FONCTIONS GENERALES

Licence de ce document

Permission est accordée de copier, distribuer et/ou modifier ce document selon les termes de la "Licence de Documentation Libre GNU" (GNU Free Documentation License), version 1.1 ou toute version ultérieure publiée par la Free Software Foundation.
En particulier le verbe "modifier" autorise tout lecteur éventuel à apporter au document ses propres contributions ou à corriger d'éventuelles erreurs et lui permet d'ajouter son propre copyright dans la page "Registre des éditions" mais lui interdit d'enlever les copyrights déjà présents et lui interdit de modifier les termes de cette licence.
Ce document ne peut être cédé, déposé ou distribué d'une autre manière que l'autorise la "Licence de Documentation Libre GNU" et toute tentative de ce type annule automatiquement les droits d'utiliser ce document sous cette licence. Toutefois, des tiers ayant reçu des copies de ce document ont le droit d'utiliser ces copies et continueront à bénéficier de ce droit tant qu'ils respecteront pleinement les conditions de la licence.

La version papier de ce document est la *1.0* et sa version la plus récente est disponible en téléchargement à l'adresse http://www.frederic-lang.fr.fm

Copyright © 2004 Frédéric Lang (frederic-lang@fr.fm)

Registre des éditions

Version	Date	Description des modifications	Auteur des modifications
1.0	20 février 2005	Mise en ligne du document	© Frédéric Lang (frederic-lang@fr.fm)

SOMMAIRE

I) PRESENTATION **5**

II) LE COMPILATEUR "CC" **6**
1) ORGANISATION D'UN PROGRAMME 6
2) PHASE "PRE-COMPILATION" 7
3) PHASES DE COMPILATION ET D'ASSEMBLAGE 7
4) EDITION DE LIENS 8

III) EXEMPLE D'UNE DECOMPOSITION DE PHASE EN MINI-PROJET **9**
1) UN SEUL SOURCE 9
2) PLUSIEURS SOURCES 10
3) MISE EN LIBRAIRIE 11
4) MISE A DISPOSITION DE LA LIBRAIRIE 11

IV) ACCES A L'ENVIRONNEMENT **12**
1) INTRODUCTION 12
2) LES PARAMETRES DE "MAIN()" 12
3) VALEUR DES VARIABLES D'ENVIRONNEMENT 13

V) LANCEMENT D'UNE COMMANDE SHELL **14**

VI) GESTION DES ERREURS **15**
1) LES VARIABLES 15
2) FONCTION "STRERROR()" 16
3) FONCTION "PERROR()" 16
4) PRINCIPALES VALEURS DE "ERRNO" 17

INDEX **18**

I) PRESENTATION

UNIX s'est imposé comme système d'exploitation dans le domaine des stations de travail.

Le succès d'UNIX a entraîné celui de TCP/IP en tant que protocole d'échange sur les réseaux. Malgré l'arrivée progressive de protocoles normalisés OSI, TCP/IP reste très utilisé, y compris par les systèmes d'exploitation non-UNIX.

UNIX présente la caractéristique fondamentale d'être écrit en langage C. De ce fait, contrairement à d'autres systèmes comme le DOS, il est accessible par des fonctions que l'on peut appeler directement à partir de routines du langage C.

Il n'est donc pas nécessaire, d'utiliser des fonctions systèmes spécifiques servant en quelque sorte d'interface entre le langage et l'assembleur (fonctions "int86()", "intdos()", etc. d'accès aux registres des processeurs Intel). Ici l'aspect " matériel" est totalement pris en charge par le système et le programmeur ne peut qu'utiliser les points d'entrée (appels systèmes ou fonctions regroupées dans des bibliothèques) proposés par UNIX. Néanmoins, cet avantage est restreint par la multitude des fonctions implémentées ; et il est bien difficile d'en avoir une vision globale.

