A long-exposure night photograph of a city street. The image shows light trails from cars and streetlights, creating a sense of motion. The foreground is a dark, paved area with a brick pattern. In the background, there are buildings and more light trails. A semi-transparent white box is overlaid on the image, containing the text "Module 04 – Gestion dynamique de la mémoire".


# Module 04 – Gestion dynamique de la mémoire

# Objectifs

- Pointeurs et opérations
  - Pointeurs de données
  - Passage par copie de pointeur
  - Pointeurs de fonctions
- Mémoire dynamique (Tas)
  - Allocation / libération
  - Arithmétique des pointeurs, lien avec les tableaux
  - Fuite mémoire
- Notion de propriétaire (RAII)
- Notation UML : Composition / Agrégation

# Pointeurs

- Un pointeur est un type de données qui permet de représenter l'adresse mémoire d'une donnée ou d'une fonction



Address	Content	Name	Type	Value
90000000	00	iii	int	000000FF (255 <sub>10</sub> )
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	sss	short	FFFF (-1 <sub>10</sub> )
90000005	FF			
90000006	1F	ddd	double	1FFFFFFFFFFFFFFF (4.4501477170144023E-308 <sub>10</sub> )
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptr	int*	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

# Pointeurs sur une donnée

- Pour indiquer que l'on veut un pointeur, on utilise **l'étoile** (« \* ») lors de la déclaration précédée du type de la valeur pointée

```
int *p = nullptr;
```

- Pour obtenir l'adresse d'un lvalue, on utilise **l'opérateur** « & » (à ne pas confondre avec son utilisation pour déclarer des références)

```
int v = 42;  
int *p = &v;
```

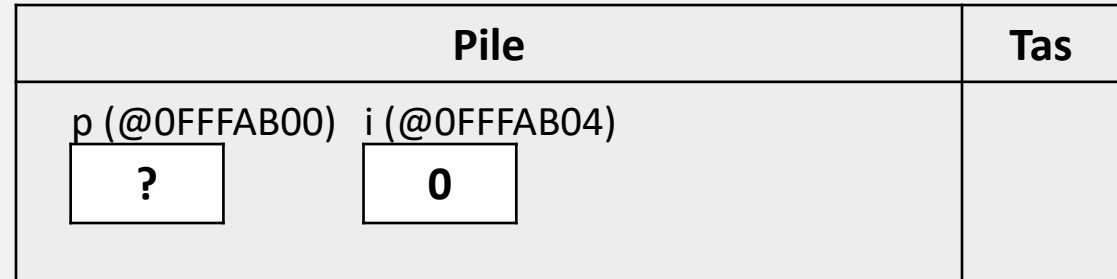
- Pour accéder à la valeur de l'adresse mémoire, on va utiliser **l'opérateur** « \* » (i.e. hors déclaration)

```
int v = 42;  
int *p = &v;  
std::cout << *p << std::endl;
```

# Pointeurs sur une donnée – Mémoire

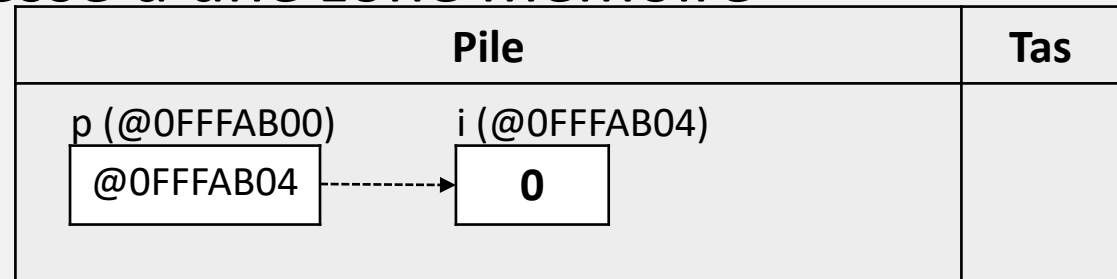
- La déclaration utilise \*

- `int* p;`
- `int i = 0;`



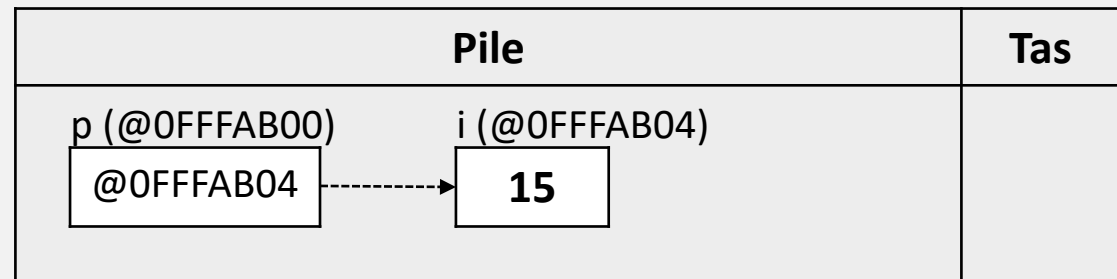
- & est l'opérateur obtenir l'adresse d'une zone mémoire

- `p = &i;`



- \* est l'opérateur de déréférencement et permet d'accéder à la valeur qui se trouve à une adresse :

- `*p = 15;`



# Pointeurs sur une donnée

- Une valeur de type pointeur étant lui-même une donnée, on peut aussi créer un pointeur sur un pointeur
  - Si on veut un pointeur sur un pointeur d'entier

