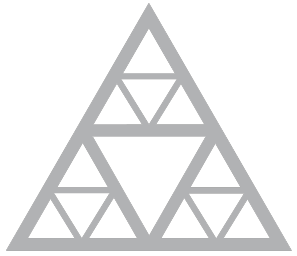
Introduction à la programmation en C++

Allocation dynamique

Nicolas Audebert

Vendredi 18 octobre 2018



Avant toute chose

Rendus de TP et des exercices

Les rendus se font sur Educnet.

1. Le code rendu **doit compiler**.
2. Le code rendu doit **être propre** (indentation, noms de variables clairs).
3. Le code rendu doit **être commenté** (réponses aux questions, fonctionnement du code).
4. Rassembler le code dans une seule archive (`.zip`, `.rar`, `.tar.gz`, etc.).

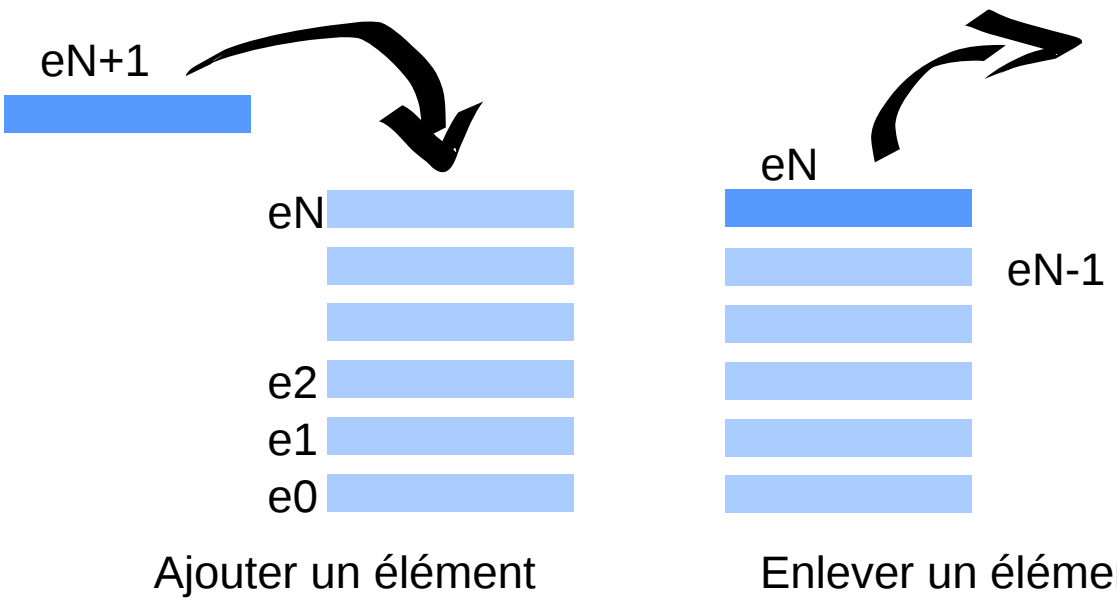
Plan de la séance

Pile des fonctions

Les appels aux fonctions sont gérés à l'aide d'une pile.

- **Entrer dans une fonction** : ajouter un élément à la pile
- **Sortir d'une fonction** : enlever un élément à la pile

La pile des fonctions permet de garder en mémoire l'ordre d'appel des fonctions. Chaque étage de la pile contient un **contexte d'exécution**. Ce mécanisme permet l'utilisation de **fonctions récursives**.



Exemple - Calcul de $n!$

Expression mathématique sous forme récursive

$$fact(n) = \begin{cases} n \times fact(n-1)\ , & \text{si } n > 0 \\ 0\ , & \text{sinon.} \end{cases}$$

Implémentation récursive en C++

```
int fact(int n){
    if(n <= 0){
        // Cas de base
        return 1;
    } else {
        // Formule de récurrence
        return n * fact(n - 1);
    }
}
```

Tableaux dynamiques

Le tas

Le tas est une autre zone mémoire (différente de la pile) qui est accessible dynamiquement à la demande du programme.

```
int taille = 1e7; // taille pas forcément constante

int* tab = new int[taille]; // réserve la place dans le tas
// utilisation du tableau comme un tableau classique
...
delete[] tab; // désalloue la mémoire occupée dans le tas
```

- La taille du tableau n'a pas à être connu au moment de la compilation,
- Le tableau peut changer de taille au cours du programme,
- Il ne faut pas oublier le `delete [] nom` .

Plan de la séance

Tableaux 2D

Les tableaux 2D de taille constante sont autorisés en C++.

```
// Déclaration d'un tableau 2D
double tab2D[5][3];
// Accès aux éléments
for(int i=0; i<5; i++){
    for(int j=0; j<3; j++){
        tab2D[i][j] = i*j;
        // ATTENTION : pas de "tab2D[i,j]"
        cout << tab2D[i][j] << " ";
    }
    cout << endl;
}
// Initialisation
int t2D[2][3] = {{1,2,3},{4,5,6}};
```

En fait, `int t2D[2][3];` est un tableau de tableaux : `t2D[0]` et `t2D[1]` sont des tableaux de 3 cases.

Tableaux 2D et fonctions

On peut utiliser les tableaux 2D dans les fonctions, mais il faut en spécifier les dimensions dans la signature de la fonction :

```
void init(int t[2][3], int val){
// Passage implicite par référence
    for(int i=0; i<2; i++){
        for(int j=0; j<3; j++){
            t[i][j] = val;
        }
    }
}

void f(){
    int tab[2][3];
    init(tab, 0); // appel de la fonction sur la variable tab
}
```

Limitations

Fonctions génériques

Cas 1D : il est possible de faire des fonctions génériques

```
void init(int t[], int taille, int val){
    for(int i=0; i<taille; i++){
        t[i] = val;
    }
}
```

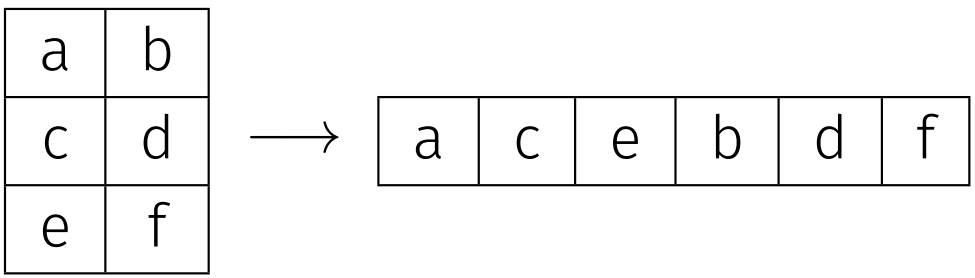
Cas 2D : ce n'est pas possible

```
void init(int t[][], int rows, int cols, int val){...} //ERREUR
```

→ Réécriture de code ? une fonction pour chaque taille de tableau ?

