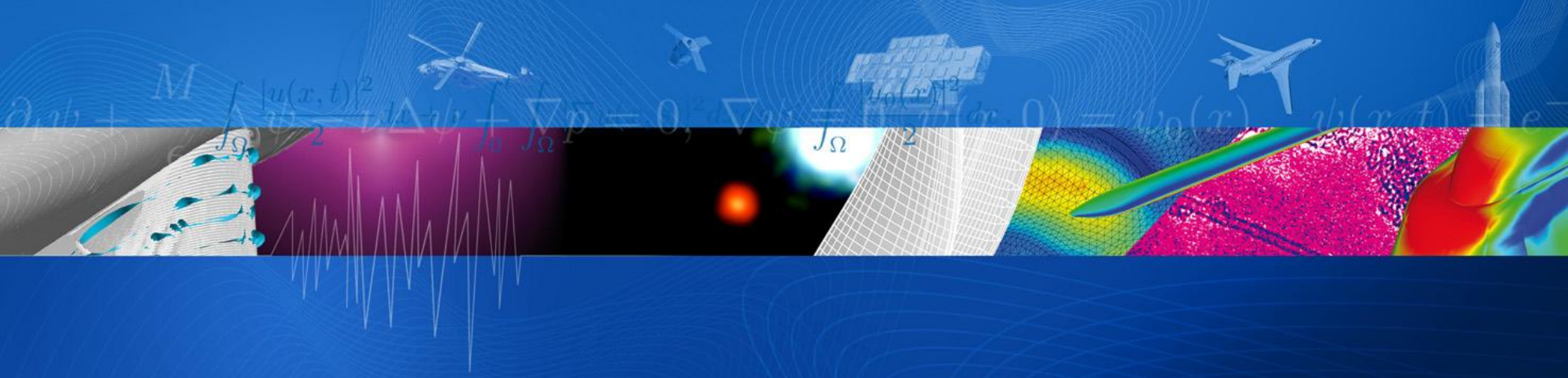


Introduction à la programmation C++

Allocation dynamique

Nicolas Audebert

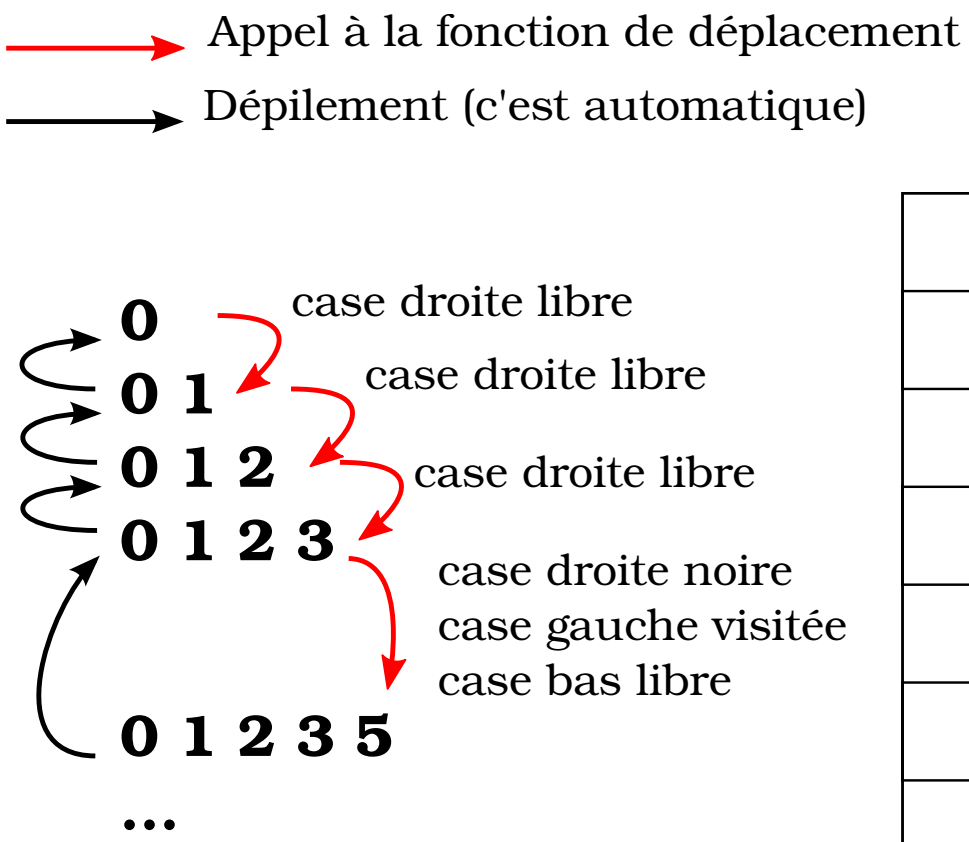
Mercredi 9 novembre 2016

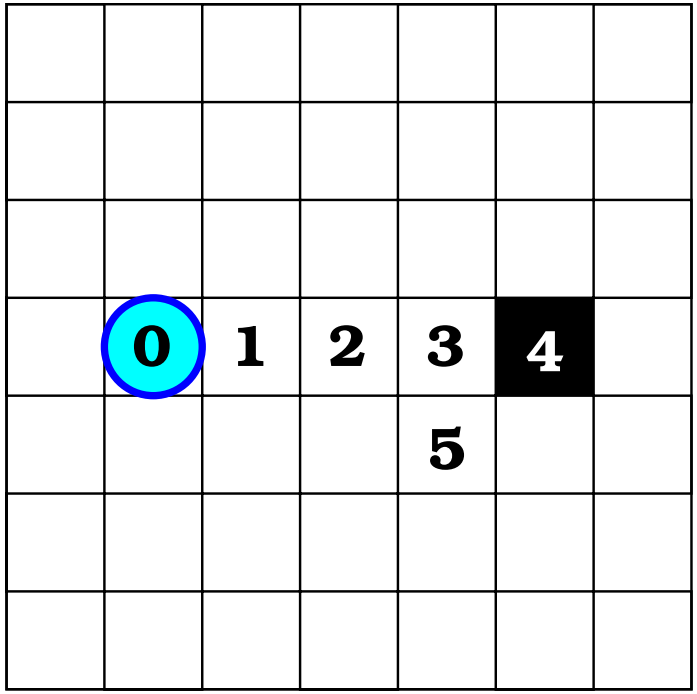
ONERA
THE FRENCH AEROSPACE LAB

retour sur innovation

Plan de la séance

Robot





Évolution de la pile des appels

Plan de la séance

Exercice Robot

DS machine : Décimales

Tableaux 2D

Allocation dynamique

Structures et allocation dynamique

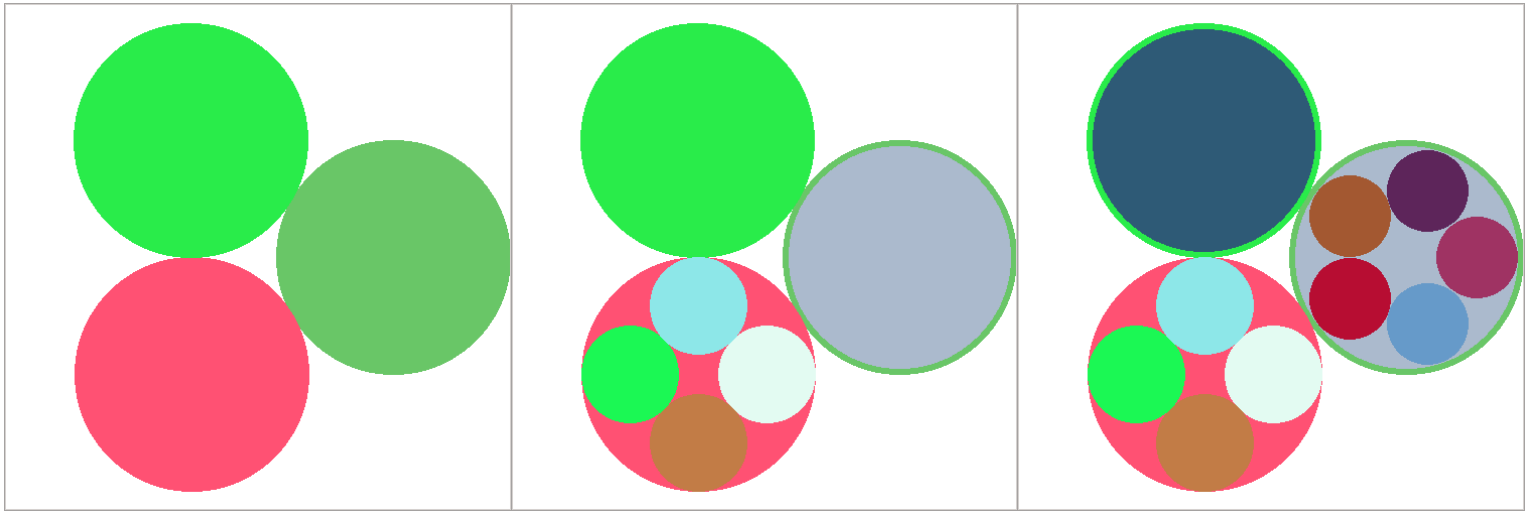
Boucles, break et continue

TP du jour

Décimales

Idée

- Représenter visuellement les décimales d'un nombre réel
- Stocker les cercles dans une file
- Afficher les disques dans une fenêtre



Décimales

Extraction des décimales

- Q1 : fonction `donne_decimale` : attention à passer par référence pour pouvoir modifier `x` !
- Q2 : affichage des décimales de π : OK

File d'attente

- Q3 : définir une structure `Cercle` : *cf.* les TP
- Q4 : définir une structure `File` : n est la taille (variable) du tableau, mais en pratique le tableau est **statique** de taille maximale 300.
- Q5 : fonction `push` : ajoute un élément à la fin de la file. Attention aux indices (`tab[n]` $\rightarrow$ erreur !).
- Q6 : fonction `pop` : renvoie le premier élément de la file. Attention à bien décaler les éléments suivants.
- Q7 : protéger les fonctions par des `assert` : vérifier que la taille de la file est cohérente (ne pas faire `pop` sur une file vide, ne pas dépasser la taille maximale de la file). Ne pas oublier d'inclure `<cassert>`.

