Aide mémoire : Langage C

Ce document n'est pas un résumé exhaustif des commandes C mais un aide-mémoire volontairement raccourci afin de permettre une acquisition rapide des premiers rudiments de programmation en C.

1 Compilation-Exécution

xemacs Un programme C est écrit dans un fichier ASCII que l'on peut donc composer avec un `éditeur` ASCII quelconque comme xemacs sous Unix-Linux ou Notepad sous windows.

cc/gcc Une fois écrit, le programme doit être lu et transcrit en langage machine par un programme appelé `compilateur`. Le compilateur vérifie la syntaxe et l'organisation du code. La commande `cc -o nom_exec programme1.c programme2.c ...` (ou `gcc`) crée un fichier exécutable nommé `nom_exec` correspondant au code C écrit dans les fichiers `programme1.c`, `programme1.c`,...

execution L'exécutable produit par le compilateur est en fait la traduction en un langage directement compréhensible par la machine du code C. C'est un `programme` qui peut donc être exécuté.

E/S (I/O) Un programme peut agir sur différentes Entrées/Sorties. On désigne souvent l'Entrée Standard comme étant clavier et la Sortie Standard l'écran. Néanmoins, d'autres sorties peuvent être utilisées comme les Entrées/Sorties sur fichiers.

2 Structure d'un programme C

instruction Une instruction est constituée d'un ensemble de commandes, de fonctions et de variables qui peut tenir sur plusieurs lignes. Une instruction simple se termine par un point-virgule. Une instruction peut aussi être un ensemble d'instructions entouré d'accolades, appelé `bloc`, sans point-virgule à la fin.

main Un programme C est un ensemble d'instructions. Il est écrit dans un bloc entre accolades précédé de `int main()`, ce bloc est appelé `programme principal`.

fonctions Une fonction est un ensemble d'instructions qui permet de décomposer un programme en tâches plus petites. Elle possède un nom `fct1(...)` que l'on fait suivre de `paramètres` entre parenthèses. Le programme principal est donc aussi une fonction.

***.h *.c** La fonction main est écrite dans un fichier portant l'extension `.c`. Un programme divisée en fonctions peut être écrit dans plusieurs fichiers. Dans ce cas, on sépare les parties déclaratives dans des fichiers d'extensions `.h` (headline) contenant les en-têtes (prototype) des fonctions et le corps des fonctions dans d'autres fichiers `.h`. On peut compiler séparemment chaque fichiers du programme par la commande
`cc -c fct1.c` qui produit le fichier `fct1.o`. A la fin, on lie (link) tous les fichiers `*.o` avec le main pour obtenir l'exécutable.

#include Au début d'un programme C, avant le main ou une fonction, on peut inclure des fichiers contenant les entêtes des fonctions par la commande `#include "fct1.h"`. Aussi, on peut inclure l'entête de librairies C par la commande `#include<stdio.h>`.

/*...*/ On peut ajouter des commentaires à un programme entre `/*` et `*/`.

3 Types simples et mots-clés

identificateur Un identificateur est un nom utilisable pour désigner une variable ou une fonction. Il est fait d'un seul mot composé de lettres, de chiffres et de l'underscore_ mais doit commencer par une lettre. Attention, le C fait la différence entre minuscule et majuscule.

mots-clés Certains mots sont réservés comme les commandes du langage C et sont écrits en minuscule. (ex: `for, int, while, system,...`).

variables Une variable C est repérée par un nom-identificateur. Toute variable doit être déclarée en début de bloc en écrivant son type suivi de son identificateur (ex: `int i;`).

type Une variable possède un type qui désigne ce que peut contenir une variable. Les types de base courant sont :

`char`	un caractère.
`int`	un nombre entier.
`float`	un nombre en virgule flottante simple précision.
`double`	un nombre en virgule flottante simple précision.
`void`	absence de type (ex: une fonction qui ne renvoie pas de valeur).
`long int`	nombre entier de précision étendue
`long double`	nombre flottant de précision étendue

constantes On appelle constante une valeur précise écrite dans un programme. par exemple `1` ou `2` ou `1000` sont des constantes de type entier; `5.0` ou `10.99` de type flottant; `'c'` ou `'/'` de type char (attention il s'agit d'un seul caractère),...

