

ALLOCATION DYNAMIQUE EN C++

Les fameux pointeurs. Parlons-en!



QUELLES SONT LES DIFFÉRENCES
ENTRE LE C++ ET LE C#?

→
AU NIVEAU DU CODE ?

→
AU NIVEAU DE L'EXÉCUTION ?



DIFFÉRENCES FONDAMENTALES

En C# :

- Le code est compilé en « ByteCode » qui, lui, est interprété.

En C++ :

- Le code est compilé en langage machine, qui lui, est directement exécuté.

Cela a plusieurs implications.



LES « VM » : VIRTUAL MACHINES

Tout code interprété est (généralement) exécuté sur une VM.

- La VM est un CPU virtuel sous stéroïdes!

Cette VM simplifie grandement le travail du programmeur.

Une des choses que gère la VM est la mémoire.

- La VM garde trace de toute allocation de mémoire et fait le ménage de temps en temps.



C++ : LE LANGAGE CASSE-COU

À l'inverse, un langage compilé n'a rien de tel.

- Vous conduisez en manuel!

Vous n'avez accès qu'aux commandes du matériel.

- Rien de plus.

Cela inclut la gestion de la mémoire.



LA PILE

Rappel avant le reste...



LA PILE (OU PILE D'APPELS) (1 DE 2)

La pile fait partie de n'importe quel langage et est le premier niveau de gestion de la mémoire.

La pile contient toutes les méthodes en cours à un moment « X ».

- Ci-contre, une méthode « A » a appelé une méthode « B » qui a appelé une méthode « C ».

Pile d'Appels





LA PILE (OU PILE D'APPELS) (2 DE 2)

En plus de représenter l'ordre d'appel des méthodes, chaque élément de la pile contient toutes les variables utilisées dans la méthode.

- Lorsqu'une méthode est appelée, ses variables sont créées et placées sur la pile.
- Lorsqu'une méthode se termine, elle est retirée de la pile avec ses variables.

Pile d'Appels





EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var + tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    return B(var + 1);
}
```



**APPELONS LA FONCTION A AVEC
LA VALEUR « 5 » EN PARAMÈTRE.**



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

➔ int A(int var)
{
    return B(var + 1);
}
```

La mémoire pour la fonction A est allouée.

A

- var : 146497187



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

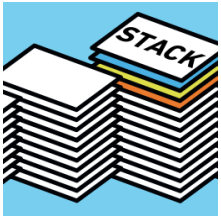
int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    ➡ return B(var + 1);
}
```

Ensuite, le code s'exécute et on peut placer des valeurs dans ces emplacements mémoire.

A

- var : 5



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    ➡ return B(var + 1);
}
```

On effectue un appel à « B ».

Cela va créer un nouvel élément sur la pile.

A

- var : 5



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}
```

```
→ int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}
```

```
int A(int var)
{
    return B(var + 1);
}
```

La mémoire de « B » est alloué sur la pile.

Il peut y avoir des éléments sur la pile plus gros que d'autres.

B

- var : 94188157
- multiple : 0.11499f

A

- var : 5



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}
```

```
→ int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}
```

```
int A(int var)
{
    return B(var + 1);
}
```

Pour un processeur,
seul l'élément sur le
dessus de la pile est
en cours d'exécution.

B

- var : 94188157
- multiple : 0.11499f

A

- var : 5



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}
```

```
→ int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}
```

```
int A(int var)
{
    return B(var + 1);
}
```

La pile a une taille fixe (en megabits), déterminée à la compilation par le compilateur.

D'où les « Stack Overflow ».

B

- var : 94188157
- multiple : 0.11499f

A

- var : 5



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    return B(var + 1);
}
```

Notez que « var » est copié, pas déplacé.

« var » est une variable pour « B ».

B

- var : 6
- multiple : 0.11499f

A

- var : 5



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    return B(var + 1);
}
```

Remarquez que « multiple » n'est pas encore initialisé à « 50 ».

Le code n'en est pas là.

B

- var : 6
- multiple : 0.11499f

A

- var : 5



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    return B(var + 1);
}
```

Là, c'est fait.

