IFT 2035 – Été 2019 David Langlois Monette (867171)

Rafael David Higuera Silva (20130914)

TP2 – Malloc et comptage de références

Expérience de développement

La difficulté la plus importante est probablement le fait que c'est difficile de debugger pour vérifier si une partie de l'algorithme fonctionne ou non. Dans le travail de ré-implémenter une librarie « malloc », c'est rarement à la compilation qu'il y a des problèmes (contrairement au devoir précédent), et surtout à l'exécution. Lorsqu'on travaille sur, par exemple, la partie de l'algorithme qui trouve un espace libre dans les blocs existants à la suite d'un appel à la fonction « mymalloc », il y a plusieurs cas possibles, et c'est difficile de concevoir tout le code d'un coup, et ensuite de vérifier ce qui ne fonctionne pas avec un gros test. Nous avons trouvé que c'était mieux de procéder par petits ajouts. Par exemple, on fait un appel à « mymalloc » et le premier bloc est plein sauf un endroit avec un espace suffisant. Faire un test pour s'assurer qu'on a le résultat escompté. Ensuite, même scénario, mais avant l'endroit avec l'espace suffisant, on a un espace qui est trop petit pour la mémoire demandée. On s'assure avec le même appel de « mymalloc » que l'algorithme fournit encore le bon résultat, et qu'il n'a pas inséré l'objet ou il n'y avait pas de place (et de ce fait, «overwrité » une partie des données).

Au début, quelques discussions ont été nécessaires afin de comprendre ce qu'on devait faire dans un premier temps, et pour ensuite se mettre d'accords sur les détails que l'algorithme que l'on allait concevoir. Comme cela faisait longtemps qu'on n'avait pas codé en C, une révision dans « tutorialpoint » a été nécessaire pour nous mettre à jour. On a alors visualisé deux routes, deux manières de faire, pour gérer la structure des blocs. Celle avec un liste de nœuds présentée en classe, mais on a conclu que cela ne serait pas agréable a gérer une liste de nœuds avec des pointeurs dans le tas, déjà que gérer des pointeurs ce n'est pas facile. On s'est dit aussi que de toute façon c'est une structure qui prend O(n) pour la plupart des actions comme «delete » ou rechercher un nœud, ce qui est beaucoup moins efficace. La solution qu'on a pris, c'est d'utiliser une liste dynamique, mais on s'est rendu compte assez vite qu'en C, la liste dynamique n'existe pas, donc on devrait initialiser nos propres listes et les gérer. Comme la pile est plus petite que le tas, on s'est dit qu'on aller utiliser le tas pour stocker nos données. On a créé des structures qui va nous permettent de gérer facilement nos données. Au cours de l'implémentation, on a eu des difficultés à suivre notre propre code, car progressivement cela devenait compliqué de voir où les données sont stockées et comment sortir les données avec les pointeurs. Évidemment, on s'est heurtés à plusieurs « segmentation faults » au cours du développement. Aussi, une difficulté important provient du fait de passer d'un langage tel que Java et avoir accès au « stack trace » qui nous dit exactement où le code a planté, à un langage tel que C qui nous donne pas

beaucoup d'informations sur les erreurs. La partie la plus difficile du devoir était vraiment de gérer les pointeurs pour notre « array » dynamique et de bien comprendre le flux d'opérations. La partie la plus facile est de comprendre la syntaxe de C comparé à Haskell. Finalement, une importante étape de « nettoyage » du code a été nécessaire, car cela devenait très difficile à se retrouver dans le code. Donc, on a fait beaucoup de fonctions séparées qui ont des noms significatifs et qui rendent bien compte du flux du code. Cette ultime étape de rendre le code lisible nous aide premièrement, puis aideront certainement aussi les personnes qui vont continuer à coder par la suite. Dans un tel travail où nous sommes plusieurs à travailler le même code, c'est même nécessaire de faire cette tâche au fur et à mesure qu'on avance pour la lisibilité entre coéquipiers.

