



UNIVERSITÉ
CÔTE D'AZUR

Programmation impérative en C

Cours 4. UNIX, compléments sur les pointeurs

Olivier Baldellon

Courriel : `prénom.nom@univ-cotedazur.fr`

Page professionnelle : `https://upinfo.univ-cotedazur.fr/~obaldellon/`

LICENCE 2 — FACULTÉ DES SCIENCES ET INGÉNIERIE DE NICE — UNIVERSITÉ CÔTE D'AZUR

- ▶ QCM noté la semaine prochaine : **Jeudi 30 novembre à 10h30 en amph**
 - au programme : cours 1 à 3
 - ▶ types de bases
 - ▶ structures de contrôle (**if**, **while**, **for**)
 - ▶ Opérations binaires
 - ▶ tableaux et chaînes
 - ▶ pointeurs
 - ▶ structures, énumérations et unions
 - ▶ etc.

- ▶ Examen terminal : **Lundi 15 janvier à 8h00**

- 🍃 **Partie I. Entrées/Sorties**
- 🍃 Partie II. Interagir avec Unix
- 🍃 Partie III. Compléments sur les pointeurs
- 🍃 Partie IV. Divers
- 🍃 Partie V. Table des matières

- ▶ Un fichier est représenté par un pointeur vers le type `FILE`.
 - ▶ on parle de **descripteur de fichier**.
 - ▶ Il ne faut pas oublier de le fermer après utilisation.

```
FILE* fichier = fopen("fichier.txt","r");  
...  
fclose(fichier);
```

*.c

- ▶ Un fichier peut-être ouvert selon plusieurs modes.
 - Le mode lecture `"r"` (*read*)
 - ▶ si le fichier n'existe pas, `fopen` renvoie le pointeur `NULL`
 - Le mode écriture `"w"` (*write*)
 - ▶ si le fichier n'existe pas, `fopen` le crée.
 - ▶ sinon, il **supprime** son contenu.
 - Le mode écriture en ajout `"a"` (*add*)
 - ▶ si le fichier n'existe pas, `fopen` le crée.
 - ▶ sinon, il écrit à la suite du contenu préexistant.

- ▶ On peut utiliser deux fonctions de `stdlib` pour écrire dans un fichier :
 - un caractère unique : `fputc(char caractere, FILE* fichier);`
 - une chaîne de caractère : `fputs(char* chaine, FILE* fichier);`

```
FILE* fichier = fopen("fichier.txt","w");
fputs("Bonjour ",fichier);
fputs("à toi\n",fichier);
fputc('B',fichier);
fputc('o',fichier);
fputc('b',fichier);
fputc('\n',fichier);
fclose(fichier);
/* Oops, j'ai oublié de signer !*/
FILE* fichier = fopen("fichier.txt","a");
fputs("--\nSigné : M. le prof\n",fichier);
fclose(fichier);
```

*.C

```
Bonjour à toi
Bob
--
Signé: M. le prof
```

FICHIER.TXT

- ▶ Pour lire un fichier, on peut lire le parcourir caractère par caractère :
 - dans `stdlib` : `fgetc(FILE* fichier)`;
 - On s'arrête lorsque la fonction renvoie EOF (*end of file*).

```
Bonjour à toi
Bob
--
Signé : M. le prof
```

FICHER.TXT

```
void histoire_d_o(char* nom_fichier) {
    char c;
    FILE* fichier = fopen(nom_fichier,"r");
    while ( (c=fgetc(fichier)) != EOF) {
        fputc(c, stdout);
        if (c=='o') fputs("oooooo",stdout);
    }
    fclose(fichier);
}
```