B

- var : 6
- multiple : 50.0f

A

- var : 5



EXEMPLE : LA PILE

Appel à « C », et donc, ajout sur la pile.

```
→ int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    return B(var + 1);
}
```

C

- var : 7395422477
- tableau : [85554][78499]

B

- var : 6
- multiple : 50.0f

A

- var : 5



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    return B(var + 1);
}
```

Copie de « var ».

C

- var : 300
- tableau : [85554][78499]

B

- var : 6
- multiple : 50.0f

A

- var : 5



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    return B(var + 1);
}
```

Initialisation du tableau.

C

- var : 300
- tableau : [7][5]

B

- var : 6
- multiple : 50.0f

A

- var : 5

SUPER

HYPER

MEGA

GIGA

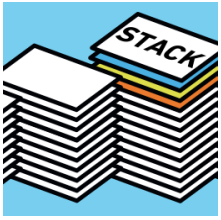
ULTRA

IMPUR

ANTI

VOUS VENEZ DE CONSTATER CE QUE L'ON APPELLE

L'ALLOCATION STATIQUE!!!!



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = {0, 0};
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    return B(var + 1);
}
```

Remarquez que l'on spécifie la taille exacte du tableau.

Le compilateur ne pose pas de question et s'exécute. Dans le code, il va prévoir l'allocation d'exactement 2 cases de tableau.

La preuve, diapositive suivante...



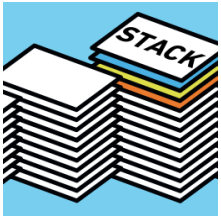
LA FONCTION C EN C++

```
int C(int var)
{
    int tableau[2] = { 7,5 };
    return tableau[0] * var +
           tableau[1] * var;
}
```



LA FONCTION C EN ASSEMBLEUR

```
C(int):  
push rbp  
mov rbp, rsp  
mov DWORD PTR [rbp-12], edi  
mov DWORD PTR [rbp-8], 7  
mov DWORD PTR [rbp-4], 5  
mov edx, DWORD PTR [rbp-8]  
mov eax, DWORD PTR [rbp-4]  
add eax, edx  
imul eax, DWORD PTR [rbp-12]  
pop rbp  
ret
```



LA FONCTION C EN ASSEMBLEUR

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}
```

```
C(int):
push rbp
mov rbp, rsp
sub rsp, 12
mov DWORD PTR [rbp-12], edi
mov DWORD PTR [rbp-8], 7
mov DWORD PTR [rbp-4], 5
mov edx, DWORD PTR [rbp-8]
mov eax, DWORD PTR [rbp-4]
add eax, edx
imul eax, DWORD PTR [rbp-12]
pop rbp
ret
```



LA FONCTION C EN ASSEMBLEUR

Ceci, c'est la taille en octets de l'élément ajouté sur la pile.

$$3 * 32 \text{ bits} = 96 \text{ bits} = 12 \text{ octets}$$

C'est ainsi que l'allocation est faite.

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}
```

```
C(int):
push rbp
mov rbp, rsp
sub rsp, 12
mov DWORD PTR [rbp-12], edi
mov DWORD PTR [rbp-8], 7
mov DWORD PTR [rbp-4], 5
mov edx, DWORD PTR [rbp-8]
mov eax, DWORD PTR [rbp-4]
add eax, edx
imul eax, DWORD PTR [rbp-12]
pop rbp
ret
```

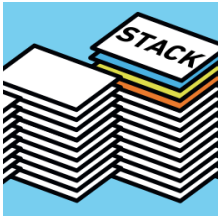


LA FONCTION C EN ASSEMBLEUR

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}
```

Fait intéressant, il y a une petite optimisation par le compilateur ici.

```
C(int):
push rbp
mov rbp, rsp
sub rsp, 12
mov DWORD PTR [rbp-12], edi
mov DWORD PTR [rbp-8], 7
mov DWORD PTR [rbp-4], 5
mov edx, DWORD PTR [rbp-8]
mov eax, DWORD PTR [rbp-4]
add eax, edx
imul eax, DWORD PTR [rbp-12]
pop rbp
ret
```



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    return B(var + 1);
}
```

Initialisation du tableau.

C

- var : 300
- tableau : [7][5]

B

- var : 6
- multiple : 50.0f

A

- var : 5



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    return B(var + 1);
}
```

Retour de C à B, donc
retrait de C de la pile.

