Rappel c

Les includes importent les libraries C dans le code en entete,

il existe plusieurs types de libraries:

<stdio.h>	Main, printf, scanf	<string.h>	Str(), strcpy ( ), strlen ( ),strcmp
<stdlib.h>	Allocation mémoire (malloc, free, realoc)	<math.h>	Cos(), sin(), sqrt(),
<assert.h>	Fonction assert, permet de verifier la condition	<ctype.h>	Tolower() , isspace() , isdigit()
<file.h>	Sscanf() etc...	<stddef.h>	size_t ()

Pour la compilation: «*gcc - Wall fichier.c -o fichier*» et «*gcc - Wall -lm fichier.c -o fichier*» pour les libaries
Pour l'execution: ./fichier

sizeof: int taille = sizeof(tab)/sizeof(tab[0]);

Dans un fonction la taille doit toujours être en parametre et preceder la tab)

```
int main(void){
  int tab[] = {1,2,3,4};
  int t =foo(tab);
}
int foo(tab [] tab){
 int taille = sizeof(tab)/sizeof(tab[0])
 return taille -1;
}
```

Erreur

> sizeOf() doit tjr être dans la main

> La taille du tab (int n) doit preceder le tab dans la fonction

>Utiliser size_t (un type unsigned)plutot que int moins de place en mémoire

```
int main(void){
  int tab[] = {1,2,3,4};
  size_t taille =sizeof(tab)/sizeof(tab[0])
  int t = foo(taille, tab);
}
int foo(int taille, tab [] tab){
 return taille -1;
}
  //cette fonction est inutile
  juste pour comprendre
```

- Les types non signés ne peuvent qu'avoir des valeurs >0. Different de entier signés (negatif / positf)
- 0 = false 0 =! true
- a = expr;

  a = b = c*d; a = (b = c*d) ;

  est la même chose que
- (condition) ? val1 : val 2 (True : False)

```
c = (a < b)? a-1 : b+2;
```

- Effet de bord

```
x = 7;
y = x ++;
```

incrementation de x après affectaction

```
printf("x=%d y=%d\n") ; /* affiche x=8 y=7 */
```

```
int x = 7, y ;
y = ++ x;
```

incrementation de x avant affectaction

```
printf("x=%d y=%d\n") ; /* affiche x=8 y=8 */
```

```
#include <stdio.h>

double somme(int nb, double tab[]);

int main(void){
  double tab[] = {-4.8, 6.1, 57.0, 23.99, -11.32, 4.5};
  int n = sizeof(tab)/sizeof(tab[i]);
  double s = somme(n, tab);
  printf("somme = %f\n",s);
}

double somme(int nb_elem, double t[]){
  double s=0;
  for(int i=0 ; i < nb_elem; i++)
    s += t[i];
  return s;
}
```

déclaration ou prototype de la fonction somme( ) pas de corps de fonction, juste les types de paramètres.

définition de la fonction somme()

Les noms de paramètres peuvent être quelconques (pas forcement les mêmes que dans la définition de la fonction).

#Define en C

```
#define  NB_ELEM  5

/* définition d'une constantes symbolique avec la
directive define, ce n'est pas une instruction, pas de
; à la fin */

int tab[ NB_ELEM ];
int s=0;

/* ici vient le code pour remplir tab */

for(int i = 0; i < NB_ELEM ; i++ ){
   s+=tab[i];
}
```

Enum

```
//Par défaut ici BLUE=0, RED=1 et GREEN=2
enum color{BLUE, RED, GREEN}; //création enum

//ici BLUE=1, RED=2 et GREEN=3
enum color{BLUE=1, RED=2, GREEN=3}; //création enum

//Sans typedef
enum color new_couleur; //nouvelle var de type enum color
new_couleur=RED; //affectation valeur

//Avec typedef
typedef enum color color;
color new_couleur;
new_couleur=RED
```

La seule utilisation de goto tolérée dans c c'est pour sortir d'une boucle imbriquée.

```
#include <stdio.h>
#include <limits.h>
int somme_ligne( int nb_l, int nb_c, int tab[nb_l][nb_c] ){
  int s = 0;
  int i;
  for( i = 0; i < nb_l; i++ ){
    for( int j = 0 ; j < nb_c ; j++ ){
      if( tab[i][j] < 0 )
      goto et;
    }
  }
  return INT_MAX;

et:
  for( int j = 0 ; j < nb_c ; j++ ){
    s += tab[i][j];
  }
```

SYMBOLE	TYPE	IMPRESSION COMME
%d ou %i	int	entier relatif
%u	int	entier naturel (unsigned)
%o	int	entier exprimé en octal
%x	int	entier exprimé en hexadécimal

%c	int	caractère
%f	double	rationnel en notation décimale
%e	double	rationnel en notation scientifique
%s	char*	chaîne de caractères

Les spécificateurs %d, %i, %u, %o, %x peuvent seulement représenter des valeurs du type int ou unsigned int. Une valeur trop grande pour être codée dans deux octets est coupée sans avertissement si nous utilisons %d.