*.C

```
Boooooonjoooooour à tooooooi
Boooooob
--
Signé : M. le proooooof
```

SHELL

- ▶ Pour lire un fichier, on peut le parcourir ligne par ligne :
 - `char* fgets(char* chaine, int taille, FILE* fichier);`
 - La fonction lit le texte et le copie dans un tampon (une chaîne).
 - ▶ jusqu'à obtenir un caractère `'\n'` ou `'\0'`
 - ▶ ou jusqu'à remplir le tampon.
 - ▶ elle renvoie NULL lorsque le texte est fini.

```
int taille_tampon = 100;
char* tampon = malloc(taille_tampon * sizeof(char));
FILE* fichier = fopen("àlire.txt", "r");
while ( fgets(tampon, taille_tampon, fichier) != NULL ){
    printf("[%s", tampon);
}
fclose(fichier);
```

*.c

- Ci dessous, l'affichage obtenue avec un tampon de taille 100 puis 5.

```
[Voici une première ligne
[voilà la seconde
```

SHELL

```
[Voici un[e pr[emi[ère [lign[e
[voilà l[à se[cond[e
```

SHELL

► Sous certain système d'exploitation non Unix, il faut préciser si l'on modifie un fichier binaire "rb" et "wb".

► `size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream)`

► `size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream)`

```
FILE *fichier_lecture = fopen("entree.bin", "rb");
char c;
fread(&c, sizeof(char), 1, fichier_lecture);

FILE *fichier_ecriture = fopen("sortie.bin", "wb");
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
fwrite(&i, sizeof(int), 5, fichier_ecriture);
```

*.C

- ▶ `printf` affiche sur la sortie standard

```
printf("%d,%d\n",12,23);
```

*.c

- ▶ `fprintf` affiche le formatage directement dans un fichier

```
FILE* fichier = fopen("fichier.txt","w");  
fprintf(fichier, "%d,%d\n",12,23);
```

*.c

- ▶ `fprints` affiche le formatage directement dans une chaîne
 - ▶ renvoie un entier : nombre de caractères écrits.
 - ▶ « utile » pour détecter (trop tard) un dépassement de tampon...

```
char chaine[100];  
int a;  
a = sprintf(chaine, "%d,%d\n",12,23);  
if (a>100) {  
    fprintf(stderr, "Oups, dépassement tampon\n");  
    exit(1);  
}
```

*.c

- ▶ Pour lire une variable sur l'entrée standard, on utilise la fonction `scanf`
 - ▶ mêmes identifiants que pour `printf` (sauf pour les flottants)
 - ▶ Les résultats sont stockés dans des pointeurs.
 - ▶ Renvoie le nombre de valeurs lues correctement.

	float	double	long double
Décimale	<code>%f</code>	<code>%lf</code>	<code>%Lf</code>
Scientifique	<code>%e</code>	<code>%le</code>	<code>%Le</code>

```
int ent; float flot; int n;
printf("Veuillez entrer un entier suivi d'un flottant :");
n = scanf("%d %f", &ent, &flot);
if (n==2) printf("Valeur lues : %d et %f\n", ent, flot);
```

*.c

- ▶ Fonctionne mal de manière interactive avec des entrées peu fiables.
- ▶ Pour l'instant on reste sur `lire_entier`

- ▶ Voici un programme ; en rouge, les valeurs entrées par l'utilisateur.

```
Entrez un nombre : 15
J'ai lu 15
Entrez un nombre : 17
J'ai lu 17
```

stdout

- ▶ Que se passe-t'il si on y écrit autre chose qu'un entier ?
 - ▶ Les autres appels à `scanf` feront absolument n'importe quoi...
 - ▶ C'est ce genre de petits détails qui rendent le C si attachant. ♥

```
Entrez un nombre : JAMAIS !
J'ai lu 0
Entrez un nombre : J'ai lu 0
```

stdout

- ▶ Une solution : faire appel à `vider_buffer` après chaque `scanf`
 - ▶ Prenez le temps pour admirez un instant l'écriture dense et absconse qui fait la fierté du vrai développeur C.

```
void vider_buffer() {
    int c;
    while((c=getchar()) != '\n' && c != EOF);
}
```

