# Programmation Avancée en Langage C
# Séances de Travaux-Pratiques

Ph. Langevin

Année Universitaire 2010

## Contents

**1 Préambule** **2**

**2 Commandes de base** **2**
- 2.1 man . . . . . 2
- 2.2 Recherche . . . . . 2
- 2.3 pipe . . . . . 3
- 2.4 Redirection . . . . . 3

**3 Compilation** **3**

**4 gdb** **4**
- 4.1 Division par zéro . . . . . 4
- 4.2 Erreur de segmentation . . . . . 5
- 4.3 Débordement de pile . . . . . 5
- 4.4 Boucle infinie . . . . . 5
- 4.5 Débordement de tableau . . . . . 5

**5 La fonction main** **5**
- 5.1 Retour d'une commande . . . . . 6
- 5.2 Retour de main . . . . . 6
- 5.3 Retour de main . . . . . 6
- 5.4 Ligne de commande . . . . . 6
- 5.5 Environnement . . . . . 7
- 5.6 Traitement des options . . . . . 7

**6 Flux** **8**

**7 Zones mémoires** **8**

**8 Cadre de pile** **10**

**9 Projet** **11**
9.1 Types et variables . . . . . . . . . . 11
9.2 Mise en place du projet . . . . . . . . . . 12
9.3 Générateur de grilles . . . . . . . . . . 12
9.4 Parties aléatoires . . . . . . . . . . 13

# 1 Préambule

L'objectif de ces séances de travaux-pratiques est d'approfondir quelques notions concernant la `programmation en langage C` sous `linux`. Les exercices proposés seront traités au début des séances de travaux-pratiques, le reste du temps sera consacré au développement d'un jeu de *bataille navale* classique.

# 2 Commandes de base

La programmation sous `linux` implique la connaissance d'un minimum de commandes.

## 2.1 man

La commande de survie par excellence. Quand le système est correctement installé, la plupart des commandes et fonctionnalités sont décrites dans les pages du manuel.

- Retrouver le rôle des options `-Wall`, et `-g` du compilateur `gcc`.
- Quel est le rôle de la commande `indent` ?
- Quel fichier faut-il inclure pour utiliser la fonction sin dans un programme en langage C ? Nom symolique de la bibliothèque à relier lors de la compilation ?

## 2.2 Recherche

Il est souvent utile de localiser de l'information.

- Utiliser la commande `find` pour localiser le fichier `math.h` dans le système de fichiers.

- Utiliser la commande `grep` pour trouver les occurrences de “PI” dans le fichier `math.h`.

### 2.3 pipe

L’opérateur | permet de chainé les commandes.

- Compter le nombre de lignes du `math.h` avec `wc`.
- Dans le fichier `math.h`, combien de lignes contiennent le motif “PI” ?

### 2.4 Redirection

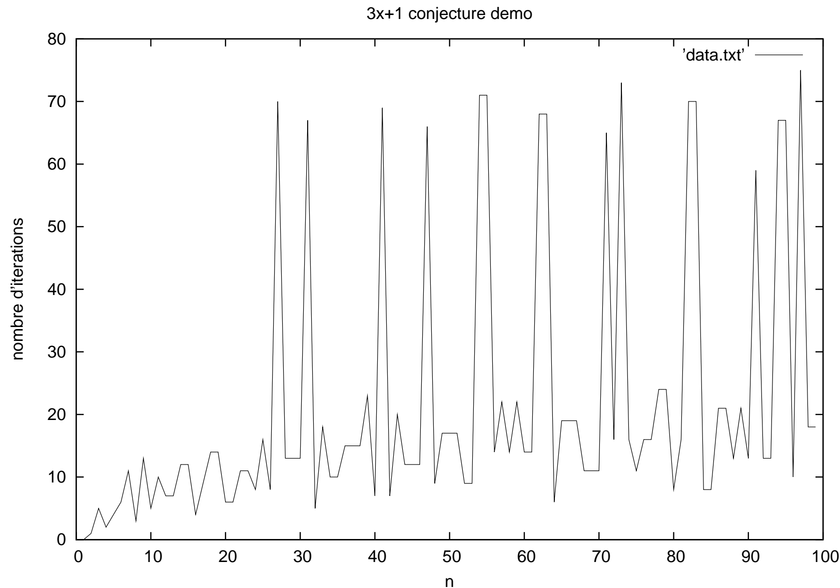
Les opérateurs > et < permettent de rediriger les entrées-sorties.

- Utiliser `history` pour obtenir l’historique des commandes utilisées.
- Faire un fichier `memo.txt` en redirigeant la sortie de `history`.

## 3 Compilation

Utiliser la commande `wget` pour télécharger l’archive http://langevin.univ-tln.fr/cours/PALC/tps/exemple-make.tar. Détarer cette archive avec la commande `tar xvf`, pour se rendre dans le répertoire `exemple-make`.