Pour pouvoir traiter correctement les arguments du type long, il faut utiliser les spécificateurs %ld, %li, %lu, %lo, %lx.

Structure

Struct est comparable à un objet en java, il peut contenir des valeurs (int, char, nom etc..) signé/ non signé et meme d'autres struct

1.

```
struct point{
  int x;
  int y;
} ;
```

```
typedef struct point  point ;
```

le type — le nom "alias" du type struct point

typedef définit le nom alias "point" pour le type de données "struct point". A partir de ce moment on peut écrire "point" à la place de "struct point".

2.

Possible de définir une structure et le nom alias en même temps:

```
typedef struct point{
int x;
int y;
} point;
```

le type — le nom "alias" du type struct point

3. Il existe des struct dit anonymes qui sont souvent des struct imbriqués cad dans d'autres struc

```
struct Foo {
  struct {
    int hi;
    int bye;
  } bar;
};
```

Il y'a plusieurs maniere d'Initialiser une structure

1. initialiser l'element directement à la creation de la structure

```
struct point p ={ .x = 5, .y= 5}
```

2.

```
struct point p;
p.x = 5;
p.y = 5;
```

3.

```
struct point p;
```

~~p ={ .x = 5, .y= 5};~~ xInterdit

```
p= (struct point){ .x = 5, .y= 5};
```

il faut caster en struct point

On peut initialiser une structure dans une structure

```
typedef struct point{
  int x;
  int y;
} point ;

typedef struct rectangle{
    point pa;
    point pb;
} rectangle;

rectangle r = { .pa = { .x = 1, .y = 1 },
                .pb = { .x = 2, .y = 3} };

rectangle d;

d.pa.x = 5; d.pa.y = 5;
d.pb.x = 8; d.pb.y = 23;
```

pa — pb

les memes regles sont appliquables pour l'initlisation des struct ci dessus

Ces structures sont souvent anonyme et donc utilisable que dans le meme fichier .c

Une structure peut être appelé en parametre d'une fonction

```
void changePoint(point p, int x ){
  p.x = x;
}
```

Faux, Ici on ne retourne pas le point le changement n'est pas effectué

```
point changePoint(point p, int x ){
 Point pf = { .x = x, .y = p.y};
 return pf;
} // l'objet est retourné
```

Les pointeurs

Les pointeurs arithémtiques

Les pointeurs ont une adresse en mémoire cad qu'il pointe vers un type, il indique uniquement le type qu'il point

Pour initlisé un pointeur on peut faire type *a; (type: int, char etc..)

```
int *a;
```

ici a contient l'adresse de D

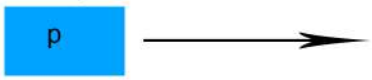
```
int d = 8;

a = &d;
```

Mais on peut changer la valeur en changeant plus l'adresse mais l'objet avec *pointeur;

```
*a = 12;  /* mettre la valeur 12 à l'adresse stockée
           * dans a */
```

Si un pointeur est null, et qu'on lui attribue une valeur alors erreur en mémoire

```
int *p = null
```

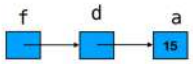


*p = 10; ERREUR FIN DU PROGRAMME

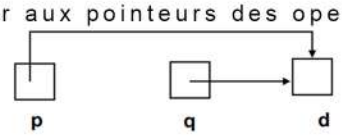
car aucune valuer n'est associé au pointeur en mémoire

on peut faire des pointeurs de pointeurs

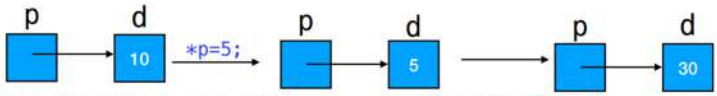
```
int a=15;
int *d;   /* d un pointeur vers un int   */
int **f;  /* f un pointeur vers un "int *"  */
d = &a;    f = &d;
```



On peut appliquer aux pointeurs des operation




```
int d;

int *p = &d;

d = 10;

*p = (*p) * 2 + 5;      /* d prend la valeur 2*10 + 5 = 25 */

*p += 3;      /* incrémenter de 3 la valeur stockée à l'adresse p;
               * d reçoit 28 */

++(*p) ; /*   incrémenter un int qui se trouve à l'adresse donnée
          *   par p, d == 29   ++ s'applique à la valeur qui se trouve
          *   à l'adresse p     */

int *q = p ;  /* les pointeurs p et q contiennent l'adresse de d */

(*q)++; /* (*q)++ augmente la valeur int à l'adresse q,
         *  d == 30 */
```



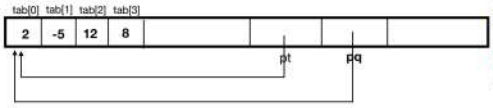
*p = *p + *p * *p

on conserve la derniere version de *p connue avant l'affectation (donc 5)

ici *p = 5 + 5*5 =30

Les pointeurs et les tableaux

C'est la meme chose




```
int *pt = &tab[0];
```
est equivalent
```
int *pq = &tab;
```
car on commence à t[0]