*.c

- ▶ Lire
 - Dans un fichier `fgetc`, `fgets` et `fscanf`
 - Sur l'entrée standard `getchar` et `scanf`
 - Depuis une chaîne de caractère `sscanf`
- ▶ Écrire
 - Dans un fichier `fputc`, `fputs` et `fprintf`
 - Sur l'entrée standard `putchar`, `puts` (ajoute `'\n'`) et `printf`
 - Dans une chaîne de caractère `sprintf`
- ▶ Le détail des fonctions se trouve dans la documentation.
- ▶ Ces fonctions font parties de la bibliothèque `stdlib.h`
- ▶ **INTERDICTION FORMELLE D'UTILISER `gets`!**

- 🍃 Partie I. Entrées/Sorties
- 🍃 Partie II. Interagir avec Unix
- 🍃 Partie III. Compléments sur les pointeurs
- 🍃 Partie IV. Divers
- 🍃 Partie V. Table des matières

- ▶ La documentation des fonctions standards du C (`stdlib`, `stdio`) sont accessibles via la commande `man`.
 - ▶ Attention : risque de conflit,
 - ▶ `man printf` donne la documentation de la fonction du shell
 - ▶ On peut rechercher avec `man 3 printf`.
 - ▶ Le 3 sert à préciser que l'on cherche une fonction d'une bibliothèque et non pas un exécutable.
- ▶ Pour les adorateurs de GNU/Emacs ❤️ : `M-x man`
 - ▶ `M-x woman` fonctionne aussi.

- ▶ La bibliothèque `stdlib` définit trois fichiers fort utiles :
 - `FILE * stdin`.
 - ▶ pour lire les données par l'utilisateur au clavier.
 - `FILE * stdout`.
 - ▶ pour afficher avec `printf`
 - `FILE * stderr`.
 - ▶ pour afficher des erreurs.
- ▶ En fait, les deux appels suivants sont équivalents :
 - `printf("%d\n", i)`
 - `fprintf(stdout, "%d\n", i)`

- ▶ Du point de vue du C,
 - ▶ aucunes différences entre les entrées au clavier et celles venant d'une pipe.
 - ▶ Autant utiliser des redirections, c'est plus rapide à tester.

```
#include <stdio.h>
int main(void) {
    int i,k;
    int somme = 0;
    for (i=0; i<2; i++) { /* Je lis deux entiers*/
        printf("entier> ");
        scanf("%d",&k);
        somme += k;
    }
    printf("la somme vaut %d\n",somme);
    return 0;
}
```

*.C

```
olivier@valrose:~ $ ./pipe_de_base
entier> 4
entier> 5
La somme vaut 9
olivier@valrose:~ $ seq 4 5 | ./pipe_de_base
entier> entier> La somme vaut 9
```

SHELL

- ▶ Comment utiliser proprement notre programme dans une pipe ?
 - ▶ `int isatty(int i)` de la bibliothèque `unistd.h` (=POSIX)
 - ▶ Renvoie vrai si l'argument est en mode interactif (*is a tty* \approx terminal)
 - ▶ 0 : `stdin` 1 : `stdout` 2 : `stderr`

```
if (isatty(0)) fprintf(stderr, "entier> ");  
  
scanf("%d",&k); k++; /* Je lis k et je l'incrémente */  
  
if (isatty(1)) printf("On obtient %d\n",k);  
else          printf("%d\n",k);
```

*.c

```
olivier@valrose:~ $ ./pipe  
entier> 12  
On obtient 13  
olivier@valrose:~ $ echo 20 | ./pipe  
On obtient 21  
olivier@valrose:~ $ echo 20 | ./pipe | head  
21  
olivier@valrose:~ $ X=$(./pipe)  
entier> 20  
olivier@valrose:~ $ echo $X  
21  
olivier@valrose:~ $ X=$(echo 11 | ./pipe | ./pipe | ./pipe)  
olivier@valrose:~ $ echo $X  
14
```

SHELL

```
int main(int argc, char* argv[]) {  
    int i;  
    for (i=0; i<argc; i++) {  
        printf("%d : %s\n",i, argv[i]);  
    }  
    return 0;  
}
```