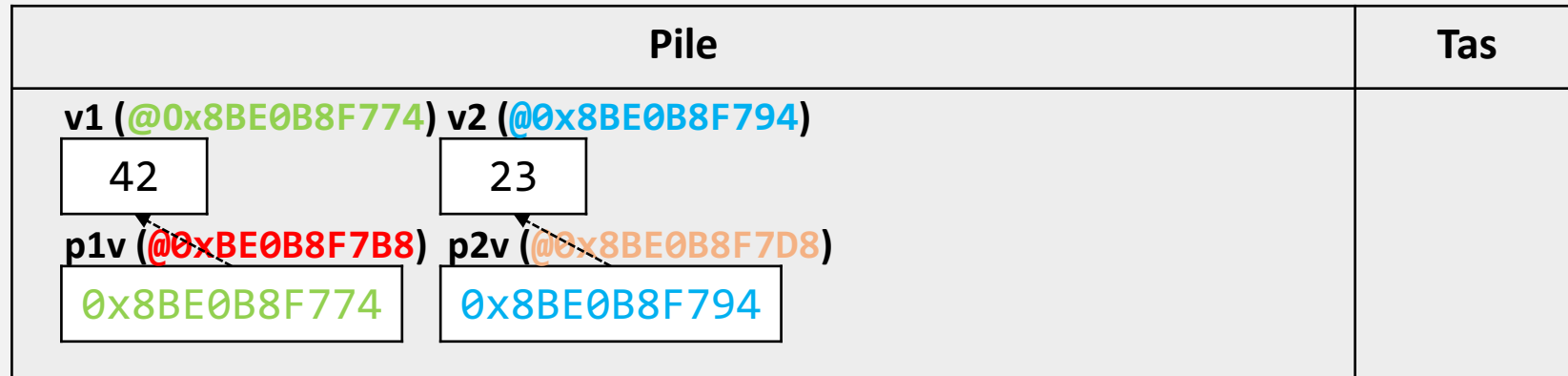
`int **p = nullptr; // On va lire le type de droite à gauche`  


```
int v = 42;
int *pv = &v;
int **pp = &pv;
std::cout << **pp << std::endl;
```



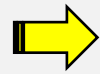
# Pointeurs sur une donnée – Mémoire – Démo 1

```
int v1 = 42;  
int v2 = 23;  
int* p1v = &v1;  
int* p2v = &v2;
```

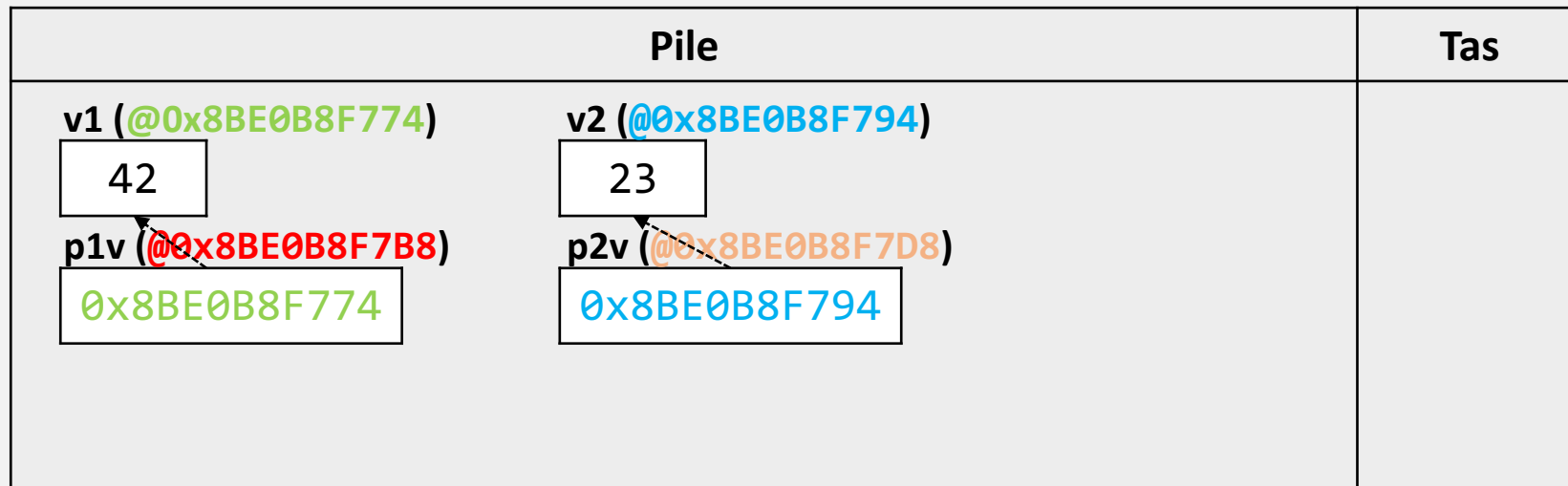


```
int - v1 (0x0000008BE0B8F774) = 42  
int - v2 (0x0000008BE0B8F794) = 23  
int* - p1v (0x0000008BE0B8F7B8) = 0x0000008BE0B8F774 -> 42  
int* - p2v (0x0000008BE0B8F7D8) = 0x0000008BE0B8F794 -> 23
```

# Pointeurs sur une donnée – Mémoire – Démo 2



```
p1v = &v2;  
*p1v = 11;  
int** ppv = &p1v;
```

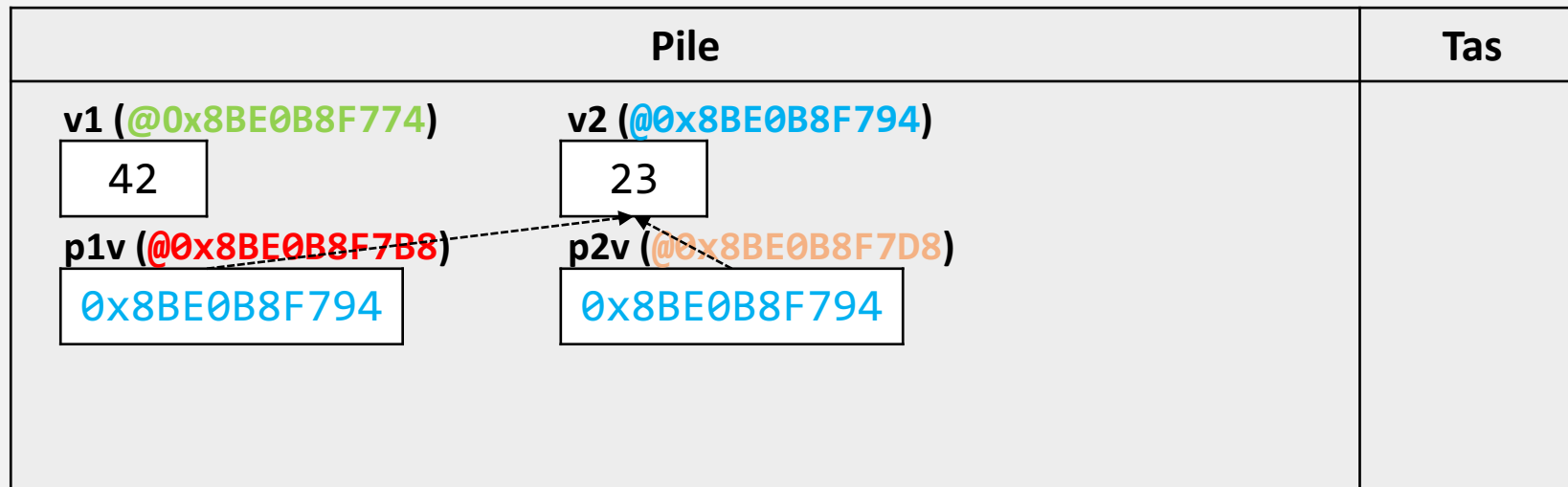


```
int - v1 (0x0000008BE0B8F774) = 42  
int - v2 (0x0000008BE0B8F794) = 23  
int* - p1v (0x0000008BE0B8F7B8) = 0x0000008BE0B8F774 -> 42  
int* - p2v (0x0000008BE0B8F7D8) = 0x0000008BE0B8F794 -> 23
```



