

C++



Les pointeurs
14 | APR | 2020
Dorian.H Mekni

C++

Les pointeurs | déclaration -> C

Déclaration de pointeurs en C++ relativement standard
si une étude approfondi du C a été au préalable.

```
1  #include <iostream>
2
3  int main()
4  {
5
6      int x = 12;
7      int *pExample{nullptr};
8
9      pExample = &x;
10
11     std::cout << *pExample << std::endl;
12
13     return 0;
14
15 }
```



```
12
Program ended with exit code: 0
```

C++

Les pointeurs | allocation dynamique

voilà en C++ la syntaxe la plus standard,
voir ancienne pour l'allocation dynamique.

```
1  #include <iostream>
2
3  int main()
4  {
5
6      int *pExample{new int(12)}; // Le new Serait en C un malloc()
7
8      std::cout << *pExample << std::endl;
9
10     delete pExample;
11
12     return 0;
13
14 }
15
```



```
12
Program ended with exit code: 0
```

C++

Les pointeurs | Allocation dynamique ++

Cette méthode permettait de remplacer les fameux malloc() héritée du C

```
1 #include <iostream>
2
3 int main()
4 {
5
6     int *pExample{new int(12)};
7     int *tableau{new int[100]};
8
9     std::cout << *pExample << std::endl;
10    std::cout << *tableau << std::endl;
11
12    delete[] tableau;
13    delete pExample;
14    // Cela permet de vider l'espace mémoire allouée
15
16    return 0;
17
18 }
```



```
12
0
Program ended with exit code: 0
```

C++

Les pointeurs | Mémo

```
/*
```

```
Le pointeur unique -> Un seul propriétaire ayant accès à une ressource. Une fois le pointeur détruit, la ressource est libérée de la mémoire.
```

```
Le pointeur partagé -> plusieurs propriétaires peuvent accéder à la même ressource. Aussi longtemps que des éléments utilisent la ressource, cette dernière est mobilisée. Dès que les pointeurs sont détruits, la ressource est libérée.
```

```
Le pointeur faible -> Il vérifie que notre ressource existe, si c'est le cas on pourra le transformer en pointeur partagé
```

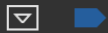
```
*/
```

C++

Les pointeurs | <memory>

Méthode dans laquelle il n'y aura pas de fuite mémoire
et grâce à laquelle nous n'avons pas à nous occuper de libérer de l'espace mémoire.
Ne pas oublier d'ajouter l'entête `#include<memory>`

```
3 int main()
4 {
5
6     std::unique_ptr<int> ptOperator{std::make_unique<int>(12)};
7
8     std::cout << *ptOperator << std::endl;
9
10    return 0;
11
12 }
13
```



```
12
Program ended with exit code: 0
```

C++

Les pointeurs | auto

Toujours dans le même exemple,
mais cette fois en allégeant la syntaxe avec l'utilisation d' auto ->

```
3 int main()
4 {
5
6     auto ptOperator{std::make_unique<int>(12)};
7
8     std::cout << *ptOperator << std::endl;
9
10    return 0;
11
12 }
13
```



```
12
Program ended with exit code: 0
```

C++

Les pointeurs | []

En revanche pour l'initialisation du tableau dans ce cas de figure, il nous faudra insérer les crochets pour que le compilateur comprenne qu'il ne s'agit pas d'une variable

```
1  #include <iostream>
2  #include <memory>
3  int main()
4  {|
5
6      auto tableau{std::make_unique<int[]>(100)};
7
8      return 0;
9
10 }
11
```


C++

Les pointeurs | pointeur partagé

Voilà la syntaxe pour l'initialisation d'un pointeur partagé

```
1  #include <iostream>
2  #include <memory>
3  int main()
4  {
5
6      std::shared_ptr<int> ptrNumber{std::make_shared<int>(12)};
7
8      return 0;
9
10 }
11
```

Les pointeurs | La conversion

Pour la conversion d'un pointeur unique à un pointeur brut, on utilise la méthode `.release()` au moment d'initialiser le pointeur en question sans oublier de l'effacer avec `delete`

```
1  #include <iostream>
2  #include <memory>
3  int main()
4  {
5
6      auto ptrNumber{std::make_unique<int>(12)};
7      int *pt = ptrNumber.release();
8
9      delete pt;
10     pt = nullptr;
11
12     return 0;
13
14 }
15
```

C++

->