#define On peut définir des "variables-constantes" universelles pour tout un fichier en écrivant en début de fichier (ex:`#define PI 3.14`).

const On peut définir qu'une variable ne sera pas modifiée en utilisant le qualificatif `const` avant le type lors de la déclaration. Dans le cas d'un tableau, cela interdit de modifier ses éléments.

tableau Une façon de déclarer un tableau est de désigner son type, son identificateur puis ses dimensions entre crochets. Par exemple, `int M[4][3];` déclare un tableau de 5*4=20 entiers numérotés `M[0][0], M[0][1], M[1][3]` ou `M[4][3]`.

typedef On peut renommer un type complexe par la commande `typedef` (ex: `typedef matrice int[50][50];`)

4 Variables et opérateurs

déclaration Les variables utilisées dans un bloc doivent être définies en liste au début du bloc (après l'accolade ouvrante). On indique alors son type. Attention, une variable ainsi déclarée n'est connue que dans le bloc et deux variables de même nom contenues dans des blocs différents n'ont aucun rapport.

affectation Une variable peut être affectée à une valeur constante ou à la valeur d'une autre variable par l'opérateur `=`.

Par exemple :
```
int x,y; /* déclaration de 2 variables entières x et y*/
y=3; /* y prend la valeur 3 */
x=y; /* y prend la valeur 3 */
```

+-*/% L'opérateur arithmétique `+` additionne, `-` soustraie, `*` multiplie, `/` divise, `%` donne le modulo de nombres situés de part et d'autre. (ex: `1*5+2*5+9*(4/2)*(5%2)` est la valeur `33`).

<>==!= Entre deux valeurs on peut faire les comparaisons classiques `<,>,<=,>=`, l'égalité `==` et l'inégalité `!=`.

TRUEFALSE&&||! On réalise des opérations logiques entre des valeurs entières particulières: `TRUE` est la valeur vraie et vaut `1`, `FALSE` est la valeur faux et vaut `0`. Ainsi on peut évaluer `1!=2` comme étant une expression valant `TRUE`, c'est-à-dire `1`. On peut donc additionner logiquement des valeurs logiques avec le "et" `&&`, le "ou" `||`, et donner le contraire d'une valeur avec `!`. Par exemple, `(1==2)&&(!(3==5-2))` est la valeur `FALSE` donc `0`.

incrementation On appelle incrémentation d'un entier le fait de lui ajouter la valeur 1. On peut le faire avec l'instruction `i=i+1;` ou avec l'opérateur `++` (ex: `i++;`). Idem pour la décrémentation avec l'opérateur `--`.

5 Structures des contrôle

if else L'instruction de test a pour syntaxe: (la partie `else` peut être ommise.)

```
if (expr_logique) instruction1 else instruction2
```

switch Cette instruction de tests a pour syntaxe:

```
switch(expression){
      case val1 : instruction1
      case val2 : instruction2
      ...
      default : instructiond
}
```

Elle fait exécuter `instruction1` si `expression==val1`, `instruction2` si `expression==val2`, ... , si aucun des cas n'est vérifié `instructiond` est exécutée (cette dernière partie est facultative). Si l'on veut qu'après l'exécution de `instructioni` , le cas suivant ne soit pas traité, on place après `instructioni` la commande `break;`.

while Une boucle "while" de syntaxe

```
while (expr_logique) instruction1
```

signifie que l'instruction `instruction1` est répétée tant que `expr_logique` est vraie.

for Une boucle "for" de syntaxe

```
for (expr1;expr2;instruction3) instruction4
```

est équivalent à

```
expr1;
while (expr2){
   instruction4;
   instruction3;
}
```

Par exemple: `for (i=0;i<10;i++) V[i]=1;` remplit les 10 valeurs d'un tableau de taille 10 avec le nombre 1.

indentation L'indentation est une façon de mettre en page un programme de manière à ce que l'imbrication des boucles, des tests et des fonctions soient lisibles. Le programme précédant en est un exemple où l'on peut facilement repérer où commence et finit la boucle "while".