Documentation

Structure générale

Notre structure générale consiste de blocs et d'objets. Les blocs sont les morceaux de mémoire contigüe demandés à « malloc », et les objets représentent les différentes données qui sont stockées dans ces blocs. Notre librairie « mymalloc » gère ainsi les objets contenus dans les blocs, et ne fait des appels à « malloc » que lorsqu'il n'y a plus d'espace dans les blocs du «playground » pour générer un nouveau bloc de mémoire contigüe. Nous avons nommé «playground » la structure de données qui gère toute la libraire « mymalloc » et qui est contenue dans la pile par opposition aux blocs/objets qui sont tous des données contenues dans le tas. Bref, le «playground » contient les adresses de tous les blocs alloués par « malloc » dans un seul tableau dynamique. L'espace de ce tableau est aussi générée par un appel à « malloc », afin que la mémoire du tableau soit contigüe. Ceci nous permet enfin de traiter le tableau de blocs tel qu'un «array », où on peut facilement passer de l'adresse d'un bloc au suivant avec l'arithmétique des pointeurs. Nous avons préféré cette méthode (toutes les adresses des blocs en un seul «array ») plutôt que d'utiliser une liste chainée parce qu'on trouvait ça plus intuitif à utiliser, et on trouvait dans ce cas précis que les avantages de l'«array » étaient supérieurs à celui de la liste chainée. Plus en sera dit à ce sujet dans la prochaine section. À l'origine, le tableau peut contenir 10 adresses de blocs, et lorsqu'un bloc supplémentaire est nécessaire, l'algorithme fait un appel à « realloc » pour trouver un espace contigüe pour le double, c'est-à-dire une vingtaine de blocs. Si le tableau est plein à 20 adresses, il alloue donc un espace pour 40 adresses, et ainsi de suite. Notre algorithme gère aussi si le programmeur redonne des ressources et que le tableau de blocs commence à être vide. Dans un cas ou on a moins du quart des adresses utilisées dans le tableau, on trouve un espace avec 2 fois moins de mémoire et on copie le contenu du tableau (dans ce cas-ci, les 24 adresses de blocs restantes).

Détails sur les trois structures de données

Voici la description en détails des structures de données qui sont à la base de notre algorithme :

1) « Struct objet » :

Le but de cette structure est de nous permettre de garder un inventaire de toutes les espaces de données demandées par l'utilisateur avec la fonction « mymalloc ». Donc, il y aura dans la « struct objet» l'adresse de la mémoire dans le tas où on va allouer l'espace mémoire que l'utilisateur veut. On doit aussi sauvegarder les références que l'utilisateur veut (identifications) pour cet espace mémoire. On a aussi besoin de savoir la taille de l'espace. La structure est donc :

« struct objet {

unsigned int size;

unsigned int numberOfReferences;

void* startPointer;

unsigned int blockId;

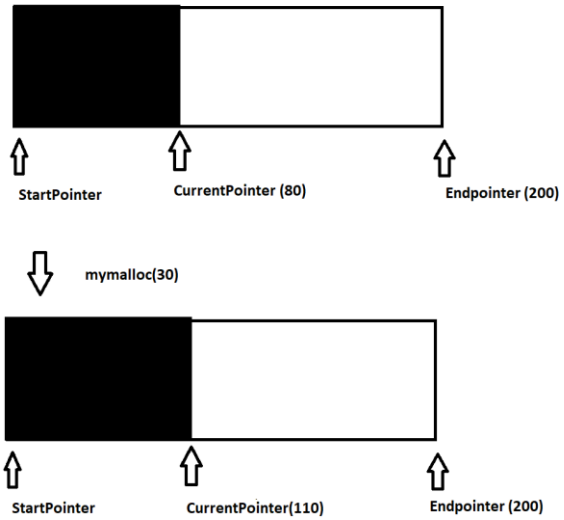
}; »

Par la suite, pour limiter nos appels à « malloc » et pouvoir simuler un réel travail de gestion de mémoire, il nous faut une structure « englobante » que nous avons nommé simplement bloc. Les blocs sont des espaces de mémoire contigue qui sont le résultat d'appels à « malloc ».