Operation pointeur = operation sur le tab

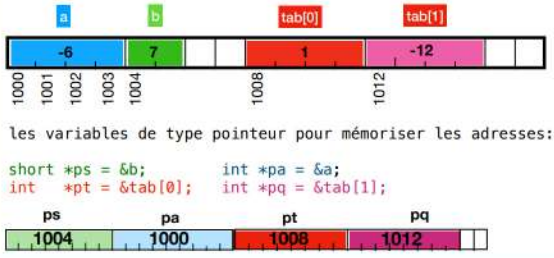
```
unsigned int tab[]={1,2,3,4,5,6,7,8};

unsigned int *p = &tab[3]; on commence le pointeur sur t[3];

*(p-1) = *(p+1) + *(p+2); ici on ajoute à t[2] = t[4] + t[5];
```

prendre un int qui se trouve à l'adresse **p+1** et un int qui se trouve à l'adresse **p+2**, additionner et mettre le résultat à l'adresse **p-1**

équivalent à : `p[-1] = p[1]+p[2];`

Les tableaux/pointeur en mémoire

```
int a; short b; int tab[] = {1,-12};

a = -6 ; b = 7;
```



les variables de type pointeur pour mémoriser les adresses:

```
short *ps = &b;        int *pa = &a;
int   *pt = &tab[0];   int *pq = &tab[1];
```


Les pointeurs de structure

on observe deux manieres d'initialiser

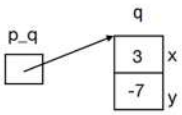
-> On créer l'objet On lui attribue un pointeur

->On attribue de la mémoire au pointeur

(*p).membre = p-> membre

(*p).x = 5; (c'est la meme) p->x = 5;

```
typedef struct {
   double x;
   double y;
}point;

point q = { .x = 3, .y =-7 };
point *p_q = & q
point *p;
p = malloc( sizeof(point) );
p->x = 2.8;  /* équivalent à  (*p).x = 2.8 */
p->y = -6.9;
p_q->x = p->x + p->y;
```



Les pointeurs génériques

```
int tab[]={3,4,5,6};

int *p = &tab[1];

void *t=p;  /* p et t contient la même adresse */

char *c=t;
```

Nous pouvons faire une affectation entre un pointeur générique et un autre pointeur sans projection de types, c'est-à-dire sans "cast"). C garantie que la valeur du pointeur est préservée par ces affectations.

A quoi sert le pointeur générique?

Arithmétique de pointeurs ne s'applique pas aux pointeurs génériques:

~~(t + 1) et (t - 1)~~

n'ont pas de sens si t un pointeur générique (déplacement de combien d'octets ? void n'est pas un type.)

L'application de l'opérateur * n'a pas de sens pour le pointeurs génériques :

~~int k = *t + 2;~~

Mémoire

*L'import de void *malloc(taille* sizeof(Type)), *calloc(), void *realloc(pointeur, taille), void free(pointeur)*

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

Utilisation de malloc

```
double *tab = malloc( n * sizeof(double) );
if(tab == NULL){   /* toujours vérifier si malloc() réussit */
   perror("malloc");
   exit(1);
}
```

`void perror( const char *s )` affiche un message d'erreur, à utiliser uniquement quand un appel fonction échoue, dans <stdio.h>

`void exit( int status )` termine l'exécution de programme avec le code status, dans <stdlib.h>

Utilisation de assert

```
assert( tab != NULL );
```

`assert( condition )` si la condition est fausse (dans le sens du C) alors l'affichage d'un message qui donne le nome de fichier source et la ligne dans le code, et le programme termine

En définissant la constante NDEBUG avant #include <assert.h> on désactive les assertions:

```
#define NDEBUG
#include <assert.h>
```

l'option -DNDEBUG à la compilation.

Utilisation de free

```
int *pointeur = malloc(int*sizeof(int));
free(pointeur);
```

Utilisation de realloc

realloc va copier les elements du pointeur dans un autre endroit de la mémoire, ajouter l'espace demander et supprimé l'ancien espace alloué

```
int *list_chiffre = malloc(10*sizeOf(int));
realloc(list_chiffre, taille_de_list_chiffre + 1);
```

Utilisation de calloc

`void *calloc(size_t nb_elem, size_t elsize)`

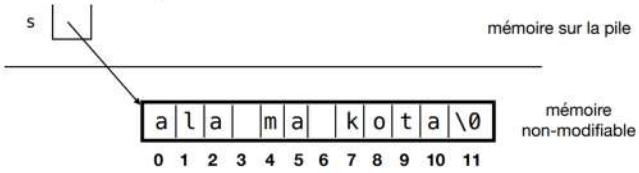
`calloc()` alloue un tableau de nb_elem éléments, chaque élément de taille elsize d'octets. De plus `calloc()` met à 0 tous les bits de la mémoire allouée. `calloc()` retourne l'adresse du premier octet de la mémoire allouée ou NULL en cas d'échec.

```
long *tab = calloc(100, sizeof(double));

/* tab - tableau de 100 éléments double
 * initialisés à 0                    */
```

