernière erreur rencontrée durant un appel système ou un appel à une fonction des librairies standards.

- ```
  #include <assert.h>
  void assert(int expression);
  ```

  La macro `assert()` termine l'exécution du programme si `expression` est évalué à faux (0). Ces gardes peuvent être désactivées en définissant le symbole `NDEBUG` avant d'inclure `assert.h` (en utilisant `#define`)

exemple *perror.c*

```c
/* perror example */
#include <stdio.h>

int main ()
{
  FILE * pFile;
  pFile=fopen ("unexist.ent","rb");
  if (pFile==NULL)
    perror ("The following error occurred");
  else
    fclose (pFile);
  return 0;
}

/*Affiche:

The following error occurred: No such file or directory*/
```

exemple *sterror.c*

```c
/* strerror example */
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main ()
{
  FILE * pFile;
  pFile = fopen ("unexist.ent","r");
  if (pFile == NULL)
    printf ("Error opening file unexist.ent: %s\n",strerror(errno));
  return 0;
}

/*Affiche:

Error opening file unexist.ent: No such file or directory*/
```

Le *tracing*

- Le *tracing* consiste à ajouter des `printf` dans le code pour afficher des messages de débogage (valeur d'une variable, résultat de fonction, ...). Cela permet de savoir pendant l'exécution où on se trouve dans le code.

```c
int main(int argc, char **argv) {
    int x, y;
    /* du code jouant avec x et y */
    if (x < y) {
        printf("main: (x < y)\n");
        fflush(stdout);
    }
    else {
        printf("main: (y >= x)\n");
        fflush(stdout);
    }

    return 0;
}
```

Le *tracing* conditionnel

- Malheureusement, les `printf` peuvent vite devenir nombreux et gênant, surtout lorsqu'on a fini de déboguer le programme
- On peut résoudre ce problème en utilisant les directives de compilation `#ifdef ... #endif` et `#define`

```
#ifdef __DEBUG
  printf("Mon message de debug!\n");
  fflush(stdout);

#endif
```

- Pour activer les messages de debugage, il suffit de définir la constante voulue (`__DEBUG` dans l'exemple) à l'aide de la directive `#define`
- On peut également utiliser un symbole de debug différent pour chaque module.

Débogueur symbolique

- Un autre façon de déboguer un programme est d'utiliser un débogueur symbolique. Ce type de débogueur permet :
  - d'exécuter un programme pas à pas
  - d'afficher les valeurs intermédiaires des variables
  - de placer des *breakpoint's*
  - d'examiner la pile (ou *stack*) des appels de fonction
  - ...
- D'une manière plus générale, il permet de comprendre et surtout de localiser l'erreur (*segmentation fault*) !

GDB (I)

- GDB = **G**NU **D**e**b**ugger
- Debogueur symbolique pour linux
- Interface interactive, en mode texte
- Avant de l'utiliser, il est nécessaire de compiler le programme avec l'option -g, afin de rajouter des informations symboliques dans l'exécutable (noms des fonctions, variables, ...)

```
[cmeuter@litpc34 Archi]$ gcc  -Wall  -g  bug.c  -o  bug
```

GDB (II)

- Ensuite on peut lancer GDB en passant l'exécutable en paramètre :