2) « Struct bloc » :

Le but de cette structure consiste à gérer l'allocation des espaces dans la mémoire qui sont de taille standard. Cela va nous permettre de simuler une espace mémoire gérée manuellement.

On a besoin de savoir où est l'espace allouée dans le tas. On a aussi un autre pointeur qui nous dit où se termine l'espace de mémoire dans le tas. Puis, on a un pointeur qui nous indique où on est dans l'espace mémoire. Donc, a chaque fois qu'on a un nouvel objet, le pointeur va avancer, puis si on est pas arrivés à la fin, cela veut dire qu'il a encore de l'espace dans ce bloc. Cette image montre visuellement le bloc et ses pointeurs.



Puis, la structure bloc possède des informations additionnelles pour gérer les structures de type objet qui vont enregistrer l'emplacement exact et la taille de l'objet. Pour l'ajout, la structure objet ne fait pas beaucoup de sens, mais elle s'avère essentielle durant l'opération de retrait. Puisqu'une partie de l'espace dans le bloc n'est pas nécessairement à la fin, on a besoin de connaître la location de cet espace. Bref, on a besoin d'avoir des informations dans la « struct object » qui vont nous permettre de connaitre exactement leur emplacement mémoire et on pourra par la suite utiliser ces espaces « effacés » pour l'ajout d'autres objets de taille égale ou moindre. Pour cela, chaque structure de bloc garde une liste d'objets, et le nombre d'objets dans le bloc consiste donc en la taille de l' « array ». Voici ce à quoi ressemble la structure d'objet :

« struct bloc {

void* startPointer;

void* endPointer;

void* currentPointer;

object* objects;

unsigned int size;

unsigned int numberOfObjects;

}; »

Finalement, on a besoin d'une structure « englobante » pour les blocs, qui nous permettra de gérer le tout. C'est ce que fait la stucture « playground » dans notre algorithme.

3) « Struct playground »

Voici ce à quoi ressemble cette structure :

« struct playgroundOfBlocks
{
  unsigned int size;
  unsigned int numberOfBlocks;
  block* blocks;
}; »

Trouver un espace mémoire pour un objet

Lorsque la fonction « mymalloc » est appelée, l'algorithme fait le tour des blocs (dans l'ordre) et regarde s'il y a des objets qui ont les references a 0. Si c'est le cas, c'est donc qu'on peut écrire par-dessus cet objet; l'algorithme regarde alors l'espace pris par cet objet. Si l'espace de cet objet est égal à l'espace mémoire recherché, on remplace alors l'ancien objet par le nouvel, en prenant soin d'indiquer que cet objet est maintenant utilisé. Si l'espace de l'ancien objet (« nombre des references = 0»), et qu'on cherche une plage de mémoire inférieure, alors on créer un nouvel objet au début de l'adresse de l'ancien objet, puis on fixe le début de l'ancien objet à l'adresse de fin du nouvel objet. Si évidemment l'espace du vieux objet n'est pas suffisant pour l'appel à « mymalloc », on continue à faire le tour des blocs en cherchant des objets dont la propriété « numberOfReferences» est fixée à « 0 », et dont l'espace utilisé est suffisant ! Par la suite, on cherche s'il a de la place entre la fin du dernier objet et la fin du bloc. Ce n'est que lorsqu'on arrive à la fin du dernier bloc sans succès qu'on créer un nouveau bloc en faisant un appel à « malloc ». Nous verrons comment cet appel est fait dans la prochaine section.