Solution

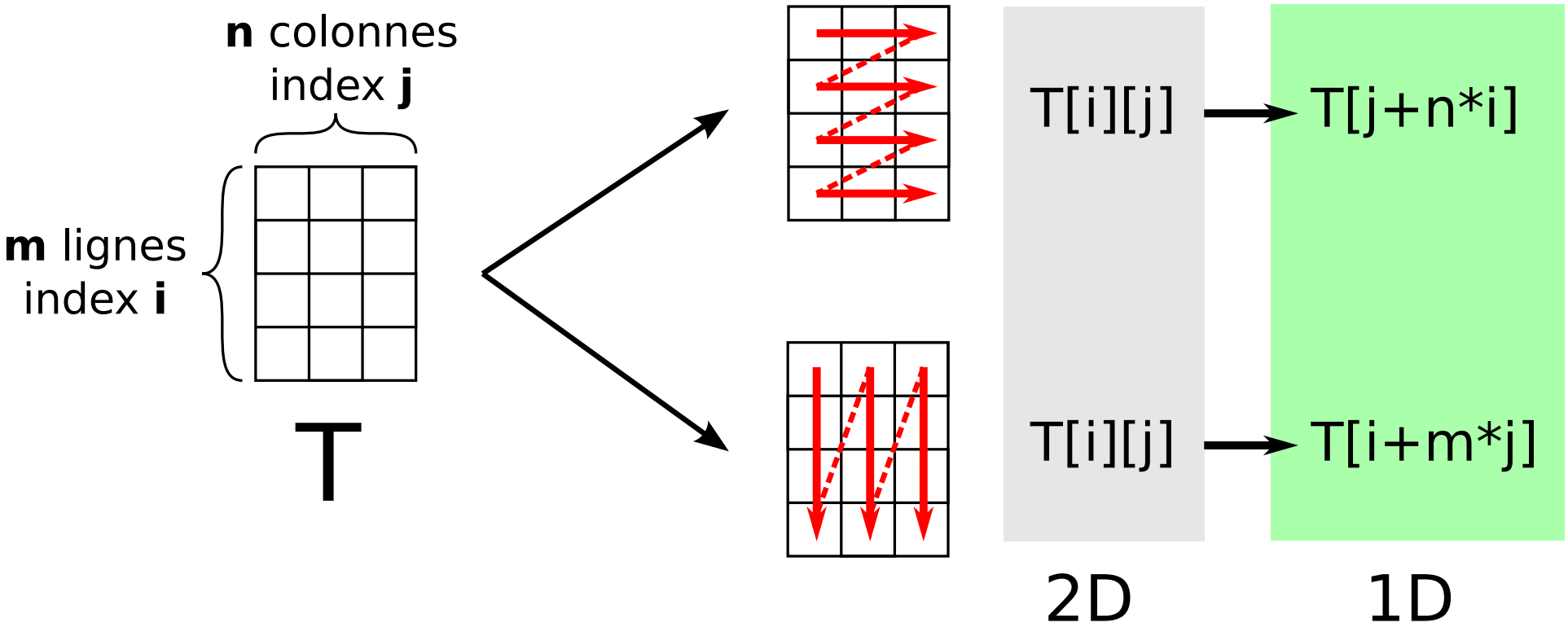
On utilise des toujours des tableaux à 1 dimension.



Cette solution permet de gérer autant de dimension qu'on le souhaite.

Parcourir un tableau 2D $\rightarrow$ 1D

On utilise soit le **parcours en lignes**, soit le **parcours en colonnes**.



Utilisation dans les fonctions

Il est désormais possible d'utiliser des fonctions génériques.

```
double fill(double mat[], int rows, int cols, double val){
    for(int i=0; i<rows; i++){
        for(int j=0; j<cols; j++){
            mat[j+cols*i] = val;
        }
    }
}

void prod_mat_vec(double mat[], int rows, int cols, double vec[],
                                                    double sol[]){
    for(int i=0; i<rows; i++){
        sol[i] = 0;
        for(int j=0; j<cols; j++){
            sol[i] += mat[j+cols*i]*vec[j];
        }
    }
}
```

Plan de la séance

Tableaux de taille variable

Allocation dynamique et tableaux 2D

Pas de possibilité de faire des tableaux 2D avec allocation dynamique (tableaux de taille variable).