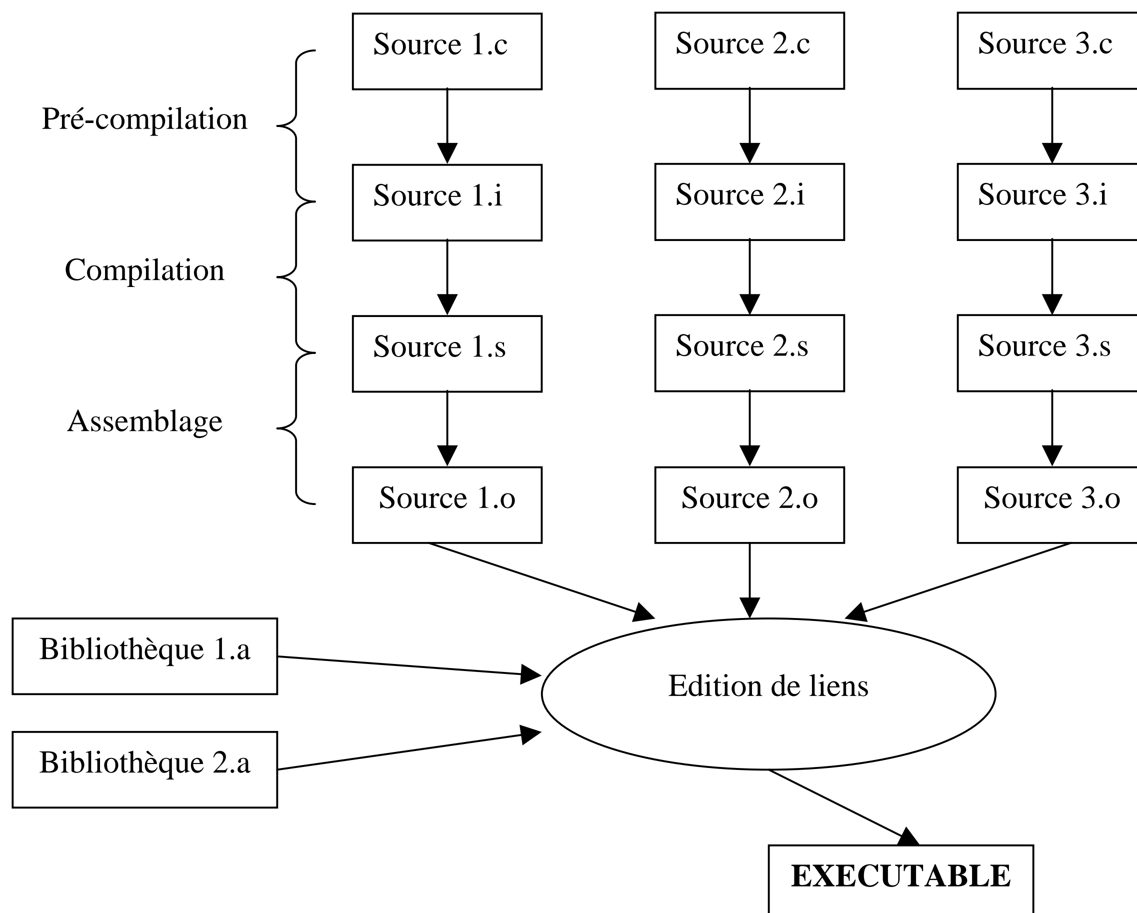
Il est bien évident que l'apprentissage du C est un préalable à l'étude de ce cours.

Eventuellement les sujets suivants seront revus :
- ✓ Utilisation des pointeurs
- ✓ Compréhension et utilisation des structures
- ✓ Appels de fonctions

II) LE COMPILATEUR "CC"

1) Organisation d'un programme

Un programme peut se composer d'un ou de plusieurs sources. Mais dans tous les cas, le cheminement de la compilation est toujours le même.



2) Phase "pré-compilation"

Le but de cette phase est de traiter les macro définitions et les inclusions de fichiers. La pré-compilation traite les lignes commençant par un "#" (#define, #include, #ifdef, etc.), son action est primordiale pour le développement d'applications paramétrées, modulaires et portables.

Options de "cc" pour la pré-compilation :

- ✓ -D *identificateur* : insère un ligne "#define identificateur" en début de source
- ✓ -D *identificateur=valeur* : insère un ligne "#define identificateur=valeur" en début de source
- ✓ -I *répertoire* : spécifie au compilateur d'utiliser aussi "répertoire" comme pouvant contenir des headers ".h". Par défaut, les fichiers inclus avec *#include <xxx.h>* (avec des caractères "<>") seront cherchés dans "/usr/include" ; éventuellement aussi dans "*répertoire*" si l'option "-I" est positionnée. Les fichiers inclus avec la directive *#include "xxx.h"* (avec des guillemets) seront cherchés par rapport au nom *Unix* donné.
- ✓ -P : S'arrête à la pré-compilation et génère un fichier ".i". Cette option n'est pas standard
- ✓ -E : S'arrête à la pré-compilation mais retranscrit le résultat à l'écran.

