Langage C

Arnaud Sangnier
sangnier@irif.fr

Tableaux de tableaux
Allocation dynamique

Retour sur les tableaux de tableaux

- Prenons un tableau de tableaux d'entiers (où chaque sous-tableau contient deux entiers

```
int tab[4][2]={{1,2},{10,20},{100,200},{1000,2000}} ;
```

- On veut faire une fonction qui affiche le contenu de tab avec chaque sous-tableau affiché sur une ligne

```
void affiche(int t[][],size_t t1,size_t t2){
  for(size_t i=0;i<t1;++i){
    for(size_t j=0;j<t2;++j){
      printf("%d ",t[i][j]);
    }
    printf("\n");
  }
}
```

Retour sur les tableaux de tableaux

- Prenons un tableau de tableaux d'entiers (où chaque sous-tableau contient deux entiers

```
int tab[4][2]={{1,2},{10,20},{100,200},{1000,2000}} ;
```

- On veut faire une fonction qui affiche le contenu de tab avec chaque sous-tableau affiché sur une ligne

```
void affiche(int t[][],size_t t1,size_t t2){
  for(size_t i=0;i<t1;++i){
    for(size_t j=0;j<t2;++j){
      printf("%d ",t[i][j]);
    }
    printf("\n");
  }
}
```

FAUX

- Il faut préciser dans l'en-tête de la fonction la taille des sous-tableaux

Retour sur les tableaux de tableaux

- Prenons un tableau de tableaux d'entiers (où chaque sous-tableau contient deux entiers)

```
int tab[4][2]={{1,2},{10,20},{100,200},{1000,2000}} ;
```

- On veut faire une fonction qui affiche le contenu de tab avec chaque sous-tableau affiché sur une ligne

```
void affiche(int t[][2],size_t t1){
  for(size_t i=0;i<t1;++i){
    for(size_t j=0;j<2;++j){
      printf("%d ",t[i][j]);
    }
    printf("\n");
  }
}
```

- **Il faut préciser dans l'en-tête de la fonction la taille des sous-tableaux**

Exemple

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

void affiche( int [][2],size_t);

int main(){
  int t[4][2]={{1,2},{10,20},{100,200},{1000,2000}};
  affiche(t,4);
  return EXIT_SUCCESS;
}

void affiche(int t[][2],size_t t1){
  for(size_t i=0;i<t1;++i){
    for(size_t j=0;j<2;++j){
      printf("%d ",t[i][j]);
    }
    printf("\n");
  }
}
```

tab-tab.c

Retour sur les tableaux de tableaux

- Mais du coup ici on passe un tableau en arguments d'une fonction

```
void affiche(int t[][2],size_t t1){
  for(size_t i=0;i<t1;++i){
    for(size_t j=0;j<2;++j){
      printf("%d ",t[i][j]);
    }
    printf("\n");
  }
}
```

- **Est-ce-que son type est int ** (un pointeur de pointeur) ?**
  - **NON !!!!!!!!!!**
- En fait il s'agit d'un pointeur vers un tableau de taille 2, le type de t s'écrit : **int (*) [2]** (ou dans la fonction int (* t)[2])
- Rappelez vous les tableaux ne sont pas des pointeurs
- **EN RÉSUMÉ :** évitez autant que faire se peut les tableaux à plusieurs dimensions, et préférez l'utilisation de l'allocation dynamique (prochain cours)

Exemple

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

void affiche(int (*)[2],size_t);

int main(){
  int t[4][2]={{1,2},{10,20},{100,200},{1000,2000}};
  affiche(t,4);
  return EXIT_SUCCESS;
}

void affiche(int (*t)[2],size_t t1){
  for(int (*p)[2]=t; p<t+t1;++p){
    for(int *p2=*p; p2<(*p)+2;++p2){
      printf("%d ",*p2);
    }
    printf("\n");
  }
}
```

tab-tab2.c

Exemple

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

void affiche(int (*)[2],size_t);

int main(){
  int t[4][2]={{1,2},{10,20},{100,200},{1000,2000}};
  affiche(t,4);
  return EXIT_SUCCESS;
}

void affiche(int (*t)[2],size_t t1){
  for(int (*p)[2]=t; p<t+t1;++p){
    printf("Valeur de p : %p\n",p);
    for(int *p2=*p; p2<(*p)+2;++p2){
      printf("Adresse de p2 : %p Valeur : %d \n",p2,*p2);
    }
    printf("\n");
  }
}
```

tab-tab3.c

Allocation de zone mémoire

- Comment faire une fonction qui crée l'équivalent d'un tableau d'entiers de 100 cases avec à la i-ème case, la valeur i ?
- Fausse bonne idée :