*.C

```
olivier@valrose:~ $ ./moi -j "aime beaucoup" --la choucroute  
0 : ./moi  
1 : -j  
2 : aime beaucoup  
3 : --la  
4 : choucroute
```

SHELL

- ▶ Le code de retour de la fonction `main` permet de coder des booléens en shell.

```
int main(int argn, char* args[]) {  
    int nombre;  
    sscanf(args[1], "%d", &nombre);  
    if (nombre%2 == 0) { /*nombre pair*/  
        return 0;  
    } else {  
        return 1;  
    }  
}
```

*.C

```
olivier@valrose:~ $ if ./pair 3; then echo P; else echo I; fi  
I  
olivier@valrose:~ $ if ./pair 6; then echo P; else echo I; fi  
P  
olivier@valrose:~ $ ./pair 3 && echo C'est pair  
olivier@valrose:~ $ ./pair 6 && echo C'est pair  
C'est pair  
olivier@valrose:~ $ ./pair 3 || echo C'est impair  
C'est impair
```

SHELL

- ▶ Le code de retour permet aussi d'indiquer les erreurs.
 - ▶ 0 : pas d'erreur
 - ▶ ici 1 et 2 correspondent à deux erreurs distinctes.

```
int main(int argn, char* args[]) {
    int nombre, retour;
    if (argn<2) {
        printf(stderr,"Pas assez d'argument !\n");
        return 1;
    }
    retour = sscanf(args[1],"%d", &nombre);
    if (retour==0) {
        fprintf(stderr,"Ce n'est pas un nombre idiot !\n");
        return 2;
    }
    printf("%d\n",2*nombre);
    return 0;
}
```

*.C

```
olivier@valrose:~ $ ./double && echo ":)"
Pas assez d'argument !
olivier@valrose:~ $ ./double jeu && echo ":)"
Ce n'est pas un nombre idiot !
olivier@valrose:~ $ ./double 3 && echo ":)"
6
:)
```

SHELL

- ▶ Si une fonction rencontre un problème comment lever une exception ?
 - ▶ Et ben on peut pas !
 - ▶ On utilise pour cela le code de retour de la fonction.
- ▶ Quelques exemples (à connaître) :
 - ▶ `scanf` renvoie le nombre de valeurs lues (donc 0 en cas d'erreur).
 - ▶ `malloc` renvoie NULL s'il n'a pas réussi à allouer de la mémoire.
 - ▶ `getc` renvoie EOF en cas d'erreur ;
 - ▶ `fopen` renvoie NULL en cas d'erreur
- ▶ Il faut absolument lire la valeur d'erreur à chaque fois
- ▶ En C on ne prévient pas en cas d'erreur, c'est au codeur de se renseigner en regardant la valeur de retour de ses fonctions.

*.c

```
#include <stdio.h>
#include <stdlib.h>

void print_file(FILE *fichier){
    char c;
    while ( (c=getc(fichier)) != EOF) fputc(c,stdout);
}

int main(int argc, char* argv[]) {
    int i;
    FILE * fichiers;
    char * nom_fichier;
    if (argc==1) { /* argv[0] : nom du programme */
        print_file(stdin); /* stdin : entrée standard */
    } else {
        for (i=1; i<argc; i++) {
            nom_fichier = argv[i];
            if ((fichier=fopen(nom_fichier,"r")) == NULL) {
                fprintf(stderr,"Erreur lecture %s\n",nom_fichier);
                exit(1);
            }
            print_file(fichier);
            fclose(fichier);
        }
    }
    return 0;
}
```

- 🍃 Partie I. Entrées/Sorties
- 🍃 Partie II. Interagir avec Unix
- 🍃 Partie III. Compléments sur les pointeurs
- 🍃 Partie IV. Divers
- 🍃 Partie V. Table des matières

- ▶ Il existe un pointeur particulier en C : NULL
 - ▶ c'est le pointeur qui ne pointe jamais vers rien
 - ▶ utilisé comme code d'erreur pour les fonctions renvoyant un pointeur.