Constantes inclues dans le pré-processeur et utilisables :

- ✓ **__FILE__** (avec deux underscores) : Nom du source (valeur de type "chaine de caractère")
- ✓ **__LINE__** (avec deux underscores) : Numéro de la ligne courante (valeur de type "int")
- ✓ **__DATE__** (avec deux underscores) : Date de compilation (valeur de type "chaine de caractère")
- ✓ **__TIME__** (avec deux underscores) : Heure de compilation (valeur de type "chaine de caractère")
- ✓ **__ANSI_C__** (avec deux underscores) : Vaut à 1 si le compilateur est à la norme "ansi" (valeur de type "int")

3) Phases de compilation et d'assemblage

Ces deux phases ont pour objectif de produire un module objet. Un module objet est très proche d'un exécutable, mais il peut contenir des liens non résolus (tels que des appels à des fonctions définies dans d'autres sources). Le module objet contient donc des informations pour l'édition de liens.

Options courantes :

- ✓ -O : Optimiser le code généré
- ✓ -g : Insère des informations de déboguage (*cf.* debug, dbx, gdb, etc.)
- ✓ -ql : Insère des informations d'analyse post-mortem (*cf.* lprof)
- ✓ -c : Arrête la compilation après la phase d'assemblage

D'autres options permettent de :

- ✓ Forcer la conformité avec différentes normes (ANSI, POSIX, etc.)
- ✓ Générer du code spécifique à un type de processeur
- ✓ Jouer sur les optimisations (vitesse d'exécution, taille du code, etc.)

Ces options ne sont pas toujours présentes.

4) Edition de liens

L'édition de liens est la phase qui permet de produire un programme exécutable par le système à partir des différents modules qui le composent et des différentes bibliothèques utilisées. Cette phase doit résoudre les références croisées et les définitions externes existant dans les modules composant une application.
Il existe différents modes d'édition de liens :

✓ Statique : tous les liens sont résolus, l'exécutable se suffit à lui-même.

✓ Dynamique immédiat : les liens référençant les objets des bibliothèques ne sont résolus qu'au chargement de l'exécutable (juste avant l'exécution à proprement parler).

✓ Dynamique retardé : un lien référençant un objet d'une bibliothèque n'est résolu que lors de la première référence à cet objet.
La portée de cette phase est donc variable et dépendante du mode. Elle commence après l'assemblage et peut se terminer pendant l'exécution du programme.

Avantages / désavantages :

✓ Mode statique :
  - Gain en vitesse d'exécution (pas d'accès aux bibliothèques)
  - Problèmes de mise à jours des bibliothèques (nécessité de refaire une édition de liens)
  - Perte en espace disque (redondance d'information)

✓ Mode dynamique :
  - Accès disque pour charger les bibliothèques
  - Pas de redondance
  - Mise à jours des bibliothèques automatiquement prise en compte

Options courantes :

✓ -L *répertoire* : pour spécifier le répertoire d'une bibliothèque

✓ -l *nom* : pour réaliser une liaison statique avec libnom.a

✓ -ln : pour forcer une liaison statique avec toutes les bibliothèques

✓ -o *nom* : pour spécifier le nom de l'exécutable

D'autres options permettent aussi de générer des exécutables pour différents systèmes d'exploitation (DOS, OS/2, Xenix, etc.).

III) EXEMPLE D'UNE DECOMPOSITION DE PHASE EN MINI-PROJET

Ce projet a pour but de montrer pourquoi et comment on définit un prototype. Le programme a uniquement pour but d'utiliser deux fonctions "*carre()*" et "*cube()*"; chacune d'elle recevant un nombre et renvoyant son carré et/ou son cube.

1) Un seul source

Pour cette première phase, on décide qu'un seul source suffira. C'est ce que l'on appelle un projet "monolithique".

✓ Source "*prog.c*"

```
/* Fonction carré */
unsigned long carre(
	short nb)				/* Nombre à opérer */
{
	/* Renvoi valeur carré */
	return((long)nb * nb);
}

/* Fonction cube */
long cube(
	short nb)				/* Nombre à opérer */
{
	/* Renvoi valeur cube */
	return((long)nb * nb * nb);
}

/* Fonction principale */
main()
{
	/* Déclaration des variables */

	/* Programmation */
	printf("Carré de 2 = %lu\n", carre(2));
	printf(Cube de 3 = %ld\n", cube(3));
}
```

Compilation

✓ cc prog.c –o prog

2) Plusieurs sources

Dans cette seconde phase, on décide que c'est plus facile à maintenir avec plusieurs fichiers sources. Cela devient un projet"modulaire".

