A. Erreurs fréquentes

A.1. Erreurs sur les pointeurs

A.1.1. Pointeurs non initialisés

```
(1)

int   x;
int * px;

x = *px;
```

```
(2)

char * s;

scanf("%s", s);
```

Dans le premier cas, le pointeur px n'est pas initialisé donc x prend n'importe quelle valeur (quand le programme ne provoque pas d'erreur).

Dans le deuxième cas, la mémoire n'a pas été réservée, ainsi le scanf() nous réserve quelques surprises. s est bien déclaré, mais lui non plus n'est pas initialisé et prend n'importe qu'elle valeur, il pointe donc sur une zone inconnue et le scanf va essayer de mettre les caractères à cet endroit générant une erreur de segmentation. Sur le même principe, les pointeurs que l'on donne aux fonctions strcpy(), gets(), scanf() doivent être initialisés et un espace mémoire suffisant réservé. En effet, ces fonctions ne font pas d'allocation mémoire.

A.1.2. Pointeurs constants

Imaginons que l'on veuille copier une chaîne de caractères s1 dans une chaîne de caractères s2 avec le code suivant :

```
char s1[10] = "chaine 1";
char s2[10];

s2 = s1;
```

Dans cette suite d'instructions, l'affectation est illégale : on essaie de modifier s2 qui est un pointeur constant. Une solution serait d'utiliser l'instruction strcpy() de l'interface string.h.

```
strcpy(s2, s1);
```

A.1.3. Renvoi d'un pointeur sur une variable locale

```
int * calculer()
{
  int x = 0;

  printf("%d", x++);
  return &x;
}

void main()
{
  int * py = calculer();

  printf("%d", *py);
}
```

A priori, le pointeur py reçoit une valeur calculée dans la fonction calculer() par un pointeur. Ce n'est pas le cas car, en fait, la fonction calculer() renvoie un pointeur sur la variable x locale à la fonction et qui n'existe plus en dehors de la fonction. Le pointeur renvoyé par la fonction ne pointe donc plus sur une adresse valide dès que la fonction est terminée. Le résultat est indéterminé (fonction du contexte d'exécution).

A.1.4. Modification d'une copie et non de l'original

Prenons l'exemple simple : incrémentation d'une variable grâce à une fonction...

Exemple 1 :

```
int incrementer(int x)
{
  return ++x;
}

void main()
{
  int x = 0;

  printf("%d", incrementer(x));
  /* affiche 1 */
  printf("%d", incrementer(x));
  /* affiche 1 */
}
```

Exemple 2 :

```
int incrementer2(int * x)
{
  return ++(*x);
}

void main()
{
  int x = 0;
  printf("%d\t", incrementer2(&x)); /* affiche 1 */
  printf("%d\n", incrementer2(&x)); /* affiche 2 */
}
```

Pourquoi dans l'exemple 1, la variable du programme principal x n'est-elle pas incrémentée alors que l'on a bien utilisé l'opérateur ++ ?

Simple, dans la fonction incrementer(), la variable x qui est incrémentée n'est pas la variable passée en paramètre mais une copie de celle-ci. Pour s'en convaincre, il suffirait de placer un printf() dans la fonction.

Tous les paramètres en C sont passés par valeur... Pour modifier la variable, il faut passer celle-ci **par adresse** en utilisant un pointeur (cf exemple 2).