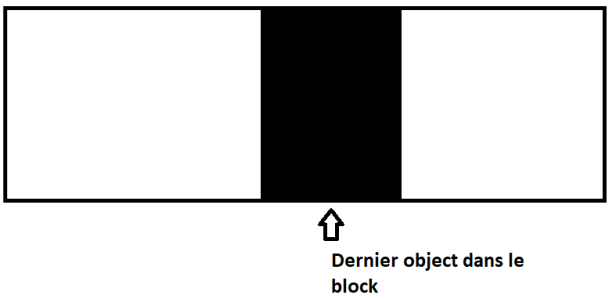
Grosseur des blocs alloués

Lorsqu'un appel est fait à « mymalloc » d'une grandeur de disons « 60k » (« 60 * 1024 octets »), et que cet espace n'est pas disponible de façon contigüe dans les blocs déjà contenus dans le «playground », notre librairie commence par vérifier si le nombre demandé est supérieur à « 4k »(« 4096 octets »). Si le nombre demandé est inférieur, l'algorithme fait un appel à « malloc » pour un espace de « 4096 octets ». Cette taille est choisie de façon arbitraire parce que c'est une valeur relativement petite (un processus de « 4k » est négligeable sur la mémoire d'un ordinateur contemporain) sans être trop volumineuse non plus. Avec « 4k », on peut stocker pas mal de données primitives (environ 1000 ou 2000 données de type « int » selon le processeur), mais ne sera évidemment pas suffisante pour des structures de données plus étendues. Bref, après le test initial qui regarde si la valeur demandée à « mymalloc » est supérieure à « 4ko » (ici « 60k > 4ko », donc « true »), alors il répète le test mais avec une valeur doublée de « 8k ». Dans notre exemple, la valeur retournée de ce test sera encore « true » puisque « 60k>8k », et donc la boucle se poursuivra avec une valeur incrémentée de « 16k » (« 8k *2 »). Finalement, lorsqu'on arrivera à « 60k > 64k », le test retourne finalement « false », et un appel est fait à « malloc » pour obtenir un bloc de « 64k », qui deviendra un nouveau bloc de mémoire dans l'espace du «playground ».

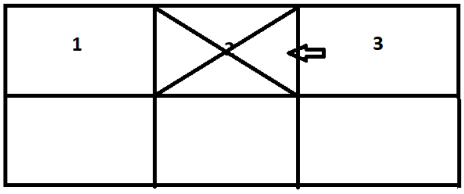
Visualisation des fonctions avec des effets spéciaux :

-Cas spéciaux lors du l'appelle à myfree:

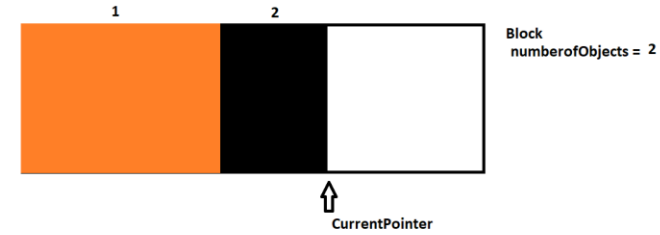
On efface le dernier objet dans un block :



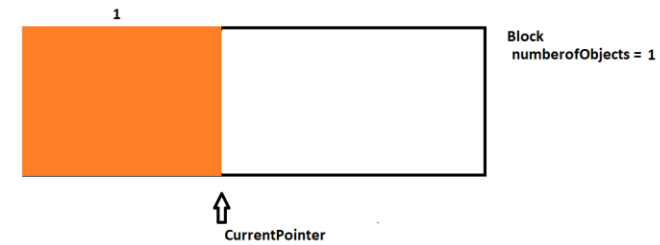
Effacement du block 2 ,car tout le block devient libre après l'effacement du dernier object