# Pointeurs sur une donnée – Mémoire – Démo 2

```
p1v = &v2;  
*p1v = 11;  
int** ppv = &p1v;
```

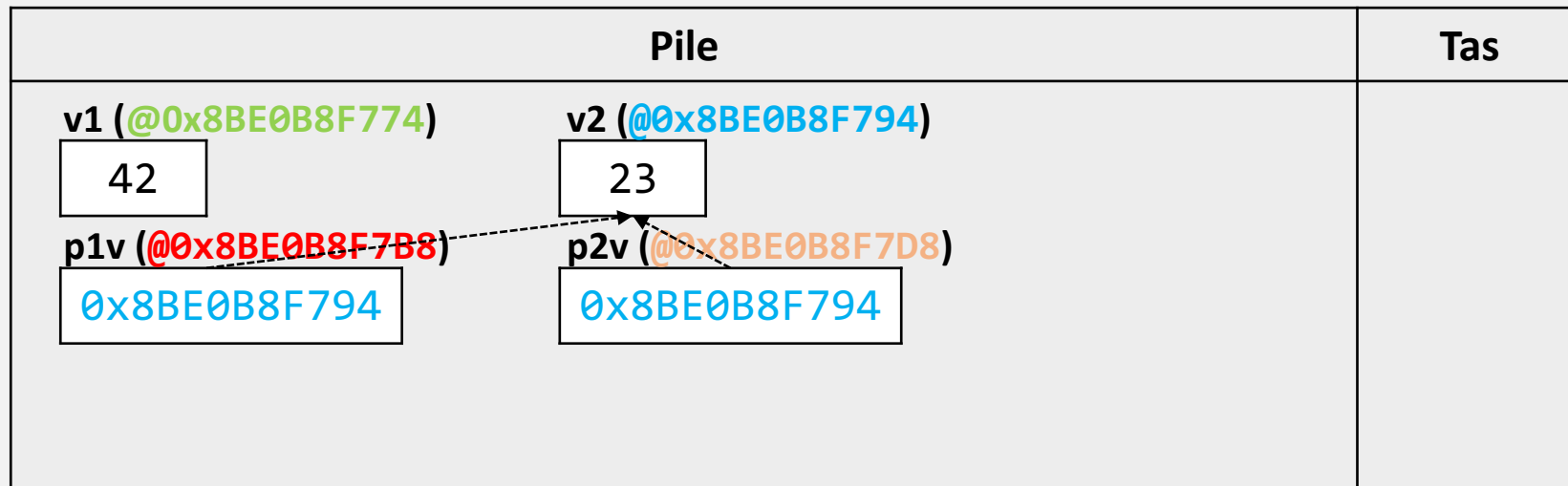


```
int - v1 (0x0000008BE0B8F774) = 42  
int - v2 (0x0000008BE0B8F794) = 23  
int* - p1v (0x0000008BE0B8F7B8) = 0x0000008BE0B8F794 -> 23  
int* - p2v (0x0000008BE0B8F7D8) = 0x0000008BE0B8F794 -> 23
```

# Pointeurs sur une donnée – Mémoire – Démo 2



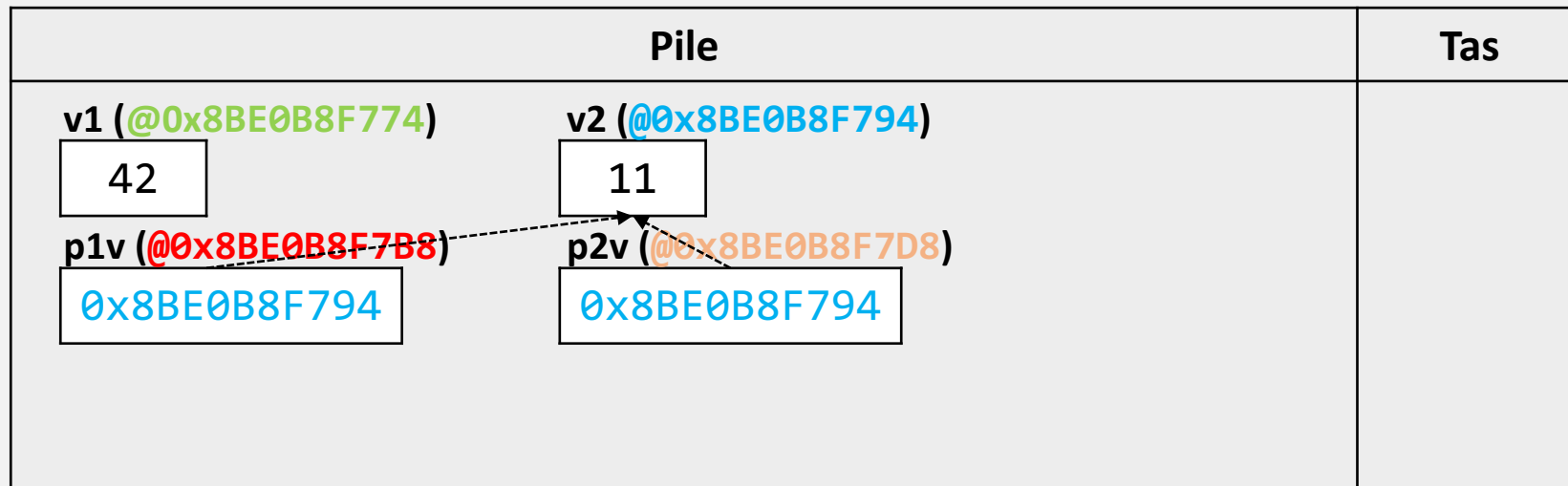
```
p1v = &v2;  
*p1v = 11;  
int** ppv = &p1v;
```



```
int - v1 (0x0000008BE0B8F774) = 42  
int - v2 (0x0000008BE0B8F794) = 23  
int* - p1v (0x0000008BE0B8F7B8) = 0x0000008BE0B8F794 -> 23  
int* - p2v (0x0000008BE0B8F7D8) = 0x0000008BE0B8F794 -> 23
```