- ▶ En C, on peut définir des pointeurs de fonction.
 - `float (*vpf) (double, char*);`
 - ▶ déclaration de `vpf` comme étant une variable de type pointeur sur fonction prenant en paramètres un `double` et un `char*` et renvoyant un `float`

 - `char *f (int)`
 - ▶ déclaration de `f` comme étant une constante de type pointeur sur fonction prenant en paramètre un `int` et renvoyant un `char*`.

 - `typedef int (*tpf) (int);`
 - ▶ déclaration de `tpf` comme étant un type pointeur sur fonction prenant un `int` en paramètre et renvoyant un `int`.

*.C

```
void map(int t[], int n, int (*f) (int)) {
    int i;
    for (i=0; i<n; i++) {
        t[i] = f(t[i]);
    }
}

int doubler(int i) {
    return 2*i;
}

int incrementer(int i) {
    return i+1;
}

int main() {
    int tableau[] = {1, 2, 3};
    map(tableau,3,doubler);
    printf("%d %d %d\n", tableau[0], tableau[1], tableau[2]);
    /* affiche 2 4 6 */
    map(tableau,3,incrementer);
    printf("%d %d %d\n", tableau[0], tableau[1], tableau[2]);
    /* affiche 3 5 7 */

    return 0;
}
```

```
typedef struct arbre_struct {  
    struct arbre_struct* gauche;  
    struct arbre_struct* droite;  
    int racine;  
} arbre;  
  
int est_entier(arbre* a) {  
    return (a->gauche==NULL && a->droite==NULL);  
}
```

*.C

*.c

```
arbre* entier(int i) {
    arbre* a = malloc(sizeof(arbre));
    a->gauche = NULL;
    a->droite = NULL;
    a->racine = i;
    return a;
}

arbre* expression(int i, arbre* exp1, arbre* exp2) {
    arbre* a = malloc(sizeof(arbre));
    a->gauche = exp1;
    a->droite = exp2;
    a->racine = i;
    return a;
}
```

*.c

```
int calcul(int operation, int x, int y) {
    switch (operation) {
        case '+': return x+y;
        case '-': return x-y;
        case '*': return x*y;
        case '/': return x/y;
        default:
            fprintf(stderr, "Calcul non autorisé\n");
            exit(1);
    }
}

int valeur(arbre* expr) {
    int gauche;
    int droite;
    if (est_entier(expr)) {
        return expr->racine;
    } else {
        gauche = valeur(expr->gauche);
        droite = valeur(expr->droite);
        return calcul(expr->racine, gauche, droite);
    }
}
```

```
void bucheron(arbre* expr) {  
    if (est_entier(expr)) {  
        free(expr);  
    } else {  
        bucheron(expr->gauche);  
        bucheron(expr->droite);  
        free(expr);  
    }  
}
```

*.C

```
int main(void) {  
    arbre* e1 = entier(3);  
    arbre* e2 = entier(5);  
    arbre* a1 = expression('+', e1, e2);  
    arbre* a2 = expression('*', a1, entier(10));  
    printf("e1=%d\n", valeur(e1)); /* affiche 3 */  
    printf("e2=%d\n", valeur(e2)); /* affiche 5 */  
    printf("a1=%d\n", valeur(a1)); /* affiche 8 */  
    /* bucheron(a1); cause une erreur ligne suivante*/  
    printf("a2=%d\n", valeur(a2)); /* affiche 80 */  
    bucheron(a2);  
    return 0;  
}
```

*.c

- 🍃 Partie I. Entrées/Sorties
- 🍃 Partie II. Interagir avec Unix
- 🍃 Partie III. Compléments sur les pointeurs
- 🍃 **Partie IV. Divers**
- 🍃 Partie V. Table des matières