- Lister le contenu du répertoire.
- Utiliser `cat` pour afficher le contenu du fichier `makefile`.
- Lancer `make` pour fabriquer le fichier `data.ps`.
- Utiliser `evince` pour voir l’image postscript.




# 4 gdb

Utiliser la commande `wget` pour télécharger l'archive http://langevin.univ-tln.fr/cours/PALC/tps/exemple-gdb.tar. Détarer cette archive avec la commande `tar xvf`, pour se rendre dans le répertoire `exemple-gdb`.

- Lister le contenu du répertoire.
- Utiliser `cat` pour afficher le contenu du fichier `makefile`.
- Lancer les compilations.
- Le fichier `makefile` contient une erreur à corriger.

## 4.1 Division par zéro

Lancer le programme : `./zero.exe 5`. Il termine avec l'erreur *Exception en point flottant*. Pour faire un diagnostic, charger le programme dans le debugger `gdb zero.exe`, sous le prompt de `gdb`, lancer `run`. Une fois la ligne d'erreur identifiée, confirmer le diagnostic par `print x`.

### 4.2 Erreur de segmentation

Lancer le programme : `./table.exe`. Il termine avec l'erreur *Segmentation fault*. Pour faire un diagnostic, charger le programme dans le debugger `gdb table.exe`, sous le prompt de `gdb`, lancer `run`. Une fois la ligne d'erreur identifiée, faire un diagnostic par `print y`, puis `print *y`.

### 4.3 Débordement de pile

Lancer le programme : `./pile.exe`. Il termine avec l'erreur *Segmentation fault*. Pour faire un diagnostic, charger le programme dans le debugger `gdb pile.exe`, sous le prompt de `gdb`, lancer `run`. Une fois la ligne d'erreur identifiée, faire un diagnostic par `backtrace`, `list`, `whatis s`, `whatis data_t`.

### 4.4 Boucle infinie

Lancer le programme : `./infini.exe 13`. Il ne termine pas. Pour faire un diagnostic, identifier le `pid` du processus, par exemple : `ps aux | grep infini`, ou encore `top`. Intercepter le processus avec `gdb --pid (valeur)`. Sous le prompt de `gdb`, lancer `list`, `whatis x`.

Pour confirmer le diagnostic, charger le programme dans le debugger `gdb infini.exe`, sous le prompt de `gdb`, lancer `break proc`, `run 13`, `display x`, `next`, `next`, `next` etc...

### 4.5 Débordement de tableau

Lancer le programme : `./range.exe`. Il ne termine pas. Utiliser `gdb` pour déterminer la procédure ou fonction qui part dans une boucle infinie. Essayer de faire un diagnostic en relançant le programme sous `gdb`.

## 5 La fonction main

La fonction `main` fait le lien entre l'exécutable et le système : la valeur de retour de `main` est la valeur transmise au système aprés l'exécution. Avant de lancer l'exécutable, le système empile le nombre d'arguments, l'adresse d'un tableau contenant les arguments, et l'adresse d'un tableau contenant l'environnement.

## 5.1 Retour d'une commande

La valeur retournée par une commande est récupérée par le système. Cette valeur est disponible au niveau du shell dans la variable `$?`. Dans la plupart des cas, la valeur de retour 0 signale un comportement normal. Tester les valeurs de retour de `grep` : sur un fichier qui n'existe pas, sur un motif introuvable, et sur un motif présesent dans le fichier.

## 5.2 Retour de main

Compiler le programme (option `-Wall` ) :

```
void main( void ) { }
```

Commentaire ?

## 5.3 Retour de main

La valeur retournée par `main` est récupérée par le système. Écrire un exécutable constitué d'une fonction `main` de prototype

```
int main( void )
```

pour lire une valeur (`scanf`) qui sera transmise au système.

## 5.4 Ligne de commande

Ecrire le programme ci-dessous dans un fichier (`main.c`), compiler un exécutable (`main.exe`). Lancer `./main.exe too -v 123`, puis `echo $?`.

```
4  int main( int argc, char* argv[] )
5  {
6    int i;
7
8    printf("\nNombre d'arguments : %d", argc );
9
10   for( i = 0 ; i < argc ; i++)
11     printf("\n      argv[ %d ] = %s", i, argv[ i ] );
12
13   return 9;
14 }
```

## 5.5 Environnement

Ecrire le programme ci-dessous dans un fichier (`env.c`), compiler un exécutable (`env.exe`). Lancer `./env.exe`. Commenter.

```
2  #include <stdlib.h>
3
4  int main( int argc, char* argv[] , char **env)
5  {
6    printf("\nEnvironnement de %s", argv[0] );
7    while ( *env != NULL ) {
8          printf("\n%s", *env);
9          env++;
10   }
11   return EXIT_SUCCESS;
12 }
```