Affichage des disques

- Q8 : fonction `affiche_chiffre` : dessiner n disques pour la k-ème décimale et ajouter (avec `push` !) le disque dans la file. Attention aux conversion `double` vers `int`.
- Q9 : fonction `affiche_nombre` : créer une file puis combiner `donne_decimale` et `affiche_chiffre`.

Décimales

Mise en pratique

- Q10 : tester `affiche_nombre` avec π.
- Q11 : générer un nombre réel aléatoire et l'afficher. Utiliser `rand()` à bon escient.

Erreurs courantes

- Passage par copie au lieu d'un passage par référence
- Erreur sur les indices : **un tableau de taille n commence à 0 et finit à $n-1$**
- Conversion `double` vers `int` mal gérées
- Oubli de l'initialisation : il faut toujours initialiser avec les variables avec des valeurs saines
- Oubli de `openWindow()` avant l'affichage des disques

Plan de la séance

Tableaux 2D

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

C++

```
double tab2D[5][3];

// acces aux elements
for(int i=0; i<5; i++){
    for(int j=0; j<3; j++){
    tab2D[i][j] = i*j;
        // pas de tab2D[i,j]
        cout << tab2D[i][j] << " ";
    }
    cout << endl;
}
```

C++

```
//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 :

C++

```
// il faut specifier les tailles dans les arguments
void init(int t[2][3], int val){ // toujours passage par reference
    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

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

C++

```
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

C++

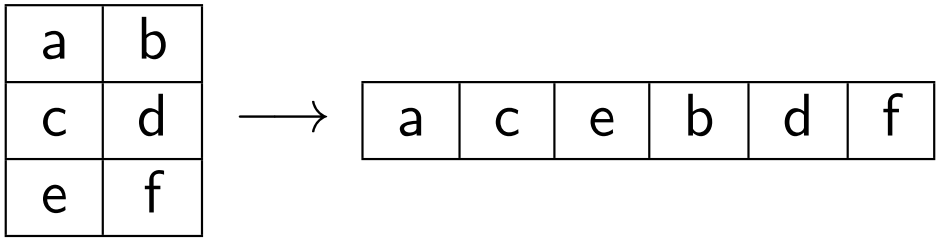
```
void init(int t[][], int rows, int cols, int val){...} //ERREUR
```

Problème

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

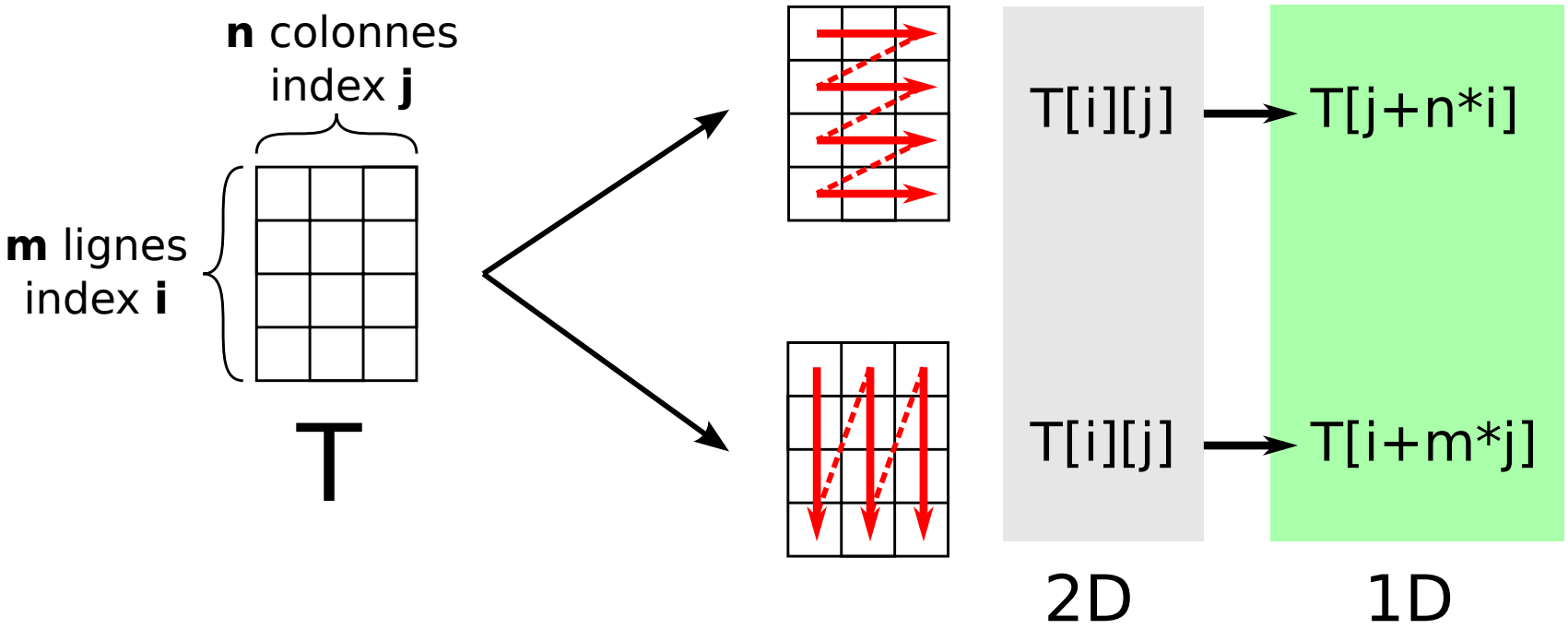
Solution

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



Utilisation dans les fonctions

Il est désormais possible d'utiliser des fonctions génériques.

C++