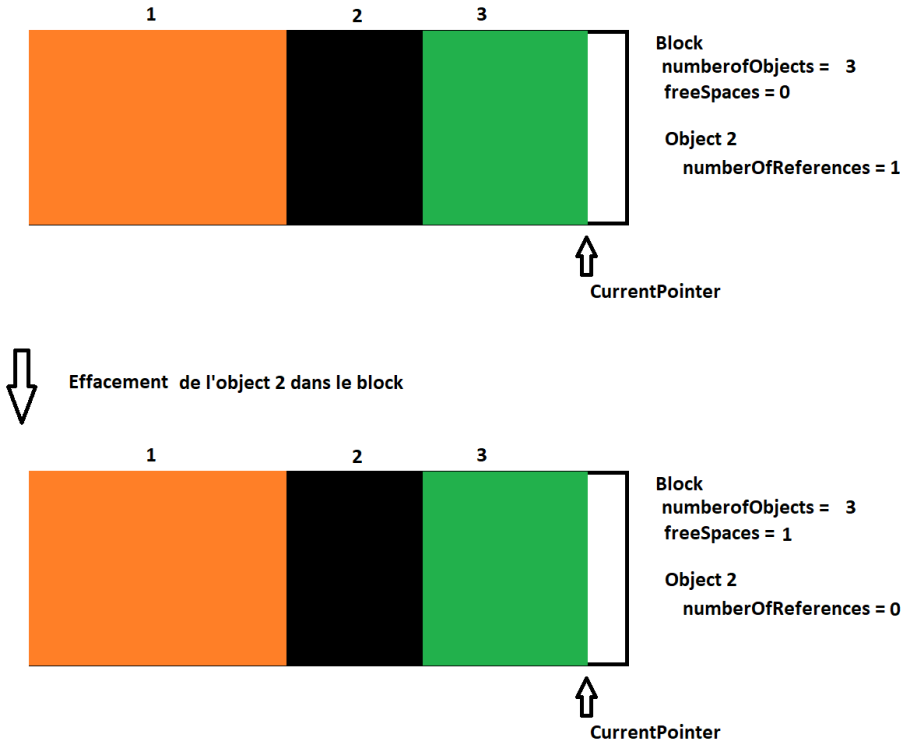
On efface un objet qui se trouve en dernier dans un block :



Effacement de l'object 2 dans le block

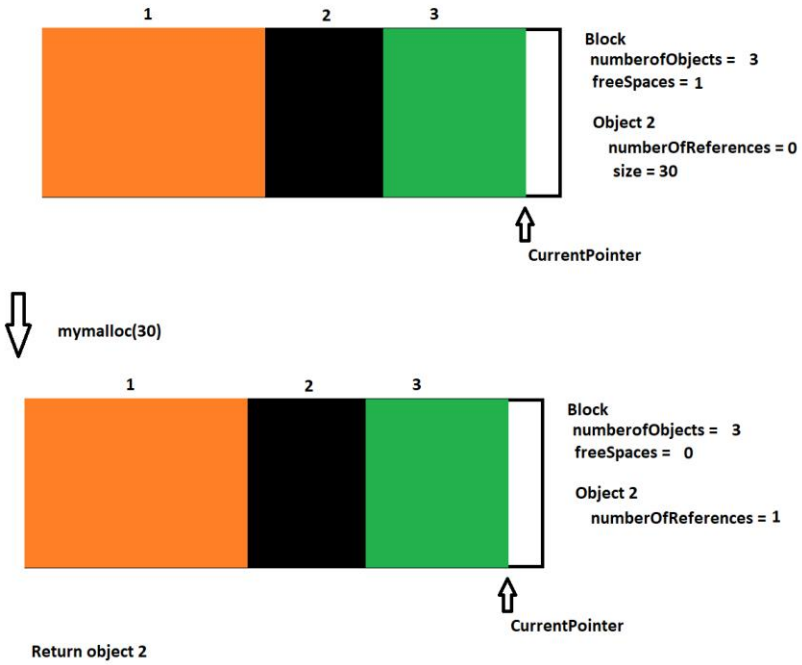


On efface un objet qui se trouve au milieu du block :