## 5.6 Traitement des options

La fonction `getopt` permet de traiter la ligne commande d'un programme. Ecrire le programme ci-dessous dans un fichier `option.c`, compiler un exécutable (`option.exe`), tester `./option.exe -a 12 -b345 -f toto`, `./option.exe -a -b`, `./option.exe -a`, `./option.exe -x`.

```
5  int main( int argc, char *argv[] )
6  { int opt;
7    char *optlist ="a:b:f:h";
8    int  a = 1, b = 0;
9    char *nom = NULL;
10   while ( ( opt = getopt( argc, argv , optlist) ) > 0  )
11           switch( opt ){
12           case 'a': a  = atoi( optarg ); break;
13           case 'b': b  = atoi( optarg ); break;
14           case 'f': nom = strdup(optarg); break;
15           case 'h': printf("\nhelp:");
16           default : printf("\nusage de %s : %s\n", argv[0], optlist);
17                     exit ( 1 );
18           }
19
20   printf("a=%d b=%d f=%s", a, b, nom);
21   printf("\n");
22   return 0;
23 }
```

## 6 Flux

Le programme ci-dessous lit un fichier pour convertir son contenu en majuscules.

```
 5 int main (int argc, char *argv[] )
 6 { int car;
 7   FILE *src, *dst;
 8   src = fopen( argv[1], "r" );
 9   dst = stdout;
10   if ( src == NULL ) {
11     fprintf(stderr, "\nfichier introuvable!");
12     src = stdin;
13   }
14   while ( ! feof( src) ) {
15     car = fgetc( src );
16     if ( car != EOF ) {
17       if ( isalpha( car ) )
18         car = toupper( car );
19     fputc( car , dst );
20     }
21   }
22   fclose( src );
23   return 0;
24 }
```

1. Utiliser `wget` pour télécharger la source :
   http://langevin.univ-tln.fr/cours/PALC/tps/exemple-flux/flux.c

2. Modifier le code pour écrire dans un fichier dont le nom est celui de la source préfixé par “out-”. ( `sprintf` ).

3. Ecrire un programme `sifr.c` pour faire un chiffrement affine du fichier source. Seuls Les caractères représentants des lettres seront chiffrés. ( `isalpha`, `islower`). Pour chiffrer un caractère majuscule $x$, on pourra utiliser la formule :

   $$x \mapsto {}'A' + \big(a * (x - {}'A') + b \pmod{26}\big)$$

   où $a$ et $b$ forment la clé de chiffrement.

## 7 Zones mémoires

Le programme ci-dessous permet de tracer les adresses des différents objets composants un programme. Il termine par un appel de la fonction `system`

pour afficher une description des zones mémoires.

```
7
8  typedef struct {
9          char   *id;
10         void   *adr;
11 } data;
12
13 char *ptr;
14 data  mem[64], tmp;
15 int   max = 0;
16 char  bfr[64];
17
18 void proc( int x, int*y) { int l; // trace ?
19 }
20
21 int main(void)
22 {
23 int i;
24 mem[max].id = "& max"; mem[max].adr = & max; max++;
25 mem[max].id = "& ptr"; mem[max].adr = & ptr; max++;
26 mem[max].id = "  ptr"; mem[max].adr = ptr;   max++;
27 // trace ?
28 proc( max, &max);
29
30 ptr = (char*) malloc(1024);
31 mem[max].id = " ptr (alloue)"; mem[max].adr = ptr;  max++;
32
33 // tri ?
34
35 for( i = 0; i < max; i++ )
36  printf("\n%-12p : %12u : %s", mem[i].adr, (uint) mem[i].adr, mem[i].id);
37 printf("\n\nZones memoires :\n");
38 sprintf(bfr, "cat /proc/%d/maps | grep -v so", getpid() );
39 system( bfr );
40 printf("\n");
41 return 0;
42 }
```

1. Utiliser `wget` pour télécharger la source :
   http://langevin.univ-tln.fr/cours/PALC/tps/exemple-mem/mem.c

2. Tracer toutes les variables globales du programme.

3. Trier le tableau `mem` avant de l'afficher.

4. Tracer l'adresse de la fonction `proc`.

5. Modifier `proc` pour tracer toutes les adresses en rapport avec cette fonction.

6. Vérifier la localisation des variables.

# 8 Cadre de pile

Avant un appel de fonction, les paramètres et l'adresse de retour sont empilés. La fonction construit son cadre de pile en réservant un espace pour les variables locales, le contenu du registre de base est empilé (sauvegarde), avant de prendre la valeur du pointeur de pile.

