TP9 - Consolidation

Dans ce TP, nous allons retravailler la notion d'ownership.

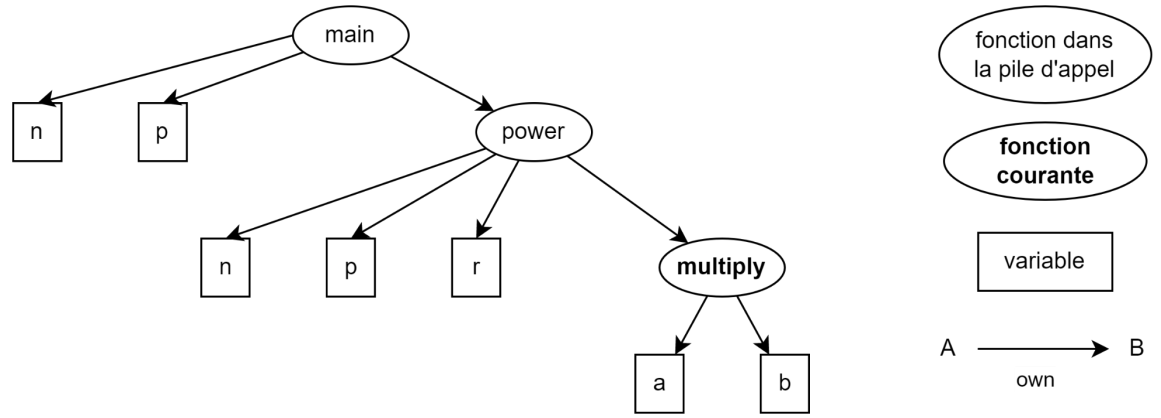
1- Introduction aux graphes d'ownership

Afin de pouvoir visualiser l'ownership, nous allons introduire un mode de représentation de celui-ci, que nous appelerons "graphe d'ownership". Il permet de décrire, dans un programme à un instant T, les entités valides ainsi que les relations d'ownership entre elles.

Voici quelques exemples de graphes d'ownership pour différents programmes à différents instants (on indique dans le code les appels effectués avec // ←). Essayez de comprendre comment les relations sont représentées, et n'hésitez pas à vous mettre en groupe et à intéragir avec le chargé de TP pour vérifier que ce que l'on vous montre est clair pour vous.

A. Variables de pile

Code: https://godbolt.org/z/a7GvhParb



Ci-dessus, on peut voir que la fonction `main` own deux variables `n` et `p`, que la fonction `power` own trois variables `n`, `p` et `r`, et que la fonction `multiply` own deux variables `a` et `b`.

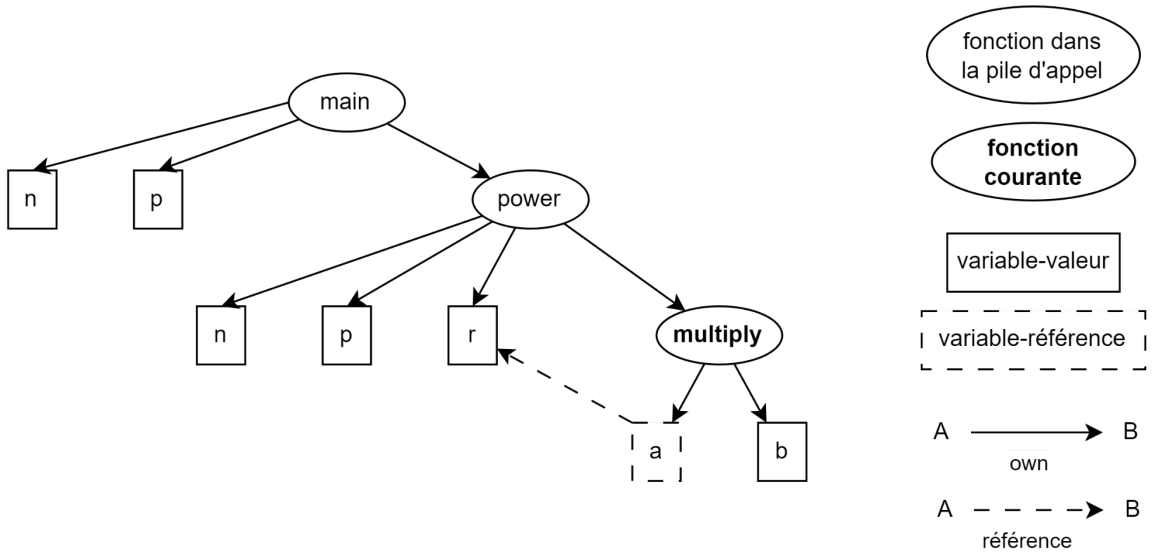
Dans le graphe d'ownership, on représente également la **transitivité de l'ownership**. `main` appelle `power`, qui elle même appelle `multiply`, donc `main` own par transitivité les instances `a` et `b` créées par `multiply`.