▶ Avant

```
all: exercice1 exercice2 exercice3

exercice1: exercice1.c
    gcc -Wall -pedantic -ansi exercice1.c -o exercice1

exercice2: exercice2.c
    gcc -Wall -pedantic -ansi exercice2.c -o exercice2

exercice3: exercice3.c
    gcc -Wall -pedantic -ansi exercice3.c -o exercice3
```

Makefile

▶ Maintenant, inutile de répéter la règle pour chaque exercice

- ▶ \$< correspond à la **source**
- ▶ \$@ correspond à la **destination**

```
CC = gcc
OPTIONS = -Wall -ansi -pedantic
all: exercice1 exercice2 exercice3

%: %.c
    $(CC) $(OPTIONS) $< -o $@
```

Makefile

- ▶ Le code suivant compile sans *warning* et s'exécute sans erreurs.
 - ▶ on accède à `tab[2]` sans l'initialiser.
 - ▶ on accède à `tab[3]` hors tableau.
 - ▶ On ne libère pas la mémoire.

```
int main(void) {  
    int i;  
    int * tab = malloc(sizeof(int)*3);  
    tab[0] = 10;  
    tab[1] = 11;  
    for (i=0; i<4; i++) {  
        printf("%d ", tab[i]);  
    }  
    printf("\n");  
    return 0; /*Affiche 10 11 0 0 */  
}
```

*.c

- ▶ L'utilitaire `valgrind` fait une analyse de l'exécution de votre code.
 - ▶ très lent
 - ▶ permet de trouver de nombreux bug!
 - ▶ `valgrind ./fuite`

* . C

```
==1112917== Memcheck, a memory error detector
[...]
==1112917== Command: ./fuite
[...]
==1112917== Use of uninitialised value of size 8
==1112917==   at 0x48BB9BB: _itoa_word (_itoa.c:177)
==1112917==   by 0x48C6D68: __vfprintf_internal (vfprintf-process-arg.c:164)
==1112917==   by 0x48BC56A: printf (printf.c:33)
==1112917==   by 0x1095E3: main (in /home/olivier/C/fuite)
[...]
==1112917== Invalid read of size 4
==1112917==   at 0x1095CC: main (in /home/olivier/C/fuite)
==1112917== Address 0x4a4e04c is 0 bytes after a block of size 12 alloc'd
==1112917==   at 0x48407B4: malloc (vg_replace_malloc.c:381)
==1112917==   by 0x109592: main (in /home/olivier/C/fuite)
10 11 0 0
==1112917== HEAP SUMMARY:
==1112917==   in use at exit: 12 bytes in 1 blocks
==1112917== total heap usage: 2 allocs, 1 frees, 1,036 bytes allocated
==1112917==
==1112917== LEAK SUMMARY:
==1112917==   definitely lost: 12 bytes in 1 blocks
==1112917==   indirectly lost: 0 bytes in 0 blocks
==1112917==   possibly lost: 0 bytes in 0 blocks
==1112917==   still reachable: 0 bytes in 0 blocks
==1112917==   suppressed: 0 bytes in 0 blocks
[...]
==1112917== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)
```

Merci pour votre attention

Questions



Hello world !



Cours 4 — UNIX, compléments sur les pointeurs

Annonces

Partie i. Entrées/Sorties

Fichier

Écrire dans un fichier

Lire dans un fichier (`fgetc`)

Lire dans un fichier (`fgets`)

Fichiers binaires et non-binaires

Affichage et formatage

La fonction `scanf`

Vider un tampon

Bilan

Partie ii. Interagir avec Unix

Unix et la documentation

Unix et les pipes

L'entrée standard

L'art de la pipe Unix

Donner des arguments à `main`

Codes de retour (shell)

Codes de retour (shell) : exemple

Codes d'erreur en C

Exemple : la commande `cat`

Partie iii. Compléments sur les pointeurs

Pointeurs et fonctions

Exemple

Créer un type arbres

Faire pousser des arbres

Travailler avec des arbres

Libérée, délivrée...

Utilisation

Partie iv. Divers

Makefile

Valgrind

Valgrind : exemple

Partie v. Table des matières