Université de Reims Champagne-Ardenne
UFR Sciences Exactes et Naturelles
Licence Informatique
INFO0402: Programmation Orientée Objet

TD N°4

Exercice 1: Gestion des pointeurs

1. en C ou C++, rappeler comment sont gérés les allocations mémoires pour les tableaux statiques.
2. en C++, rappeler comment sont gérés les tableaux dynamiques ?
3. pourquoi le mode de gestion des tableaux dynamiques pose potentiellement des problèmes ?
4. rappeler le fonctionnement des constructeurs/destructeurs pour les tableaux.

Solution:

1. Rappel :

   (a) Les tableaux statiques associés à une variable automatique sont associé à la portée de la variable (alloués à la déclaration, libéré en fin de portée).

   (b) Les tableaux statiques dans les structures sont intégrés dans l'objet, donc associé à la portée de l'objet créé.

   voir les exemples suivants :

```
struct A { int a[10]; }
// sizeof(A) = 10*sizeof(int)
void fun() {
    A   u;    // allocation du tableau dans la structure
    int v[10];  // allocation d'un tableau de 10 éléments
    ...
} // libération auto de a et b
```

2. les tableaux dynamiques doivent être alloués explicitement et désalloués explicitement. Voir les exemples suivants :

```
struct A { int *a; }
void fun() {
    A   u;    // u.a = nullptr; // devrait être initialisé
    int *v;     // v = nullptr;   // devrait être initialisé
    // allocation explicite
    u.a = new int[10]; // devrait être dans le constructeur
    v = new int[10];
    ...
    // desallocation explicite
    delete [] u.a;     // devrait être dans le destructeur
    delete [] v;
}
```

3. Les problèmes sont les suivants :
   - si une allocation est faite au cours d'une fonction, alors tous les chemins de retour doivent libérer la mémoire avant le retour,
     problème compliqué par les exceptions qui sont des chemins exceptionnels en cas d'erreur.
   - tout objet qui alloue de la mémoire dynamique doit prévoir sa désallocation (en général dans le destructeur).

— toute allocation dynamique peut échouer, en conséquence, toute opération allouant de la mémoire peu échouer (en particulier copie et assignation).

4. même fonctionnement en statique ou en dynamique :
   — `x = new A(10)` alloue un A et lance le constructeur `A(int)`, `y = new A[10]` alloue 10 `A` et lance 10 constructeurs par défaut.
   — `delete x` lance le destructeur et désalloue, `delete [] y` lance le destructeur 10 fois et désalloue.

Exercice 2: Pile d'entier

On voudrait écrire une classe qui permette de gérer une pile d'entiers. L'une des capacités des piles est de se retrouver pleine : on considérera qu'une pile ne peut être redimensionnée.

1. classe `StaticStack` basée sur un tableau statique de taille constante :
   (a) Définir une classe `StaticStack` basée sur un tableau statique de taille constante `size`=10 (constante de classe).
   (b) Donner le `sizeof` de cette classe.
   (c) Donner le constructeur, le destructeur.
   (d) Ajouter les méthodes pour vérifier l'état de la pile (vide ou pleine), push, pop, top.
   (e) Donner le constructeur par copie.
   (f) Donner l'assignation par copie.
2. classe `DynamicStack` basée sur un tableau dynamique de taille constante : on veut modifier la classe précédente de manière à ce que la taille de la pile puisse être fixée dans le constructeur (mais non modifiable).
   (a) Définir une classe `DynamicStack` basée sur un tableau dynamique
   (b) Donner le `sizeof` de cette classe.
   (c) Quelle caractéristique possède le pointeur vers les données de la pile ?
   (d) Donner les constructeurs et le destructeur.
   (e) Donner le constructeur par copie et l'assignation par copie.
3. classe `SmartStack` basée sur un tableau dynamique de taille constante utilisant un pointeur intelligent :
   (a) Quel est l'intérêt de faire appel à un pointeur intelligent ?
   (b) Définir une classe `SmartStack` basée sur un pointeur intelligent.
   (c) Donner le constructeur, le destructeur.
   (d) Donner le constructeur et l'assignation par copie.

Solution:

1. classe `StaticStack`
   (a) définition du code :

```
class StaticStack {
private:
  static const size_t   size = 10;  // si non statique: un par classe
  int    stack[size];
  size_t  pos;
public:
    ...
};
```

   (b) `sizeof(StaticCast) = sizeof(size_t) + size*sizeof(int)`
   (c) constructeurs et destructeur :

```
StaticStack() : pos(0) { cout << "Cd" << endl;  }
~StaticStack() { cout << "D" << endl; };
```

   (d) état pile, push, pop, top

```
bool isEmpty() const { return (pos == 0); }
bool isFull()  const { return (pos == size); }
size_t Size() const { return size; }
bool Push(const int v) {
    if (isFull()) return false;
    else {  stack[pos++] = v; return true; }
}
bool Top(int &v)
    { if (isEmpty()) return false; v = stack[pos - 1]; return true; }
bool Pop(int &v)
    { if (isEmpty()) return false; v = stack[--pos]; return true; }
```