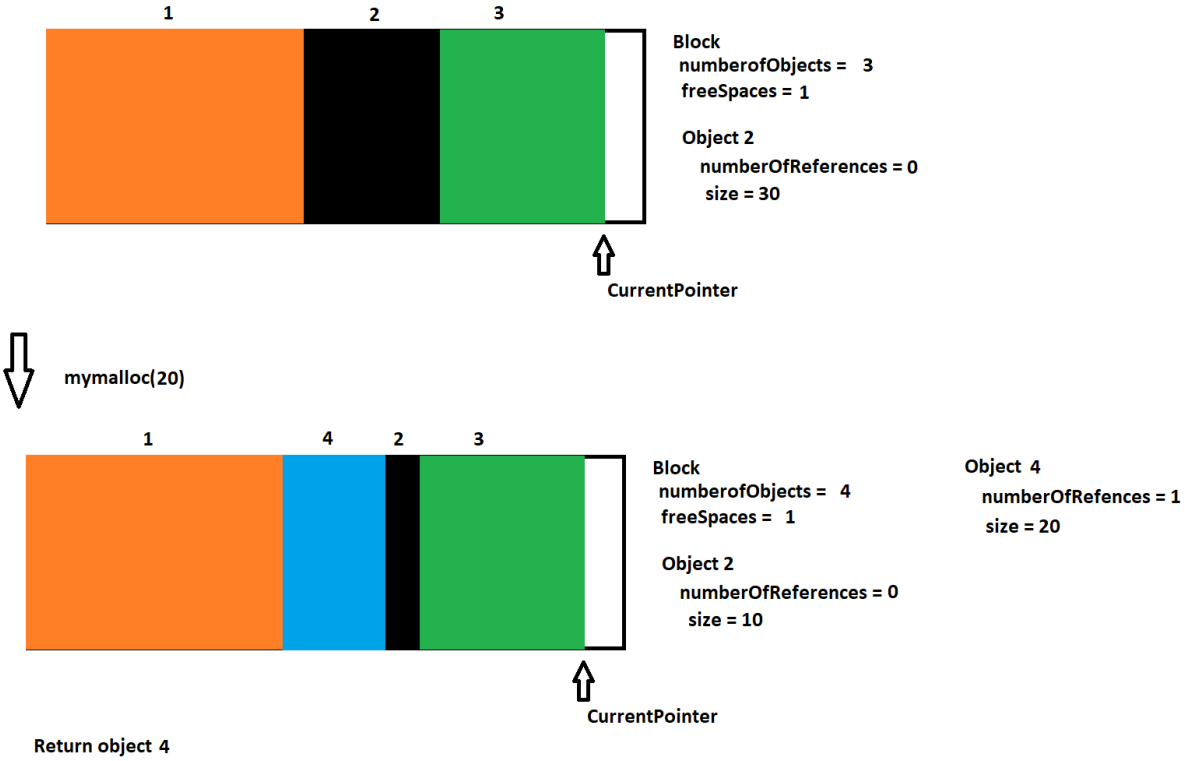


-Cas spéciaux lors du l'appelle à mymalloc :

Ajout dans un espace exactement de même ne taille que l'objet qui est libre :



Ajout dans un espace de plus petite taille que l'objet qui est libre :



Description de la fonction <<refinc>>

La méthode pour ajouter 1 à une référence d'un objet est simple, car après avoir mis à jour la valeur de la référence, il n'y a plus d'effet. Donc, on cherche l'objet avec le même pointeur puis on ajoute un 1 a le nombre de références.

Remarques générales

Notre algorithme réserve la taille demandée avec la fonction « mymalloc » exactement sans arrondir. C'est ce qui nous semblait le plus naturel. Nous n'avons pas aligné les adresses à l'intérieur d'un bloc sur un multiple d'octets. Cela serait un clairement une des premières améliorations importantes à faire au point de vue de l'optimisation si on continuait à travailler sur notre librairie. Si les adresses étaient alignées sur un multiple d'octets, cela permettrait de réduire les opérations de lecture afin de trouver la bonne adresse, et par exemple, l'opération d'ajout d'un objet serait beaucoup plus rapide.

Finalement, on n'a pas fait une liste de tests exhaustifs pour voir si tout fonctionne correctement, mais on a quand même testé en profondeur au fur et à mesure qu'on implantait des nouvelles fonctions. À chaque étape, on pensait aux différents cas possibles, et on trouvait des petits tests qui nous assuraient que l'algorithme fonctionnait tel que prévu dans tous les cas différents.