Chaine de caractère en C

Char *s en mémoire

```
char *s ;
s = "ala ma kota";
```




```
printf("%c", s[4]);        OK, écrit :  m
printf("%s", s) ;          écrit  :  ala ma kota
printf("%s", s+4);         écrit  :  ma kota
printf("%s", s+9);         écrit  :  ta
```

```
s[4] = 'z';   ←   erreur d'exécution, tentative de modifier la mémoire non-modifiable (non-writable)
```

`sizeof(char) == 1`

Donc inutile de faire le calcul de taille avec sizeOf pour malloc par exempl

```
#include <ctype.h>
```

`int isalnum(int)`	lettre ou chiffre
`int isalpha(int)`	lettre
`int isascii(int)`	caractère ascii
`int isblank(int)`	'\t' ou ' ' et d'autres caractères blancs (dépend de la langue locale)
`int isdigit(int)`	un chiffre décimal de '0' à '9'
`int islower(int)`	lettre minuscule
`int isupper(int)`	lettre majuscule
`int isspace(int)`	un de caractères '\t', '\n', '\v', '\f', '\r', ' '
`int isxdigit(int)`	chiffre hexadécimal
`int ispunct(int)`	caractères imprimables, sauf l'espace, les lettres et chiffres

Chaine modifiable (important)

Il faut note que les deux chaînes sont placées par le compilateur dans une zone de mémoire non-modifiable, c'est-à-dire accessible uniquement en lecture. Mais les deux chaînes sont utilisées de façon très différente dans `char *s = "do mi";` et `char t[] = "halo";`.
Supposons que ce fragment de code fasse partie d'une fonction. Au moment de l'appel de la fonction, la variable `s` et le vecteur `t` sont placés sur la pile.
Dans `char *s = "do mi";` la variable `s` est initialisée avec l'adresse de la chaîne non-modifiable `"do mi"`.
Par contre pour initialiser le vecteur `t` les caractères de la chaîne non-modifiable `"halo"`, y compris le caractères nul, sont copiés dans le vecteur `t`. A partir de ce moment le vecteur `t` et la chaîne non-modifiable original `"helo"` n'ont plus rien à voir un avec l'autre.

=

```
char *s = "do mi" non modifiable
char t[] = "halo" modifiable
```

Il faut differencier char [] qui est modifiable car sur la pile et char * qui est non modifiable

Ne pas oublier le +1 pour le '\0' pour malloc()

```
char *s="first";
char *t="last";
//n'oubliez pas +1 pour l'octet avec null
char *res = malloc( strlen(s) + strlen(t) + 1);
strcpy(res, s); //copier d'abord s à l'adresse res
strcat(res, t); //ajouter t dans le string pointé par res
```

Initialiser des char // pas la meme modification

```
char t[]={ 'a', 'b', 'c', 'd' };
char *s = t;
char *u = &t[2];
```

Ni la suite de caractères pointée par `s` ni la suite pointé par `u` ne sont pas de chaînes de caractères, puisque il n y a pas de `'\0'` pour terminer la chaîne.

`const char *s` indique que la fonction ne modifie pas le string d'adresse `s`.
Par contre, quand le paramètre est sans `const`, la fonction peut modifier le string pointé par le paramètre.

3 Fonction essentiels

-size_t strlen(const char *s)

-int strcmp(const char *s, const char *t)

-char *strcpy(char *dest, const char *src)

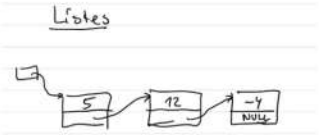
Exemple strlen() // subit opération arth.

```
char *s = "abcdef";
char  v[] = "rstuwxyz";
size_t i = strlen( s ); // i == 6
i = strlen( s + 2 );    // i == 4
i = strlen( &s[3] )     // i == 3
i = strlen( v );        // i == 8
i = strlen( &v[2] )     // i == 6
i = strlen( s + 6 )     // i == 0, s+6 est l'adresse de caractère nu
                        // termine la chaîne pointé par s
```

Liste chainée

Le meme principe que les tp de java, on recréait un system de liste avec une struct

Exemple de liste



Action sur les listes à connaitre: suppression(), insertion(), parcours() par exemple ici supp:

```
/* supprimer word de la liste. On ne suppose llus que la liste soit tréies
*/
void list_remove( list l, const char *word){
    list precedent = l;
    list courant = l->suivant;

    /* avancer tant que le mot courant différent du mot recherché
     * On ne supoose pas que la liste soit triée */
    while( courant != NULL && strcmp(courant->mot, word ) != 0 ){
        precedent = courant;
        courant = courant->suivant;
    }

    if( courant == NULL )
        return;

    /* supprimer courant */
    precedent -> suivant = courant -> suivant;
    free(courant);
    return;

}
```

Arbre bineaire

Un arbre binaire de recherche doit respecter cette propriété

pour chaque sommet s :