On peut faire deux remarques supplémentaires avec ce graphe :

- la variable `n` de `power` et la variable `n` de `main` ne correspondent pas à la même instance, et ne sont donc pas reliées entre elles. En effet, les paramètres de `power` sont passés par valeur : une copie est faite, et une toute nouvelle instance de `n` est créée à l'intérieur de `power`.
- la variable `r` de `main` n'est pas référencée dans ce graphe. Effectivement, au moment de l'appel à `power`, bien que l'espace mémoire ait été réservé pour la variable, celle-ci n'a pas encore été affectée, puisque l'appel à `power` n'est pas terminé. On considère donc que son cycle de vie n'a pas encore démarré, et on ne la représente pas dans ce graphe.

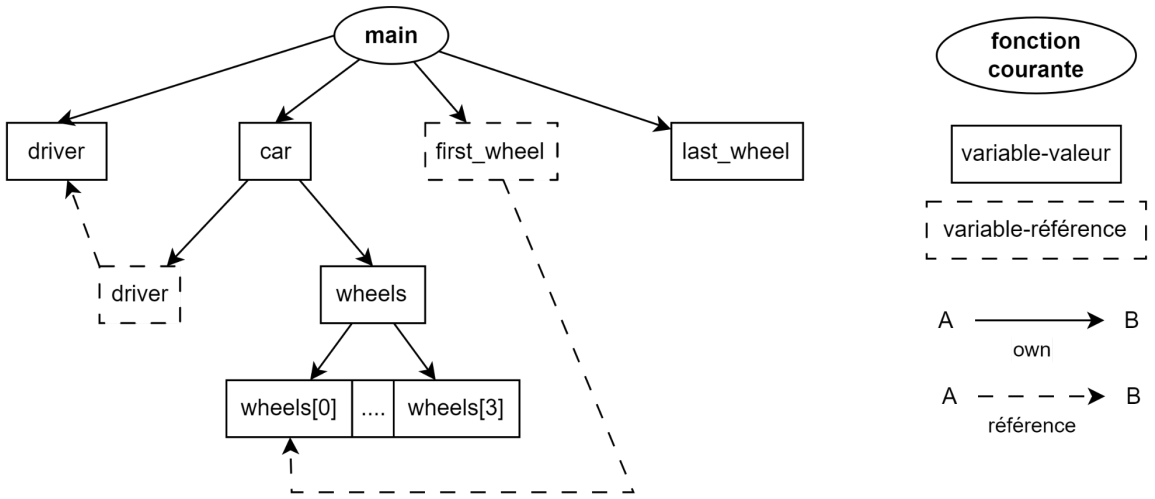
B. Références

Code: https://godbolt.org/z/ooPvEbWdK



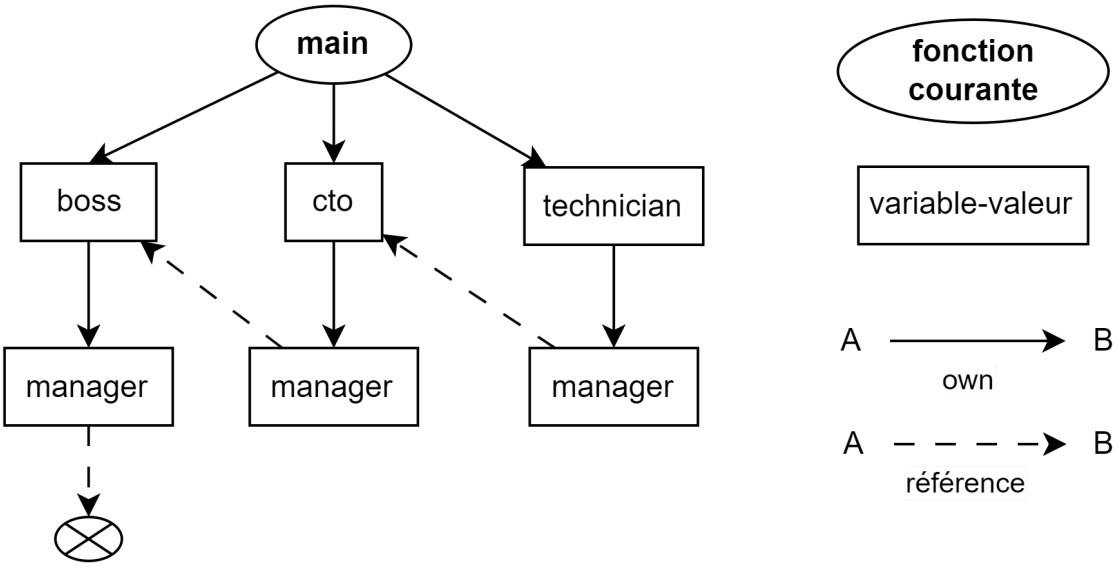