Dans le cas où « myfree » reçoit une adresse jamais retournée par « mymalloc », notre fonctionne vérifie d'abord dans tous les blocs afin de voir si l'adresse fait partie d'un bloc. Si l'adresse n'est pas dans le bloc, la fonction sort sans faire appel la fonction <<updateNumberOfReferences>>, donc il n'aura pas d'effet de bord. Si par chance, l'utilisateur trouve un pointeur qui est dans le bloc, on s'assure par la suite que l'objet est égale au pointeur l'utilisateur a passé comme paramètre, donc on s'assure que on applique les modifications a un objet que l'utilisateur a demandé avec la fonction <<mymalloc>>.

Si « refinc » reçoit un pointeur qui a été retourné par « mymalloc » mais qui a déjà été désalloué (car le nombre de références est à 0), on entrera dans la fonction «updateNumberOfReferences», et une condition surviendra telle que si le nombre de références est égal à 0, il faut retourner 0. Puisque l'objet n'a plus de références, c'est un objet qui a été effacé, et donc il n'est plus nécessaire de le gérer.

Effets non désirés dans notre algorithme

Si on faisait un test où on demandait en boucle de l'espace pour un objet de 4095 octets, puis pour un objet d'un seul octet. Si on répétait cela disons 1000 fois, et qu'après on utilisait « myfree » pour tous les objets de 4095 octets, on aurait encore besoin de tous les blocs créés (1000 blocs de 4096 octets) pour stocker les objets d'un octet, puisque notre algorithme ne comporte pas de procédé pour compacter la mémoire. Bref, cela voudrait dire que malgré que seulement 1/4096 de la mémoire demandée à « malloc » est véritablement utilisée, on ne peut redonner la mémoire, et il y aurait éventuellement un manque de mémoire ! Le problème est que même si on voudrait essayer d'arranger il y a une impossibilité à cause du système de pointeurs. Explication* : on ne peut pas déplacer les pointeurs d'objets qui sont de taille 1 vers un seul bloc, car on ne pourra modifier le pointeur que l'utilisateur a pour les objets de taille 1. Dans d'autres langages comme java, on n'a pas accès à la mémoire, donc disons le <<garbage collector>> pourrait déplacer la variable vers un autre espace mémoire sans que nous on se rend compte, car on ne fait qu'utiliser la variable et non pas au pointeur. Par contre, lors d'appels de <<mymalloc>>, ces espaces vont être remplis.

Un autre cas qui est loin d'être optimal dans notre algorithme survient lorsqu'on fait des petites demandes de mémoire avec « mymalloc ».Par exemple, disons que demande 100*1 byte, il faudrait faire 100 structures d'objets où chaque objet prend 16 octets de mémoire avec une structure de block qui prend 28 octets. Bref, on a au final 16 octets utilisés pour sauvegarder un octet. Ceci est clairement inefficace, mais ce sont les informations nécessaires pour que notre algorithme fonctionne. Il y a probablement des moyens plus efficaces pour prendre moins d'espace. Nous pensons que peut-être il est possible d'enlever le « padding » automatique que le système fait quand la sauvegarde des structures est effectuée. Ou, peut-être est-il possible d'avoir des blocs destinés à entreposer des structures primitives (qui possèdent chacune un code binaire), ce qui permet à chaque fois à l'algorithme de connaitre l'adresse de la donnée suivante, nous permettant de pouvoir négliger de préciser les informations de pointeurs, et sauver ainsi de nombreux précieux octets à chaque fois. Ceci

toutefois semble assez complexe à implémenter et par contrainte de temps et de connaissance on a décidé de laisser ces cas de côté et de les mentionner dans ce rapport.

Améliorations

L'amélioration qui fera que le code soit plus vite, même amène un temps constant d'accès serait un dictionnaire qui prendrait comme clé le pointeur de l'objet et comme valeur l'objet qui serait stocker dans chaque bloc. Cette amélioration fera que notre code aille un accès constant à chaque fois qu'on veut faire des modifications a cout de sauvegarder un pointeur de plus pour les clès.

*Explique dans la classe par le professeur Vincent Archambault-B