A.2. Erreurs diverses

A.2.1. scanf

```
int x;
scanf ("%d", x);
```

Il faut passer les variables par adresse et dans ce cas ne pas oublier le "**&**". (c'est logique, la fonction change le contenu de la variable).

malloc / free

Lorsque l'on alloue de la mémoire, il ne faut pas oublier de la rendre au système après utilisation. A tout malloc() (ou similaire alloc(), calloc()) doit correspondre un free(). Si dans le programme est manipulée une structure "évoluée" (liste chaînée, arbre ...), il ne faut pas oublier d'appeler une méthode de rendu de mémoire.
Vous pouvez utiliser la commande valgrind pour vérifier s'il y a fuite de mémoire. Plus d'information sur http://valgrind.org/docs/manual/quick-start.html

A.2.3. oubli du break

```
int choix = 0;
...
switch (choix)
{
  case 1  : action1();
  case 2  : action2();
  default : action_default();
}
```

Si choix vaut 1, les fonctions action1(), action2() et action_default() seront appelées. Si action2() ne doit pas être appelée, il ne faut pas oublier le mot-clé ***break*** .

A.2.4 affectation ou test ?

```
int a = 0;
```

```
while (a = 1) {
  ...
}
```

Le code précédent est une boucle sans fin ... Il ne faut pas confondre le test d'égalité `==` et l'opérateur d'affectation `=`.

A.2.5 nom du programme ?

Il est des noms de programme que l'on trouve adapté dans certaines situations mais qui ont des conséquences inattendues. Voici deux exemples.

Ne jamais appeler un programme *null* car c'est un nom réservé pour le système (le périphérique nul */dev/null*).

Vous avez appelé votre programme *test* et il ne se passe rien lors de l'appel ? A moins d'y accéder par *./test* ? C'est normal, *test* est une commande UNIX. Faites un petit *man* pour vérifier.

Annexe B : Fonctions d'E/S bas niveau

Ouverture/Création de fichier

int **creat**(char * nom, int permission);

int **open**(char * nom, int mode, int permission);

Fermeture de fichier

int **close**(int fd);

Lecture/Écriture

int **read** (int fd, char * tampon, int n);

int **write**(int fd, char * tampon, int n);

Accès séquentiel

long **tell** (int fd);

long **lseek**(int fd, long offset, int origine);

Descripteurs de fichier réservés

0	stdin
1	stdout
2	stderr
3	fichier spécial E/S
4	imprimante

Mode	Description
O_RDONLY	ouverture en lecture seule
O_WRONLY	ouverture en écriture seule
O_RDWR	ouverture en lecture/écriture

Origine	Description
0 ou **SEEK_SET**	début du fichier
1 ou **SEEK_CUR**	position courante
2 ou **SEEK_END**	fin de fichier

Annexe C : Fichiers binaires

Ouverture/Création de fichier
FILE * **fopen**(const char * nom, char * mode);
Fermeture de fichier
Int **fclose**(FILE * fic);
Lecture/Écriture
size_t **fread**(void *ptr, size_t taille, size_t nbobj, FILE * fic);
size_t **fwrite**(const void *ptr, size_t taille, size_t nbobj, FILE * fic);
Accès séquentiel
long **ftell**(FILE * fic);
int **fseek**(FILE * fic, long deplacement, int origine);
Divers
int **feof**(FILE * fic);
int **ferror**(FILE * fic);
int **fflush**(FILE * fic);

Constante	Description
BUFSIZE	taille du buffer
FOPEN_MAX	nombre de fichiers ouvrables

Origine	Description
0 ou **SEEK_SET**	début du fichier
1 ou **SEEK_CUR**	position courante
2 ou **SEEK_END**	fin du fichier

Annexe D : Comment calculer le temps écoulé en C

Ceci est juste un préambule au calcul du temps en C sous UNIX. Pour de plus amples renseignements, consulter le **Kernighan et Ritchie** (p 261) ou les pages **man**. Ceci doit aussi être valable pour la norme ANSI.

Quelques fonctions...
clock_t **clock**();
time_t **time**(time_t * t);
double **difftime**(time_t * t2, time_t * t1);

Fichier d'entête
time.h

Constantes
CLK_TCK
CLOCKS_PER_SEC (ANSI C)

Pour toute opération, il faut bien entendu ne pas oublier d'inclure le fichier time.h.
La fonction **clock**() renvoie une **estimation** du temps d'utilisation CPU depuis le début du programme ou **-1** en cas d'échec. Il faut diviser cette valeur par l'une des deux constantes données pour obtenir un temps en **secondes**. Si vous voulez une unité de temps différente (plus précise par exemple), je vous conseille d'aller voir ce que vaut cette valeur ou de faire une petite conversion en double.