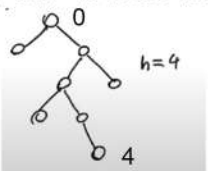
valeurs (sous-arbre gauche de s) < valeur (s) < valeurs (sous-arbre droit de s)

comme en EA

Exemple appel recursif

```
void arbre_afficher( arbre a ){
  if( a == NULL )
    return;
  printf("%s ", a->info);
  arbre_afficher( a->gauche );
  arbre_afficher( a->droit );
}
```

Hauteur = la profondeur maximal sur toutes les feuilles

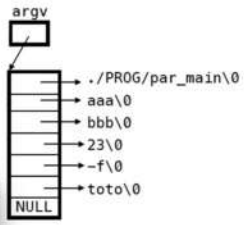


Arbre vide = (hauteur = -1)

Un arbre est une struct avec des infos
on simule un arbre. On peut comparer ça en Java avec les objets Noeud qui ont un fils Noeud G et D

Supposons que le programme compilé par_main suivant se trouve dans le répertoire PROG

```c
/*  par_main.c   */
#include <stdio.h>
int main(int argc, char *argv[]){

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




```
$ ./PROG/par_main aaa bbb 23 -f toto
./PROG/par_main
aaa
bbb
23
-f
toto
```

Main et exit

Les arguments compris dans argv sont les entrée du progra

A la position argv[0] il y'a toujours le chemin et le nom du programme

- Si return n'est pas spécifié dans le main, le progr s'arrete normalement, il considere l'operation comme un return 0 (uniquement possible dans le main).
- return 0 dans le main = exit(0)
- exit(0) = pas d'erreur, sinon exit(1...n) il s'agit d'une erreur

Ces fonctions convertissent

```c
#include <stdlib.h>
double   atof( const char *s ); // convertit s en double
int      atoi( const char *s ); // convertit s en int
long     atol( const char *s ); // convertit s en long
```

char *getenv(const char *name)

la fonction getenv() (stdlib.h) prend en paramètre le nom d'une variable d'environnement et retourne la valeur de cette variable.

Exemple : getenv("SHELL")
retourne sur mon portable "/bin/bash"

```c
/* variables.c */
#include <stdlib.h>
#include <std
int main(){
  char *var = getenv( "TOTO" );
  if( var == NULL )
    return 0;

  int d = atoi( var );
  for(int i = 0 ; i < d ; i++){
    printf("TOTO=%d ", d);
  }

}
```

```
PROG — -bash • Emacs — 31×6
$ TOTO=3 ./variables
TOTO=3 TOTO=3 TOTO=3 $
```

```
PROG — -bash • Emacs — 31×6
$ export TOTO=4
$ ./variables
TOTO=4 TOTO=4 TOTO=4 TOTO=4 $
```

Variable d'environnement enrengistrer dans le shell

```c
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
#include<string.h>

typedef struct pile_amortie {
    int occupation;
    int capacite;
    int *elements;
} pile_amortie;

void affiche_pile(pile_amortie *pile) {
    printf("capacite : %d\n", pile->capacite);
    printf("occupation : %d\n", pile->occupation);
    for (int i = 0; i<pile->occupation; i++) {
        printf("%d ", pile->elements[i]);
    }
    printf("\n\n");
}

pile_amortie *allouer_pile_amortie() {
    pile_amortie *pile = malloc(sizeof(pile_amortie));
    assert(pile != NULL);
    int *new_elements = malloc(sizeof(int));
    assert(new_elements != NULL);
    pile->occupation=0;
    pile->capacite=1;
    pile->elements=new_elements;
    return pile;
}

void libere_pile_amortie(pile_amortie *pile) {
    free(pile->elements);
    free(pile);
}

int empile_pile_amortie(pile_amortie *pile, int n) {
    if (pile->capacite == pile->occupation+1) {
    // Il faut etendre le tableau
        int *new_elements = realloc(pile->elements, (pile->capacite*2)*sizeof(int));
        // ne pas faire p = realloc(p, ...) car si realloc echoue, on perd l'adresse
        // de p et on a une fuite de memoire

        if (new_elements == NULL) {
            return -1; // l'allocation echoue
        }
        //free(pile->elements); // free l'ancien tableau
        pile->elements = new_elements; // ajouter le nouveau
        pile->capacite *= 2;
    }

    // ajouter l'element sur la pile
    pile->elements[pile->occupation] = n;
    pile->occupation ++;
    return 0;
}

int depile_pile_amortie(pile_amortie *pile, int *e) {
    if (pile->occupation <= 0) {
        return -1;
    }
    *e = pile->elements[pile->occupation -1];
    pile->occupation -= 1;

    if (pile->capacite > 2 && pile->occupation*4 <= pile->capacite) {
        int* new_elements = realloc(pile->elements, pile->capacite/2);
        if (new_elements == NULL) {
            return -1;
        }
        pile->elements = new_elements;
        pile->capacite = pile->capacite/2;
    }
    return 0;
}