# Pointeurs sur une donnée – Mémoire – Démo 2

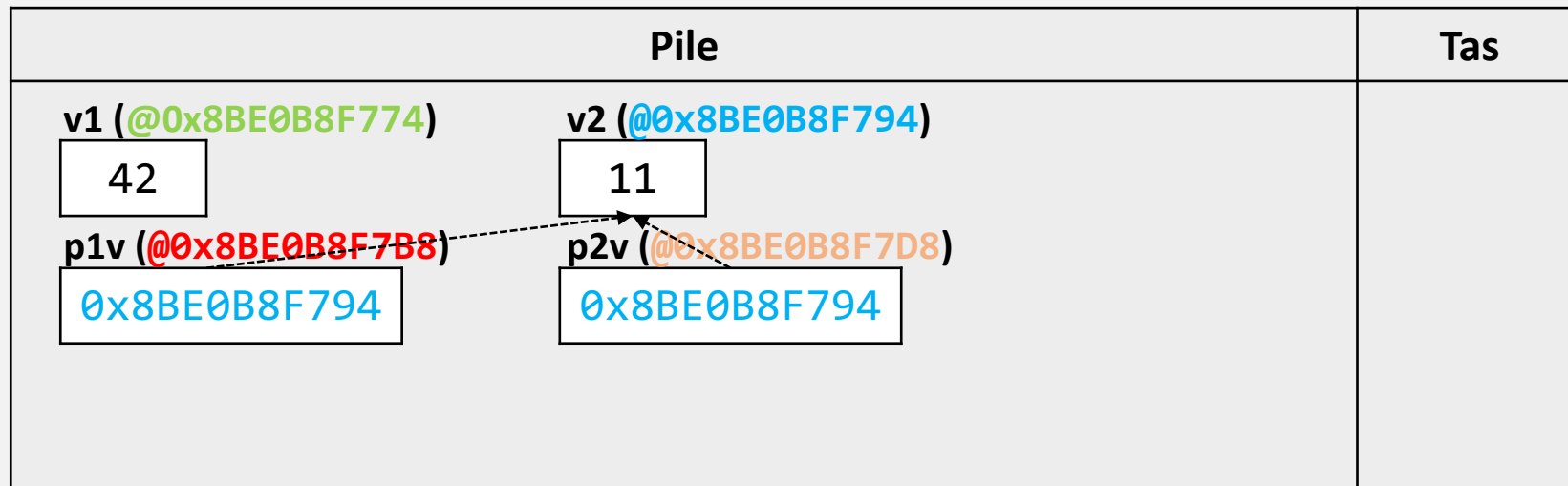
```
p1v = &v2;  
*p1v = 11;  
int** ppv = &p1v;
```



```
int - v1 (0x0000008BE0B8F774) = 42  
int - v2 (0x0000008BE0B8F794) = 11  
int* - p1v (0x0000008BE0B8F7B8) = 0x0000008BE0B8F794 -> 11  
int* - p2v (0x0000008BE0B8F7D8) = 0x0000008BE0B8F794 -> 11
```

# Pointeurs sur une donnée – Mémoire – Démo 2

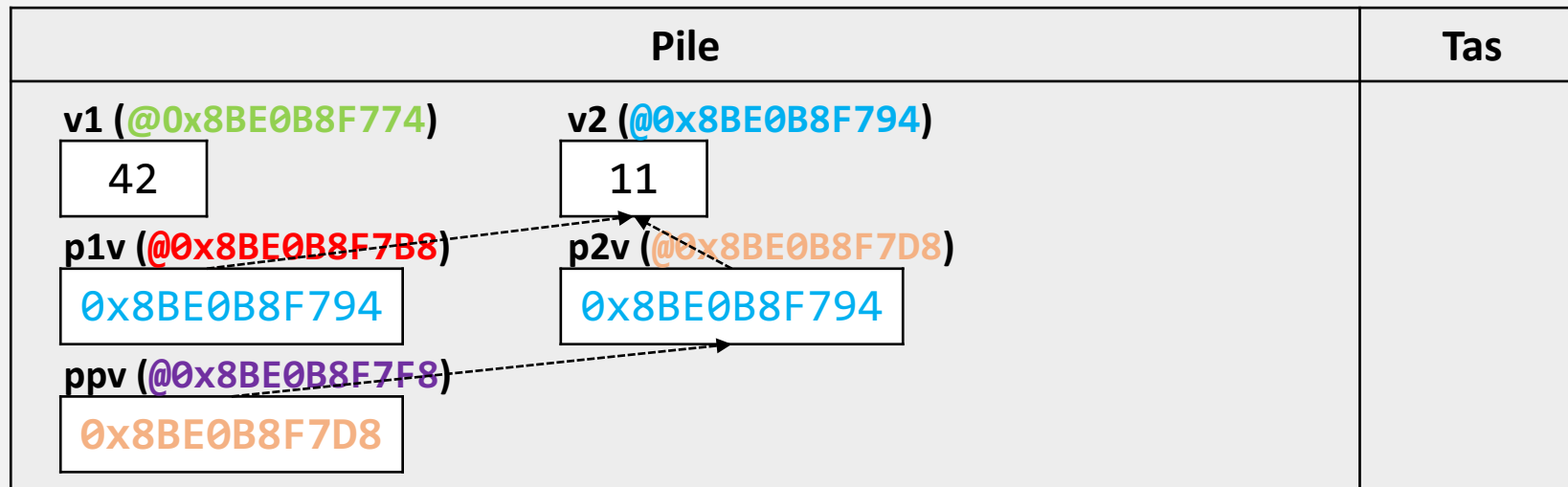
 `p1v = &v2;  
*p1v = 11;  
int** ppv = &p1v;`



```
int - v1 (0x0000008BE0B8F774) = 42
int - v2 (0x0000008BE0B8F794) = 11
int* - p1v (0x0000008BE0B8F7B8) = 0x0000008BE0B8F794 -> 11
int* - p2v (0x0000008BE0B8F7D8) = 0x0000008BE0B8F794 -> 11
```