Solution

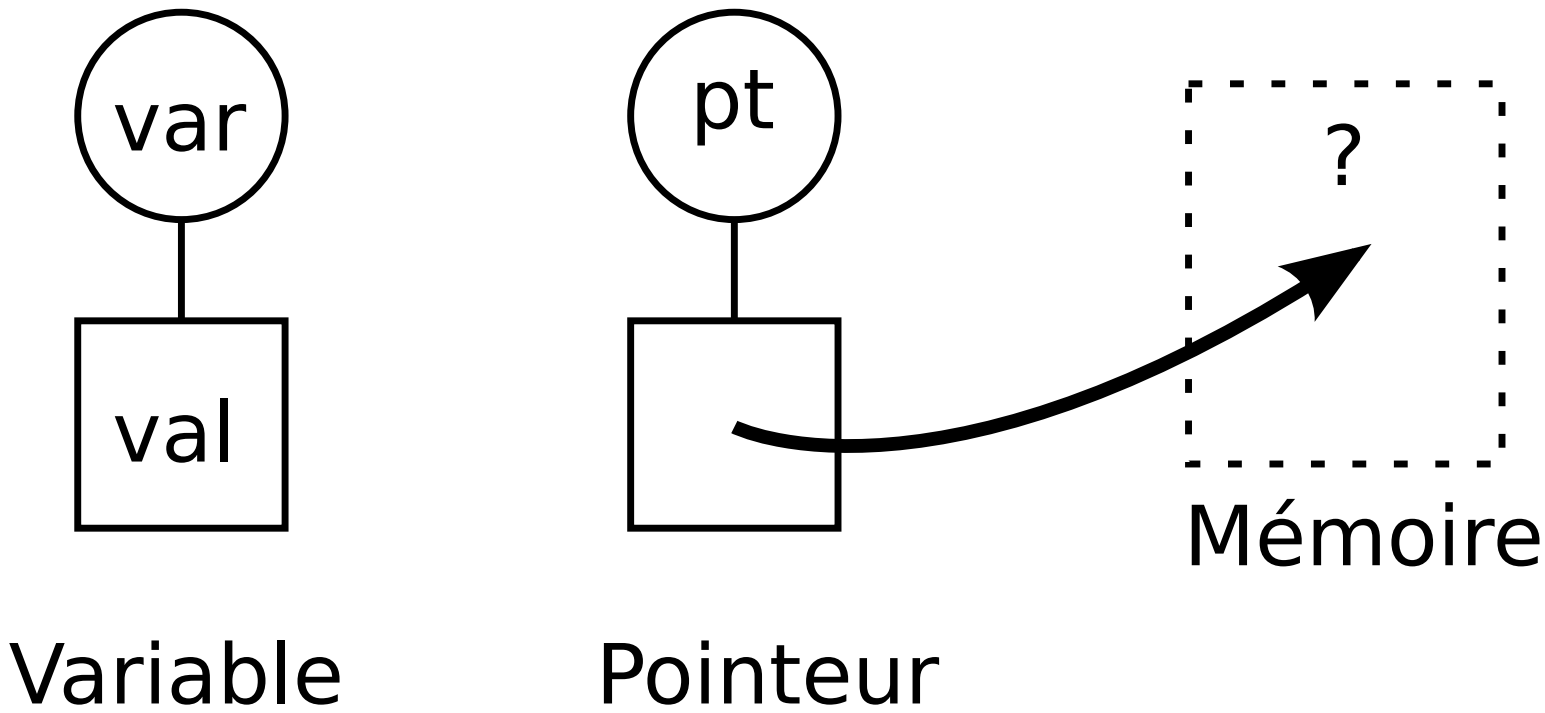
On fait des tableaux 1D, comme précédemment.

```
int m = ... ;
int n = ... ;
double* A = new double[m*n];
double* x = new double[m];
double* y = new double[n];
...
void prod_mat_vec(A,m,n,x,y);
...
delete[] A;
delete[] x;
delete[] y;
```

Pointeurs

Définition

Un **pointeur** est une variable qui stocke une adresse vers une zone mémoire (tableau ou variable) dans la pile ou dans le **tas**.



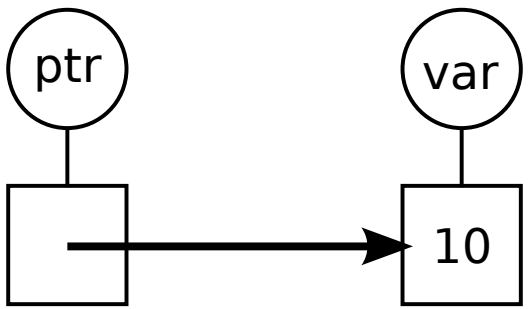
Déclarer un pointeur

On utilise le caractère `*`.

```
int* ptr; // un pointeur vers un entier
```

Pour récupérer l'adresse d'une variable on utilise le `&`

```
int* ptr; // un pointeur vers un entier
int test = 10;
ptr = &test; // le pointeur redirige vers test
```



Pointeurs et mémoire

L'intérêt d'utiliser des pointeurs avec des variables classiques est limité.

Des pointeurs pour le tas

Les pointeurs sont la porte d'entrée vers le tas (la mémoire de l'ordinateur).

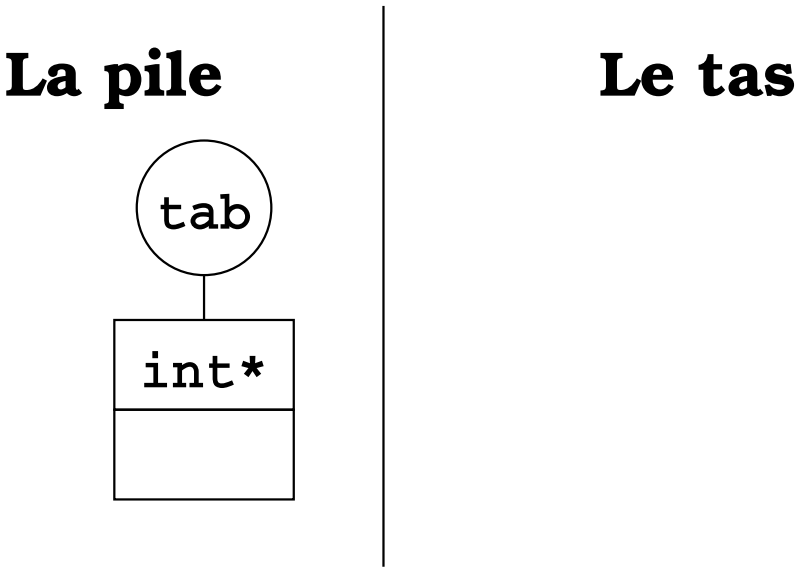
- Créer une variable dans le tas : **`new`**
- Supprimer une variable dans le tas : **`delete`**

Pointeurs et tableaux

```
double* tab;
int n=5;
tab = new double[n];

for(int i=0; i<n; i++){
    tab[i] = 2*i;
}

delete[] tab;
```

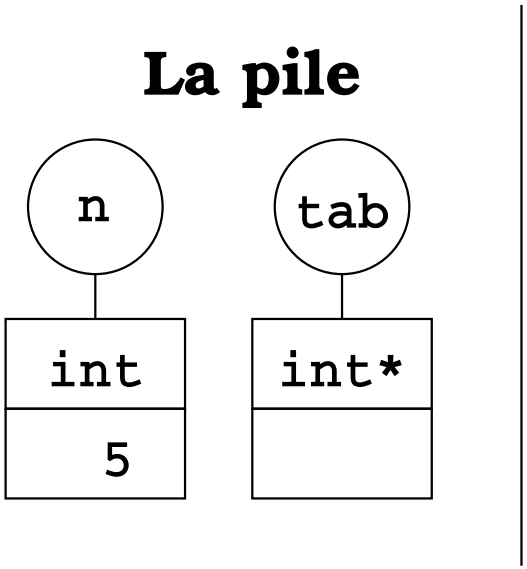


Pointeurs et tableaux

```
double* tab;
int n=5;
tab = new double[n];

for(int i=0; i<n; i++){
    tab[i] = 2*i;
}

delete[] tab;
```