pile_amortie *copie_pile_amortie(pile_amortie *pile) {
    pile_amortie *copie = malloc(sizeof(pile_amortie));
    int *new_elements = malloc(pile->capacite * sizeof(int));
    if (copie == NULL || new_elements == NULL) {
        return NULL;
    }
    memmove(new_elements, pile->elements, pile->capacite * sizeof(int));
    copie->elements = new_elements;
    copie->capacite = pile->capacite;
    copie->occupation = pile->occupation;
    return copie;
}

int main() {
    printf("---- ALLOUER ----\n");
    pile_amortie *pile = allouer_pile_amortie();
    affiche_pile(pile);
    printf("---- EMPILER ----\n");
    for (int i = 0; i<10; i++) {
        empile_pile_amortie(pile, i);
        affiche_pile(pile);
    }
    int e;
    printf("---- DEPILER ----\n");
    for (int i = 0; i<10; i++) {
        depile_pile_amortie(pile, &e);
        printf("depile : %d\n", e);
        affiche_pile(pile);
    }

    printf("---- LIBERER ----\n OK\nQ");
    libere_pile_amortie(pile);
    return 0;
}
```

Piles

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

typedef struct{
    size_t size;
    int content[];
} buffer;

void print_buffer(buffer *pb) {
    // afficher
    for (int i = 0 ; i<size ; i++) {
        printf("%s", buffer->content[i], " ");
    }
    printf("\n");
}

buffer *alloc_buffer(size_t size) {
    // allouer le buffer
    buffer *b = malloc(sizeof(buffer) + size * sizeof(int));
    b -> size = size;
    return b;
}

void write_buffer (buffer *pb, const char *file_name) {
    // écrire
    FILE *f = fopen(file_name, 'w');
    fwrite(pb, sizeof(buffer) + pb->size * sizeof(int), 1, file_name);
}

buffer *read_buffer (const char *file_name) {
    // lire
    buffer *b = sizeof(buffer);
    fread(b, file_name);
    return b;
}
```

Manipulation de fichiers

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

int copie(FILE*, FILE*);

int main(int argc, char const *argv[]) {

    FILE *src = fopen(argv[1],"r");
    assert(src != NULL);

    FILE *dst = fopen(argv[2],"w");
    assert(dst != NULL);

    copie(src,dst);

    fclose(src);
    fclose(dst);

    return 0;
}

int copie(FILE *fsrc, FILE *fdst) {

    while((int c = fgetc(fsrc)) != EOF) {
        int ret = fputc(c,fdst);
        if(ret == EOF)
            return -1;
    }
    return 0;
}
```

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

int main(int argc, char const *argv[]) {

    for(int i=1;i<argc;i++) {
    //ne pas copier caractere par caractere
        FILE *f = fopen(argv[i],"r");
        assert(f != NULL);
        int c = fgetc(f);
        while(c != EOF) {
            printf("%c",c);
            c = fgetc(f);
        }
        fclose(f);
    }

    return 0;
}
```

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

int main(int argc, char const *argv[]) {
    FILE *f = fopen(argv[1],"w");
    assert(f != NULL);
    for(int i=0;i<atoi(argv[2]);i++)
        fputs("une ligne\n",f);
    return 0;
}
```

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

int main(int argc, char const *argv[]) {

    int k = atoi(getenv("LIG"));

    for(int i=1;i<argc;i++) {
        FILE *f = fopen(argv[i],"r");
        assert(f != NULL);
        int c = fgetc(f);
        while(c != EOF) {
            printf("%c",c);
            c = fgetc(f);
        }
        fclose(f);
    }

    return 0;
}
```

commandes

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

typedef struct element element;
struct element {
    int val;
    element *previous;
    element *next;
};

element *const_list() {
    element *e = malloc(sizeof(element));
    assert(e != NULL);
    return e;
}

int isempty_list(element *L) {
    return L->next == NULL || L->previous == NULL;
}

void add_first_list(element *L, int v) {
    element *e = malloc(sizeof(element));
    assert(e != NULL);

    e->val = v;

    if (L->next == NULL) {
        e->next = L;
        e->previous = L;
        L->next = e;
        L->previous = e;
    } else {
        e->next = L->next;
        e->next->previous = e;
        e->previous = L;
        e->previous->next = e;
    }
}
```

```c
void add_last_list(element *L, int v) {
    if (isempty_list(L)) {
        add_first_list(L, v);
        return;
    }
    element *e = malloc(sizeof(element));
    assert(e != NULL);

    e->val = v;
    element *tmp = L->next;
    while (tmp->next != L) {
        tmp = tmp->next;
    }

    e->previous = tmp;
    e->previous->next = e;
    e->next = L;
    e->next->previous = e;
}


int len_list(element *L) {
    if (isempty_list(L)) return 0;
    element *tmp = L->next;
    int len = 0;
    while(tmp != L) {
        len += 1;
        tmp = tmp->next;
    }
    return len;
}

void print_list(element *L) {
    if (isempty_list(L)) {
        printf("list vide");
        return;
    }
    element *tmp = L->next;
    while(tmp != L) {
        printf("%d ", tmp->val);
        tmp = tmp->next;
    }
    printf("\n");
}
```