```
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

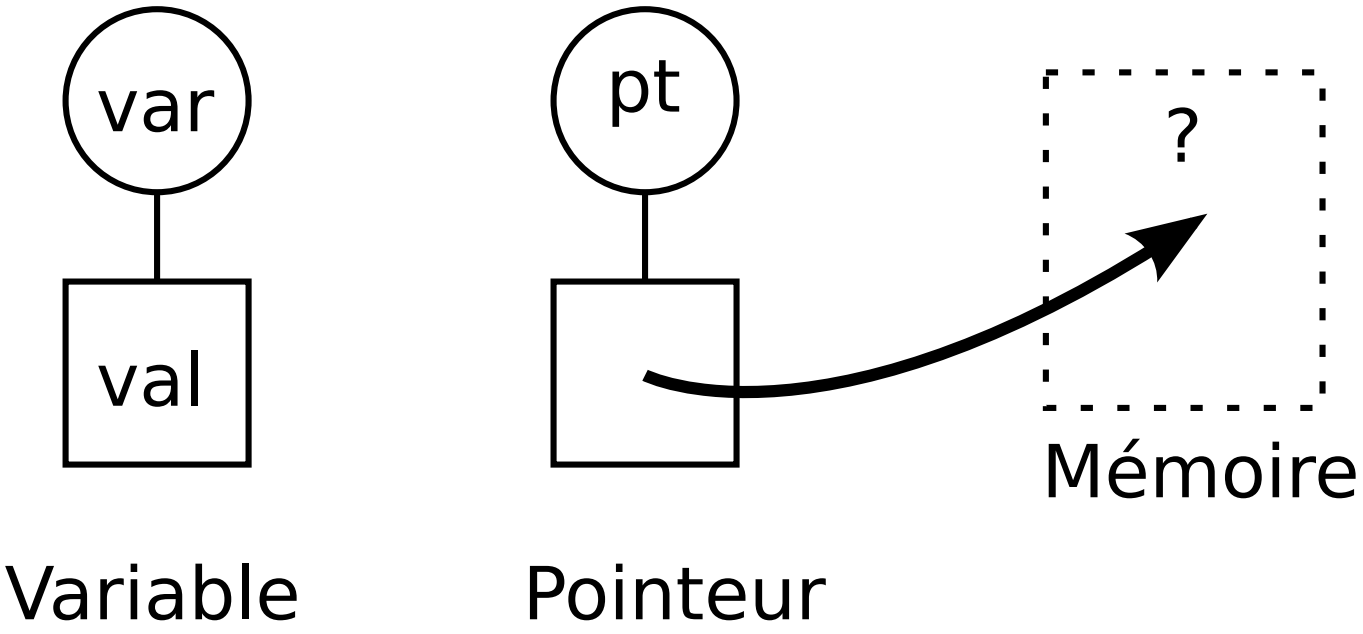
On fait des tableaux 1D, comme précédemment.

C++

```
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;
```

C'est quoi ?

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 *.

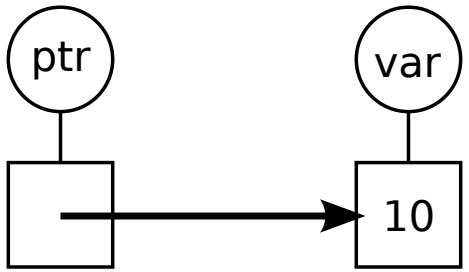
C++

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

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

C++

```
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).

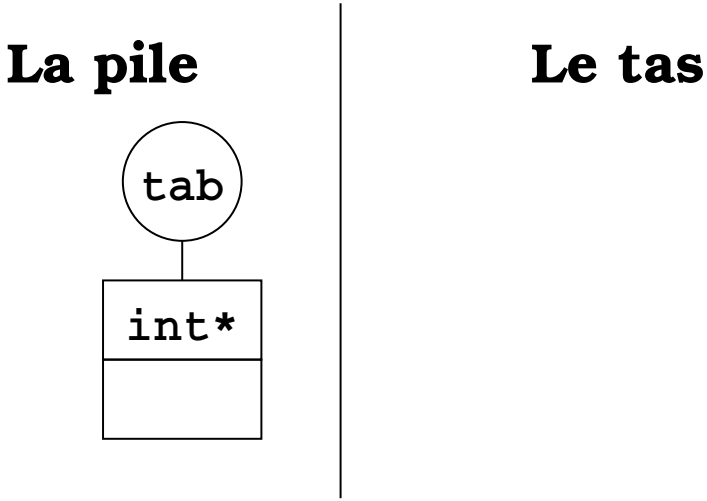
- Créer une variable dans le tas : **`new`**
- Supprimer une variable dans le tas : **`delete`**

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;
```