6 Fonctions et paramètres

return Une fonction possède un type de retour. C'est-à-dire qu'une fonction peut être évaluée et manipulée comme une "variable-constante" de ce type. A la fin d'une fonction, on doit placer l'instruction `return(val);` qui permet de donner la valeur `val` à la fonction. Après son exécution, la fonction prend cette valeur à l'endroit où elle a été appelée. Dans le cas d'un type `void`, la fonction ne retourne rien.

déclaration Une fonction est définie en 2 étapes. Une première fois elle est déclarée dans un fichier en-tête (`.h`) avec la syntaxe:

```
type_retour nom_fct(type1 param1, type2 param2,...);
```

puis elle est implémentée (codée) dans un fichier `.c` avec la syntaxe:

```
type_retour nom_fct(type1 param1, type2 param2,...){
     ...
     return(val);
}
```

Les variables `param1` de type `type1`,... peuvent être utilisées comme des variables du programme. L'intérieur du corps de la fonction peut inclure des appels à d'autres fonctions et même des appels à la fonction elle-même (récurrence).

appel Une fonction peut être appelée par la syntaxe `fct1(var1,var2,...)`. Ainsi, les variables `param1`,... prend la valeur de la variable `var1`,... Il faut donc que les types concordent. On peut aussi faire appel à une fonction en passant en paramètres des constantes.

passage En C, le passage de paramètres se fait toujours par valeurs. C'est-à-dire qu'en retour, les variables `var1`,... ne prennent pas les valeurs des variables de la fonction `param1`,... Pour réaliser un tel passage, il faut travailler avec les pointeurs (voir section 7).

7 Pointeurs et allocations de mémoire

pointeur Une variable possède une addresse-mémoire correspondant à l'emplacement où elle est stockée. On peut récupérer l'adresse d'une variable `x` par la commande `&x`.

type pointeur Une adresse est de type pointeur. On peut définir par exemple une variable de type "pointeur sur entier" par la déclaration `int *p;`. Ainsi si `x` est une variable entière, on peut affecter `p` à la valeur de l'adresse de `x` par l'affectation `p=&x;`.

taille Toutes les variables n'ont pas la même taille en mémoire, on peut déterminer la taille d'un type par la commande `sizeof()` qui prend en argument un type et renvoie un nombre de type `size_t`. Par exemple, `sizeof(char)` vaut en général 1 octet.

allocation Plutôt que définir une variable du programme pour chaque donnée à stocker, on peut directement alouer de l'espace mémoire pour stocker une donnée sans lui donner un nom. On parle d'allocation mémoire. On alloue la mémoire par la commande (voir section 10):

```
void *malloc(size_t taille);
```

qui renvoie un pointeur sur un espace mémoire réservé à un objet de taille `taille`. Par exemple, on peut réserver un pointeur sur un entier par les instructions `p=(int *) malloc(sizeof(int));` où `p` est un pointeur sur entier. La parenthèse devant le malloc s'appelle un casting, c'est-à-dire que l'on indique que la fonction malloc qui est de type pointeur su void va renvoyer dans ce cas un pointeur sur entier.

désallocation Une fois une zone mémoire allouée, cette zone reste allouée tant qu'elle est utilisée par le programme. La mémoire non utilisée par le programme ne sera libérée que lorsque le système la collectera et la libérera à un moment non prévisible par le programme. C'est

pourquoi on doit désallouer la mémoire réservée après utilisation.