Ce coup-ci, la fonction `multiply` applique et stocke le résultat directement dans son premier paramètre. Pour que le changement soit répliqué dans la fonction appelante, la paramètre `a` est passé par référence. La variable `r` de `power` et la variable `a` de `multiply` constituent donc une seule et même entité.

Code: https://godbolt.org/z/ra51avrKf



Question : Dans le graphe ci-dessus, pourquoi n'a-t-on pas représenté de relation entre `last_wheel` et `wheels[3]` ?

Code: https://godbolt.org/z/n5eo1d8a1



Question : Dans le graphe d'ownership, comment sont représentés les pointeurs-nus ? Comment est représenté un pointeur nul ? En terme de code, quelles sont les différences principales entre un pointeur-nu et une référence ?

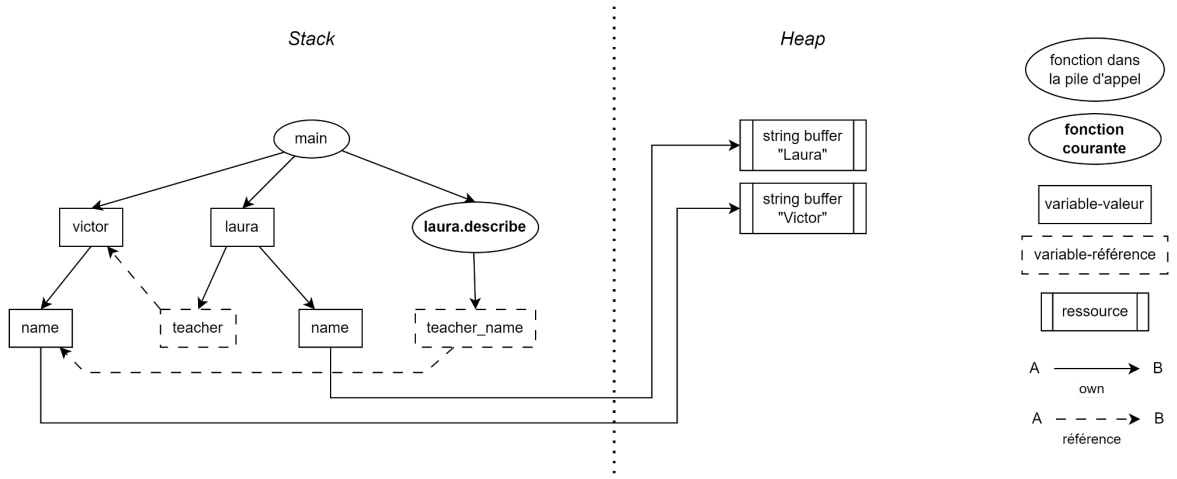
Pratique : Dessiner le graphe d'ownership correspondant au code ci-dessous. Vous pouvez omettre de représenter les attributs `size`.