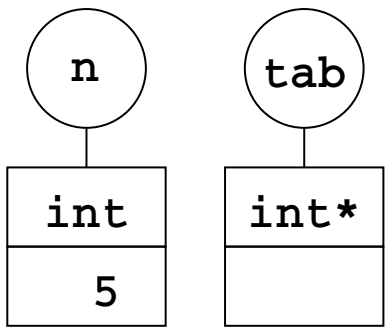


Pointeurs et tableaux

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

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

delete[] tab;
```

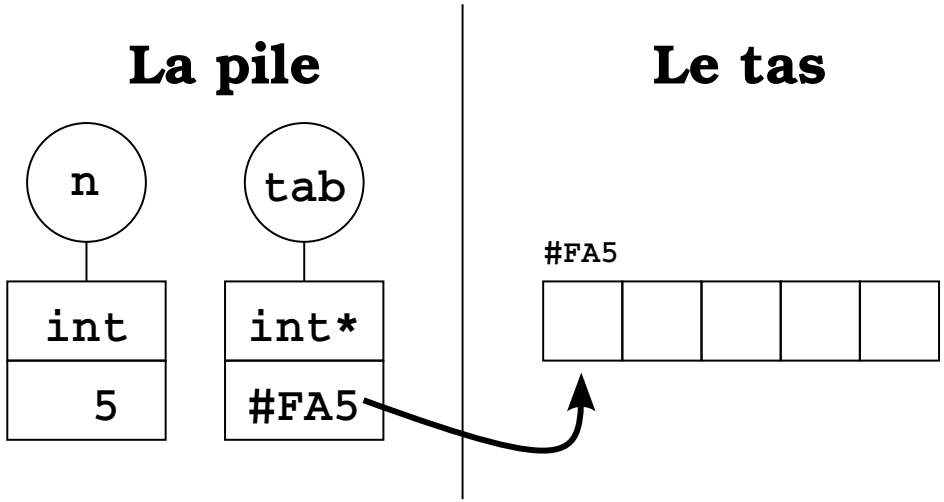


Pointeurs et tableaux

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

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

delete[] tab;
```

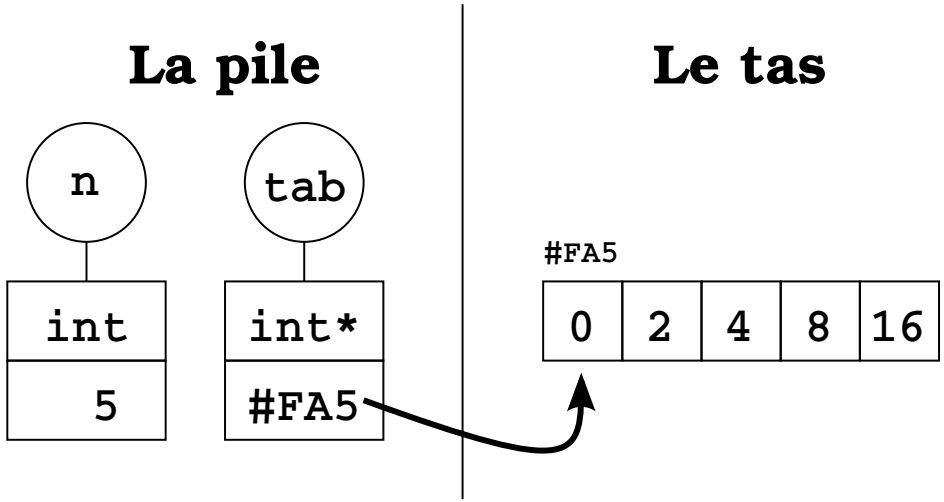


Pointeurs et tableaux

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

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

delete[] tab;
```

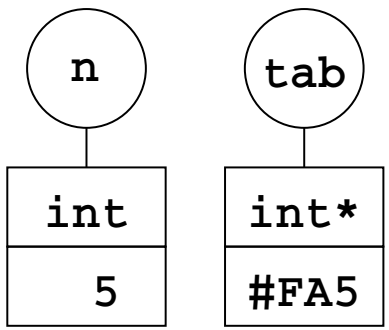


Pointeurs et tableaux

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

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

delete[] tab;
```