`void free(void *p)` libère la zone mémoire pointée par `p`

accès On accède aux données pointées par une variable pointeur par les opérateurs `*`. Par exemple, après avoir alloué un pointeur sur entier par un malloc, l'instruction `*p` peut être manipulée dans le programme comme un entier.

paramètres Si l'on veut qu'en sortie de fonctions, une variable "conserve" les modifications faites sur elle dans la fonction, on doit passer en paramètre l'adresse de cette variable. Par exemple:

```
void ajoute_cinq(int *p){
        *p=*p+5;
}
```

est une fonction qui ajoute 5 à la variable dont on passe l'adresse en paramètre. Si une variable du programme est un entier `x`, le fait de faire appel à la fonction par l'instruction `ajoute_cinq(&x)` modifie le contenu de `x` dans le programme en lui ajoutant 5.

tableau En fait, un tableau est un espace mémoire où l'on a alloué plusieurs espace-mémoire d'un même type les uns à côtés des autres (on parle de mémoire contigüe). L'indice d'un tableau `[0], [1],...` correspond en fait au numéro de l'emplacement réservé. Conséquemment, un tableau est en fait un pointeur! On alloue une mémoire contigüe par un calloc avec pour taille le nombre d'éléments fois la taille d'un élement ou alors par la commande:

```
void *calloc(size_t nelt, size_t taille);
```

où `nelt` est le nombre d'élément du tableau.

On définit et alloue aisemment un tableau d'entiers à une dimension par les lignes:

```
int *M;
M=(int*) calloc(10,sizeof(int));
```

Un tableau d'entiers à deux dimensions est en fait un tableau de pointeurs!

```
int **M;
M=(int**) calloc(10,sizeof(int*));
for(i=0;i<10;i++)
  M=(int*)calloc(10,sizeof(int));
```

affectation ATTENTION, l'affectation d'un "tableau" `M` dans un "tableau" `T` n'est jamais qu'une affectation entre deux pointeurs! L'instruction `T=M;` signifie qu'après cette instruction que les pointeurs `T` et `M` vont pointés sur le même tableau.

8 Struct et structures de données

struct On peut construire des types composés avec le type `struct`. Il permet de regrouper plusieurs variables dans une structure fixe. Sa syntaxe est:

```
struct nom_type { type1 var1; type2 var2; ...}
```

On peut alors déclarer une variable de ce type par `struct nom_type nom;` et on déclare ainsi les variables `nom.var1, nom.var2,...` qui se manipulent comme des variables simples.

struct...* On peut allouer une variable de type `struct` et ainsi réserver de la place mémoire pour tous ses champs. On peut accéder au champ bien sûr par la commande `*` (ex: `(*pnom).var1`) mais on peut aussi utiliser l'opérateur `->` (ex: `pnom->var1`).

data struct. Un des champs d'un type `struct` peut être de type pointeur et même de type pointeur sur ce même type `struct`. Ce schéma permet par exemple de coder la structure de données appelée liste chaînée d'entiers:

```
struct elt {
    int i;
    struct elt *suivant;
}
```

9 Manipulation de chaînes de caractères

chaîne Une chaîne de caractères est en fait un tableau de type `char`, donc de type `char *`. Elle se termine par le caractère `'\0'` (attention c'est un seul caractère!). On peut définir une constante-chaine par une liste de caractères entre `“”`. Par exemple:

```
char *s=(char*) calloc(20,sizeof(char));
s=”Bonjour”;
```

manipulations Si l'on veut opérer des manipulations de chaînes comme la copie ou l'addition de deux chaînes, on doit utiliser des fonctions de la librairies `<string.h>` (voir section 10). Dans la suite les chaînes `s1` et `s2` sont de `type const char *`:

`char* strcpy(s1,s2);` copie `s2` dans `s1`, retourne `s1`

`char* strncpy(s1,s2, int n);` copie au plus `n` caractères de `s2` dans `s1` (complète par des `\0` si `s2` trop petite).