Le résultat est transmis par les registres.

```
13
14 typedef unsigned long long ullong;
15
16 ullong proc( ullong x, ullong y)
17 { ullong r = 0;
18   r = x + y;
19   return r;
20 }
21
22 int main(void)
23 {
24 ullong     s;
25          s = proc( 23, 5 );
26 return 0;
27 }
```

1. Utiliser `wget` pour télécharger la source :
   http://langevin.univ-tln.fr/cours/PALC/tps/exemple-pile/pile.c

2. Compiler une exécutable avec l'option `-g`.

3. Charger le programme sous `gdb`.

4. Placer un point d'arrêt sur la fonction `proc`

5. Lancer le programme jusqu'au point d'arrêt.

6. Déterminer les adresses de $x$, $y$, $r$.
7. Déterminer le contenu du pointeur de pile (esp).
8. Déterminer le contenu du regsitre de base (ebp).
9. Reconstituer le cadre de pile.
10. Quelle est l'adresse de retour ?

# 9 Projet

L'objectif du projet est de programmer le jeu de bataille navale tel qu'il est décrit dans wikipaedia.

## 9.1 Types et variables

```
1  #ifndef BNC_H
2  #define BNC_H
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #define  NB 6
7  #define touche 1
8  #define rate   2
9  #define coule  3
10 #define nul    4
11 #define opt    5
12 #define Opt    6
13 typedef struct {
14   char   nom[16];
15   int    vie;
16   int    num[10][10];
17   int    val[NB];
18   int    tir[10][10];
19   int    adv[10][10];
20 } grille;
21 /*
22    nom       : nom du joueur
23    vie       : points de vie
24    num[i][j] : numero du navire occupant la case ixj
25    val[k]    : valeur du navire k
26    tir[i][j] : tir recu sur la case ixj
```

```
27    adv[i][j] : information sur l'adversaire
28 */
29
30 void  init(grille *g, char* nom);
31 // initialise une grille aleatoirement
32 void print(grille  g);
33 // affiche une grille
34 int attaque( int i, int j, grille *y );
35 // un tir (i,j) sur y
36 void defense(int i, int j, grille *x, int r);
37 // mise a jour (i,j) sachant r
38 #endif
```

## 9.2 Mise en place du projet

Il s'agit de créer un répertoire `projet` qui contient 4 fichiers :

1. Un fichier de définition `bnc.h`. Il suffit d'utiliser `wget` pour télécharger à l'adresse :http://langevin.univ-tln.fr/cours/PALC/tps/projet/bnc.h

2. Le code des fonctions communes `bnc.c`. Dans un premier temps, les fonctions `init` et `print` ne font rien. Par exemple,

```
void print( grille x) { }
void init( grille *x, char*s){}
```

3. Une source `test.c` qui contient une fonction `main` qui appelle les fonctions `init` et `print` :

```
int main( void )
{
 grille x;
 init( &x , ``test'');
 print( x );
 return 0;
}
```

4. Un fichier `makefile` construit sur le modèle de la section (3).

Les fichiers `bn.o` et `test.exe` seront compilés automatique ment par `make`.

## 9.3 Générateur de grilles

Il s'agit de coder les fonctions `init` et `print`.

```
Alice : 16
      A  B  C  D  E  F  G  H  I  J        A  B  C  D  E  F  G  H  I  J
| 0 | +  .  .  +  4  4  4  4  .  .  | 0 | .  .  .  .  .  +  .  .  .  .
| 1 | +  .  +  +  .  +  +  +  .  .  | 1 | +  .  .  +  +  .  +  .  .  .
| 2 | .  .  +  +  .  .  .  .  3  .  | 2 | .  +  .  +  .  +  .  .  +  .
| 3 | +  .  .  .  .  +  +  +  3  .  | 3 | .  .  .  .  +  .  +  .  +  .
| 4 | .  +  .  .  .  .  .  +  3  .  | 4 | .  .  .  .  .  .  .  .  +  .
| 5 | .  +  +  +  +  .  +  .  .  .  | 5 | +  .  .  .  .  .  #  .  +  .
| 6 | .  1  1  .  2  #  2  +  +  .  | 6 | .  .  +  .  .  .  .  +  +  +
| 7 | .  +  .  +  .  .  .  .  .  .  | 7 | +  +  .  .  .  +  #  .  .  .
| 8 | .  +  5  #  5  5  5  5  .  .  | 8 | .  .  .  .  .  +  .  .  .  .
| 9 | .  .  +  .  .  .  .  .  .  .  | 9 | .  .  .  +  .  #  .  .  +  +
      A  B  C  D  E  F  G  H  I  J        A  B  C  D  E  F  G  H  I  J
```

### 9.4 Parties aléatoires

Implanter les fonctions :

- `int attaque ( int i , int j , grille * y )` pour obtenir le résultat d'un tir en $i \times j$ sur la grille $y$.
- `void defense ( int i , int j , grille * x , int r )` pour mettre mise à jour les informations de la grille $x$.
- `void partie(void)` pour jouer une partie aléatoire sans stratégie.