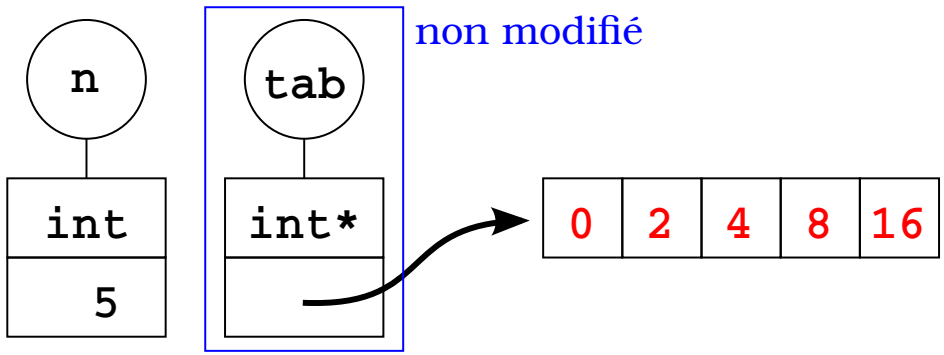


Modifier la variable / tableau désigné par le pointeurs

- 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;
```

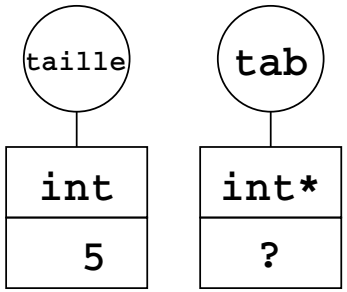


Modifier le pointeur

- 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;
```

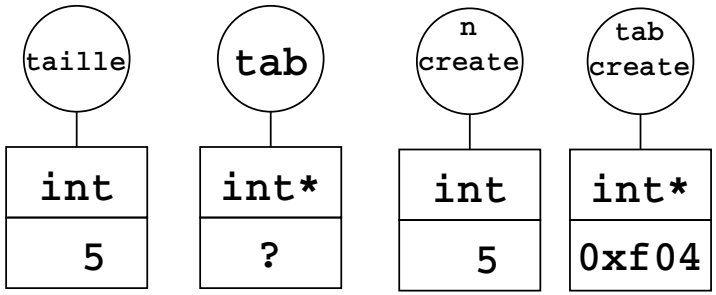


Modifier le pointeur

- 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;
```

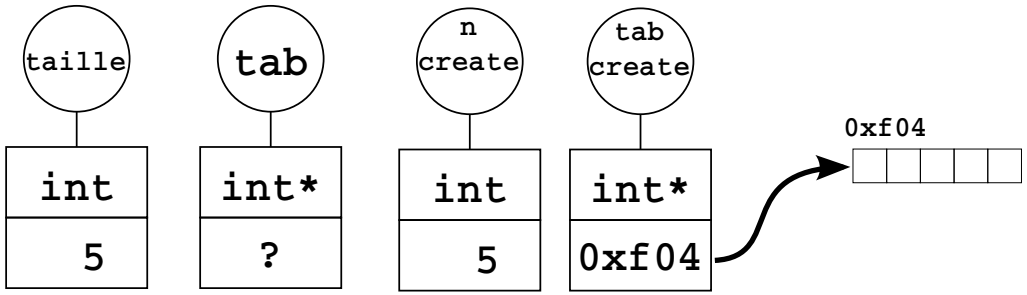


Modifier le pointeur

- 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;
```

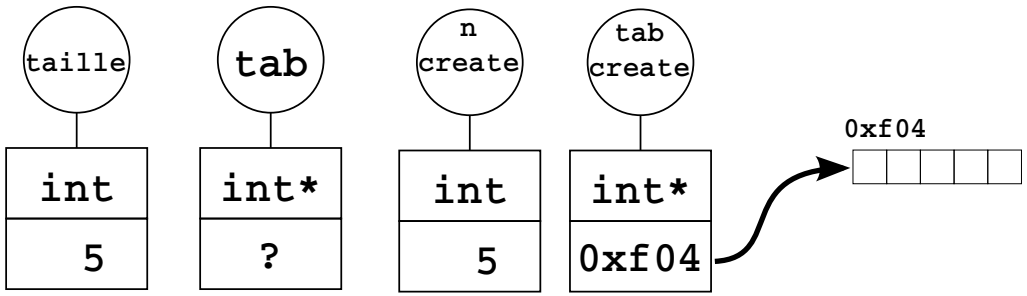


Modifier le pointeur

- 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;
```

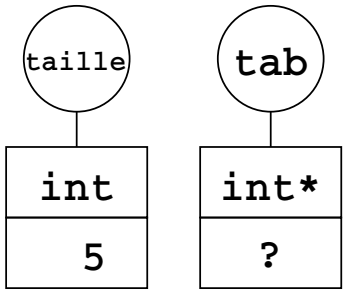


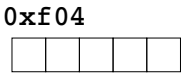
Modifier le pointeur

- 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;
```



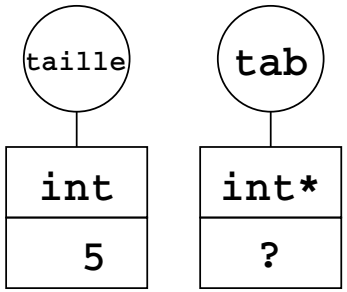


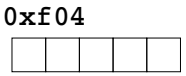
Modifier le pointeur

- 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;
```



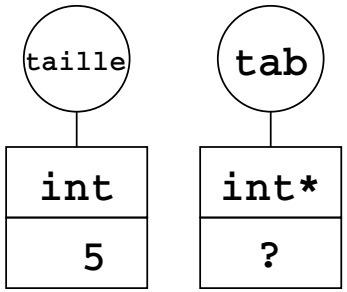


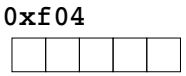
Modifier le pointeur

- 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
```