ATTENTION : cette fonction doit être supportée par le système pour donner un résultat correct (sinon, on a juste le droit à une différence de temps!). Exit donc notre bon cygwin ! D'après le *man* Linux, sur certaines machines, **clock**() remet le compteur à **0** toutes les 72 minutes.

ATTENTION : La norme C autorise l'initialisation de la fonction par n'importe quelle valeur. Pour calculer le temps, il est donc conseillé de faire un premier appel à la fonction en début de programme et le second quand il faut puis de faire la différence.

La fonction **time()** renvoie l'heure calendaire (depuis le 1er janvier 1970 0h00:00). Si le pointeur *t* n'est pas nul, la variable contiendra aussi cette valeur.

La fonction **difftime()** permet de calculer la différence $t_2 - t_1$ (en secondes).

Annexe E : Erreurs à l'exécution

Je veux juste parler un tout petit peu de ces erreurs d'exécution qui nous pourrissent la vie, surtout quand on manipule (plus ou moins bien) des pointeurs. Je fais bien entendu allusion aux très célèbres *core dump* ou *segmentation fault*.

L'erreur de segmentation se produit lorsque le programme essaye d'accéder à une zone mémoire qui ne lui est pas réservée (allouée). En général, on obtient une image de la mémoire (core dump). Le remède avec une telle erreur est de vérifier le comportement de pointeurs (initialisation, allocations suffisantes).

Le core dump est une copie [partielle] de la mémoire (core) au moment où l'erreur s'est produite. On a alors un fichier .core qui peut être très gros et qu'il est conseillé d'effacer quand on n'en a plus besoin (après déboguage).

Annexe F : La librairie standard

La syntaxe des principales fonctions de la librairie standard est donnée ci-dessous. Une liste exhaustive de toutes les fonctions disponibles figure à l'annexe B de l'ouvrage de Kernighan et Richie. Pour obtenir plus d'informations sur ces fonctions, il suffit de consulter les pages de `man` correspondant.

F.1. Entrées-sorties `<stdio.h>`

F.1.1. Manipulation de fichiers

Prototype de fonction	action
`FILE * fopen(char *reference, char *mode);`	ouverture d'un fichier
`void fclose(FILE *stream);`	fermeture d'un fichier
`int fflush(FILE *stream);`	force le vidage du tampon associé au `stream`. `stream=NULL` : tous les tampons

F.1.2. Entrées et sorties formatées

Prototype de fonction	action
`int printf(char *format, ...);`	écriture sur la sortie standard
`int scanf(char *format, ...);`	lecture depuis l'entrée standard
`int fprintf(FILE *stream, char *format, ...);`	écriture sur un fichier
`int fscanf(FILE *stream, char *format, ...);`	lecture depuis un fichier
`int sprintf(char *s, char *format, ...);`	écriture dans la chaîne de caractères s
`int sscanf(char *s, char *format, ...);`	lecture depuis la chaîne de caractères s

F.1.3. Ecriture et lecture de caractères

Prototype de fonction	action
`int getchar(void);`	lecture d'un caractère depuis l'entrée standard
`int putchar(int c);`	écriture d'un caractère sur la sortie standard
`char * gets(char *s);`	lecture d'une chaîne de caractères sur l'entrée standard