# Pointeurs sur une donnée – Mémoire – Démo 2

```
p1v = &v2;  
*p1v = 11;  
int** ppv = &p2v;
```



```
int - v1 (0x0000008BE0B8F774) = 42  
int - v2 (0x0000008BE0B8F794) = 11  
int* - p1v (0x0000008BE0B8F7B8) = 0x0000008BE0B8F794 -> 11  
int* - p2v (0x0000008BE0B8F7D8) = 0x0000008BE0B8F794 -> 11  
int** ppv (0x0000008BE0B8F7F8) = 0x0000008BE0B8F7D8 -> 0x0000008BE0B8F794 ->-> 11
```

# Pointeurs sur une donnée – const

- Contrairement aux références qui sont constantes (les références pas les valeurs), les pointeurs ne le sont pas et peuvent donc être affectés et réaffectés à n'importe quel moment

```
int v1 = 42;  
int v2 = 23;  
int *p = &v1;  
[...]  
p = &v2;
```

# Pointeurs sur une donnée – const

- On peut utiliser le mot clef « const » pour rendre le pointeur constant et / ou la valeur constante

```
int v1 = 42;  
int v2 = 23;  
int * const pc = &v1;  
*pc = 23;  
pc = &v2; // Erreur compilation
```

```
int v1 = 42;  
int v2 = 23;  
const int *pvc = &v1;  
// ou const int * pvc = &v1;  
*pvc = 23; // Erreur compilation  
pvc = &v2;
```

```
int v1 = 42;  
int v2 = 23;  
const int* const pcvc = &v1;  
*pcvc = 23; // Erreur compilation  
pcvc = &v2; // Erreur compilation
```



# Cas des pointeurs objet

- L'opérateur « \* » n'est pas prioritaire sur l'opérateur point « . ». Il faut donc utiliser des parenthèses pour forcer la priorité
- C++ propose une écriture simplifiée « -> »


```
Televiseur t;  
Televiseur* pt = &t;  
  
// *pt.allumer(); // Erreur de compilation  
(*pt).allumer();  
// Écriture simplifiée :  
pt->allumer();
```

# Fonctions – Exemples – Passage par copie de pointeur

```
void echangerPointeurs(int *p_v1, int *p_v2) {  
    int temp = *p_v1;  
    *p_v1 = *p_v2;  
    *p_v2 = temp;  
}  
  
int v1 = 42;  
int v2 = 13;  
→ echangerPointeurs(&v1, &v2);  
// ...
```

Pile		Tas
v1 (@0FFFFAB00)	v2 (@0FFFFAB04)	
42	13	


# Fonctions – Exemples – Passage par copie de pointeur



```
void echangerPointeurs(int *p_v1, int *p_v2) {  
    int temp = *p_v1;  
    *p_v1 = *p_v2;  
    *p_v2 = temp;  
}  
  
int v1 = 42;  
int v2 = 13;  
echangerPointeurs(&v1, &v2);  
// ...
```

Pile		Tas
v1 (@0FFFAB00)	v2 (@0FFFAB04)	
42	13	
p_v1(@0FFFAB08)	p_v2 (@0FFFAB0C)	
@0FFFAB00	@0FFFAB04	
temp (@0FFFAB10)		
42		


# Fonctions – Exemples – Passage par copie de pointeur



```
void echangerPointeurs(int *p_v1, int *p_v2) {  
    int temp = *p_v1;  
    *p_v1 = *p_v2;  
    *p_v2 = temp;  
}  
  
int v1 = 42;  
int v2 = 13;  
echangerPointeurs(&v1, &v2);  
// ...
```

Pile		Tas
v1 (@0FFFAB00)	v2 (@0FFFAB04)	
13	13	
p_v1(@0FFFAB08)	p_v2 (@0FFFAB0C)	
@0FFFAB00	@0FFFAB04	
temp (@0FFFAB10)		
42		


# Fonctions – Exemples – Passage par copie de pointeur



```
void echangerPointeurs(int *p_v1, int *p_v2) {  
    int temp = *p_v1;  
    *p_v1 = *p_v2;  
    *p_v2 = temp;  
}  
  
int v1 = 42;  
int v2 = 13;  
echangerPointeurs(&v1, &v2);  
// ...
```

Pile		Tas
v1 (@0FFFAB00)	v2 (@0FFFAB04)	
13	42	
p_v1(@0FFFAB08)	p_v2 (@0FFFAB0C)	
@0FFFAB00	@0FFFAB04	
temp (@0FFFAB10)		
42		

# Fonctions – Exemples – Passage par copie de pointeur



```
void echangerPointeurs(int *p_v1, int *p_v2) {  
    int temp = *p_v1;  
    *p_v1 = *p_v2;  
    *p_v2 = temp;  
}  
  
int v1 = 42;  
int v2 = 13;  
echangerPointeurs(&v1, &v2);  
// ...
```

Pile		Tas
v1 (@0FFFFAB00)	v2 (@0FFFFAB04)	
13	42	
p_v1(@0FFFFAB08)	p_v2 (@0FFFFAB0C)	
<del>@0FFFFAB00</del>	<del>@0FFFFAB04</del>	
temp (@0FFFFAB10)		
<del>42</del>		

# Fonctions – Exemples – Passage par copie de pointeur

```
void echangerPointeurs(int *p_v1, int *p_v2) {  
    int temp = *p_v1;  
    *p_v1 = *p_v2;  
    *p_v2 = temp;  
}  
  
int v1 = 42;  
int v2 = 13;  
echangerPointeurs(&v1, &v2);  
// ...
```



Pile		Tas
v1 (@0FFFAB00) 13	v2 (@0FFFAB04) 42	

v1 et v2 modifiées !