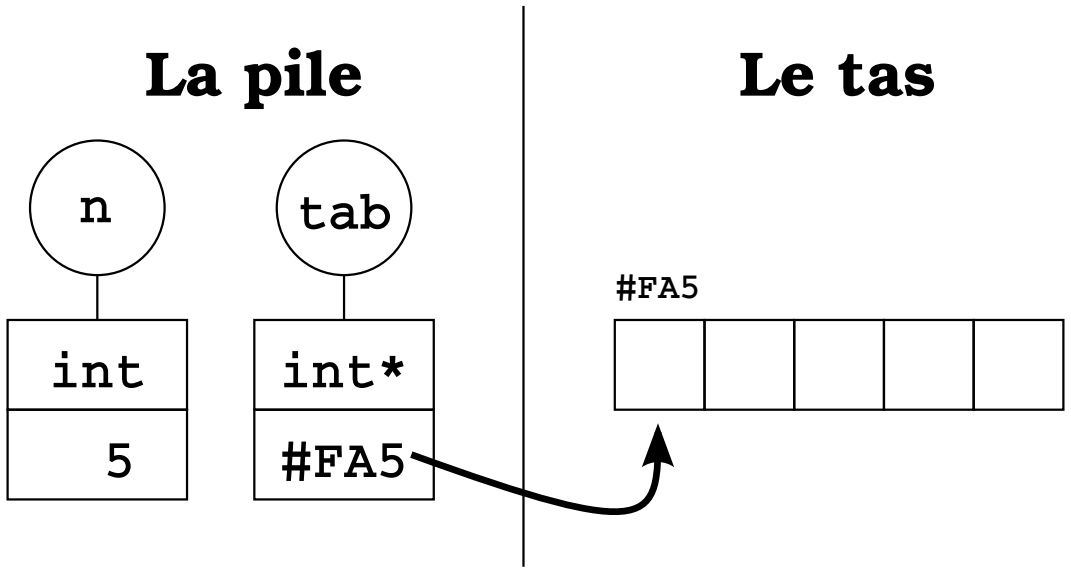


Pointeurs et tableaux

```
double* tab;
int n=5;
tab = new double[n];

for(int i=0; i<n; i++){
    tab[i] = 2*i;
}

delete[] tab;
```

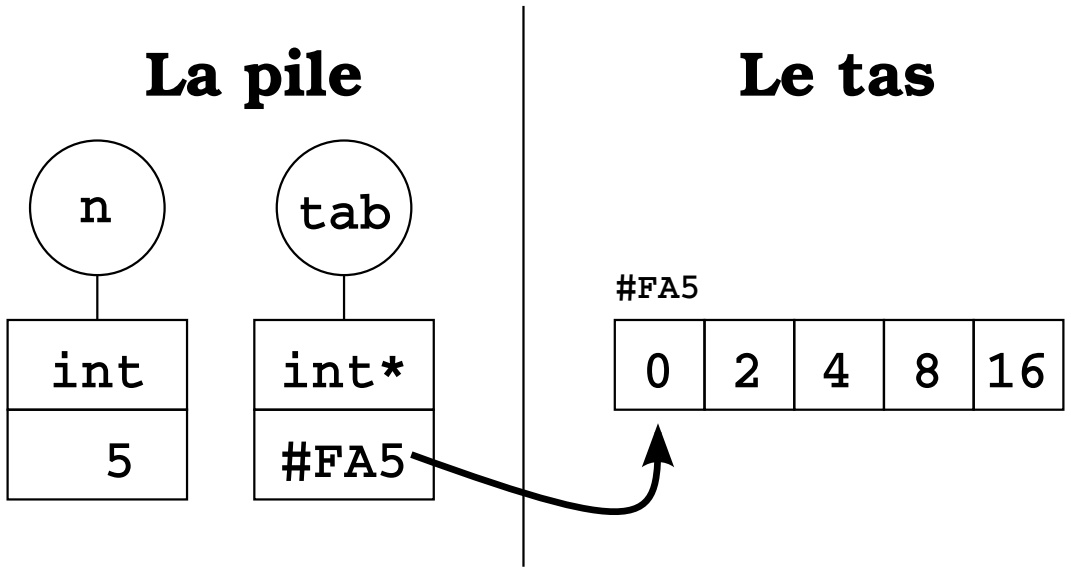


Pointeurs et tableaux

```
double* tab;
int n=5;
tab = new double[n];

for(int i=0; i<n; i++){
    tab[i] = 2*i;
}

delete[] tab;
```

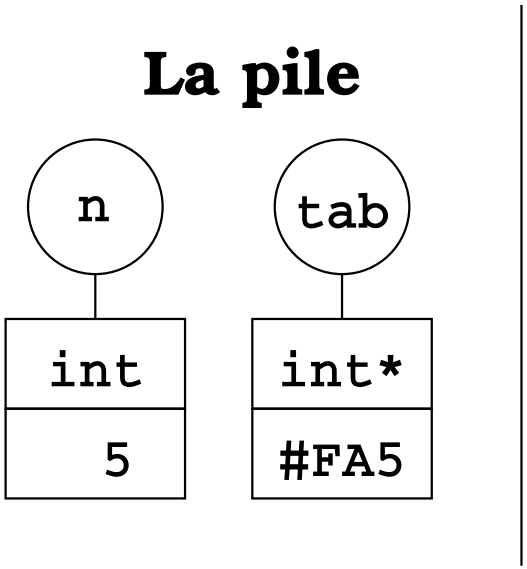


Pointeurs et tableaux

```
double* tab;
int n=5;
tab = new double[n];

for(int i=0; i<n; i++){
    tab[i] = 2*i;
}

delete[] tab;
```