`char* strcat(s1,s2);` concatène `s2` à la suite de `s1`, retourne `s1`

`int strcmp(s1,s2);` compare les chaînes `s1` et `s2`, retourne une valeur négative si inférieure, nulle si égale et positive si supérieure.

`size_t strlen(s1);` retourne la longueur de `s1`.

...

10 Librairies et librairies standards

librairies Un code C compilé peut devenir une librairie, c'est-à-dire un ensemble de fonctions utilisables comme des commandes simples du langage C ou des fonctions faites par le programmeur.

librairie standard Il existe plusieurs librairies dites standard fournies avec le compilateur C. Il suffit d'inclure leurs entêtes au début des fichiers.

#include<math.h> Cette librairie contient des fonctions mathématiques classiques. Dans la liste suivante, `x` est de type `double` et les fonctions retournent un `double`: `sin(x)`, `cos(x)`, `tan(x)`, `asin(x)`, `acos(x)`, `atan(x)`, `sinh(x)`, `cosh(x)`, `tanh(x)`, `exp(x)`, `log(x)`, `log10(x)`, `pow(x,y)` `x` puissance `y`, `sqrt(x)` racine carrée, `ceil(x)` entier supérieur, `floor(x)` entier inférieur, `fabs(x)` valeur absolue,... Attention, il faut fréquemment ajouter `-lm` dans la ligne de compilation.

#include<stdlib.h> Cette librairie contient des fonctions de conversion de nombres, allocations et autres.
`double atof(const char *s)` convertit la chaine de caractères `s` en `double`.
`int atoi(const char *s)` convertit la chaine de caractères `s` en `int`.
`int rand()` retourne un entier pseudo-aléatoire compris entre `0` et `RAND_MAX`.
`int system(const char *s)` passe la chaîne `s` à l'environnement pour exécution.
`int abs(int n)` valeur absolue de l'entier `n`.
`calloc, malloc, free` commande d'allocation mémoire (voir section 7).

#include<string.h> Manipulation de chaîne de caractères (voir section 9).

#include<stdio.h> Opération sur les Entrées/Sorties, clavier, écan, fichiers (voir section 11).

11 Entrées/Sorties

flot Un flot est une source ou une destination de données qui peut être associée à un disque ou à un autre périphérique. La librairie `stdio.h` peut gérer entre autre les flots en mode texte, c'est-à-dire une suite de lignes constituées chacune de caractères et terminées par le caractère `\n`. Ce caractère revient à faire un retour-chariot-avance-ligne sous la plupart des systèmes.

flot standard Au lancement d'un programme un flot standard entrée (clavier) et un flot standard sortie (souvent l'écran) sont déjà ouverts.

ecriture On écrit sur le flot standard de sortie avec la fonction:

```
printf(format,type1 var1, type2 var2,...)
```

Le format est une chaîne de caractères contenant des caractères à afficher et des codes de mise en forme des variables. Par exemple:

`printf("Bonjour");` affiche `Bonjour`,

`printf("x=%d",i);` affiche `x=5`, si la variable entière `i` vaut 5,

`printf("Il y a %f Litres de substance",qte);`
affiche `Il y a 1.5 litres de substance`,
si la variable flottante `qte` vaut 1.5,

Les codes pour l'affichage formatée de variable sont `%d` pour `int`, `%f` pour `double`, `%g` pour `double` avec format scientifique,`%c` pour `char`, `%s` pour `char*` (affiche jusqu'à `'\0'`).

lecture On peut lire l'entrée standard par la commande:

```
scanf(format,type1 *var1, type2 *var2,...)
```

Le format suit les mêmes principes que pour `printf` mais on doit passer en argument l'adresse de la variable à remplir. Par exemple:

`scanf("%c",&c);` saisie un caractère dans la variable `c` de type `char`;

Par contre pour saisir une chaîne de caractère, c'est-à-dire un tableau, il faut penser à allouer la mémoire:

```
char *s=(char*) calloc(50,sizeof(char));
```