# Pointeurs de fonctions

- Déclaration :

<type retour> (\*<nomVariable>)([<listeParamètres>]

```
void (*p_saluer)(const std::string&)
```

```
void demoPointeurFonction(void (*p_saluer)(const std::string&))  
{  
    p_saluer("Pierre-Francois");  
    // ou (*p_saluer)("Pierre-Francois");  
}
```

# Pointeurs de fonctions

```
void demoPointeurFonction(void (*p_saluer)(const std::string&)) {  
    p_saluer("Pierre-Francois");  
    // ou (*p_saluer)("Pierre-Francois");  
}
```

```
void saluerFrancais(const std::string& p_nom) {  
    std::cout << "Bonjour " << p_nom << " !" << std::endl;  
}
```

```
void saluerBasque(const std::string& p_nom) {  
    std::cout << "Kaixo " << p_nom << " !" << std::endl;  
}
```

```
demoPointeurFonction(saluerFrancais);  
demoPointeurFonction(&saluerBasque);  
demoPointeurFonction(  
    [] (const std::string& p_nom)  
    {  
        std::cout << "Hello " << p_nom << " !" << std::endl;  
    }  
);
```

# Allocation dynamique de mémoire

- Pour allouer de la mémoire dynamiquement, on utilise l'opérateur « **new** » suivi du type de donnée à allouer
- « **new** » si l'allocation est un succès, il renvoie un pointeur sur la première case mémoire de la donnée
- Il y a principalement deux opérations qui sont réalisées par l'opérateur « **new** » :
  - Allocation dynamique de la mémoire dans le tas (Réservation de `sizeof(<type>)`)
  - Appel du constructeur décrit

```
Televiseur* pt = new Televiseur();
```

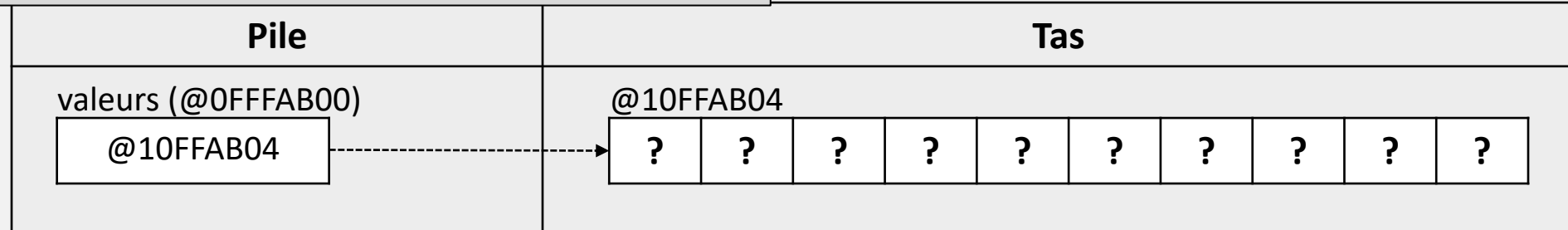
# Libération dynamique de mémoire

- Pour libérer / désallouer de la mémoire dynamiquement, on utilise l'opérateur « **delete** » suivi de l'adresse
- Il est conseillé et **obligatoire** dans le cours, d'affecter la valeur « **nullptr** » après chaque « **delete** »
- Il y a principalement deux opérations qui sont réalisées par l'opérateur « delete » :
  - Appel du destructeur
  - Libération de la mémoire

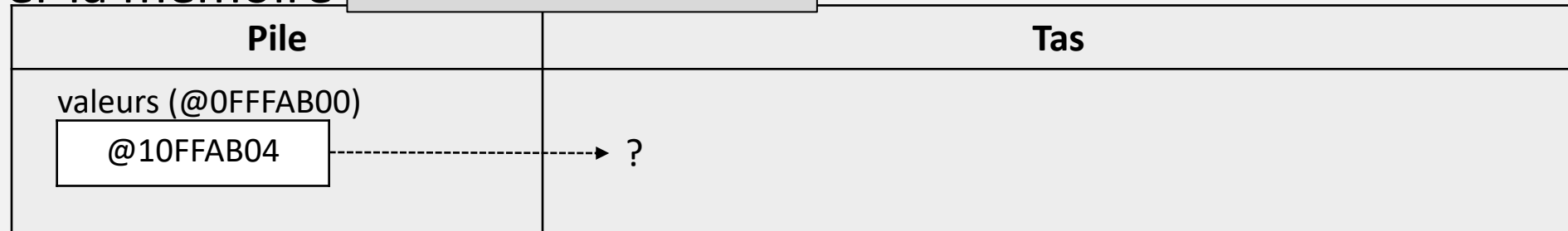
```
delete pt;  
pt = nullptr;
```

# Allocation dynamique de mémoire – Tableaux

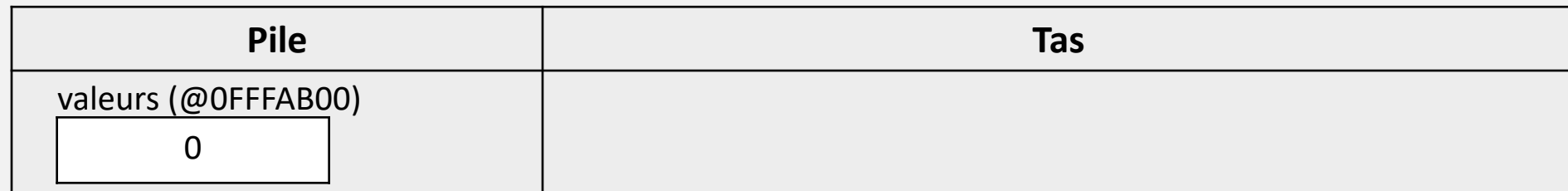
- `int* valeurs = new int[10];`



- Libérer la mémoire `delete[] valeurs;`



- nullptr peut être assigné au pointeur (bonne pratique) : `valeurs = nullptr;`



# Arithmétique pointeurs

- Les pointeurs sont des entiers. On peut appliquer des opérations d'addition/soustraction
- Pour réaliser ces opérations, le compilateur va utiliser la quantité de mémoire nécessaire à représenter une donnée (`sizeof(<typePointé>)`)
- L'indice des tableaux correspond à cet arithmétique :
  - Si `t` est de type `int*` (`int []`), qu'un `int` se code sur 4 octets et que `t = 0xFFA0`
  - $t[0] \Leftrightarrow *(t + 0) = *(0xFFA0 + 0 * 4) = *(0xFFA0)$
  - $t[1] \Leftrightarrow *(t + 1) = *(0xFFA0 + 1 * 4) = *(0xFFA4)$
  - $t[2] \Leftrightarrow *(t + 2) = *(0xFFA0 + 2 * 4) = *(0xFFA8)$