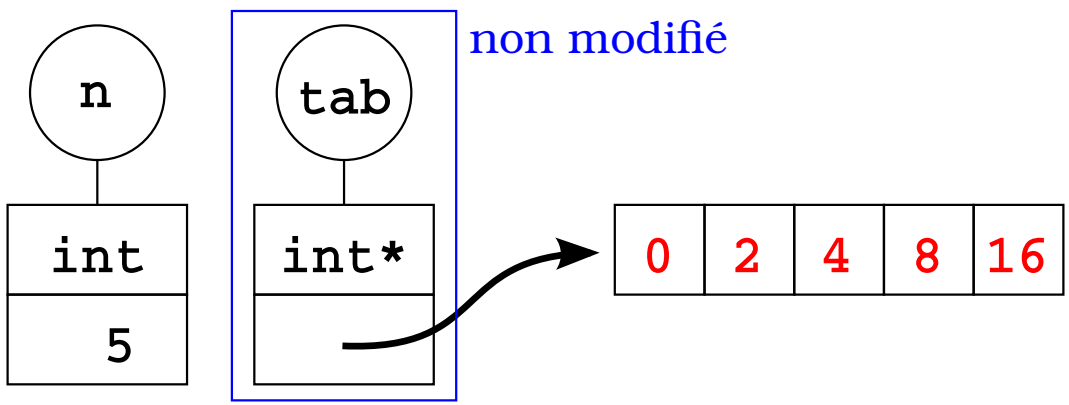


Modifier la variable / tableau désigné par le pointeurs

- Pas besoin de passage par référence : on ne modifie pas le pointeur (l'adresse), seulement les valeurs stockées dans la zone de la mémoire désignées par le pointeur.
- On peut utiliser les fonctions créées pour les tableaux statiques.

```
void fill(double* tab, int n){
  for(int i=0; i<n; i++)
    tab[i] = 2*i;
}

double* t;
int taille=5;
t = new double[taille];
fill(t,taille);
delete[] tab;
```

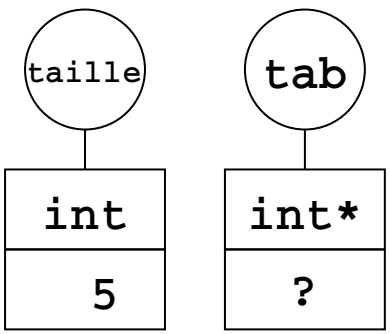


Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){
  tab = new double[n];
  cout << tab << endl;
}

double* t;
int taille=5;
create(t,taille);
cout << t << endl;
delete[] tab;
```

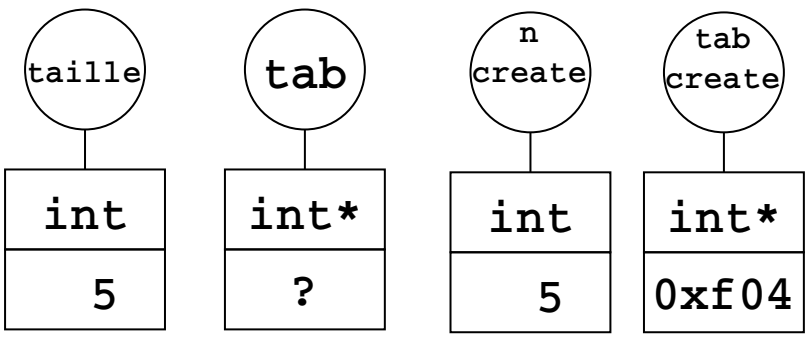


Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){
  tab = new double[n];
  cout << tab << endl;
}

double* t;
int taille=5;
create(t,taille);
cout << t << endl;
delete[] tab;
```

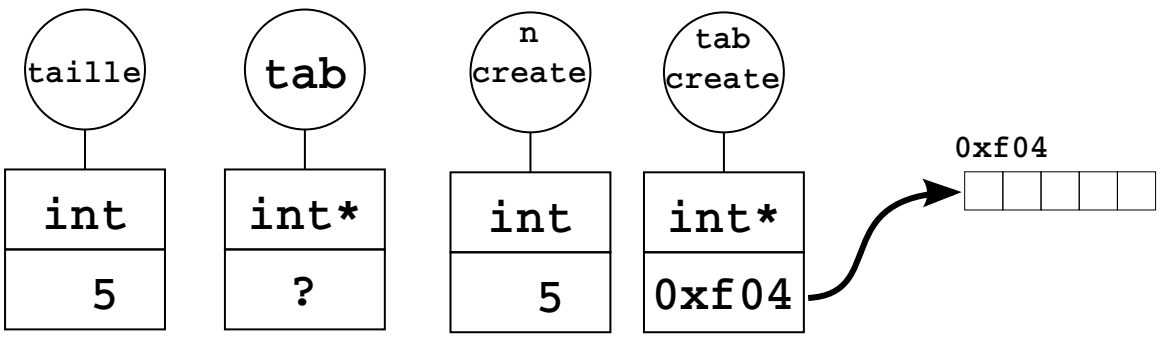


Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){
  tab = new double[n];
  cout << tab << endl;
}

double* t;
int taille=5;
create(t,taille);
cout << t << endl;
delete[] tab;
```

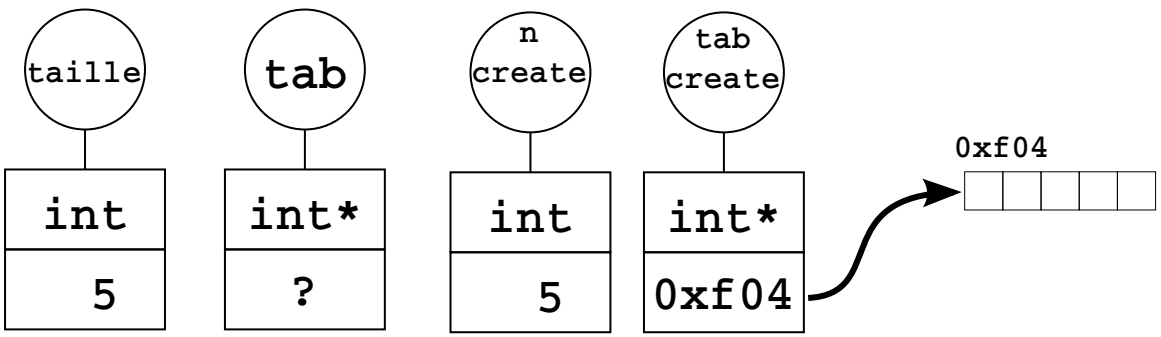


Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){
  tab = new double[n];
  cout << tab << endl;
}                              --> 0xf04

double* t;
int taille=5;
create(t,taille);
cout << t << endl;
delete[] tab;
```