Code: https://godbolt.org/z/Psaz31vYq

C. Mémoire allouée sur le tas

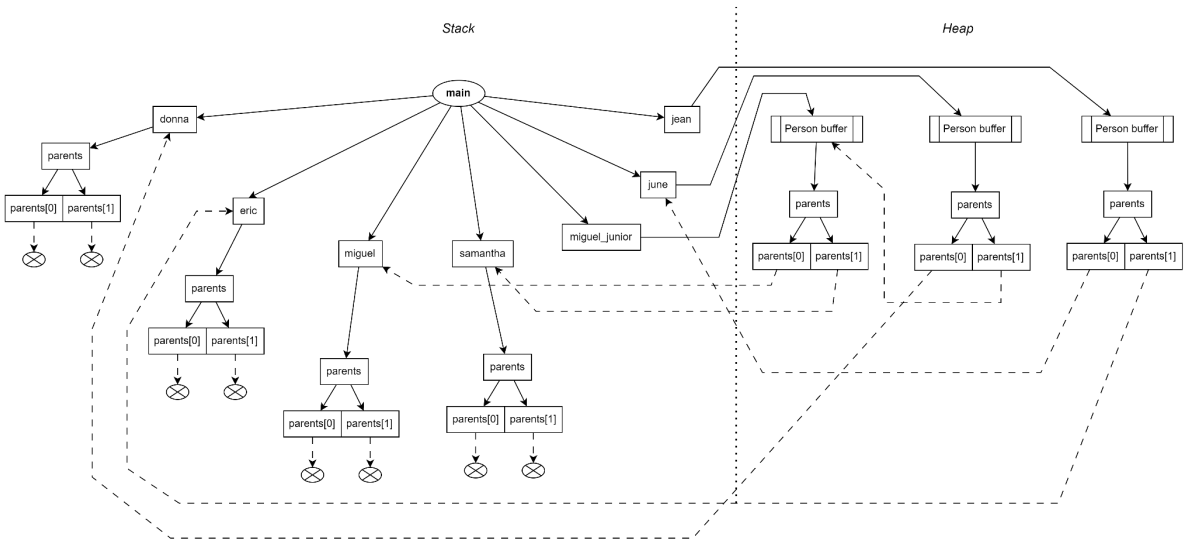
Lorsqu'un élément R est alloué sur le tas via un objet O qui gère sa durée de vie (par exemple std::string, std::vector ou std::unique_ptr), alors O est owner de R.

Code: https://godbolt.org/z/8zoeWq46P



Par exemple, dans le graphe ci-dessus, `laura.name` est une std::string dans laquelle on a réservé de l'espace pour écrire "Laura". `laura.name` own donc le buffer alloué sur le tas pour cette chaîne de caractères.

Code: https://godbolt.org/z/sK17EfEaP

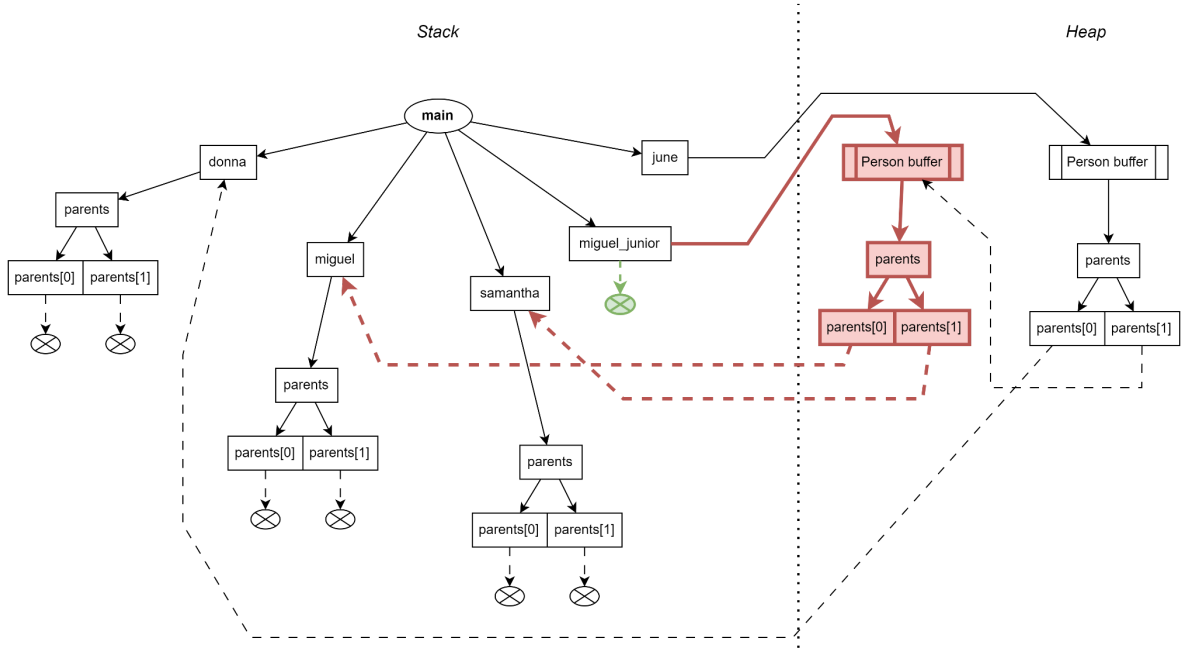


Question : Pourquoi le second parent de June ne référence pas directement la case miguel_junior ?