```
int *create_tab(){
  int tab[100];
  for(size_t i=0;i<100;++i){
    tab[i]=i;
  }
  int *tab2=tab;
  return tab2;
}
```

tab[0] = 1
2
3

- Ce comde compile sans problème
- MAIS le tableau créé se trouve dans la partie de la mémoire dédiée aux appels de fonctions (**la pile d'appels**)
- Quand on sortira de la fonction, **cette partie de la mémoire peut être réécrite**

Exemple

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

int *create_tab();

int main(){
  int *t=create_tab();
  for(int *ptr=t;ptr<t+100;++ptr){
    printf("%d \n",*ptr);
  }
  return EXIT_SUCCESS;
}

int *create_tab(){
  int tab[100];
  for(size_t i=0;i<100;++i){
    tab[i]=i;
  }
  int *t=tab;
  return t;
}
```

create-tab.c

Sur ma machine, quand je l'exécute les valeurs affichées ne sont pas celles attendues, par exemple : -519830016, 0, 16, 48, etc

Pile d'exécution vs Tas mémoire

- Quand une variable/un tableau est créé lors d'un appel à une fonction
  - Elle/Il est stocké dans la **pile d'exécution**
  - Elle/Il a duré de vie limité qui correspond à la durée de vie de la fonction
  - Ensuite elle peut être écrasé
- Donc, ce n'est pas une bonne idée de créer des variables/des tableaux dans des fonctions dont les emplacements mémoire vont ensuite être utilisées quand on sortira de la fonction !
- Comment faire autrement :
  - Il faut **allouer (réserver)** des zones dans **le tas mémoire**
  - Ces zones ne seront pas réécrites/supprimées automatiquement
  - il faudra cela dit les libérer nous-même

Allouer dynamiquement la mémoire

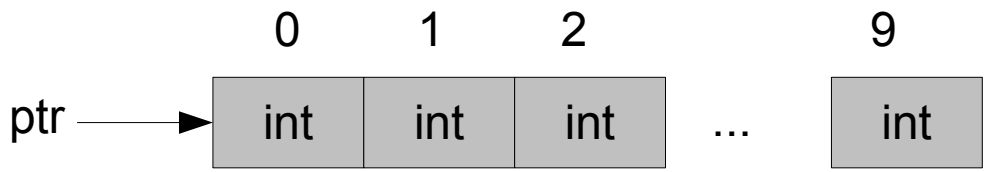
- La fonction
  - `void *malloc(size_t size)`
- Elle permet d'allouer size octets consécutifs dans la mémoire
- Son type de retour est void * pour dire qu'elle renvoie un pointeur
- Si quelque chose se passe mal,
  - la fonction renvoie le pointeur NULL
  - et elle met à jour la variable globale **errno** (voir plus loin)
  - l'allocation pourrait mal se passer si
- Si l'appel se déroule correctement, la fonction renvoie un pointeur vers une zone allouée dans le tas mémoire de size octets

Allouer dynamiquement la mémoire

- Exemple :

```
int *ptr=malloc(10*sizeof(int);
if(ptr!=NULL){
  //on peut accéder à la zone
}
```

- Si les choses se passent bien on obtient alors :



- Cette zone est dans le tas mémoire
- On peut la parcourir et y changer les données
- Le contenu de chaque ‘case’ **n’est pas initialisée**