B

- var : 6
- multiple : 50.0f

A

- var : 5



EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    ➔ return B(var + 1);
}
```

Retour de B à A, donc
retrait de B de la pile.

A

- var : 5




EXEMPLE : LA PILE

```
int C(int var)
{
    int tableau[2] = { 7, 5 };
    return tableau[0] * var +
           tableau[1] * var;
}

int B(int var)
{
    float multiple = 50.0f;
    return C(var * multiple);
}

int A(int var)
{
    return B(var + 1);
}
```



Fin du programme lorsqu'il n'y a plus rien sur la pile.



DÉCLARATION STATIQUE EN C++

Tout ce qui est déclaré sans le mot clé « new » en C++ est alloué de manière statique (donc sur la pile).

- Classes et structures incluses.

La syntaxe est différente du C#.

```
int main(char* argv, int argc)
{
    Point2D point1;
    point1.x = 12;
    point1.y = 24;
}
```



RÉFÉRENCES C# VS STATIQUE C++

Attention à ne pas tomber dans le piège du « new ».

En C#, pour créer un nouvel objet, il faut systématiquement faire « new ».

En C++, si vous allouez sur la pile, **il n'y a pas de « new »**.

C#

```
static void Main(string[] args)
{
    Point2D point1 = new Point2D();
    point1.x = 12;
    point1.y = 24;
}
```

C++

```
int main(char* argv, int argc)
{
    Point2D point1;
    point1.x = 12;
    point1.y = 24;
}
```



ALLOCATION STATIQUE ET GESTION DE LA MÉMOIRE

Tout ce qui est alloué sur la pile est géré automatiquement par le langage.

Tant que la fonction s'exécute, tout ce qui est alloué statiquement reste en mémoire.

Dès que la fonction arrête, tout ce qui fut alloué par la fonction est « retiré » de la mémoire.



LE TAS

Ou « Heap »...où il y a
allocation dynamique.



LIMITATIONS DE LA PILE

La Pile est utile pour les cas où l'on sait, à la compilation, la quantité de données que l'on a besoin.

Que faire lorsque la quantité de données est inconnue ?

- Lorsque l'utilisateur peut saisir un nombre arbitraire de données ?

Le « TAS » à la rescousse !



LE TAS (HEAP)

La « Heap » (ou le tas en français), est un emplacement mémoire où l'on peut allouer tout ce que l'on veut, au nombre que l'on veut, au moment où l'on veut.

- Tant que l'on ne manque pas de mémoire...bien sûr!



COMMENT FONCTIONNE LE TAS ET COMMENT ON L'UTILISE ?

WARNING!

ANALOGIE IDIOTE



LE TAS — UN GRAND LOT DE CASIERS





LE TAS — LOTS DE CASIERS (1 DE 3)

Vous louez un casier. On vous donne un coupon et vous pouvez y mettre ce que vous désirez dedans.

Sur le coupon, vous avez un numéro de case. Vous souffrez de problèmes de mémoire et, sans le coupon, ne pouvez vous rappeler du numéro de case.

Lorsque vous n'avez plus besoin du casier, vous retournez le coupon et quelqu'un d'autre peut utiliser la case.





LE TAS — LOTS DE CASIERS (2 DE 3)

Vous pouvez demander le nombre de casiers que vous voulez.

Si vous perdez le coupon, la case est perdue pour toujours et personne d'autre ne l'utilisera jamais.





LE TAS — LOTS DE CASIERS (3 DE 3)

Si vous êtes un gros demandeur, on peut vous donner un lot de cases consécutives. Le coupon que l'on vous donne indique la première case.

Vous devez cependant vous rappeler combien de cases vous avez réservé et faire attention à ne pas utiliser la case de quelqu'un d'autre.

Lorsque vous rendez le coupon, toutes les cases réservés sont libérés.





L'ANALOGIE FONCTIONNE AVEC LE TAS.



LE TAS — LOTS DE CASIERS (1 DE 2)

Vous louez un casier. On vous donne un coupon et vous pouvez y mettre ce que vous désirez dedans.



Vous faites un « new » d'un type précis. Le mot clé « new » vous renvoie une adresse mémoire.

Sur le coupon, vous avez un numéro de case. Vous souffrez de problèmes de mémoire et, sans le coupon, ne pouvez vous rappeler du numéro de case.



Vous conservez cette adresse mémoire dans un pointeur. Si vous perdez le pointeur, vos données sont perdues à jamais.

Lorsque vous n'avez plus besoin du casier, vous retournez le coupon et quelqu'un d'autre peut utiliser la case.



Quand les données ne sont plus nécessaires, faites un « delete » sur votre pointeur.



LE TAS — LOTS DE CASIERS (2 DE 2)

Si vous êtes un gros demandeur, on peut vous donner un lot de cases consécutives. Le coupon que l'on vous donne indique la première case.



Vous pouvez allouer un seul élément ou allouer un tableau d'éléments consécutifs. Le pointeur retourné par le « new » retourne toujours l'adresse du premier élément.

Vous devez cependant vous rappeler combien de cases vous avez réservé et faire attention à ne pas utiliser la case de quelqu'un d'autre.



Contrairement au C#, il n'est pas possible de savoir la longueur d'un tableau. Vous devez conserver cette information ailleurs.

Lorsque vous rendez le coupon, toutes les cases réservés sont libérés.



Pour supprimer un tableau, il faut faire un « delete[] » (notez les « [] »).



EN CODE, ÇA DONNE QUOI ?



LES POINTEURS EN CODE (1 DE 2)

Vous faites un « new » d'un type précis. Le mot clé « new » vous renvoie une adresse mémoire.