- ✓ Source "*carre.c*"

```
/* Fonction carré */
unsigned long carre(
	short nb)				/* Nombre à opérer */
{
	/* Renvoi valeur carré */
	return((long)nb * nb);
}
```

- ✓ Source "*cube.c*"

```
/* Fonction cube */
long cube(
	short nb)				/* Nombre à opérer */
{
	/* Renvoi valeur cube */
	return((long)nb * nb * nb);
}
```

- ✓ Source "*prog.c*"

```
/* Prototypes des fonctions afin que le compilateur les connaisse avant de les utiliser */
unsigned long carre(short);			/* Carré d'un nombre */
long cube(short);				/* Cube d'un nombre */

/* Fonction principale */
main()
{
	/* Programmation */
	printf("Carré de 2 = %lu\n", carre(2));
	printf(Cube de 3 = %ld\n", cube(3));
}
```

<u>Compilation</u>

- ✓ cc –c carre.c
- ✓ cc –c cube.c
- ✓ cc prog.c carre.o cube.o –o prog
- ✓ rm –f carre.o cube.o

3) Mise en librairie

Dans cette troisième phase, on décide que si on regroupe les fonctions annexes de famille identique dans une librairie, l'appel aux fonctions sera plus rapide. Cela est vrai car les librairies possèdent un index. Cela devient un projet de "librairie".

Source "carre.c", "cube.c" et "prog.c" identiques à ceux de la phase précédente

Compilation des modules

- ✓ cc –c carre.c
- ✓ cc –c cube.c

Création de la librairie

- ✓ ar rv math.a carre.o cube.o
- ✓ rm –f carre.o cube.o

Création de l'exécutable

- ✓ cc prog.c math.a –o prog

4) Mise à disposition de la librairie

Dans cette quatrième phase, on décide que cette librairie "*math.a*" pourrait être utile à d'autres. On la distribue donc par un vecteur quelconque (internet ou autre) tout en indiquant aux utilisateurs potentiels comment utiliser les fonctions de la librairie. Bien entendu, les sources "*carre.c*" et "*cube.c*" restent confidentiels sous copyright de l'auteur et seul sortent disponibles pour le public la librairie compilée ainsi que les prototypes des fonctions. Pour ces derniers éléments, on regroupes ces prototypes dans un *header* dont le nom est significatif.

- ✓ Header "*math.h*"

```
/* Bloc conditionnel au cas où "math.h" serait inclus par un autre header (idempotence) */
#ifndef _MATH_H_                                /* Si "_MATH_H_" non défini */
#define _MATH_H_                                /* Définition "_MATH_H_" */

/* Fonctions mathématiques */
unsigned long carre(short);                     /* Carré d'un nombre */
long cube(short);                               /* Cube d'un nombre */

#endif /* _MATH_H_*/                            /* Fin de bloc conditionnel "_MATH_H_" */
```

- ✓ Source "*prog.c*"

```
#include "math.h"                               /* Fonctions mathématiques */

/* Fonction principale */
main()
{
        /* Programmation */
        printf("Carré de 2 = %lu\n", carre(2));
        printf(Cube de 3 = %ld\n", cube(3));
}
```

Compilation

- ✓ cc prog.c math.a –o prog

IV) ACCES A L'ENVIRONNEMENT

1) Introduction

Il est possible d'indiquer à un programme des éléments en paramètres qu'il pourra recevoir et utiliser à sa convenance. Ces éléments peuvent être un fichier à imprimer, un disque à examiner, etc.
De plus, on a vu qu'il était possible de définir, grâce aux possibilités de l'interpréteur de commandes, des variables. Un certain nombre de ces variables sont exportées et deviennent donc accessibles par l'ensemble des processus descendants du processus où elles ont été déclarées (en général le shell de connexion).Tous ces éléments constituent l'environnement d'un programme.

2) Les paramètres de "main()"

main() est une fonction comme une autre. Elle reçoit donc des paramètres et renvoie une valeur.

```
int main (int argc, char *argv[], char *envp[]);
```

Explication des paramètres :

✓ int argc (vient de "argument count") : Nombre d'arguments passés au programme ; y compris le nom du programme lui-même. Il s'ensuit que cette variable ne peut jamais être à zéro.