Exemple

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

int *create(size_t);

int main(){
  int *t=create(20);
  if(t!=NULL){
    printf("[");
    for(int *ptr=t;ptr<t+20;++ptr){
      printf("%d",*ptr);
      if(ptr!=t+19){
        printf(", ");
      }
    }
    printf("]\n");
  }
}

int* create(size_t n){
  int *tab=malloc(n*sizeof(int));
  for(size_t i=0;i<n;++i){
    *(tab+i)=i;
  }
  return tab;
}
```

pointeur-simple.c

Errno et perror

- Certaines fonctions de librairies standards (voir leur page man), utilisent la variable globale errno pour mettre un code d'erreur en cas de mauvais fonctionnement
- Vous n'avez pas nécessairement à connaître ce code d'erreur
- Mais vous pouvez récupérer un message qui lui est associé grâce à la fonction :
  - **`void perror(const char *s)`**
- Cette fonction attend une chaîne de caractères
- Elle se trouve dans **<stdio.h>**
- Elle affichera la chaîne de caractères suivi du message lié à l'erreur

```
int *ptr=malloc(10*sizeof(int);
if(ptr==NULL){
  perror("Probleme d'allocation");
  exit(1);
}
```

- Quand ptr vaut NULL on sait que errno a été mis à jour
- L'appel à `exit(1)` fait terminer le programme avec 1 comme code de sortie

Exemple

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

int *create(size_t);

int main(){
  int *t=create(20);
  if(t!=NULL){
    printf("[");
    for(int *ptr=t;ptr<t+20;++ptr){
      printf("%d",*ptr);
      if(ptr!=t+19){
        printf(", ");
      }
    }
    printf("]\n");
  }
}

int* create(size_t n){
  int *tab=malloc(n*sizeof(int));
  if(tab==NULL){
    perror("Probleme create");
  }else{
    for(size_t i=0;i<n;++i){
      *(tab+i)=i;
    }
  }
  return tab;
}
```

pointeur-simple2.c

Libérer la mémoire

- Les zones allouées dynamiquement
  - sont situées dans le tas mémoire
  - et existent encore même si l'on sort de la fonction qui les a créé
- Il est **important** de les libérer grâce à la fonction
  - **`void free(void *ptr)`**
- Cette fonction libère la zone allouée pointée par le pointeur ptr
- **Si ptr vaut NULL cette fonction ne fait rien**
- Ceci permet d'**éviter les fuites mémoire** (*memory leak* en anglais)
  - Une zone allouée qui n'est plus pointée par aucune variable

```
int *ptr=malloc(10*sizeof(int);
ptr=malloc(10*sizeof(int);
```

  - Avec le deuxième malloc, plus aucune variable ne pointe sur la première zone ! Elle est réservée dans le tas mémoire et on ne pourra jamais l'effacer
  - Même si vous ne les voyez pas, les fuites mémoire sont une erreur grave de programmation

Exemple

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


int main(){
  unsigned d=0;
  while(true){
    int *t=malloc(1000000000*sizeof(int));
    if(t==NULL){
      perror("Probleme d'allocation");
      exit(1);
    }
    printf("Iteration %u\n",d);
    ++d;
  }
}
```

test-errno.c

Ici le programme alloue de plus en plus de mémoire qui n'est jamais libérée
On finit par avoir un problème avec le malloc qui échoue

Exemple

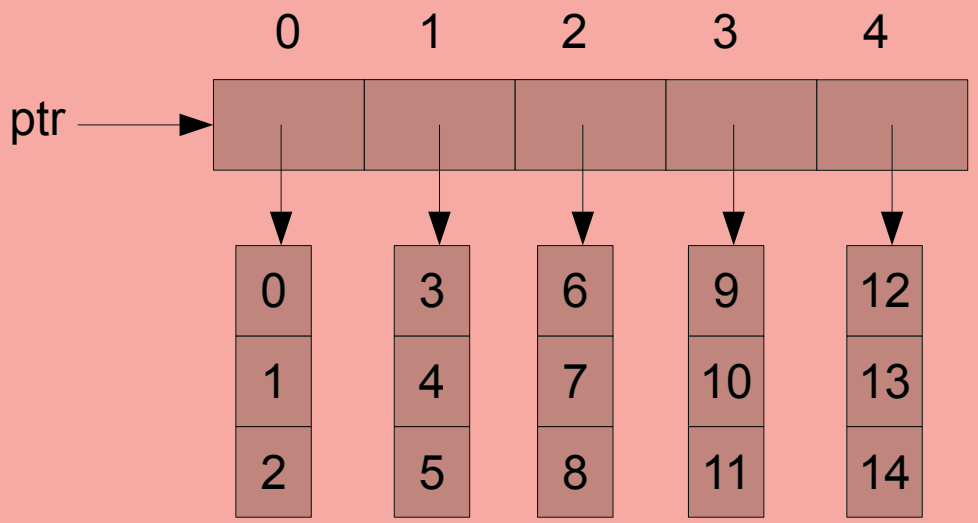
```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdbool.h>