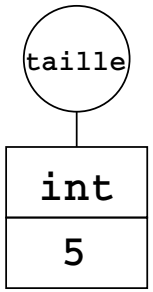
ERREUR FUITE DE MÉMOIRE

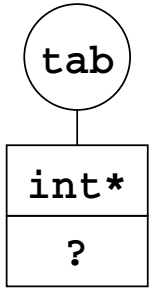
Modifier le pointeur

- 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;
```



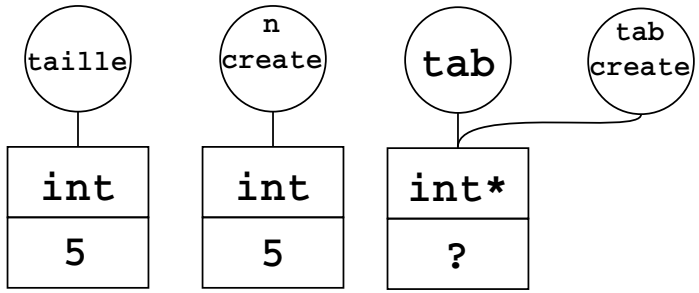


Modifier le pointeur

- 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;
```

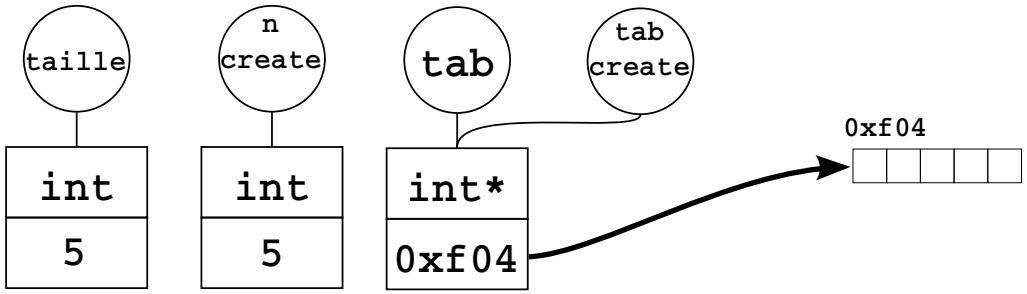


Modifier le pointeur

- 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;
```

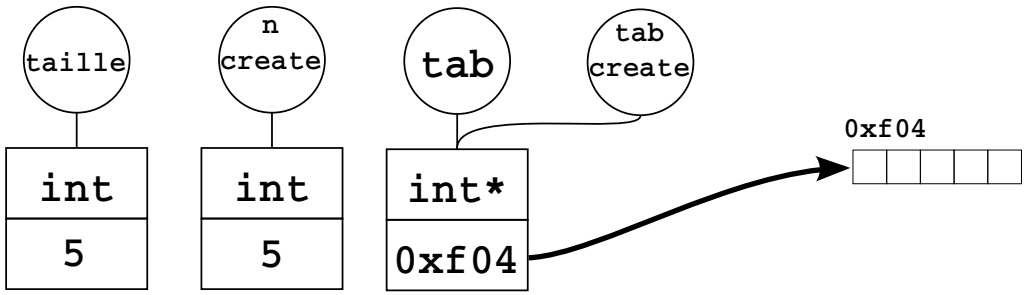


Modifier le pointeur

- 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;
```

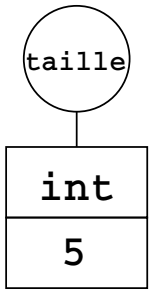


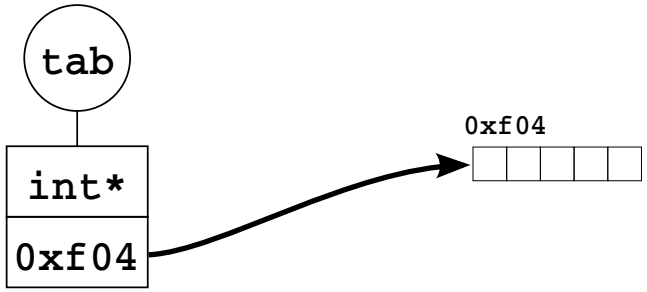
Modifier le pointeur

- 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;
}                          --> 0xf04