```
int* monEntier = new int(3);
```

Vous conservez cette adresse mémoire dans un pointeur. Si vous perdez le pointeur, vos données sont perdues à jamais.



```
//NON!!!  
monEntier = nullptr;
```

Quand les données ne sont plus nécessaires, faites un « delete » sur votre pointeur.



```
delete monEntier;
```



LES POINTEURS EN CODE (2 DE 2)

Vous pouvez allouer un seul élément ou allouer un tableau d'éléments consécutifs. Le pointeur retourné par le « new » retourne toujours l'adresse du premier élément.



```
int* monEntier = new int(3);  
int* monTableau = new int[10];
```

Contrairement au C#, il n'est pas possible de savoir la longueur d'un tableau. Vous devez conserver cette information ailleurs.



```
int size = 10;  
int* monTableau = new int[size];
```

Pour supprimer un tableau, il faut faire un « delete[] » (notez les « [] »).



```
delete[] monTableau;
```



ET COMMENT ON UTILISE UN POINTEUR ?



DÉRÉFÉRENCEMENT DE POINTEURS

Pour rappel, un pointeur est une adresse.
Si vous changez le pointeur, vous
CHANGEZ L'ADRESSE!!!!

`int* monEntier = new int(3);
monEntier = 4; //NON!!!!`

Pour changer ce qui est pointé par un
pointeur (donc, dans le HEAP à l'adresse
du pointeur), il faut déréférencer le
pointeur **AVANT** avec l'opérateur « * ».

`*monEntier = 4; //Oui`

Sur un tableau, l'opérateur « [] »
effectue déjà le déréférencement.

`int* monTableau = new int[10];
monTableau[0] = 4; //Ok`



REMARQUES SUR LE POINTEUR DE TABLEAU

Puisqu'un pointeur de tableau est l'adresse du premier élément, ceci est valide.



```
int* monTableau = new int[10];  
monTableau[0] = 4; //Oui  
*monTableau = 4; //Même chose
```

Si vous ajoutez au pointeur la taille d'un élément¹, vous faites passer le pointeur à la position suivante du tableau. Vous pouvez faire cela en incrémentant votre pointeur.



```
int* monTableau = new int[10];  
monTableau[1] = 4; //Oui  
monTableau++;  
*monTableau = 4; //Même chose
```

¹ - C'est plus complexe que ça en vérité (si le compilateur ajoute du padding), mais le principe est correct.



LA PILE ET LES POINTEURS

Les données sont stockées dans le Tas, mais le pointeur (l'adresse mémoire), est dans la pile.

```
int* tableau = new int[5];  
for(int i = 0; i < 5; i++)  
{  
    tableau[i] = i + 1;  
}
```



SUPER

HYPER

MEGA

GIGA

ULTRA

IMPUR

ANTI

TOUJOURS FAIRE VOS
DELETES!



ÊTRE VOTRE PROPRE « GARBAGE COLLECTOR »

Tout ce qui est placé dans le Tas est conservé tant qu'il n'est pas mentionné explicitement de le supprimer.

➡ `int* monEntier = new int(3);`
`int* monTableau = new int[10];`

Pour supprimer un élément, utilisez « delete ».

➡ `delete monEntier;`

Pour supprimer un tableau, utilisez « delete[] ».

➡ `delete[] monTableau;`

SI VOUS NE LE FAITES PAS,
VOUS AVEZ UNE FUITE DE
MÉMOIRE.

ET C'EST BAD!!!



POINTEURS ET CLASSES

Rien ne vous empêche d'allouer des classes sur le Tas.