int main(){
  unsigned d=0;
  while(true){
    int *t=malloc(1000000000*sizeof(int));
    if(t==NULL){
      perror("Probleme d'allocation");
      exit(1);
    }
    printf("Iteration %u\n",d);
    ++d;
    free(t);
  }
}
```

test-errno-free.c

Pointeur de pointeurs

- On a vu que les tableaux multi-dimensionnels étaient complexes à gérer
- MAIS on peut utiliser des pointeurs de pointeurs

```
int x=0;
int **ptr=malloc(5*sizeof(int*));
for(int **p=ptr;p<ptr+5;++p){
  *p=malloc(3*sizeof(int);
  for(int *p2=*p;p2<(*p)+3;++p2)){
    *p2=x;
    ++x;
  }
}
```



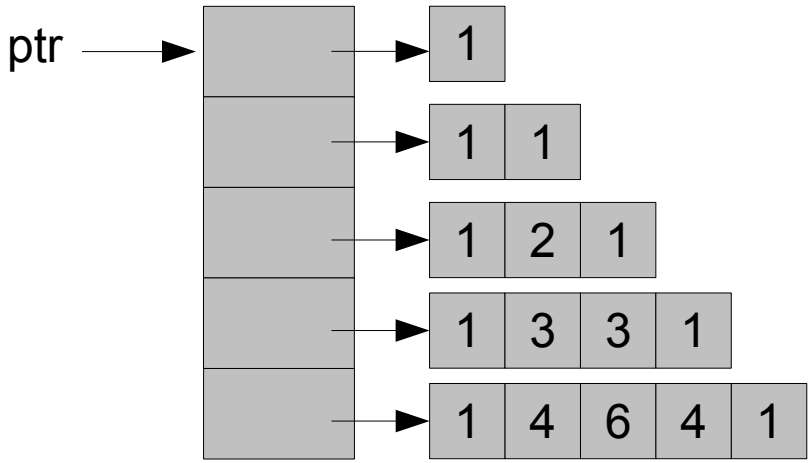
Exemple

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

int main(){
  int x=0;
  int **ptr=malloc(5*sizeof(int*));
  for(int **p=ptr;p<ptr+5;++p){
    *p=malloc(3*sizeof(int));
    for(int *p2=*p;p2<(*p)+3;++p2){
      *p2=x;
      ++x;
    }
  }
  for(size_t i=0;i<5;++i){
    for(size_t j=0;j<3;++j){
      printf("%d ",*(*(ptr+i)+j));
    }
    printf("\n");
  }
  for(int **p=ptr;p<ptr+5;++p){
    free(*p);
  }
  free(ptr);
}
```

pointeur-pointeur.c

Triangle de pascal

- On veut faire une fonction qui crée le triangle de Pascal de taille n>0
  - Il a n lignes
  - la ligne i contient i+1 éléments a[i,0], a[i,1],..., a[i,i] (pour i allant de 0 à n-1)
  - On a a[i+1,j] = a[i,j-1]+a[i,j] si j>0 et j<i+1 sinon a[i+1,j]=1 et a[0,0]=1



Exemple

```
unsigned **pascal(size_t n){
  if(n>0){
    unsigned **tp=malloc(n*sizeof(unsigned *));
    if(tp==NULL){
      return NULL;
    }
    for(size_t i=0;i<n;++i){
      *(tp+i)=malloc((i+1)*sizeof(unsigned));
      if(*(tp+i)==NULL){
        return NULL;
      }
      for(size_t j=0;j<=i;++j){
        if(i==0 || j==0 || j==i){
          *(*(tp+i)+j)=1;
        }else{
          *(*(tp+i)+j)=*(*(tp+i-1)+j-1)+*(*(tp+i-1)+j);
        }
      }
    }
    return tp;
  }else{
    return NULL;
  }
}
```