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





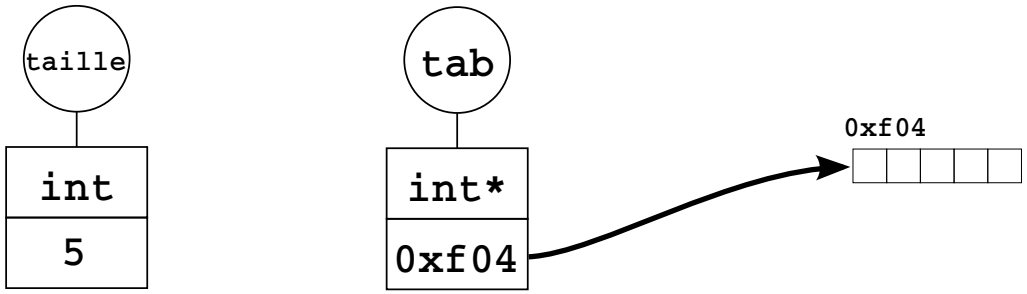
Modifier le pointeur

- 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

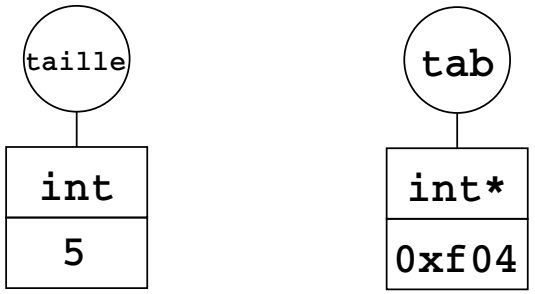


Modifier le pointeur

- 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;
```



Egalité de pointeurs

L'égalité de pointeurs est autorisée.

ATTENTION

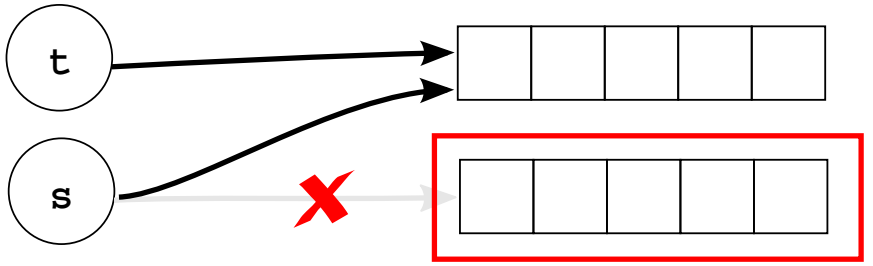
- Il y a des risques de fuite de mémoire
- Deux pointeurs égaux renvoient au même espace mémoire
- Il n'y a pas création d'un nouveau tableau

```
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

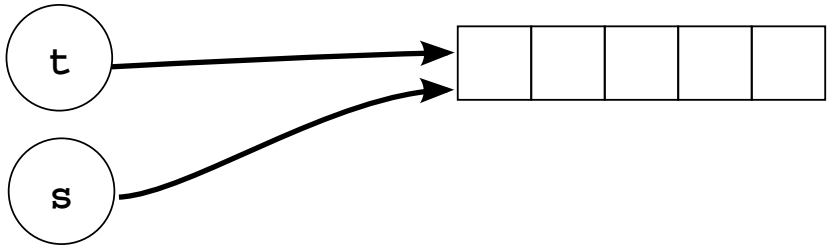
- Il y a des risques de fuite de mémoire
- Deux pointeurs égaux renvoient au même espace mémoire
- Il n'y a pas création d'un nouveau tableau

```
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

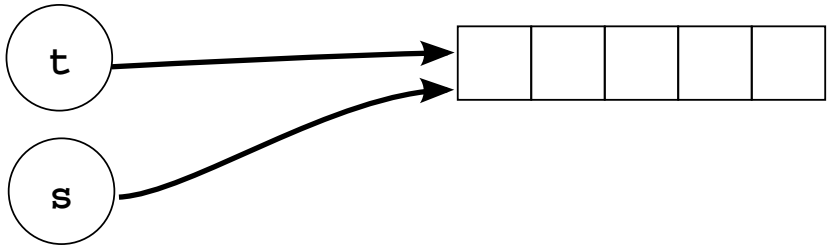
- Il y a des risques de fuite de mémoire
- Deux pointeurs égaux renvoient au même espace mémoire
- Il n'y a pas création d'un nouveau tableau

```
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.

C++

```
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.

C++

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

Structures et allocation dynamique

C++

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

void init(Vect& v);

void cree(Vect& v, int taille);

void detruit(Vect& v);

void remplit(Vect& v, double& val);

void copie(Vect& v, Vect orig);

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

C++

```
// Vect.cpp
#include "Vect.h"
void init(Vect& v){
    v.taille = 0;
}

void cree(Vect& v, int taille_v){
    assert(taille_v >0);
    v.taille = taille_v;
    v.t = new double[v.taille];
}

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.taille; i++){
        v.t[i] = val;
    }
}
```

Structures et allocation dynamique 2

C++

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

void init(Vect& v);

void cree(Vect& v, int taille);

void detruit(Vect& v);

void remplit(Vect& v, double& val);

void copie(Vect& v, Vect orig);

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

C++

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

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

Structures et allocation dynamique 2

C++

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

void init(Vect& v);

void cree(Vect& v, int taille);

void detruit(Vect& v);

void remplit(Vect& v, double& val);

void copie(Vect& v, Vect orig);

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

C++

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

int main(){
    cout << "Debut main" << endl;

    Vect v1,v2;
    init(v1);
    init(v2);

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

    copie(v2, v1);

    Vect v3 = v1 + v2;

    detruit(v1);
    detruit(v2);
    detruit(v3);
    cout << "Fin main" << endl;
    return 0;
}
```

Plan de la séance

break

L'intruction **`break`** permet de sortir d'une boucle en cours d'exécution.

C++

```
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.

C++

```
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;
}
```

C++

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

continue

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

C++

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

Plan de la séance

Exercice Robot

DS machine : Décimales

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