✓ char *argv[](vient de "argument value") : Pointeur vers un tableau de pointeurs, chacun de ceux-ci pointant vers un tableau de caractères au format ***chaîne*** (tableau contenant un caractère de valeur zéro). Chaque chaîne contiendra le nom de l'argument passé au programme. De plus, ***argv[0]*** contiendra le nom du programme lui-même. Enfin, ce tableau contiendra un pointeur supplémentaire de valeur nulle. On remarquera que cette valeur nulle se trouve dans la variable ***argv[argc]***.

✓ char *envp[](vient de "environment program"). De même format que "argv", cette variable pointe vers un tableau de chaînes de caractères. Chaque chaîne contiendra la valeur de chaque variable d'environnement qui a été exportée par un des processus ascendants et donc accessible au programme. Cette chaîne est de la forme ***variable=valeur***. De même que pour "argv", un dernier pointeur de ce tableau a pour valeur "zéro".

Valeur renvoyée (int) : un entier utilisable par le processus appelant le programme (récupéré par ***$?*** en shell).

Exemple :

```
int main (int argc, char*argv[], char *envp[])
{
	printf("Je me nomme %s et on m'a envoyé %d paramètres\n", argv[0], argc – 1);
	return(0);
}
```

3) Valeur des variables d'environnement

La primitive ***getenv()*** permet d'obtenir la valeur courante d'une variable d'environnement particulière.

```
#include <stdlib.h>
char* getenv (char *string);
```

Explication des paramètres :

✓ char *string : Chaîne contenant la variable dont on veut obtenir la valeur

Valeur renvoyée (char*) : un pointeur sur une zone mémoire statique contenant la chaîne correspondant à la variable demandée si celle-ci existe

Exemple :

```
#include <stdlib.h>                          /* Standards librairies "C" */
main()
{
        char *pt;                            /* Ptr récupérant le résultat de "getenv()" */

        pt=getenv("PATH");
        printf("PATH = %s\n ", pt);
}
```

V) LANCEMENT D'UNE COMMANDE SHELL

Il est possible, depuis le programme C, de démarrer l'exécution d'une commande shell par un appel à la primitive ***system()***. Cette fonction lance un shell qui va exécuter la commande voulue. Le shell lancé n'a aucune communication avec le programme appelant. Celui-ci reste en attente jusqu'à la fin du shell lancé.
Remarque : Il est souvent plus judicieux de reprogrammer directement la commande que l'on veut faire exécuter. en effet, La rapidité en sera accrue et il pourra y avoir communication entre programme et commande.

```
#include <stdlib.h>
int system (char *string);
```

Explication des paramètres :
- ✓ char *string : Chaîne contenant la commande que l'on veut exécuter

Valeur renvoyée (int) : un nombre contenant le statut de fin du shell lancé

Exemple :

```
#include <stdlib.h>
main()
{
	system("date");
}
```

VI) Gestion des erreurs

1) Les variables

Pour un traitement correct des erreurs, il faut exploiter à fond toutes les possibilités que le système met à notre disposition. Par défaut, un appel système échoué renvoie ***0*** ou ***-1*** dans 99% des cas. Pour fournir un complément d'information au programmeur, la variable ***extern int errno*** est remplie d'une valeur indiquant la cause réelle de l'échec. Après l'insertion du fichier ***errno.h***, la variable ***errno*** est automatiquement déclarée.
De plus, un second objet est mis à disposition du programmeur. Il s'agit de la variable ***extern const char * const sys_errlist[]***. Cette variable est un pointeur non modifiable vers un tableau de chaînes de caractères non modifiables. Chaque chaîne contient un bref descriptif de l'erreur correspondant à son indice. Ainsi, pour chaque entier ***i***, la variable ***sys_errlist[i]*** contient le texte correspondant à l'erreur ***i***.
Cette variable n'est déclarée dans aucun fichier header dans le système Unix. Par contre, sous Linux, on la trouve dans le fichier "***stdio.h***". Le programmeur désirant l'utiliser peut donc soit inclure le fichier correspondant, soit la déclarer.
Une troisième variable peut aussi être exploitée mais son utilité reste limitée. Il s'agit de ***extern int sys_nerr***. Elle contient la valeur maximale que peut prendre ***errno***, c'est aussi le nombre total de messages d'erreurs disponibles dans ***sys_errlist[]***.