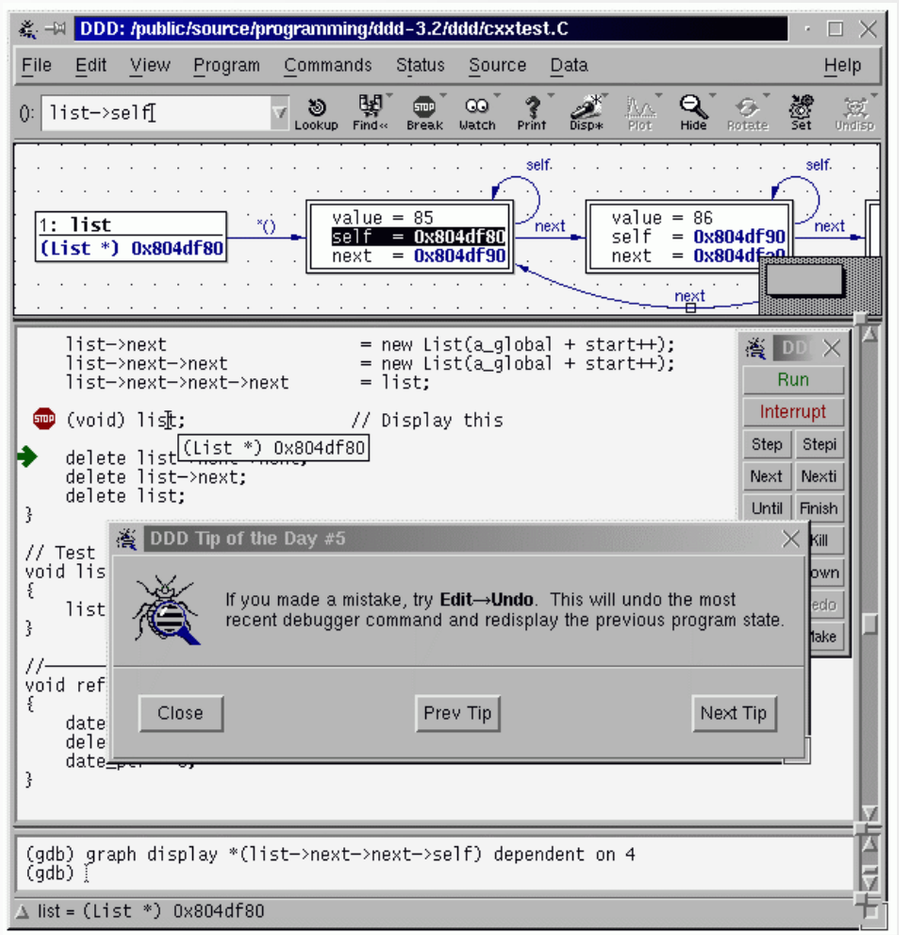
```
[cmeuter@litpc34 error]$ gdb bug
GNU gdb 6.0-2mdk (Mandrake Linux)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under
certain conditions. Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty"
for details.
This GDB was configured as "i586-mandrake-linux-gnu"...
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb)
```

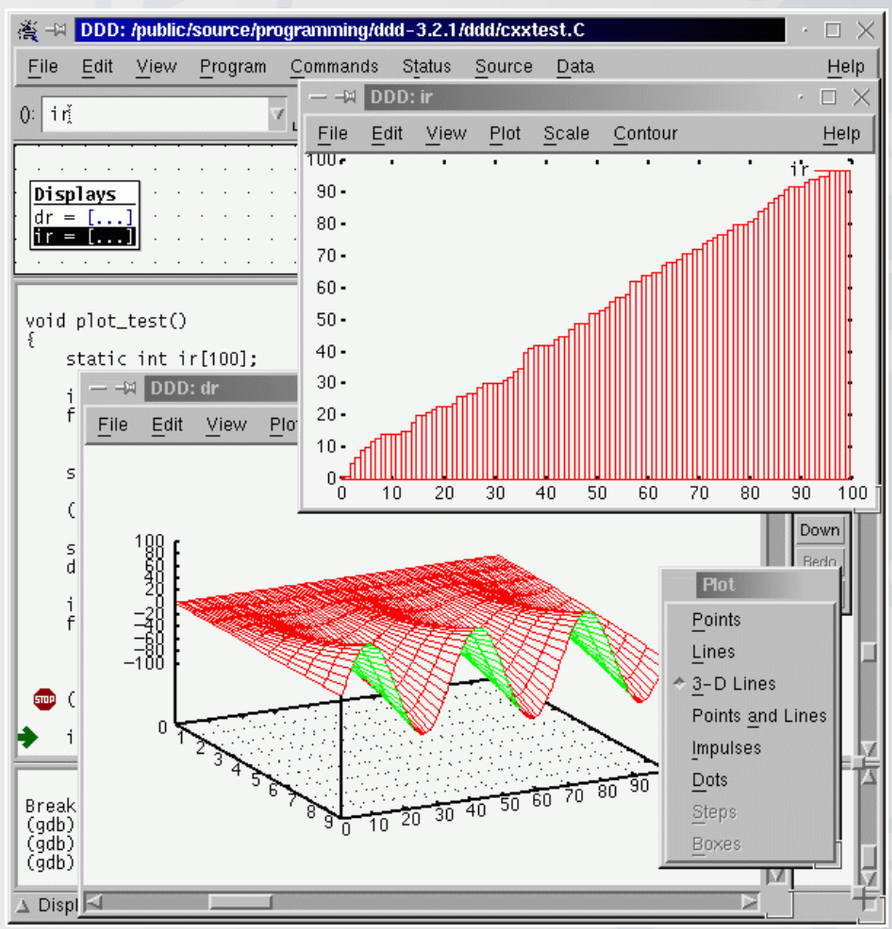
GDB (III)

- Les commandes les plus utilisées de **GDB** :
  - *`l`* : affiche **la ligne** de code **courante**, ainsi que **les 5 lignes suivantes et précédentes**.
  - *`l num`* : affiche la ligne de code *`num`*.
  - *`b num`* : place un *breakpoint* à la ligne de code *`num`*.
  - *`n`* : exécute l'instruction suivante **sans entrer** dans les appels de fonctions.
  - *`s`* : exécute l'instruction suivante **et entre** dans les appels de fonctions.
  - *`p expression`* : affiche **le résultat de l'évaluation** de *`expression`*.
  - *`r args`* : exécute le programme chargé **avec** les arguments *`args`*
  - *`bt`* : affiche le *stack* d'appel de fonction.