```c
int main(int argsc, char *argv[]) {
    element *L = const_list();
    element *M = const_list();

    for (int i = 1; i<argsc; i++) {
        int a = atoi(argv[i]);
        add_first_list(M, a);
        add_last_list(L, a);
    }

    printf("List L : \n");
    printf("longeur : %d\n", len_list(L));
    print_list(L);

    printf("\nList M : \n");
    print_list(M);
    printf("longeur : %d\n", len_list(M));

    printf("\nsuppr le premier el de M : \n");
    printf("supprime : %d\n", del_first_list(M));
    print_list(M);

    printf("\nsuppr le dernier el de M : \n");
    printf("supprime : %d\n", del_last_list(M));
    print_list(M);

    printf("\nsuppr le premier el de L : \n");
    printf("supprime : %d\n", del_first_list(L));
    print_list(L);

    printf("\nsuppr le dernier el de L : \n");
    printf("supprime : %d\n", del_last_list(L));
    print_list(L);

    printf("\nfree L\n");
    free_list(L);
    return 0;
}
```

```c
int del_first_list(element *L) {
    if (isempty_list(L)) {
        printf("Erreur : list vide\n");
        return 0;
    }

    element *e = L->next;
    int v = e->val;

    if (L->next->next != L) {
        L->next = L->next->next;
        L->next->previous = L;
    } else {
        L->next = NULL;
    }

    free(e);
    return v;
}

int del_last_list(element *L) {
    if (isempty_list(L)) {
        printf("Erreur : list vide\n");
        return 0;
    }

    element *e = L->previous;
    int v = e->val;
    if (L->previous->previous == L) {
        L->previous = NULL;
        L->next = NULL;
        printf("pb");
    } else {
        L->previous = L->previous->previous;
        L->previous->next = L;
    }

    free(e);
    return v;
}
```

```c
void free_list(element *L) {
    if(! isempty_list(L)) {
        element *tmp = L->next;
        element *suppr;
        while (tmp != L) {
            suppr = tmp;
            tmp = tmp->next;
            free(suppr);
        }
    }
    free(L);
}
```

Listes

```c
#include<assert.h>
#include "./afficheur.h"

node *const_tree(int val, node *left, node *right) {
    node *n = malloc(sizeof(node));
    assert(n != NULL);

    n->val = val;
    n->left = left;
    n->right = right;

    return n;
}

void free_tree(node *t) {
    if (t == NULL) return;
    free_tree(t->left);
    free_tree(t->right);
    free(t);
}

int size_tree(node *t) {
    if (t == NULL) return 0;
    return 1 + size_tree(t->left) + size_tree(t->right);
}

int sum_tree(node *t) {
    if (t == NULL) return 0;
    return t->val + sum_tree(t->left) + sum_tree(t->right);
}

int depth_tree(node *t) {
    if (t == NULL) return 0;
    int a = 1 + depth_tree(t->left);
    int b = 1 + depth_tree(t->right);
    if (a<b) {
        return b;
    } else {
        return a;
    }
}
```

```c
int check_abr(node *t) {
    if (t == NULL) return 1;
    else if (t->left != NULL && t->val < t->left->val) {
        return 0;
    } else if (t->right != NULL && t->val > t->right->val) {
        return 0;
    }
    int res = check_abr(t->left) + check_abr(t->right);
    return res == 2;
}
```

```c
int main() {
    node *t;

    t = const_tree(1,const_tree(3, NULL, NULL),
                   const_tree(6, const_tree(4, NULL, NULL), NULL));

    pretty_print(t);
    //free_tree(t);
    printf("---- EXERCICE 3 ----\n");
    printf("size of tree : %d\n", size_tree(t));
    printf("sum of all val : %d\n", sum_tree(t));
    printf("depth : %d\n", depth_tree(t));

    printf("\n---- EXERCICE 5 ----\n");
    node *n = NULL;
    int vals[10] = {8, 3, 1, 2, 6, 4, 7, 10, 14, 13};
    for (int i = 0; i<10; i++) {
        n = insert_abr(n, vals[i]);
    }
    pretty_print(n);
    printf("\n");
    print_abr(n);
    printf("\n");

    printf("\n---- EXERCICE 6 ----\n");
    node *s = search_abr(n, 6);
    printf("on trouve bien 6 : %d\n", s->val);

    printf("\n---- EXERCICE 7 ----\n");
    node *min = min_abr(n);
    node *max = max_abr(n);
    printf("max : %d\n", max->val);
    printf("min : %d\n", min->val);
    printf("ABR : %d\n", check_abr(n));
    printf("pas ABR : %d\n", check_abr(t));
    return 0;
}
```