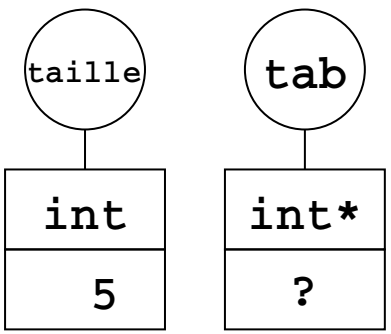


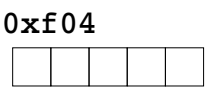
Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){
  tab = new double[n];
  cout << tab << endl;
}                          --> 0xf04

double* t;
int taille=5;
create(t,taille);
cout << t << endl;
delete[] tab;
```



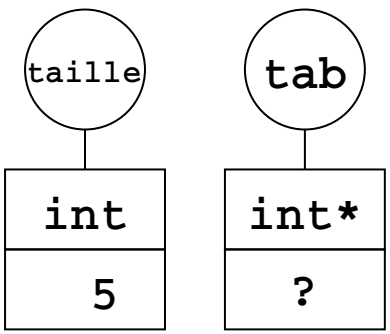


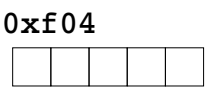
Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){
  tab = new double[n];
  cout << tab << endl;
}                          --> 0xf04

double* t;
int taille=5;
create(t,taille);
cout << t << endl;         --> ?
delete[] tab;
```



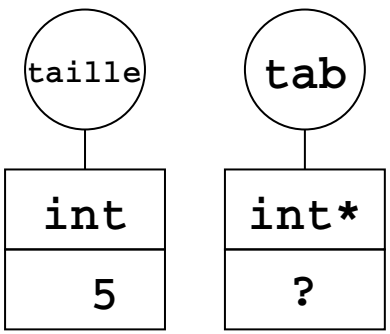


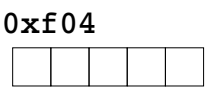
Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){
  tab = new double[n];
  cout << tab << endl;
}                          --> 0xf04

double* t;
int taille=5;
create(t,taille);
cout << t << endl;      --> ?
delete[] tab;  --> ERREUR non alloué
```





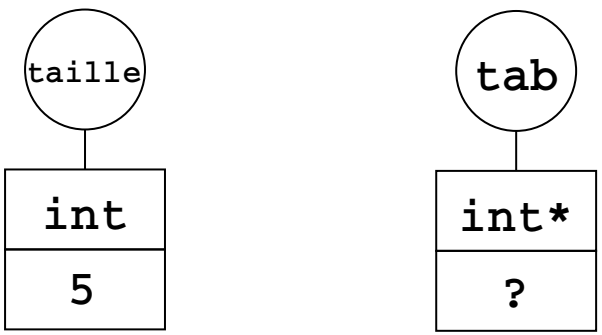
ERREUR FUITE DE MÉMOIRE

Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){
  tab = new double[n];
  cout << tab << endl;
}

double* t;
int taille=5;
create(t,taille);
cout << t << endl;
delete[] tab;
```

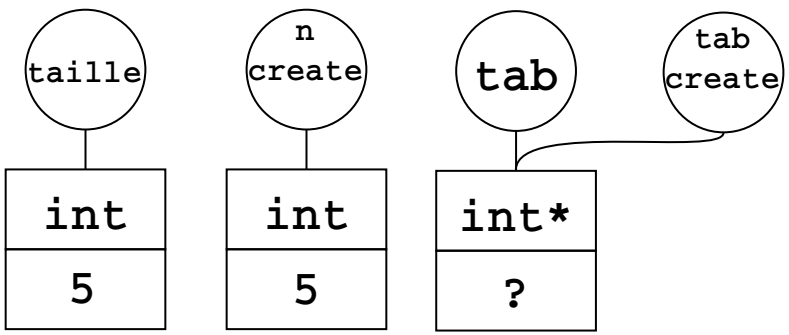


Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){
  tab = new double[n];
  cout << tab << endl;
}

double* t;
int taille=5;
create(t,taille);
cout << t << endl;
delete[] tab;
```

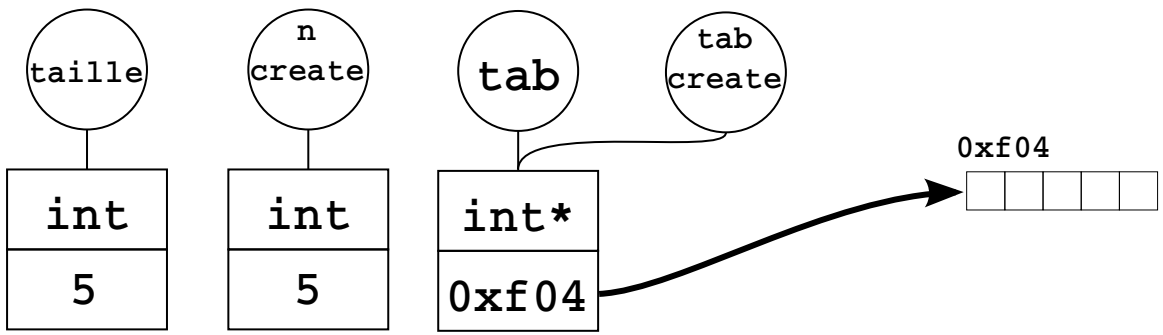


Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){
  tab = new double[n];
  cout << tab << endl;
}

double* t;
int taille=5;
create(t,taille);
cout << t << endl;
delete[] tab;
```

