Gestion de l’allocation dans le tas

Comment apporter des solutions au problème de la gestion de la mémoire

Allouer/Désallouer la mémoire dans le tas

- Allocation/Désallocation de la mémoire:

```
auto vec = new std::vector<int>();
delete vec;

auto c = new std::complex(0, -1);
std::cout << *c << std::endl;
delete c;
```

Allouer/Désallouer la mémoire dans le tas

- Allocation/désallocation d'un tableau

```
int number_of_characters = 10;
char* memory = new
          char[number_of_characters];
for(char* start = memory;
number_of_characters -- > 0; start ++)
    *start = 'a';
delete [] memory;
```

Pourquoi allouer de la mémoire dans le tas ?

- Autre stratégie d'allocation mémoire
  - **Mémoire globale:** variable partagée par toutes les fonctions et taille connue au moment de la création.
  - **Mémoire locale:** automatiquement allouée au moment de l'entrée dans la fonction et désallouée au moment de la sortie de la fonction. Taille a priori connue au moment de la création, peut-être parfois dynamique (cf. **`alloca`**).

Allocation dans le tas

- Supporte des références multiples sur le même bloc mémoire
- Supporte d'allouer un bloc mémoire en fonction du besoin
- Supporte d'ajuster la taille d'un bloc mémoire en fonction du besoin
- Date de création et de suppression contrôlée a priori par le programmeur.

Quels sont les difficultés ?

- Oublier de libérer la mémoire

  => Fuite de mémoire
- Libérer la mémoire avant que tous les opérations accédant à la mémoire soient terminées

  => Erreur d’exécution ou calcul potentiellement erroné.

Quelles sont les autres erreurs possibles ?

- Erreur dans la suppression de la mémoire
  - Allocation d'un tableau :

    auto x = new int[10];
  - Supression d'un élément:

    delete x;

Quelles sont les autres limitations ?

```
std::vector<int*> v;
for(int number_of_integers =10;
    number_of_integers > 0;
    number_of_integers --)
    v.push_back(
          new int(number_of_integers))
```

- La destruction du tableau ne détruit pas les blocs mémoires auxquels le tableau fait référence

Comment éviter ces problèmes (1)

- **Solution 1 :** bloc mémoire de taille définie au moment de la compilation

  ```
  std::array<int, 3> a = {1, 2, 3};
  ```

- La mémoire stockant les données est allouée dans le tas.
- La recopie d'un tableau est rapide, simplement le pointeur pointant sur la mémoire
- La mémoire est libérée au moment de la destruction du tableau

Comment éviter ces problèmes (2)

- **Solution 2 :** Les pointeurs dits intelligents
  - Pointeurs faisant référence à la mémoire qui a été alloué
  - Pointeurs gérant les informations nécessaires pour déterminer quand il faut désallouer la mémoire

La référence unique à une ressource

- **std::unique_ptr:**

  Pointeur référençant une valeur ou un tableau de valeurs qui lui est associé.

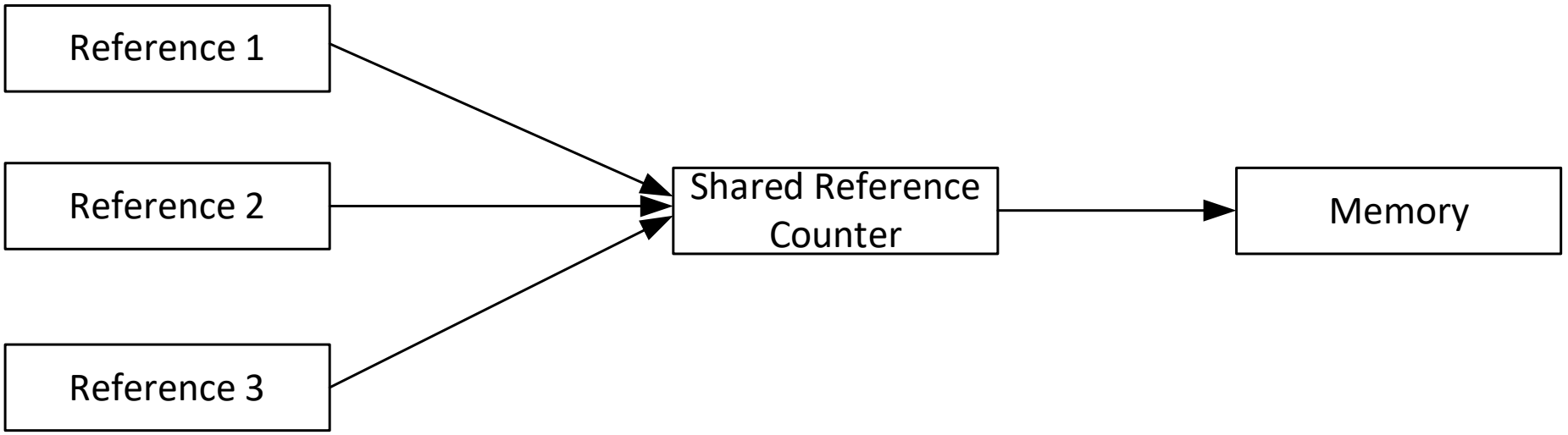
- Il est supposé être le seul gestionnaire de cette valeur ou tableau de valeurs.
- Il détruit la mémoire allouée pour la valeur ou le tableau alloué pour les valeurs au moment de sa destruction.