pascal.c

Exemple

```
void affiche_pascal(unsigned **tp,size_t n){
  for(size_t i=0;i<n;++i){
    for(size_t j=0;j<=i;++j){
      printf("%u ",*(*(tp+i)+j));
    }
    printf("\n");
  }
}

void libere_pascal(unsigned** tp,size_t n){
  for(size_t i=0;i<n;++i){
    free(*(tp+i));
  }
  free(tp);
}
```

pascal.c

Allouer la mémoire

- Autre fonction pour allouer la mémoire
  - **`void *calloc(size_t count, size_t size)`**
- Cette fonction alloue une zone dans la mémoire pour count objets de taille size (idem que malloc(count*size))
- En plus, elle initialise toute la zone allouée avec des 0

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

- malloc vs calloc
  - à cause de l'initialisation à 0, calloc peut prendre plus de temps
  - si on remplit la zone allouée juste après l'allocation, mieux vaut faire malloc
- Les zones allouées avec calloc doivent aussi être libérées avec free

Exemple

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

int main(){
  unsigned *t=malloc(10*sizeof(unsigned));
  if(t==NULL){
    perror("Probleme malloc");s
    exit(1);
  }
  for(unsigned *ptr=t;ptr<t+10;++ptr){
    printf("%u ",*ptr);
  }
  printf("\n");
  free(t);
  t=calloc(10,sizeof(unsigned));
  if(t==NULL){
    perror("Probleme calloc");
    exit(1);
  }
  for(unsigned *ptr=t;ptr<t+10;++ptr){
    printf("%u ",*ptr);
  }
  printf("\n");
  free(t) ;
}
```

malloc-calloc.c

```
>../malloc-calloc
0 1610612736 0 1610612736 2411200528 32767 1475776918 32767 2411217256 32767
0 0 0 0 0 0 0 0 0 0
```

Changer la taille de la zone allouée

- Il est possible de modifier la taille de la zone allouée
  - `void *calloc(void *ptr, size_t size)`
- Cette fonction ré-alloue une zone dans la mémoire de taille size octets
- ptr est un pointeur vers la zone allouée originale
- La fonction renvoie un pointeur vers la nouvelle zone si tout s'est bien passé et NULL sinon
- Comment elle fonctionne :
  - soit elle change la taille et ne déplace rien
  - soit elle déplace la zone pointée par ptr puis elle l'étend et **elle libère l'ancienne zone** pointée
- Cela permet soit d'augmenter la taille d'une zone mais aussi de diminuer pour libère de la place en mémoire
- En quand de déplacement, les données dans la zone sont recopiées
- Quand on n'a plus besoin de la zone allouée, il faut là aussi faire un free

Exemple

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

int main(){
  int *t=malloc(5*sizeof(int));
  assert(t!=NULL);
  int x=1;
  for(int *ptr=t;ptr<t+5;++ptr){
    *ptr=x;
    x*=2;
  }
  for(int *ptr=t;ptr<t+5;++ptr){
    printf("Adresse : %p Donnee : %d\n",ptr,*ptr);
  }
  puts("_____________________________________");
  int *t2=realloc(t,10*sizeof(int));
  assert(t2!=NULL);
  for(int *ptr=t2+5;ptr<t2+10;++ptr){
    *ptr=x;
    x*=2;
  }
  for(int *ptr=t2;ptr<t2+10;++ptr){
    printf("Adresse : %p Donnee : %d\n",ptr,*ptr);
  }
  free(t2);
  return EXIT_SUCCESS;
}
```