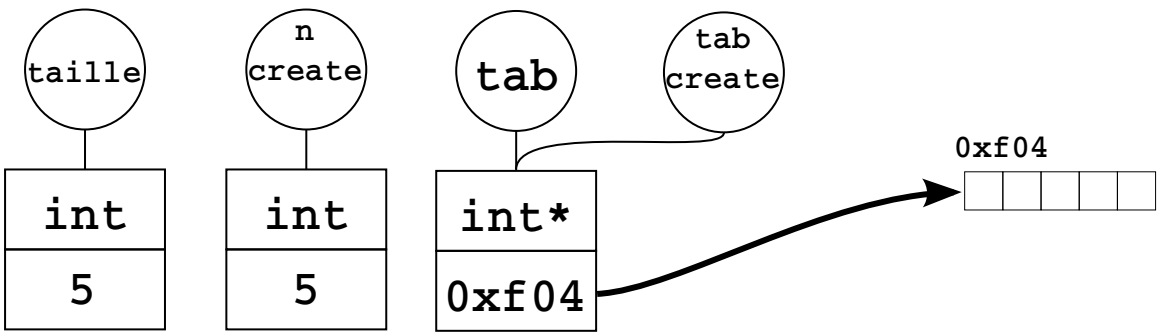


Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){
  tab = new double[n];
  cout << tab << endl;
}                          --> 0xf04

double* t;
int taille=5;
create(t,taille);
cout << t << endl;
delete[] tab;
```



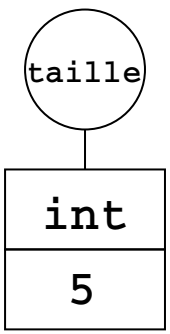
Modifier le pointeur

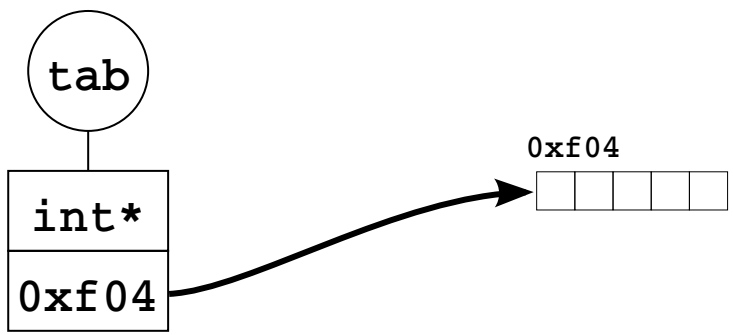
- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){
  tab = new double[n];
  cout << tab << endl;
}

double* t;
int taille=5;
create(t,taille);
cout << t << endl;
delete[] tab;
```

--> 0xf04



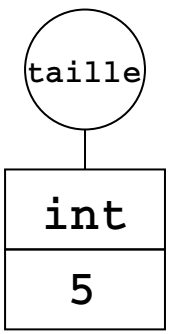


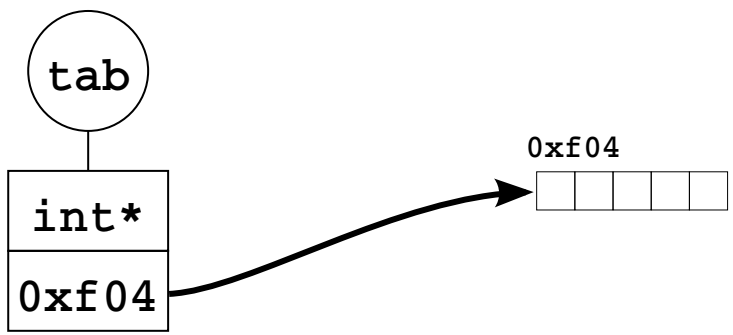
Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){
  tab = new double[n];
  cout << tab << endl;
}

double* t;
int taille=5;
create(t,taille);
cout << t << endl;  --> 0xf04
delete[] tab;
```



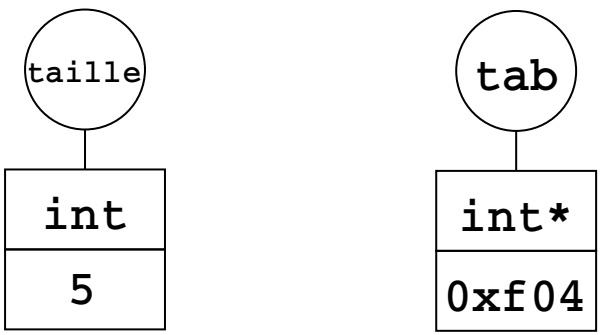


Modifier le pointeur

- Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){
  tab = new double[n];
  cout << tab << endl;
}

double* t;
int taille=5;
create(t,taille);
cout << t << endl;  --> 0xf04
delete[] tab;
```



Egalité de pointeurs

L'égalité de pointeurs est autorisée.

Attention

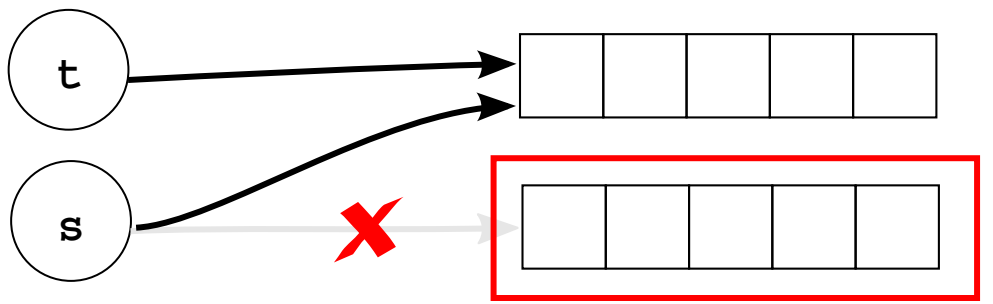
- Il y a des risques de fuite de mémoire
- Deux pointeurs égaux renvoient au même espace mémoire
- Il n'y a pas création d'un nouveau tableau

```
double* t,s;
int n=5;
t = new double[n];
s = new double[n];

s = t;

...
```

--> Fuite de mémoire



--> Fuite de mémoire

Egalité de pointeurs

L'égalité de pointeurs est autorisée.

Attention

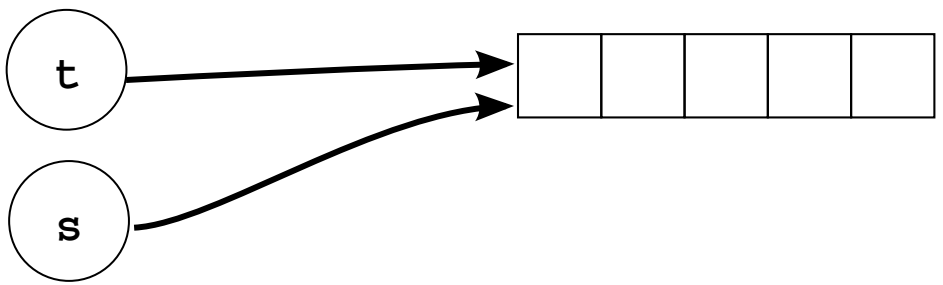
- Il y a des risques de fuite de mémoire
- Deux pointeurs égaux renvoient au même espace mémoire
- Il n'y a pas création d'un nouveau tableau

```
double* t,s;
int n=5;
t = new double[n];

s = t;

delete[] t;
delete[] s;
```

--> double délétion



Egalité de pointeurs

L'égalité de pointeurs est autorisée.