```c
void print_abr(node *t) {
    if (t == NULL) return;
    print_abr(t->left);
    printf(" %d ", t->val);
    print_abr(t->right);
}

node *insert_abr(node *t, int val) {
    if (t == NULL) {
        t = const_tree(val, NULL, NULL);
        return t;
    }

    if (val < t->val) {
        t->left = insert_abr(t->left, val);
        return t;
    } else {
        t->right = insert_abr(t->right, val);
        return t;
    }
}

node *search_abr(node *t, int val) {
    if (t == NULL) return NULL;
    else if (t->val == val) return t;
    else if (val < t->val) return search_abr(t->left, val);
    else return search_abr(t->right, val);
}

node *max_abr(node *t) {
    if (t == NULL) return NULL;
    else if (t->right == NULL) return t;
    else return max_abr(t->right);
}

node *min_abr(node *t) {
    if (t == NULL) return NULL;
    else if (t->left == NULL) return t;
    else return min_abr(t->left);
}
```

Arbres

afficheur.h

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

typedef struct node node;
struct node {
  int val;
  node *left;
  node *right;
};

void print_head (int depth, int addr) {
    if (depth > 1) {
  int pre = addr / 2;
  print_head (depth - 1, pre);
  printf ("%s", (pre % 2) != (addr % 2) ? "| " : "  ");
  return;
    }
    if (depth == 1)
  printf ("--- ");
}

void pretty_rec (node *t, int depth, int addr) {
    if (t == NULL) {
  print_head (depth, addr);
  printf ("|----N\n");
  return;
    }
    pretty_rec (t -> right, depth + 1, 2 * addr  + 1);
    print_head (depth, addr);
    char c =
  (depth == 0) ? '-' : '|';
    printf ("%c----%d\n", c, t -> val);
    pretty_rec (t -> left, depth + 1, 2 * addr);
}

// fonction principale d'affichage.
void pretty_print (node *t) {
    pretty_rec (t, 0, 0);
}
```

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

typedef struct{
int x,y;
}paire;

int main() {
    int a ;
    void *pt = &a;
    *((int *)pt) = 42;
    *((int *)pt) = ((*((int *)pt)) * (*((int *)pt)));
    printf("%s\n", a);

    paire b;
    pt = &b;

    ((paire *)pt)->y+=1;
    return 0;
}
```

```c
#include "Func.h";

static void* decale(void* f, ssize_t d){
    return (char *)f + d;
}
```

```c
#include "MultiEnsemble.h"
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct node{
    int val;
    unsigned num;
    struct node *next;
};

mset new_node(int val, unsigned num) {
    node *n = malloc(sizeof(node));
    assert(n != NULL);
    n->val = val;
    n->num = num;
    n->next = NULL;
    return n;
}

mset add_val(int val, unsigned num, mset m) {

    if (m == NULL) {
        return new_node(val, num);
    }
    if (val == m->val) {
        m->num+=num;
        return m;
    }
    if (m->val > val) {
        mset newNode = new_node(val, num);
        newNode->next = m;
        return newNode;
    }
}

mset build(int *values, size_t size) {
    node *n
    for (int i = 0 ; i<size ; i++) {
        add_val(values[i], size, n);
    }
    return n;
}
```

```c
void print_mset(mset m, short verbose) {
    int cmp = 1;
    if (verbose == NULL) {
        for (int i = 0 ; i<m->num ; i++) {
            if (m->next->val == m->val) {
                cmp+=1;
            } else {
                printf("%s", m->val, "(", cmp, ") ");
                cmp = 1;
            }
        }
    } else {
        for (int i = 0 ; i<m->num ; i++) {
            printf("%s\n", m->val, " ");
        }
    }
}

int main() {
    int* vals = malloc(sizeof(int)*6);
    vals[0] = 5;
    vals[1] = 4;
    vals[2] = 7;
    vals[3] = 11;
    vals[4] = 1;
    vals[5] = 2;
    short v;
    print_mset(build(vals, 6), v);
    return 0;
}
```

```c
#include <stddef.h>

typedef struct node node;
typedef node* mset;


mset new_node(int val, unsigned num);
mset add_val(int val, unsigned num, mset m);
mset build(int *values, size_t size);
```

MultiEnsemble.h

Généricité

```c
#include "Func.h";

fifo create_fifo( size_t capacite_init, size_t taille_elem) {
    fifo f = malloc(sizeof(struct  file));
    assert(f != NULL);
    f->first = malloc(capacite_init * taille_elem);
    assert (first != NULL);
    f->occupe = f->first;
    f->libre = f->first;
    f->last = decale(f->first, capacite_init * taille_elem);
    return f;
}

void delete_fifo(fifo f) {
    free(f->first);
    free(f);
}

int empty_fifo(fifo f) {
    if (f->first == NULL) {
        return 1;
    } else {
        return 0;
    }
}

void *get_fifo(fifo f, void *element) {
    memmove(element, f->first, sizeof(f));
    f->occupe-=1;
}
```

```c
struct file{
void *first; /*pointeur debut de tableau*/
void *last; /*pointeur fin de tableau*/
size_t te; /*taille d'un element en octets*/
void *occupe; /*pointeur premier element de la file*/
void *libre; /*pointeur le premier element libre*/
};
typedef struct file *fifo;


fifo create_fifo( size_t capacite_init, size_t taille_elem);
void delete_fifo(fifo f);
```