# Arithmétique pointeurs

```
constexpr int N = 5;
int valeurs[N] = { 1, 2, 3, 4, 5};
int* pvalueActuelle = valeurs;
int* pfin = valeurs + N;

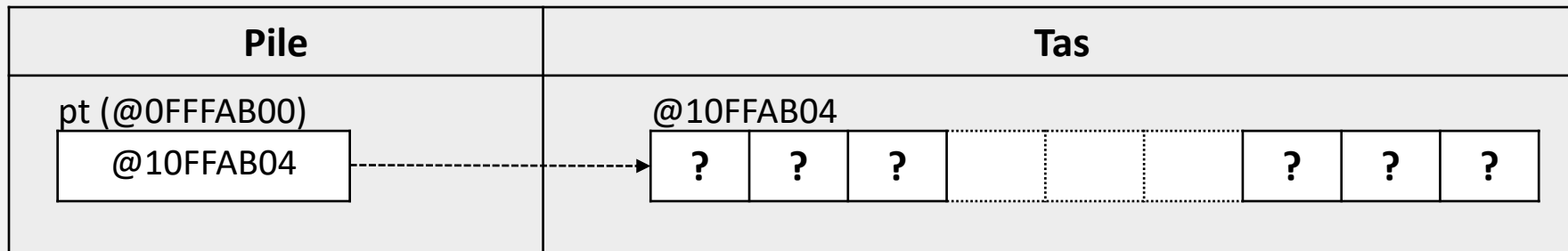
while (pvalueActuelle < pfin)
{
    std::cout
        << "0x" << std::hex << pvalueActuelle
        << std::dec << " : " << *pvalueActuelle++
        << std::endl;
}
```

```
0x0000005AC571F6DC : 1
0x0000005AC571F6E0 : 2
0x0000005AC571F6E4 : 3
0x0000005AC571F6E8 : 4
0x0000005AC571F6EC : 5
```



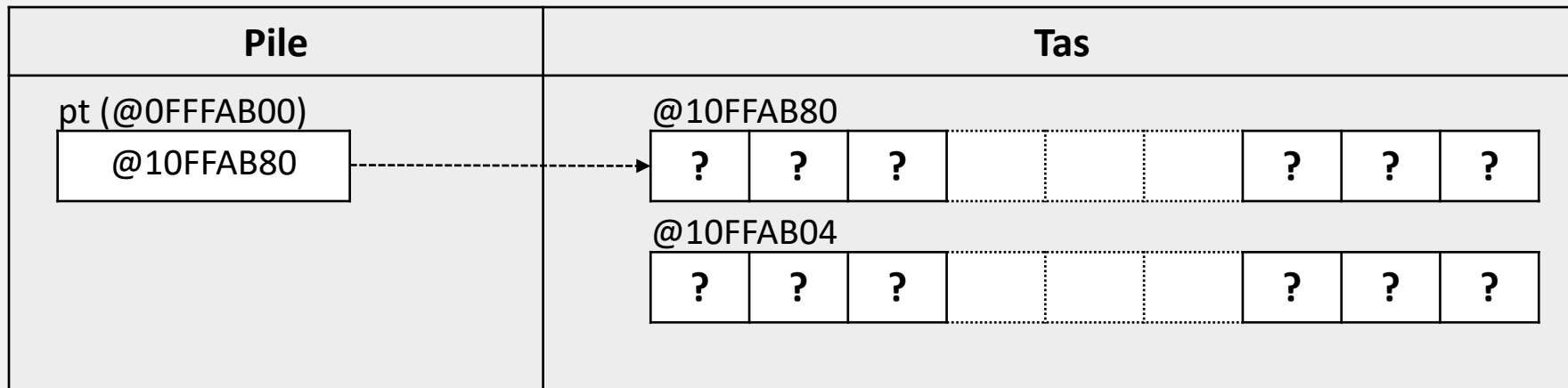
# Fuite de mémoire (memory leak)

```
void maFonction()  
{  
    int *pt = new int[100];  
    // sans delete  
}  
  
for(;;)  
{  
    maFonction();  
}
```



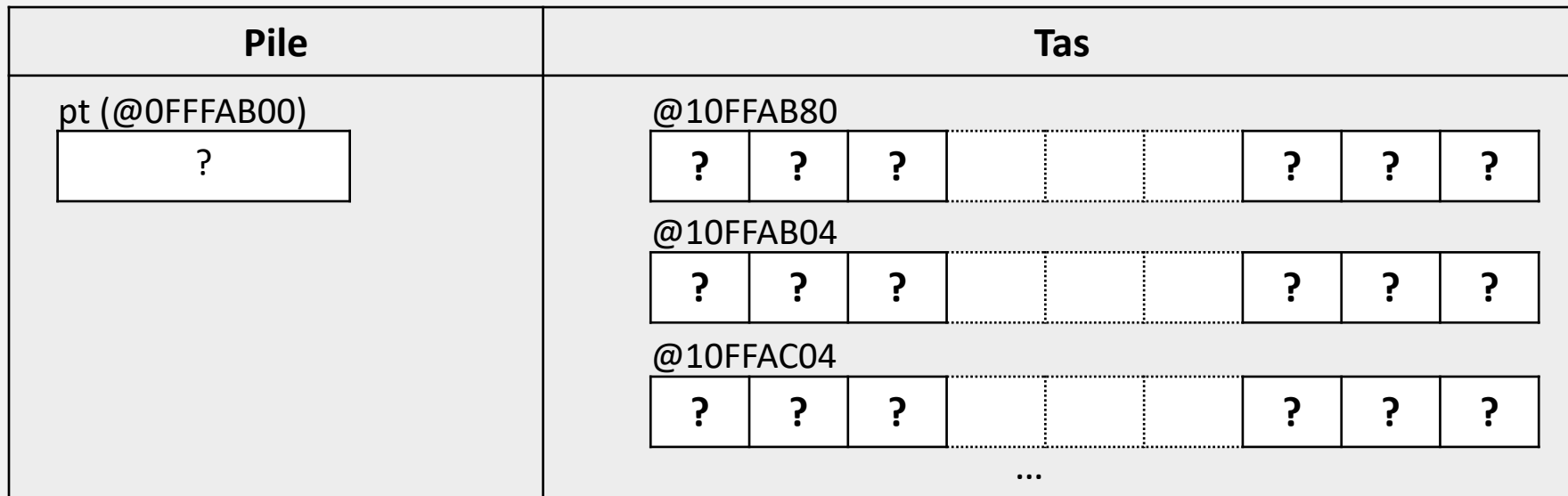
# Fuite de mémoire (memory leak)

```
void maFonction()  
{  
    int *pt = new int[100];  
    // sans delete  
}  
  
for(;;)  
{  
    maFonction();  
}
```