TD – Partie 1

- Gérer des buffers de taille fixe pour lire des données d’un fichier.
- Utiliser des std::unique_ptr pour implanter des matrices

Les références partagées sur une ressource

- `std::shared_ptr`

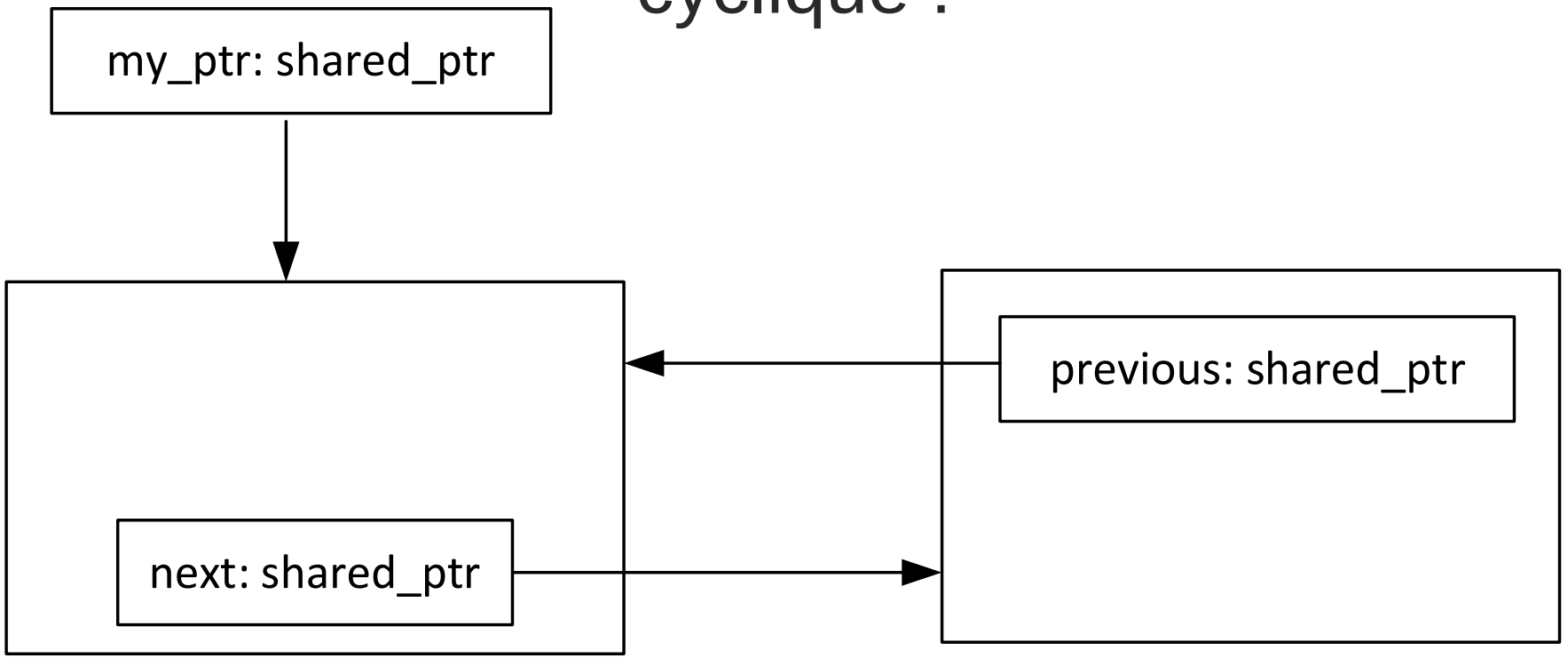


Fonctionnement d'un `shared_ptr`

- Chaque fois qu'une nouvelle référence est crée par duplication d'un `shared_ptr`
  - Le compteur de référence est incrémenté
- Chaque fois qu'un `shared_ptr` est détruit:
  - Le compteur de référence est incrémenté, si le compteur est à zéro, la mémoire contenant le compteur est détruite ainsi que la mémoire référencée.

Limites des `shared_ptr`

- Impossible de détruire une référence cyclique :



Introduction des `weak_ptr`

- `weak_ptr:` Maintient une référence aux données référencées par un `shared_ptr` tant qu'elles sont définies.
- Ne gère ni l'accès aux données, ni la persistance aux données.
- Permet d'obtenir un `shared_ptr` si la référence est encore valide.

TD – Partie 2

- Manipulation des shared_ptr
- Manipulation des weak_ptr

Allocation de mémoire non-initialisée

- Allocation par l'opérateur `new`:

```
auto array_of_strings
                = new std::string[10];
```

Alloue une zone mémoire nécessaire pour stocker au moins 10 instances de `std::string`.

Appelle pour chacune des entrées le constructeur pour initialiser la mémoire.

Allocation de mémoire non-initialisée

- C++ offre un ensemble de primitives pour permettre d'allouer un pool de mémoire non initialisée.

Intérêt: la zone mémoire est crée et il est possible de l'initialiser par recopie ou en appellant l'opérateur `new` et en passant l'adresse de la mémoire.