`scanf("%s",s);` saisie une chaîne de 49 caractères,
(après il y a le caractère `'\0'` puis il y a un dépassement de mémoire).

fichiers On peut ouvrir un flot, en particulier un flot sur les fichier. Il faut déclarer une variable-flot de type `FILE*` et l'affecter par la commandes:

```
FILE* fopen(char *nom,char* mode);
```

`nom` est le nom du fichier avec son chemin (le répertoire courant est le répertoire du programme); `mode` correspond au mode d'ouverture du fichier `"r"` pour la lecture, `"w"` pour l'écriture, `"a"` ouvre ou crée le fichier et se positionne à la fin pour écrire.

fermeture A la fin des manipulations, il faut refermer un flot avec la commande:

```
fclose(FILE *f);
```

operations On peut effectuer les mêmes opérations d'écriture et de lecture dans un fichier que sur les flots standards avec:

```
fprintf(FILE *f,format,type1 var1, type2 var2,...)
fscanf(FILE *f,format,type1 *var1, type2 *var2,...)
```

On peut aussi faire les opérations suivantes:

`fseek(FILE *f,0,int pos);` place le flot sur la position pos,

`int feof(FILE *f);` teste si on a atteint la fin du fichier,

Par exemple, le code suivant permet d'afficher une liste de nombres entiers contenus dans un fichier

```
int i;
FILE *fic=fopen("bb.txt","r");
while (!(feof(fic))){
     fscanf(fic,"%d ",&i);
     printf("%d",i);
}
fclose(fic);
```

12 Messages d'erreurs

compilation Des erreurs peuvent se produire (eh oui!) lors de la compilation: les erreurs de syntaxe sur les mots-clés, les erreurs dans le nom des variables et des fonctions, affecter une variable d'un type à une variable d'un autre type, ne pas respecter le prototype d'une fonction; Les erreurs de linkage si l'on oublie de lier des fichiers `.c` ou de mettre le `#include` des prototypes de fonctions au début d'un fichier où l'on utilise ces fonctions;...

warnings Le compilateur donne aussi des avis sur des choses étranges et litigieuses par des warnings: si l'on définit une variable sans l'utiliser, si on tente un casting non explicite, si on ne respecte pas la norme ANSI (norme d'un certain nombre de compilateurs C créée afin d'universaliser le code C),...

exécution Des erreurs peuvent se produire lors de l'exécution du programme. On les appelle des beugs (bugs), il s'agit en général d'un `aborted`, d'un `killed` ou d'un `segmentation fault` qui correspondent globalement à une erreur mémoire: le programme essaye d'écrire dans une zone mémoire interdite, il lit une donnée dans une zone qui n'est pas à lui, il lit une donnée fausse dans une de ses zones,... Ces erreurs se produisent généralement lorsque l'on a oublié d'allouer la mémoire, lorqu'on utilise trop de mémoire (il faut désallouer la mémoire non utilisée), lorsqu'on lit/écrit dans une case d'un tableau non allouée,... Il faut alors déboguer...

13 Makefile

compiler Les lignes de compilation deviennent souvent longue à copier, on peut alors utiliser un fichier de commande unix pour exécuter la ligne.

makefile Il existe sous unix des fichiers de commande exécutables par la commande unix `make` de manière à ne pas avoir à recompiler tous les morceaux d'un programme de grande taille. Ce fichier se nomme `Makefile`. Par exemple:

```
EXEC= es2 #nom de l'exécutable
SOURCES= es.c #noms des fichiers sources
CFLAGS= -O2 #options éventuelles de compilation
LIB =-lm #inclusion de librairies
OBJETS = $(SOURCES:.c=.o)
$(EXEC):$(OBJETS)# Attention, il faut un TAB juste avant gcc
      gcc -o $(EXEC) $(OBJETS) $(CFLAGS) $(LIB)
```

OUVRAGES-REFERENCES

B.W. Kernighan and D.M. Ritchie, Le langage C, 2eme ed., Masson