Attention

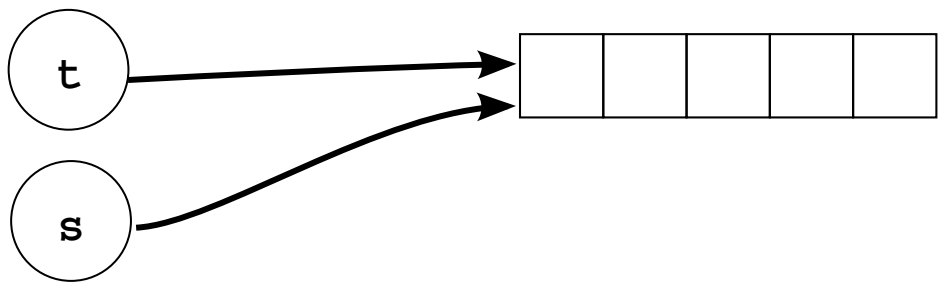
- Il y a des risques de fuite de mémoire
- Deux pointeurs égaux renvoient au même espace mémoire
- Il n'y a pas création d'un nouveau tableau

```
double* t,s;
int n=5;
t = new double[n];

s = t;

delete[] t; // ou delete[] s
```

--> OK



Egalité de pointeurs 2

Pour copier un tableau, il faut le faire terme à terme.

```
double* t,s;
int n = 100;
t = new double[n];

...

// copie du tableau
s = new double[n]; // allocation de la memoire
for(int i=0; i<n; i++){
    s[i] = t[i]; // recopie terme a terme
}
...
delete[] t; // liberation tableau t
delete[] s; // liberation tableau s
```

Plan de la séance

Des tableaux dans des structures

Il est possible d'utiliser des tableaux dynamiques dans les structures.

Attention

Surtout pas de tableaux statiques.

```
struct Vect{
    int taille; // la taille
    double* t; // le tableau, ne pas allouer
               // dans la declaration de la structure
};
```

Structures et allocation dynamique

```
// Vect.h
struct Vect{
    int n; // taille
    double* t; // tableau
};

void init(Vect& v);

void cree(Vect& v, int n);

void detruit(Vect& v);

void remplit(Vect& v, double val);

void copie(Vect& v, Vect o);

Vect operator+(Vect v1, Vect v2);
```

Structures et allocation dynamique

```
// Vect.h
struct Vect{
    int n; // taille
    double* t; // tableau
};

void init(Vect& v);

void cree(Vect& v, int n);

void detruit(Vect& v);

void remplit(Vect& v, double val);

void copie(Vect& v, Vect o);

Vect operator+(Vect v1, Vect v2);
```

```
// Vect.cpp
#include "Vect.h"
void init(Vect& v){
    v.n = 0;
}
void cree(Vect& v, int n){
    assert(n > 0);
    v.n = n;
    v.t = new double[v.n];
}
void detruit(Vect& v){
    if(v.taille > 0){
        v.taille = 0;
        delete[] v.t;
    }
}
void remplit(Vect& v, double val){
    for(int i=0; i<v.n; i++){
        v.t[i] = val;
    }
}
```

Structures et allocation dynamique

```
// Vect.h
struct Vect{
    int n; // taille
    double* t; // tableau
};

void init(Vect& v);

void cree(Vect& v, int n);

void detruit(Vect& v);

void remplit(Vect& v, double val);

void copie(Vect& v, Vect o);

Vect operator+(Vect v1, Vect v2);
```

```
// Vect.cpp
#include "Vect.h"
void copie(Vect& v, Vect o){
    detruit(v);
    cree(v, o.taille);
    for(int i=0;i<v.n;i++)
        v.t[i] = o.t[i];
}

Vect operator+(Vect v1, Vect v2){
    assert(v1.n == v2.n);
    Vect v;
    cree(v, v1.n);
    for(int i=0;i<v.n; i++)
        v.t[i] = v1.t[i]+v2.t[i];
    return v;
}
```

Structures et allocation dynamique

```
// Vect.h
struct Vect{
    int n; // taille
    double* t; // tableau
};

void init(Vect& v);

void cree(Vect& v, int n);

void detruit(Vect& v);

void remplit(Vect& v, double val);

void copie(Vect& v, Vect o);

Vect operator+(Vect v1, Vect v2);
```

```
// main.cpp
#include "Vect.h"

int main(){
    Vect v1,v2;
    init(v1);
    init(v2);

    cree(v1, 10);
    remplit(v1, 5.6);

    copie(v2, v2);

    Vect v3 = v1 + v2;

    detruit(v1);
    detruit(v2);
    detruit(v3);
    return 0;
}
```

Plan de la séance

break

L'instruction **break** permet de sortir d'une boucle.

```
for(int i=0; i<n; i++){
    bool b = f(i);
    if(!b) break; // sort de la boucle si b est faux
}
```

Pour sortir de boucles imbriquées, il faut utiliser des booléens.

```
bool stop = false;
for(int i=0; i<n; i++){
    for(int j=0; j<m; j++){
        if(i*j > 100){
            stop = true;
            break;
        }
    }
    if(stop) break;
}
```

```
bool go = true;
for(int i=0; i<n && go; i++){
    for(int j=0; j<m && go;
    ↪  j++){
        if(i*j > 100){
            go = false;
        }
    }
}
```

continue

L'instruction **`continue`** permet de passer à l'itération suivante dans une boucle (sans exécuter ce qui se trouve après le **`continue`**).

```
int i=1;
while(i< 1000){
    i++;
    if(i%2 == 1)
        continue;
    cout << i << " est pair" << endl;
}
```

Plan de la séance

Rappels

Tableaux 2D

Allocation dynamique

Structures et allocation dynamique

Boucles, break et continue

TP du jour

Le TP

Manipulation d'images.

- Tableaux 2D en allocation dynamique
- Opérations courantes sur les images (flou, inversion, contraste...)
- Manipulation de structure et d'allocation dynamique