D. Destructions

Pour représenter des changements d'états, nous avons dans le code un indicateur de début et un indicateur de fin. Le graphe met en avant les modifications qui ont eu lieu entre ces deux indicateurs : en vert, les nouvelles entités et relations, en rouge, les entités ou relations qui n'existent plus.

Code: https://godbolt.org/z/vK3MqMvvK

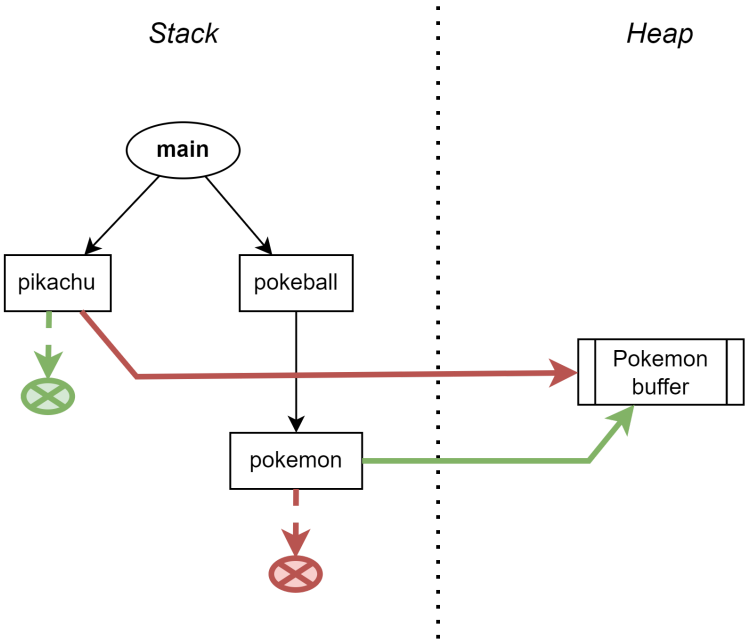


Dans le code, on réinitialise le pointeur `miguel_junior`. Cela entraîne la réassignation du pointeur à nullptr et la destruction du contenu pointé.

Question : A la suite de la destruction du contenu de `miguel_junior`, quel problème est mis en avant par le graphe d'ownership ?