`int * puts(char *s);`	écriture d'une chaîne de caractères sur la sortie standard
`int fgetc(FILE *stream);`	lecture d'un caractère depuis un fichier
`int fputc(int c, FILE *stream);`	écriture d'un caractère sur un fichier
`int getc(FILE *stream);`	équivalent de `fgetc` mais implémenté par une macro
`int putc(int c, FILE *stream);`	équivalent de `fputc` mais implémenté par une macro
`char * fgets(char *s, size_t nb, FILE *stream);`	lecture d'une chaîne de caractères depuis un fichier
`int * fputs(char *s, FILE *stream);`	écriture d'une chaîne de caractères sur un fichier

F.2. Manipulation de chaînes de caractères `<string.h>`

Prototype de fonction	action
int strlen(char *s) ;	calcule la longueur de la chaîne `s`; retourne la longueur.
`char * strcpy(char *s1, char *s2);`	copie la chaîne `s2` dans la chaîne `s1`; retourne `s1`.
`char * strncpy(char *s1, char *s2, size_t n);`	copie `n` caractères de la chaîne `s2` dans la chaîne `s1`; retourne `s1`.
`void memcopy(char *s1, char *s2, size_t n);`	recopie la zone mémoire de `s2` de `n` octets dans la zone mémoire de `s1`.
`void * memset(void *s, int c, size_t n);`	initialise les `n` octets d'une zone mémoire `s` aux caractères `c`.
`char * strcat(char *s1, char *s2);`	concatène la chaîne `s2` à la fin de la chaîne `s1`; retourne `s1`.
`char * strncat(char *s1, char *s2, size_t n);`	concatène `n` caractères de la chaîne `s2` à la fin de la chaîne `s1`; retourne `s1`.
`int strcmp(char *s1, char *s2);`	compare `s1` et `s2` pour l'ordre lexicographique; retourne une valeur négative si `s1` est inférieure à `s2`, une valeur positive si `s1` est supérieure à `s2`, 0 si elles sont identiques.
`int strncmp(char *s1, char *s2, size_t n);`	compare les `n` premiers caractères de `s1` et `s2`.
`char * strchr(char *s, char c);`	retourne un pointeur sur la première occurrence de `c` dans `s`, et `NULL` si `c` n'y figure pas.

`char * strrchr(char *s, char c);`	retourne un pointeur sur la dernière occurrence de `c` dans `s`, et `NULL` si `c` n'y figure pas.
`char * strstr(char *s1, char *s2);`	retourne un pointeur sur la première occurrence de `s2` dans `s1`, et `NULL` si `s2` n'y figure pas.

F.3. Manipulation de caractères `<ctype.h>`

Toutes les fonctions ci-dessous permettent de tester une propriété du caractère passé en paramètre. Elles renvoient la valeur 1 si le caractère vérifie la propriété et 0 sinon.

Prototype de fonction	**renvoie 1 si le caractère est**
`int isalnum(int c);`	une lettre ou un chiffre
`int isalpha(int c);`	une lettre
`int iscntrl(int c);`	un caractère de commande
`int isdigit(int c);`	un chiffre décimal
`int isgraph(int c);`	un caractère imprimable ou le blanc
`int islower(int c);`	une lettre minuscule
`int isprint(int c);`	un caractère imprimable (pas le blanc)
`int ispunct(int c);`	un caractère imprimable qui n'est ni une lettre ni un chiffre
`int isspace(int c);`	un blanc
`int isupper(int c);`	une lettre majuscule
`int isxdigit(int c);`	un chiffre hexadécimal

On dispose également de deux fonctions permettant la conversion entre lettres minuscules et lettres majuscules :

Prototype de fonction	**action**
`int tolower(int c);`	convertit `c` en minuscule si c'est une lettre majuscule, retourne `c` sinon.
`int toupper(int c);`	convertit `c` en majuscule si c'est une lettre minuscule, retourne `c` sinon.

F.4. Fonctions mathématiques `<math.h>`

Le résultat et les paramètres de toutes ces fonctions sont de type `double`. Si les paramètres effectifs sont de type `float`, ils seront convertis en `double` par le compilateur. L'inclusion de <math.h> est obligatoire si vous voulez que le compilateur fonctionne correctement.

