DAEMONLAB

STRUCTURE

- DaemonLab -

Bienvenue.

Vous allez apprendre ou réviser aujourd'hui à employer plusieurs structures de données construites.

Les sujets du jours sont :
Les tables de hash, les bases de données hiérarchique, les graphes et les réserves.

01 – Détails administratifs

Certains exercices vous demandent d'utiliser le C **et** le C++. Dans ces cas là, les règles sont les suivantes :

Chaque fonction est demandée en double. Vos interfaces C doivent également être utilisable en C++. Les versions C++ sont donc présente pour une question de confort. Usez de la constante de précompilation **__cplusplus** et de la directive **extern "C".** *Votre fichier en-tête doit être compatible avec les deux langages.* Bien entendu, votre implémentation dans un langage **doit** appeler votre implémentation dans l'autre langage, **il ne doit pas y avoir de doublon de code !** Idéalement, implémentez les deux fonctions dans le même fichier.

Vous compilerez avec g++ ou clang++.

Votre rendu doit respecter **strictement** l'ensemble des règles suivantes :

- Il ne doit contenir **aucun** fichier objet. (*.o)
- Il ne doit contenir **aucun** ichier tampon. (*~, #*#)
- Il ne doit pas contenir votre production finale (programme ou bibliothèque)

Le respect de la table des normes est obligatoire.

Vous devez écrire des tests unitaires et exploiter lcov/gcov pour disposer à chaque étape d'une couverture de code avoisinant les 100 %. Un patron de projet avec couverture de code et programmes de tests basés sur assert vous est fourni en annexe.

02 – Tables de hash

Fichier à rendre : dhash.hpp

Cet exercice est à réaliser exclusivement en C++. Vous êtes libre de réaliser un bind C mais il n'est pas exigé.

Une table de hash est un conteneur associant une clef à une valeur qui peut-être implémenté de nombreuses façons différentes. Dans notre cas, il s'agit d'un **tableau statique** de **liste**.

L'idée générale est de hasher la clef afin d'en tirer un entier. L'opérateur modulo sera appliqué à cet entier avec comme diviseur la taille du tableau : une case sera déduite de cette opération, la donnée sera alors stockée avec sa clef dans la liste chaînée située à cette case.

```
template<
      typename K,
      typename V,
      size_t hash(const K&),
      size_t array_size = 1024
> class HashMap
{
  public:
      V &set(const K &k, const V &k);
      V &get(const K &k);

      ...
};
```

Vous n'avez le droit qu'a std::array et std::list.

03 – Bases de données hiérarchique

Fichier à rendre : ddatabase.hpp

Cet exercice est à réaliser exclusivement en C++. Vous êtes libre de réaliser un bind C mais il n'est pas exigé.

Une base de données hiérarchique est une table de hash dont la clef est une chaîne de caractère et dont la valeur est :

- Soit une valeur. Pour simplifier, cette valeur sera une std::string.
- Soit, récursivement, une table de hash du même type qu'au niveau au dessus.

```
class Database
{
      ...
  public:
      // Alloue l'élément si nécessaire
      Database &operator[](const std::string &key);
      Database &operator=(const std::string &val);
      Database &operator=(int i);

      operator int (void) const;
      operator std::string (void) const;

      ...
};

std::ostream &operator<<(std:: ostream &os, const Database &d);
```

Vous êtes libre d'utiliser n'importe quel conteneur de la STL.

Ci-dessous, un exemple d'utilisation de l'objet Database.

```
int main(void)
{
      Database db;
      int i;

      db["a"]["b"] = "45";
      i = db["a"]["b"];
      assert(i == 45);
      return (0);
}
```

04 – Graphes

Fichier à rendre : dgraph.hpp

Cet exercice est à réaliser en C++. Vous êtes libre de réaliser un bind C mais il n'est pas exigé.

Les graphes sont un type de conteneur permettant d'associer un ou plusieurs nœuds entre eux à l'aide de connecteurs. Lorsque l'on instancie un graphe, on y ajoute des nœuds et on décrit les relations ensuite entre ces derniers. Les graphes sont utilisés pour représenter toutes sortes de choses, comme des réseaux de neurones ou des circuits électroniques.