E. Movements

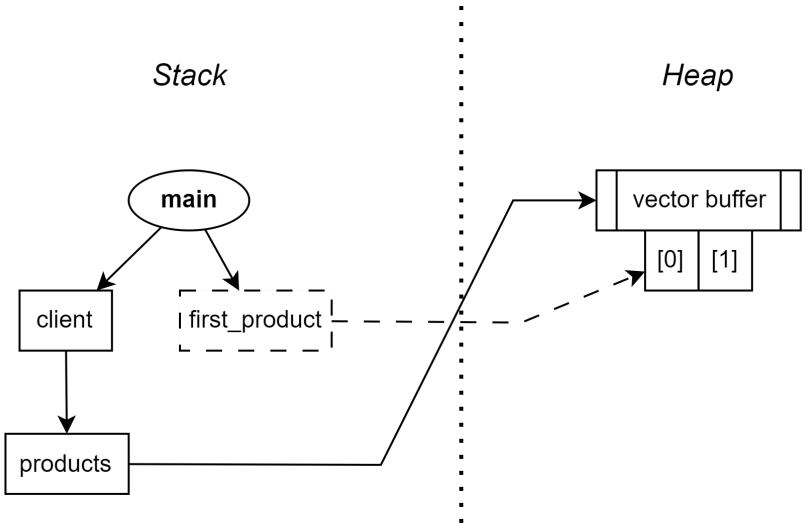
Code: https://godbolt.org/z/vzfb9cGMa



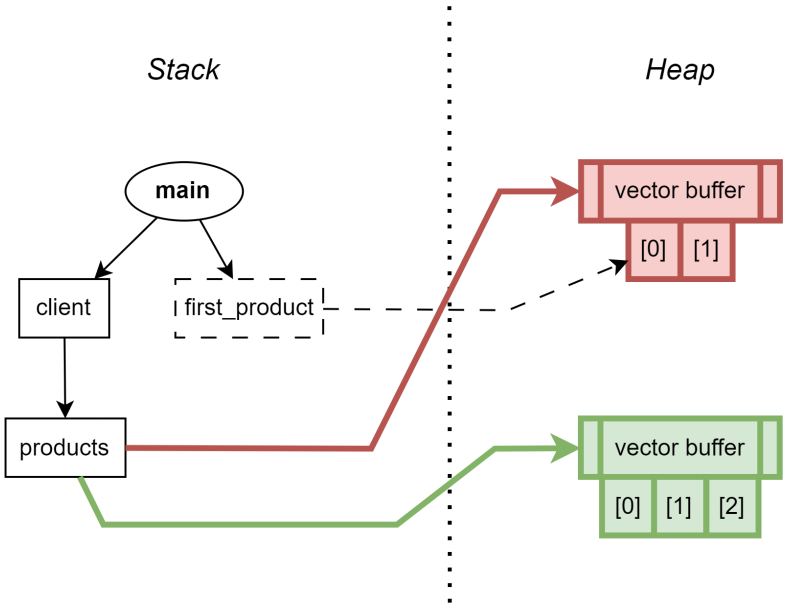
Question : Lorsqu'on écrit `pokeball.pokemon = std::move(pikachu)`, est-ce que `pikachu` est déplacé ? Que se passe-t-il réellement dans la mémoire lors d'un `move` ?

F. Insertion dans un std::vector

Situation initiale : https://godbolt.org/z/d353hnW8c



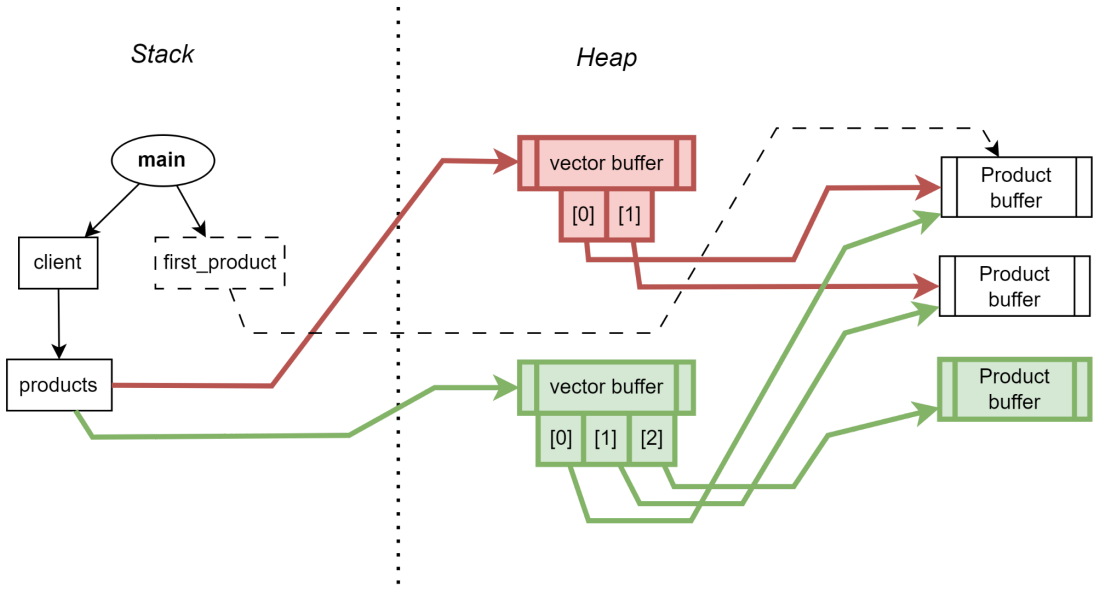
Durant l'insertion : https://godbolt.org/z/aKcooaGEc



L'insertion nécessite d'ajouter un nouvel élément dans le buffer. Or, dans notre exemple, le buffer du vector ne contient suffisamment de place que pour deux éléments. Le vector réalloue donc un nouveau buffer qui permettra de contenir suffisamment d'éléments après l'insertion.

Question : Quel problème apparaît dans le graphe ci-dessus ? Généralisez le problème et déduisez-en la série d'opérations qu'il ne faut pas faire lorsqu'on utilise un std::vector<T>. Proposez des solutions qui permettent d'éviter le problème.

