420KH2 – TP01

Vous avez pour tâche de gérer la mémoire de manière efficace dans un programme qui a les caractéristiques suivantes à l'exécution :

- les objets ne sont pas tous de la même taille;
- il n'y a pas de plafond fixe *a priori* sur le nombre d'objets qui seront créés;
- la plupart des objets occupent une taille inférieure à 128 *bytes*;
- plus concrètement, vous avez une majorité de très petits objets (32 *bytes* et moins), beaucoup de petits objets (plus gros que les très petits, mais 64 *bytes* et moins chacun), et un bon nombre d'objets de taille restreinte (dont la taille se situe dans l'intervalle (64,128] *bytes*). Vous trouverez des exemples de chaque catégorie à droite;
- il y a aussi des objets plus massifs, mais ceux-ci sont suffisamment peu nombreux pour que déléguer le travail aux versions standards de `new` et de `delete` ne soit pas un problème;
- vous avez beaucoup de mémoire vive à votre disposition. Pas une infinité, bien sûr, donc il faut que la mémoire libérée par le programme soit réutilisée par la suite, mais assez pour vous permettre de préférer économiser du temps plutôt que de l'espace.

```
// exemple de très petit objet
struct Point {
   int x = 0, y = 0;
   Point() = default;
   Point(int x, int y) noexcept
      : x{ x }, y{ y }
   {
   }
   // ...
};
class vide {};
// exemple de petit objet
class Orque {
   string nom;
   double puanteur;
   int force;
   char intelligence;
   // ...
};
// exemple d'objet de taille restreinte
class Meute {
   enum { TAILLE = 3 };
   string nom[TAILLE];
   Orque *membres[TAILLE];
   // ...
};
```

Votre mandat est de concevoir un mécanisme d'allocation dynamique de mémoire par blocs. Les principes qui sous-tendent votre démarche sont :

- la gestion de la mémoire passera par un `GestionnaireBlocs`. Les opérateurs `new` et `delete` que vous devrez implémenter prendront donc un `GestionnaireBlocs&` en tant que deuxième paramètre;
- vous n'aurez pas à implémenter `new[]` ou `delete[]` pour ce travail;
- lors d'une demande d'allocation de mémoire, un `GestionnaireBlocs` aura pour rôle d'examiner la quantité de mémoire demandée, puis de retourner un pointeur sur un bloc suffisamment grand pour combler les besoins du demandeur;
- un `GestionnaireBlocs` donnera accès (à l'insu du code client) à des blocs de taille fixe, soit des blocs de 32, 64 ou 128 *bytes*. Évidemment, le bloc de la plus petite taille possible tout en couvrant les besoins du demandeur devra être retourné;
- lors d'une demande d'un bloc de plus grande taille, le `GestionnaireBlocs` déléguera son travail à la version standard de `new`;
- lors d'une libération de mémoire, le `GestionnaireBlocs` sera responsable de rendre la mémoire libérée disponible pour répondre à des demandes d'allocation ultérieures.

Vous devrez écrire `GestionnaireBlocs` et le tester rigoureusement. Vos tests devront permettre de comparer les opérateurs `new` et `delete` standards de même que vous versions passant par un `GestionnaireBlocs`, pour montrer quantitativement s'il y a un avantage ou non à réaliser cette implémentation.

Je n'exige pas de vous que l'allocation et la libération de mémoire soient *Thread-Safe* pour ce travail pratique.

Si vous n'êtes pas en mesure d'obtenir la mémoire demandée par le code client, alors en conformité avec le standard, levez `bad_alloc`.

Contraintes humaines et échéances

Organisation humaine	Équipes de deux personnes.
Format de la remise	Imprimé. Prenez soin d'inclure votre code de test.
Date de remise	Au plus tard à la fin du cours du mardi `10` avril `2018`.

Amusez-vous bien!