Exemple :

```
#include <errno.h>                                  /* Gestion des erreurs */
extern const char *const sys_errlist[];             /* Liste des messages d'erreur */
main()
{
        int var;
        if (primitive quelconque style "open" (…) == (-1))
                printf("Erreur %d – Message=%s\n", errno, sys_errlist[errno]);
}
```

La valeur "errno" n'a de signification qu'après un appel à une primitive ayant échoué. Elle n'est jamais "remise à zéro" de manière automatique ; mais rien n'interdit de la mettre manuellement à "zéro" si le besoin s'en fait sentir.

2) Fonction "strerror()"

La fonction ***strerror()*** renvoie le message d'erreur correspondant à ***errno***. Elle renvoie en fait un pointeur sur l'index de ***sys_errlist[]*** correspondant à ***errno***.

```
#include <errno.h>
char *strerror (int error);
```

Explication des paramètres :

- ✓ int error : Numéro d'erreur dont on veut récupérer le message correspondant

Valeur renvoyée (char *) : Pointeur sur l'index de ***sys_errlist[]*** contenant le message d'erreur correspondant au numéro d'erreur

Exemple :

```
#include <errno.h>                              /* Gestion des erreurs */
main()
{
        int var;
        if (primitive quelconque style "open" (…) == (-1))
                printf("Erreur %d – Message=%s\n", errno, strerror(errno));
}
```

3) Fonction "perror()"

La fonction ***perror()*** affiche sur la sortie des erreurs un texte choisi par le programmeur immédiatement suivi du message d'erreur correspondant à ***errno***. Ce n'est qu'un affichage habillé de ***sys_errlist[]***.

```
#include <errno.h>
void perror (const char *txt);
```

Explication des paramètres :

- ✓ char *txt : Chaîne contenant un message qui sera affiché en plus du message provenant de ***sys_errlist[]***

Exemple :

```
#include <errno.h>                              /* Gestion des erreurs */
main()
{
        int var;
        if (primitive quelconque style "open" (…) == (-1))
                perror("Erreur dans la fonction xxx");
}
```

4) Principales valeurs de "errno"

N°	*Constante prédéfinie*	*Signification*
1	EPERM	Problème de permission sur le fichier
2	ENOENT	Fichier ou répertoire inexistant
3	ESRCH	Processus inexistant
4	EINTR	Interruption de l'appel système
5	EIO	Erreur d'entrée / sortie
6	ENXIO	Ressource inexistante
7	E2BIG	Liste d'arguments trop longue
8	ENOEXEC	Erreur de format dans le fichier exécutable
9	EBADF	Numéro de fichier incorrect
10	ECHILD	Pas de processus fils (en cas d'exécution de ***wait()***)
11	EAGAIN	Table de processus pleine
12	ENOMEM	Zone de swap pleine
13	EACCES	Problème de protection sur un fichier
14	EFAULT	Mauvaise adresse
15	ENOTBLK	Problème de type de fichier
16	EBUSY	Tentative pour monter un système de fichiers déjà monté
17	EEXIST	Fichier déjà existant
19	ENODEV	Le device n'existe pas
20	ENOTDIR	Le fichier spécifié n'est pas un répertoire
21	EISDIR	Le fichier spécifié est un répertoire
22	EINVAL	Argument incorrect
23	ENFILE	Table des fichiers du système pleine
24	EMFILE	Trop de fichiers ouverts par le processus
25	ENOTTY	Appel ***ioctl()*** non appliqué à un terminal
27	EFBIG	Fichier trop grand
28	ENOSPC	Plus de place sur le disque
29	ESPIPE	Déplacement du pointeur de fichier illégal
30	EROFS	Tentative de modification d'un système de fichiers monté en lecture seule
31	EMLINK	Trop de liens sur un même fichier
32	EPIPE	Pipe brisé
33	EDOM	Résultat trop grand
34	ERANGE	Résultat non représentable
35	ENOMSG	Pas de message du type désiré

INDEX

A
argc 12
argv[] 12
assemblage 7

C
cc 6
compilation 7

E
envp[] 12
errno 15

F
fonction perror() 16
fonction strerror() 16

G
getenv() 13

L
liens 8

M
main() 12

P
pré-compilation 7
projet 9
projet librairie 11
projet modulaire 10
projet monolithique 9
projet public 11

S
sys_errlist[] 15
sys_nerr 15
system() 14