# Fuite de mémoire (memory leak)

```
void maFonction()  
{  
    int *pt = new int[100];  
    // sans delete  
}  
  
for(;;)  
{  
    maFonction();  
}
```



=> Out of memory

# Comment éviter les fuites mémoires ?

- RAI : Resource Acquisition Is Initialization
  - Notion de propriétaire des ressources allouées
  - On peut transférer cette propriété (exemple : constructeur ou opération d'affectation par déplacement)
  - Le constructeur alloue des ressources
  - Le destructeur les libère
- Utiliser des pointeurs intelligents (smart pointer) – **Non vus en cours**
  - **std::unique\_ptr** : peut stocker un pointeur.
    - On peut créer ce type d'objet avec **std::make\_unique**<<Type>>([paramConstructeur])
    - Il ne peut y avoir qu'un objet de ce type vers cette adresse. Il faut utiliser **std::move** pour transférer la propriété.
  - **std::shared\_ptr** : peut stocker un pointeur qui peut être partagé par plusieurs objets de ce type.

# Constructeur et opérateur= par déplacement

```
class NewEtDep {  
public:  
    NewEtDep();  
    NewEtDep(NewEtDep&& p_aDeplacer);  
    ~NewEtDep();  
  
    NewEtDep& operator=(NewEtDep&& p_aDeplacer);  
  
private:  
    int* m_donnee;  
};
```

# Constructeur et opérateur= par déplacement

```
NewEtDep::NewEtDep(NewEtDep&& p_aDeplacer)
{
    this->m_donnee = p_aDeplacer.m_donnee;
    p_aDeplacer.m_donnee = nullptr;
}
```

On déplace l'adresse de la mémoire contenant les données => évite de faire un **new**, une **copie** et un **delete** au moment du destructeur

```
NewEtDep::~~NewEtDep() {
    if (this->m_donnee) {
        delete[] this->m_donnee;
        this->m_donnee = nullptr;
    }
}
```

# Constructeur et opérateur= par déplacement

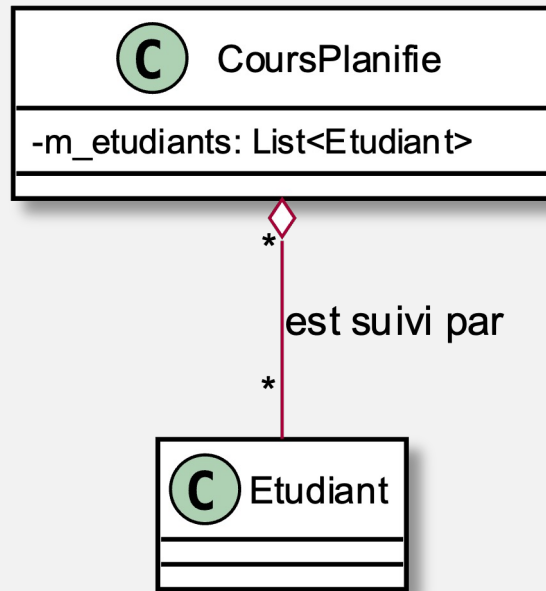
```
NewEtDep& NewEtDep::operator=(NewEtDep&& p_aDeplacer)
{
    if (this->m_donnee) {
        delete[] this->m_donnee;
        this->m_donnee = nullptr;
    }
    this->m_donnee = p_aDeplacer.m_donnee;
    p_aDeplacer.m_donnee = nullptr;

    return *this;
}
```



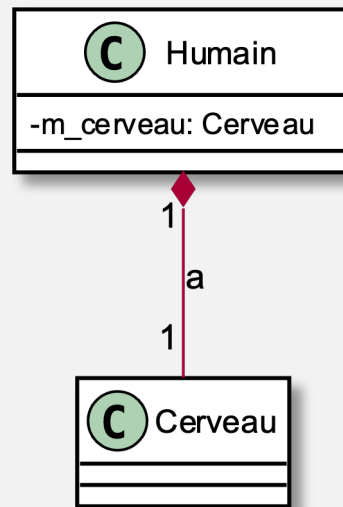
# Notation UML – Composition / agrégation

- Composition vs agrégation
  - Agrégation : les éléments existent de manière indépendante.
    - Exemple : l'étudiant existe même si le cours n'existe pas



# Notation UML – Composition / agrégation

- Composition vs agrégation
  - Composition : l'élément « enfant » n'existe pas s'il n'a pas de parent
    - Exemple : le cerveau n'existe pas seul, si on « efface » l'humain, on « efface » aussi le cerveau
    - Du côté du losange plein, la cardinalité est obligatoirement 1



# Références

- [https://fr.wikipedia.org/wiki/Pointeur \(programmation\)](https://fr.wikipedia.org/wiki/Pointeur_(programmation))
- <https://en.cppreference.com/w/cpp/memory>
- <https://learn.microsoft.com/en-us/cpp/cpp/move-constructors-and-move-assignment-operators-cpp>