realloc.c

Exemple

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

int main(){
  int *t=malloc(5*sizeof(int));
  assert(t!=NULL);
  int x=1;
  for(int *ptr=t;ptr<t+5;++ptr){
    *ptr=x;
    x*=2;
  }
  for(int *ptr=t;ptr<t+5;++ptr){
    printf("Adresse : %p Donnee : %d\n",ptr,*ptr);
  }
  int *tab=malloc(t,100*sizeof(int));
  puts("_____");
  int *t2=realloc(t,10*sizeof(int));
  assert(t2!=NULL);
  for(int *ptr=t2+5;ptr<t2+10;++ptr){
    *ptr=x;
    x*=2;
  }
  for(int *ptr=t2;ptr<t2+10;++ptr){
    printf("Adresse : %p Donnee : %d\n",ptr,*ptr);
  }
  free(t2);
  return EXIT_SUCCESS;
}
```

realloc2.c

Ici, il y a des chances que la zone soit déplacée !

Copier des zones mémoire

- On peut copier des zones mémoire, et on a deux fonctions (elle sont **dans <string.h>**) :
  - `void *memmove(void *dst, const void *src,size_t len)`
  - `void *memcpy(void *dst, const void *src,size_t len)`
- Elles copie toutes les deux len octets de src vers dst
- **Les deux zones pointées doivent être allouées et de la bonne taille** (inférieure ou égale à len)
- Elles renvoient toutes les deux dst
- Différence :
  - Pour memcpy, les deux zones pointées par dst et src ne doivent pas se chevaucher !

Exemple

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

int main(){
  int *t=malloc(5*sizeof(int));
  assert(t!=NULL);
  int x=1;
  for(int *ptr=t;ptr<t+5;++ptr){
    *ptr=x;
    x*=2;
  }
  int *t2=malloc(5*sizeof(int));
  for(int *ptr=t2;ptr<t2+5;++ptr){
    printf("Adresse : %p Donnee : %d\n",ptr,*ptr);
  }
  puts("------------------------------");
  memcpy(t2,t,5*sizeof(int));
  for(int *ptr=t2;ptr<t2+5;++ptr){
    printf("Adresse : %p Donnee : %d\n",ptr,*ptr);
  }
  free(t);
  free(t2);
  return EXIT_SUCCESS;
}
```

copy.c

t et t2 pointet bien vers des zones différentes !!

Décaler les cases d'un tableau

- On veut décaler les cases d'un 'tableau' d'une case vers la droite
- Et la première case prend la valeur de la dernière case

```
void  shift(int *t,size_t l){
  assert(l>1);
  int tmp=*(t+l-1);
  memmove(t+1,t,(l-1)*sizeof(int));
  *t=tmp;
}
```

- Ici les deux zone pointées par t+1 et t se chevauchent
- On utilise donc memmove

Exemple

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

void shift(int *,size_t);

int main(){
  int *t=malloc(5*sizeof(int));
  assert(t!=NULL);
  int x=1;
  for(int *ptr=t;ptr<t+5;++ptr){
    *ptr=x;
    x*=2;
  }
  shift(t,5);
  for(int *ptr=t;ptr<t+5;++ptr){
    printf("Adresse : %p Donnee : %d\n",ptr,*ptr);
  };
  return EXIT_SUCCESS;
}

void  shift(int *t,size_t l){
  assert(l>=1);
  int tmp=*(t+l-1);
  memmove(t+1,t,(l-1)*sizeof(int));
  *t=tmp;
}
```

shift.c

++exemple

- On veut programmer une pile d'entiers à l'aide de tableaux
- Pour cela on va se servir de la structure suivante :

```
struct pile{
  size_t pos;
  size_t taille;
  int *data;
};

typedef struct pile pile;
```

- pos indique la position où empiler et taille la taille du tableau
- On va vouloir faire des fonctions pour intialiser la pile, empiler, dépiler et libérer l'espace qu'elle occupe
- Si on empile trop de données, il faudra faire croire la taille du tableau et si à un moment la pile est trop grande par rapport au nombre de données on pourra réduire sa taille

Initialisation

-
```
struct pile{
  size_t pos;
  size_t taille;
  int *data;
};

typedef struct pile pile;
```

```
pile *init_pile(pile *p){
  if(p==NULL){
    return NULL;
  }else{
    p->pos=0;
    p->taille=10;
    p->data=malloc(10*sizeof(int));
  }
  return p;
}
```

Pop and push

```
void push(pile *p,int d){
  if(p->pos>=p->taille){
     p->taille+=10;
     p->data=realloc(p->data,(p->taille)*sizeof(int));
  }
  *((p->data)+(p->pos))=d;
  ++(p->pos);
}

int *pop(pile *p,int *d){
  if(p->pos==0){
    return NULL;
  }else{
    *d=*((p->data)+(p->pos)-1);
    (p->pos)-=1;
    return d;
  }
}
```