PARIS DESCARTES IUT

DUT Informatique - 1ère année
SDA - Structures de Données et Algorithmes

TRAVAUX PRATIQUES – Semaine n°3

Thèmes

- Allocation dynamique de mémoire
- Désallocation

Les tableaux statiques ne sont pas toujours adaptés au traitement des données. Lorsque la taille d'un tableau nécessaire au traitement des données est connue seulement au cours de l'application, on utilise l'allocation dynamique de mémoire. On s'intéressera dans ce TP à un conteneur (fondé sur un tableau d'items) dynamique. Le terme « dynamique » fait référence ici -d'une part à l'allocation dynamique de mémoire et -d'autre part à l'extension automatique de la capacité de stockage. Dans le cadre de ces *distinguos*, le but des exercices est (i) d'implémenter un tel conteneur ainsi que ses fonctions élémentaires associées et (ii) de tester ces fonctionnalités à l'utilisation de ce type de conteneur.

Exercice 1. Conteneur alloué en mémoire dynamique

Le but de cet exercice est d'utiliser les opérateurs d'allocation dynamique `new` et `delete` pour une allocation dynamique de mémoire d'un tableau d'items. La structure de données choisie est un conteneur fondé sur un tel tableau :

```
/** @brief Conteneur d'items alloué en mémoire dynamique
 *  et de capacité donnée
 */
struct ConteneurTD {
     unsigned int capacite;     // capacité du conteneur (>0)
     Item* tab;                 // tableau alloué en mémoire dynamique
};
```

Pour la gestion en mémoire dynamique et les accès aux données, la collection des fonctions retenues est la suivante : **`initialiser`**, **`detruire`**, **`lire`** et **`ecrire`**. Leur spécification vous est donnée.

1.1. Créez un nouveau projet/Solution (`sem03-tp`). Importez dans ce nouveau projet le fichier source **`ConteneurTD.cpp`** sur le disque COMMUN (répertoire `sem03/SourcesTp`).

1.2. Codez les fonctions **`initialiser`**, **`detruire`** et **`lire`** spécifiées dans le source.

1.3. Codez la fonction **`main`** qui teste un tableau de dates du nouveau type `ConteneurTD`. Codez les étapes fonctionnelles données en commentaire.

Exercice 2. Conteneur en mémoire dynamique extensible

La taille des données utilisées peut varier au cours de l'exécution d'un programme (elle s'accroît en général). Le but de cet exercice est d'étudier l'évolution de la structure de données `ConteneurTD` (décrite à l'exercice précédent) pour une extension automatique de la capacité de stockage suivant les besoins. Soit `ConteneurTDE` cette nouvelle structure de données :

```
/** @brief Conteneur d'items alloués en mémoire dynamique
 *  de capacité extensible suivant un pas d'extension donné
 */
struct ConteneurTDE {
     unsigned int capacite;     // capacité du conteneur (>0)
     unsigned int pasExtension; // pas d'extension du conteneur (>0)
     Item* tab;                 // tableau alloué en mémoire dynamique
};
```

Le principe d'extension choisi est le suivant : le besoin d'extension se manifestera lorsque la fonction `écrire` demandera l'écriture d'un nouvel item à une position *i* supérieure ou égale à la capacité courante du tableau. Dans ce cas et suivant une stratégie linéaire, la fonction `écrire` devra étendre la capacité (`capacite`) du tableau d'items à `capacite*pasExtension` (autant de fois que nécessaire). La collection des fonctions associées au conteneur et dont les spécifications vous sont données est la suivante : **`initialiser`**, **`detruire`**, **`lire`**, **`ecrire`**.

2.1. Créez un nouveau projet. Importez le fichier source **`ConteneurTDE.cpp`** sur le disque COMMUN (répertoire `sem03/SourcesTp`).

2.2. Codez les fonctions d'allocation et de désallocation **`initialiser`** et **`detruire`**.

2.3. Modifiez la fonction **`ecrire`** suivant le principe de l'extension.

2.4. Codez la fonction **`main`** qui teste un conteneur de dates du nouveau type `ConteneurTDE`. Complétez le code des étapes fonctionnelles du test données en commentaire.

Exercice 3. Des pièges et des fuites

Faites un schéma montrant l'évolution de la pile d'exécution des deux programmes suivants et répertoriez les différents problèmes qu'ils contiennent.

```
void piege(int* p) {
  std::cout << *p << std::endl;
  delete p;
}


int main() {
  int a;
  piege(&a);
  int* p = new int;
  *p = 21;
  piege(p);
  std::cout << *p << std::endl;
  return 0;
}

void egoiste() {
  int* p = new int;
  *p = 12;
}

int main() {
  for ( ; ; )
    egoiste();
  return 0;
}
```

Exercice 4. De la faiblesse des compilateurs

Les fonctions suivantes sont incohérentes. Expliquez pourquoi.

```
int* ad_locale() {
  int locale = 0;
  return &locale;
}

int& ref_locale() {
  int locale = 0;
  return locale;
}

int* ad_param(int param) {
  return &param;
}


int& ref_param(int param) {
  return param;
}
```

Pour vous en convaincre, donnez l'évolution de la pile d'exécution du programme suivant :

```
int main() {
  int* p1 = ad_locale();
  *p1 = 10;
  int& r1 = ref_locale();
  r1 = 11;
  int* p2 = ad_param(12);
  *p2 = 13;
  int& r2 = ref_param(14);
  r2 = 15;
  return 0;
}
```


Exercice 5. Fonctions retournant des pointeurs ou des références

Qu'affiche le programme suivant. Un schéma d'évolution de la pile peut vous aider à trouver la réponse à cette question.

```
int* ad_table(int* t, int size, int i) {
  if (0 > i || i >= size) {
    std::cerr << "error : bad index" << std::endl;
    exit(1);
  }
  return &t[i];
}

int& ref_table(int* t, int size, int i) {
  if (0 > i || i >= size) {
    std::cerr << "error : bad index" << std::endl;
    exit(1);
  }
  return t[i];
}

int main() {
  const int SIZE = 4;
  int tab[SIZE];
  int* tmp;
  tmp = ad_table(tab, SIZE, 0);
  *tmp = 12;
  *ad_table(tab, SIZE, 1) = 13;
  int& r = ref_table(tab, SIZE, 2);
  r = 14;
  ref_table(tab, SIZE, 3) = 15;
  for (int i = 0; i < SIZE; ++i)
    std::cout << "tab[" << i << "] = " << tab[i] << std::endl;
  return 0;
}
```

Exercice 6. Conteneur en mémoire dynamique

Reprendre l'exercice 1 (créez un nouveau projet), en réécrivant l'allocation mémoire et la désallocation avec les instructions correspondantes du langage C. Les fonctions à réécrire sont **`initialiser`** et **`detruire`**. Tester un conteneur de dates suite à cette implémentation.

Exercice 7. Conteneur en mémoire dynamique extensible

Reprendre l'exercice 2 (créez un nouveau projet) en réécrivant l'allocation mémoire et la désallocation avec les instructions correspondantes du langage C. Les fonctions à réécrire sont **`initialiser`**, **`detruire`** et **`écrire`** pour laquelle vous utiliserez la fonction **`realloc`**. Tester un conteneur de dates suite à cette implémentation.