Prototype de fonction	action
`double cos(double x);`	cosinus
`double sin(double x);`	sinus
`double tan(double x);`	tangente
`double acos(double x);`	arc cosinus
`double asin(double x);`	arc sinus
`double atan(double x);`	arc tangente
`double exp(double x);`	Exponentielle e^x
`double log(double x);`	logarithme népérien
`double log10(double x);`	logarithme en base 10
`double pow(double x, double y);`	puissance x^y
`double sqrt(double x);`	racine carrée
`double fabs(double x);`	valeur absolue
`double ceil(double x);`	partie entière supérieure
`double floor(double x);`	partie entière inférieure

F.5. Fonctions utilitaires divers `<stdlib.h>`

F.5.1. Allocation dynamique

Prototype de fonction	action
`void * malloc(size_t taille);`	allocation dynamique d'une zone mémoire de `taille` octets. Elle retourne l'adresse du début de la zone mémoire allouée.
`void * calloc(size_t nb_elt,` `size_t taille_elt);`	allocation dynamique et initialisation de la zone allouée à zéro.
`void * free(void *ptr);`	libère une zone mémoire.
`void * realloc(void *ptr, size_t taille);`	modifie la taille d'une zone mémoire préalablement allouée par `calloc` , `malloc ou realloc`.

F.5.2. Conversion de chaînes de caractères en nombres

Les fonctions suivantes permettent de convertir une chaîne de caractères en un nombre.

Prototype de fonction	action
`double atof(char *chaine);`	convertit `chaine` en un `double`
`int atoi(char *chaine);`	convertit `chaine` en un `int`
`long atol(char *chaine);`	convertit `chaine` en un `long int`

F.5.3. Génération de nombres pseudo-aléatoires

Prototype de fonction	action
`int  rand(void);`	fournit un nombre entier pseudo-aléatoire dans l'intervalle [0, `RAND_MAX`], où `RAND_MAX` est une constante prédéfinie au moins égale à 2^{15}-1.
`void srand(unsigned int germe);`	modifie la valeur de l'initialisation du générateur pseudo-aléatoire utilisé par `rand`. Elle est égale à 1 par défaut.

F.5.4. Arithmétique sur les entiers

Prototype de fonction	action
`int  abs(int x);`	valeur absolue d'un entier
`long labs(long x);`	valeur absolue d'un `long int`

F.5.5. Recherche et tri

Prototype de fonction	action

`void qsort(void *tab,`
`size_t n, size_t taille,`
`int (*comp)(void * , void *));`
	tri sur place par ordre croissant d'un tableau `tab` de `n` éléments de taille `taille.`

`int  bsearch(void *ptr, void *tab,`
`size_t n, size_t taille,`
`int (*comp)(void *, void *));`
	recherche dichotomique d'un élément ayant l'adresse `ptr`, dans un tableau trié `tab` de `n` éléments de taille `taille`.

F.5.6. Communication avec l'environnement

Prototype de fonction	action
`void abort(void);`	provoque un arrêt anormal du programme.
`void exit(int etat);`	terminaison du programme; rend le contrôle au système en lui fournissant la valeur `etat` (la valeur 0 est considérée comme une fin normale).
`int  system(char *s);`	exécution de la commande système définie par la chaîne de caractères `s`.

F.6. Date et heure `<time.h>`

Plusieurs fonctions permettent d'obtenir la date et l'heure. Le temps est représenté par des objets de type `time_t` ou `clock_t`, lesquels correspondent généralement à des `int` ou à des `long int`.

Prototype de fonction	action
`time_t  time(time_t *tp);`	retourne le nombre de secondes écoulées depuis le 1*er* janvier 1970, 0 heures G.M.T. La valeur retournée est assignée à `*tp`.
`double  difftime(time_t t1, time_t t2);`	retourne la différence `t1 - t2` en secondes.
`char *  ctime(time_t *tp);`	convertit le temps système `*tp` en une chaîne de caractères explicitant la date et l'heure sous un format prédéterminé.
`Clock_t clock(void);`	retourne le temps CPU en microsecondes utilisé depuis le dernier appel à `clock`.