Code: https://godbolt.org/z/fd7cvx5dz



Ci-dessus, on utilise des unique_ptrs pour ajouter une indirection vers les objets et éviter le problème de dangling-références causé par la réallocation des vectors.

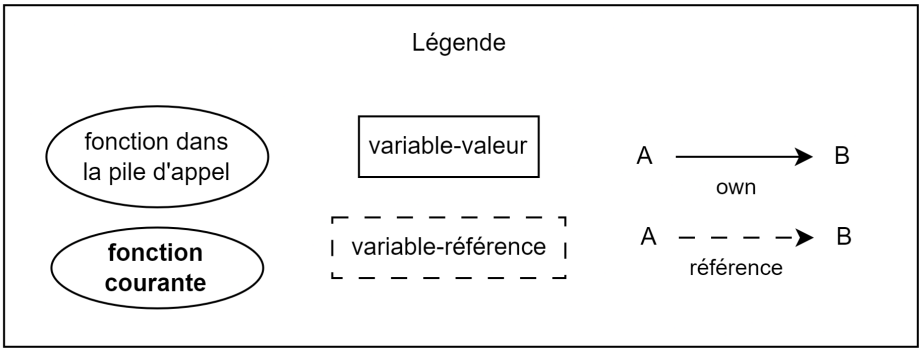
Pratique : Au lieu d'utiliser des unique_ptr, redessinez la situation en passant par des std::list plutôt que par des std::vector.

Question : Est-ce que le problème décrit ci-dessus existe toujours ? Même question si vous aviez utilisé un std::set, un std::unordered_set, un std::map et un std::unordered_map.

2- Dessiner des graphes d'ownership

Vous allez dessiner les graphes d'ownership qui correspondent aux snippets de code fournis. Vous commencerez par dessiner les fonctions et variables que l'on vous demandera de représenter dans le graphe. Vous n'êtes pas obligé de dessiner celles que l'on ne vous demande pas.
Vous devrez ensuite ajouter les ressources que ces entités ont éventuellement allouées sur le tas pour porter leurs données, ainsi que les relations *own* et *référence* qui existent entre les différents éléments.
Pour terminer, vous relèverez les éventuels problèmes mis en exergue par votre graphe.

Vous pouvez dessiner sur une feuille ou bien utiliser l'application https://app.diagrams.net/.
Pour rappel, voici la légende :



Enfin, faites bien attention à l'endroit où vous vous trouvez dans le programme (indiqué par // ←).

Code: https://godbolt.org/z/Psaz31vYq

```
main
   -  b1
   -  b2
   -  smaller_box
```

Code: https://godbolt.org/z/bMv3fr3de

```
- main
    - boxes
    - first
    - get_first_or_default
        - boxes
        - default_box
```

Code: https://godbolt.org/z/6KxT3hPYo

```
- main
    - chicken
        - egg
            - chicken
```

Question : Que va-t-il se produire à la sortie de la fonction `main`, lors de l'appel au destructeur de `chicken` ?

Code: https://github.com/Laefy/CPP_Exercises/blob/tp4-solutions/tests/Test-12.cpp (on se place ligne 60)

```
- test
    - pc
    - sacha
        - pc
        - pokeballs
        - pokedex
```

ainsi que les Pokémons capturés par Sacha.

Vous pouvez omettre les éléments redondants.

3- Dessiner un graphe d'ownership à partir d'une situation abstraite

Dans un programme, on souhaite pouvoir référencer, à partir d'une plante donnée, les instances de plantes de la même espèce qu'elle. On décrit donc la situation ci-dessous.

Dans la fonction `main`, on dispose d'une variable `species` qui contient les espèces de plantes connues. Initialement, ce conteneur dispose de 2 éléments. On définit aussi cinq instances de plantes `p1`, `p2`, …, `p5`. Les deux premières sont de la première espèce de plante, les trois dernières de la seconde espèce.

Chaque espèce de plante connaît les instances de plantes de cette espèce. Ainsi, à partir d'une instance de plante, il est possible d'accéder à son espèce, puis depuis l'espèce, de récupérer l'ensemble des plantes de la même espèce qu'elle.

Contrainte :

- À tout moment, on peut décider d'insérer de nouveaux éléments dans `species`. Cela ne doit pas introduire de problèmes.