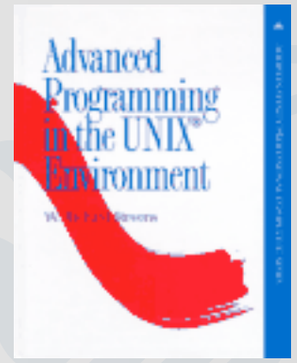
Interface à GDB

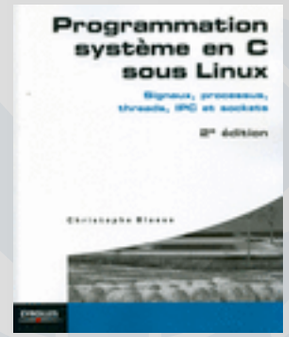
- Il existe des interfaces graphiques à GDB.
- On peut citer DDD = **D**ata **D**isplay **D**ebugger





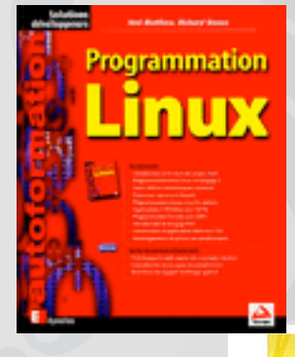
Quelques références

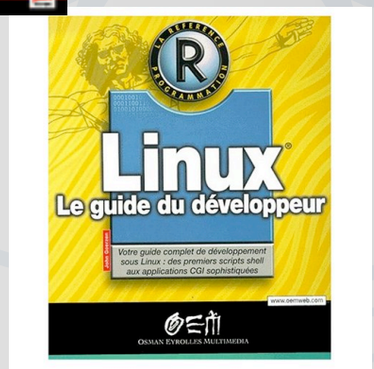




- *Advanced programming in the UNIX environment*, par Richard Stevens, Addison-Wesley, ISBN 0-201-56317-7.

- *Programmation Système en C sous Linux*, par Christophe Blaess, Eyrolles, ISBN 2-212-11054-5



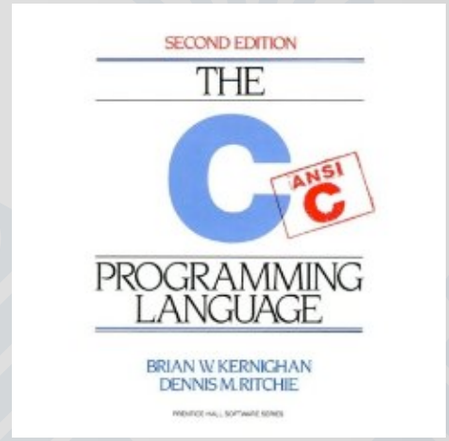


- *Programmation Linux*, par Neil Matthew et Richard Stones, Eyrolles, ISBN 1-861002-97-1

- *Linux : Le guide du développeur* par John Goerzen, Eyrolles, ISBN 2-7464-0217-3

Quelques références

- *C Programming Language (2nd Edition),* par Brian W. Kernighan et Dennis M. Ritchie , Prentice Hall. ISBN 0-13-110163-3.



- Managing Projects with GNU Make, Third Edition, par Robert Mecklenburg

- Version Control with Subversion, par C. Pilato, Ben Collins-Sussman, Brian Fitzpatrick

Ces deux derniers sont gratuits sur:

http://oreilly.com/openbook/

À vous de jouer!

1. Ecrire un module (*fichiers .h et .c*) de gestion d'une liste dynamique d'entiers, simplement liée, comprenant :
   (a) une fonction d'initialisation
   (b) une fonction d'insertion en début de liste
   (c) une fonction d'insertion en fin de liste
   (d) une fonction pour effacer une liste
2. Ecrire une programme de *démonstration* qui utilise les fonctionnalités du modules
3. Ecrire un ***Makefile*** qui permet de compiler le module et le programme *démo* en utilisant la compilation séparée