Annexe G : format de printf/fprintf/sprintf

Le format détaillé de ces fonctions est :

```
%[flag] [largeur_champ] [.[precision]] [l]caractere_conversion
```

Caractère de conversion	Signification
`d, i`	Affiche un entier signé en décimal
`o`	Affiche un entier en octal
`x, X`	Affiche un entier en hexadécimal
`u`	Affiche un entier non signé en décimal
`c`	Affiche un caractère
`s`	Affiche une chaîne de caractères
`f`	Affiche un `float` ou `double` en décimal avec 6 chiffres après la virgule sauf si une autre précision est indiquée.
`e, E`	Affiche un nombre réel en notation scientifique
`g, G`	Utilise le format `e`, `E` ou `f` selon l'exposant
`%`	Affiche le caractère `%`

L'`indicateur de précision` est constitué du point décimal (.) seul ou accompagné d'un nombre. Il ne s'applique qu'aux `e E f g G s`. Il indique le nombre de chiffres à afficher après la virgule, ou le nombre de caractères s'il est utilisé avec `s`. Le point décimal seul indique une précision de 0.

La largeur de champs indique le nombre minimum de caractères de la sortie. Cette largeur peut être représentée par :

- Un entier décimal ne commence pas par 0. La sortie sera complétée à gauche par des blancs.
- Un entier décimal commence par 0. La sortie sera complétée à gauche par des zéros.
- Le caractère `*` La valeur de l'argument suivant de type int sera interprétée comme la largeur de champs. Exemple : `int l=5, a ; printf("%*d", l, a);`

Des drapeaux (flag) peuvent être :

`-`	La sortie sera cadrée à gauche, c'est l'option par défaut.
`+`	Les nombres seront affichés avec leur signe.
`espace`	Si le premier caractère n'est pas un signe, place un espace au début.
`0`	La sortie sera complétée à gauche par des zéros.
`#`	Spécifie un format de sortie différent. Pour `o,` le premier chiffre sera `0.` Pour `x` ou `X`, le préfixe correspondant `0x` ou `0X` sera ajouté si le résultat n'est pas nul. Pour `e E f g` et `G`, la sortie comportera toujours un point décimal ; pour `f` et `G`, les zéros de terminaison seront conservés.

Annexe G : format de scanf/fscanf/sscanf

Le format de ces fonctions peut contenir :

- des espaces et des tabulations qui seront ignorés.
- des caractères ordinaires (différents de %) , dont chacun est supposé s'identifier au caractère suivant du flot d'entrée autre qu'un caractère d'espacement.
- des spécifications de conversion composé d'un %, d'un caractère facultatif de suppression d'affectation *, d'un nombre facultatif donnant la largeur maximum du champ, d'un champ facultatif de précision h, l ou L.

Caractère de conversion	Type de l'argument	Converti en
d	int *	entier sous forme décimale.
i	int *	entier. L'entier peut être sous forme octal s'il est précédé de 0, ou hexadécimal s'il est précédé par 0x ou 0X.
o	int *	entier sous forme octale.
x, X	int *	entier sous forme hexadécimale.
u	unsigned int *	entier non signé sous forme décimal.
c	char *	caractère.
s	char *	chaîne de caractères sans espace.
e, f, g	float *	nombre en virgule flottant.

Caractère de précision	Signification
h	Placé avant d, i, o, x ou u, il indique que l'argument est de type short *.
l	Placé avant d, i, o, x ou u, il indique que l'argument est de type long *. Placé devant e, f ou g, il indique que l'argument est de type double *.
L	Placé avant e, f ou g, il indique que l'argument est de type long double *.