```
Point2D* point2d = new Point2D;
```

Pour y accéder, il faut aussi le déréférencer, soit avec l'opérateur « * », soit avec l'opérateur « -> ».



```
point2d.x = 2; //NON!!!!
```

```
(*point2d).x = 2; //Oui
```

```
point2d->x = 2; //Même chose
```

Le « delete » fonctionnent comme le reste.




```
delete point2d;
```



MANIPULATION DE POINTEURS

Vous pouvez copier l'adresse d'un pointeur dans un autre.



```
Point2D* point2d = new Point2D;  
Point2D* autre = point2d;
```



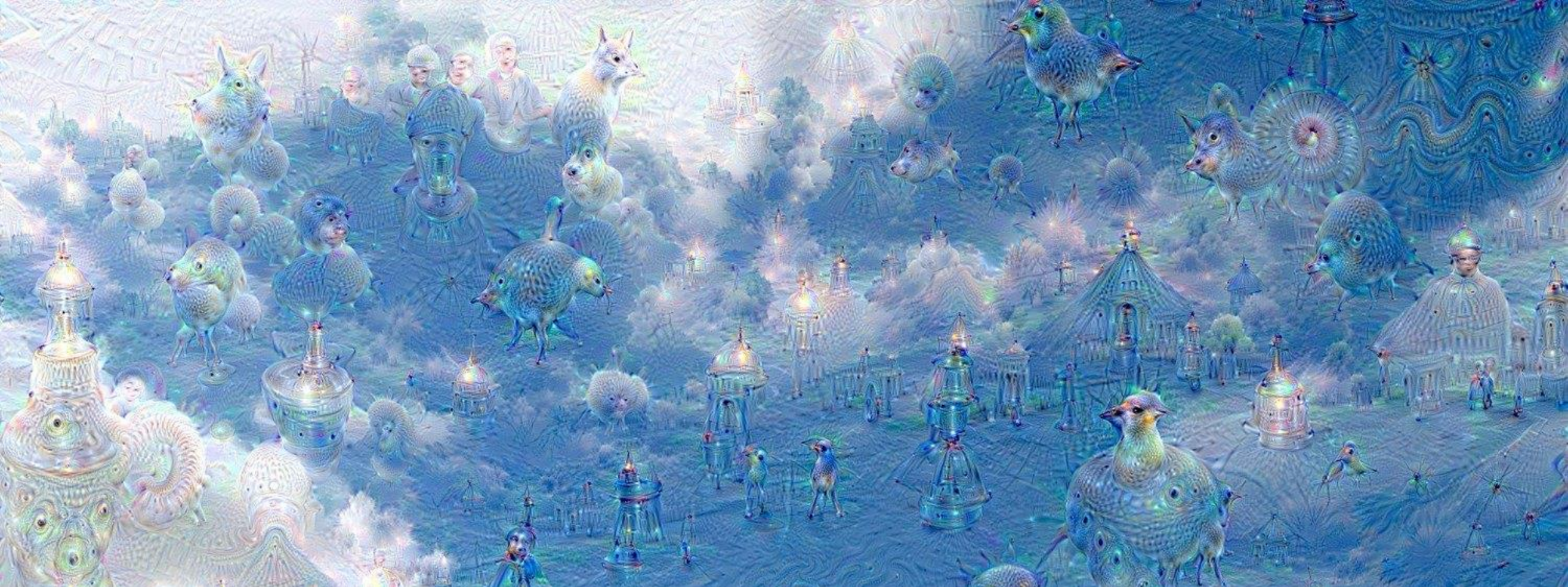
POINTEURS NULL

En C#, vous pouvez avoir des références « null ». En C++, vous pouvez avoir des pointeurs « null ».



```
Point2D* point2d = nullptr;
```

Un pointeur « null » est égal à 0.



FIN