Et l'encapsulation en C++ :

```
template <
      typename Value,
      typename Key = std::string,
      bool Oriented = false>
class Graph
{
      ...
      class Node
      {
            const Key name;
            Value value;

            Node &operator=(const Value &val);
            ...
      };
      ...

  public:
      std::vector<Node*> path(const Key &a, const Key &b);

      // Alloue l'élément si nécessaire
      Node &operator[](const Key &a);

      bool add(const Key &a, const Value &b);
      bool remove(const Key &a);

      bool link(const Key &a, const Key &b);
      bool unlink(const Key &a, const Key &b);

      ...
};
```

Le paramètre template **Oriented** indique si lorsqu'un lien est ajouté, le lien dans l'autre sens doit être ajouté ou non.

05 – Réserve

Fichier à rendre : dpool.hpp

Cet exercice est à réaliser en C.

Une réserve est un pseudo-allocateur de mémoire, dans le sens où elles limitent le nom d'appels à **malloc** et favorisent la réutilisation d'une zone mémoire précise. Lorsqu'on crée une plage, on lui précise la nature de l'élément que l'on souhaite stocker ainsi que la quantité maximale d'éléments dont on pense avoir à disposer.

L'objectif de la réserve est de délivrer une complexité très réduite avec le moins de désavantages possibles.

En l'occurrence le design en question délivre :

- Une complexité de O(1) à l'allocation comme à la libération
- Un accès aléatoire
- Une forte localité, extrêmement proche de celle d'un tableau.
- Et c'est un effet secondaire, mais il est transmissible en réseau à peu de frais.

Ses défauts sont :

- L'ordre de parcours des éléments allouées de la réserve change en fonction des allocations et des libérations
- Par rapport au tableau, l'accès à une donnée nécessite non pas une résolution d'adresse mais deux (toujours locales, néanmoins)
- Il n'est pas possible d'agrandir la réserve après allocation sans créer une autre réserve.

Ce conteneur a été imaginé par la **LibLapin** pour résoudre le problème des ressources jetables du type des « tirs » dans les jeux vidéo.

Direction la prochaine page pour le travail à effectuer !

Pour le réaliser, vous devez allouer deux tableaux : un tableau de pointeur sur élément et un tableau d'élément. Le tableau de pointeur doit être initialisé pour pointer sur les éléments du second tableau. Un entier garde l'information du nombre d'éléments alloués, au départ, il vaut 0.

- Lorsqu'une allocation a lieu, il suffit de vérifier le débordement de l'entier et renvoyer l'élément lui correspondant avant de l'incrémenter.
- Lorsqu'un libération a lieu, il suffit d'intervertir le pointeur a libérer avec le dernier pointeur alloué et de décrémenter l'entier…
- Pour parcourir tous les éléments, il suffit de parcourir le tableau des pointeurs… et de déréférencer ceux là.

```
typedef struct      s_dlab_pool
{
        ...
}                   t_dlab_pool;

t_dlab_pool  *dlab_pool_new(size_t elem_size, size_t nbr_elem);
void         dlab_pool_delete(t_dlab_pool *pool);

// Taille d'un élément
size_t       dlab_pool_element_size(const t_dlab_pool *pool);
// Nombre d'élément dans la réserve
size_t       dlab_pool_element_quantity(const t_dlab_pool *pool);
// Taille de la structure + taille des deux tableaux
size_t       dlab_pool_capacity(const t_dlab_pool *pool);

void         *dlab_pool_alloc(t_dlab_pool *pool);
void         dlab_pool_free(t_dlab_pool *pool, void *data);

typedef void (*t_dlab_pool_callback)(void *elem, void *ctx);
void         dlab_pool_foreach(t_dlab_pool *pool,
                               t_dlab_pool_callback action,
                               void *ctx);

t_dlab_pool  *dlab_pool_pack(t_dlab_pool *pool, bool dup);
t_dlab_pool  *dlab_pool_unpack(t_dlab_pool *pool, bool dup);
```

La fonction **pack** fait passer le tableau de pointeur à un tableau d'index. Si par exemple, un pointeur contenait l'adresse de la case 4 du tableau de données, cette adresse serait remplacée par 4. La fonction **unpack** fait l'inverse. Ces deux fonctions dupliquent les données si leur paramètre **dup** est vrai. La version empaquetée de la réserve peut-être envoyée sur un réseau, *à la condition évidente que l'élément stockée ne contienne pas de pointeur !*