   (e) constructeur par copie

```
StaticStack(const StaticStack& orig) : pos(orig.pos)  {
  cout << "Cc" << endl;
  if (pos) memcpy(stack, orig.stack, orig.pos*sizeof(int));
}
```

   Note : le constructeur par copie par défaut fonctionnerait car elle fait aussi une copie champs à champs.
   (f) assignation par copie

```
StaticStack& operator=(const StaticStack& orig) {
  cout << "=c" << endl;
  if (this != &orig) {
    pos = orig.pos;
    if (pos) memcpy(stack, orig.stack, orig.pos*sizeof(int));
  }
  return *this;
}
```

   Note : l'assignation par copie par défaut fonctionnerait car elle fait aussi une copie champs à champs.

2. classe `DynamicStack`
   (a) définition du code :

```
class DynamicStack {
private:
    // si non statique: un par classe et ne peut pas être initialisé
  static const size_t   defsize = 4;
  const size_t  size;
  int*            stack;
  size_t          pos;
public:
    ...
}
```

   (b) `sizeof(DynamicStack) = 2*sizeof(size_t) + sizeof(int*)`
   (c) il ne dépend pas de la taille des données pointées, et la mémoire associée doit alloué/desallouée explicitement.
   (d) constructeurs et destructeur :

```
DynamicStack(size_t n) : size(n), stack(new int[n]), pos(0) {};
DynamicStack() : DynamicStack(defsize) {};
~DynamicStack() { delete [] stack; stack = nullptr; };
```

   (e) constructeur par copie et l'assignation par copie.

```
// copy constructor
DynamicStack(const DynamicStack& orig) : DynamicStack(orig.size) {
  pos = orig.pos;
  if (pos) memcpy(stack, orig.stack, orig.pos*sizeof(int));
}
// copy assignation
DynamicStack& operator=(const DynamicStack& orig) {
  assert(orig.pos <= size);
  if (this != &orig) {
    pos = orig.pos;
    if (pos) memcpy(stack, orig.stack, orig.pos*sizeof(int));
  }
  return *this;
}
```

3. classe `SmartStack`
   (a) Permet d'associer le temps de vie d'une mémoire allouée dynamiquement à la durée de vie du pointeur intelligent. Autrement dit, lorsqu'un `unique_ptr` est détruit, la mémoire sur laquelle il pointe il détruit. **Rappel :** toutes les opérations autorisées sur un pointeur sont surchargées sur un pointeur intelligent.
   (b) définition de la classe :

```
class SmartStack {
private:
  static const size_t      defsize = 10;
  unique_ptr<int[]>  stack; // #include <memory>
  size_t  size;
  size_t  pos;
public:
    ...
}
```

   (c) constructeur et destructeur : idem `DynamicStack` sauf que le destructeur est vide.
   (d) constructeur par copie et l'assignation par copie : idem `DynamicStack`. Exception : utiliser stack.get() pour récupérer le pointeur réel à passer en paramètre à memcpy.

Exercice 3: Déplacement

1. Au regard du déplacement, quel est la caractéristique d'une variable automatique ?
2. Qu'est-ce-qui peut être déplacé ?
3. Par défaut, quels est la particularité des objets qui peuvent être déplacés ?
4. Pourquoi est-il possible d'écrire des fonctions spécifiquement pour ces cas ?
5. A quoi sert `std::move(x)` ?
6. Quelle est la conséquence de l'opération précédente sur une fonction/méthode qui effectue un déplacement ?
7. Quelles sont les principales occasions d'utiliser un objet déplacé ?
8. L'écriture suivante provoque-t-elle un déplacement : `A a = A(12);`
9. L'écriture suivante provoque-t-elle un déplacement :
```
A fun() { return A(12); };
A a = fun();
```

Solution:

1. une variable automatique ne peut jamais être déplacée : son adresse mémoire ne change pas au cours de l'exécution de la fonction dans laquelle elle est définie.
2. uniquement la propriété d'un pointeur (ou d'une ressource), qui pointe sur une zone allouée dynamiquement (danger potentiel si elle est dans le stack). Acquis par copie, mais le transfert de propriété doit s'accompagner de la perte de propriété par l'objet qui le possédait.
3. les objets temporaires (ou non nommé), typiquement des rvalues.
4. En utilisant le qualificateur `&&` qui indique que l'objet est temporaire.
5. pour un objet non temporaire `x` (lvalue), marquer qu'il peut être déplacé en le transformant en rvalue (change juste son type), et donc qu'il est donc en fin de vie. Un tel objet ne doit pas être réutilisé après déplacement (modifié par le déplacement) ou alors pour le réaffecter.
6. un déplacement doit toujours laisser l'objet dans un état cohérent, ou du moins dans un état suffisamment cohérent afin de permettre au destructeur de fonctionner correctement.
7. au moment de sa création (création à partir d'un objet temporaire) ou lors d'une assignation (copie depuis un objet temporaire), à l'appel d'une fonction à laquelle on passe un objet temporaire si le paramètre est passé par référence à une rvalue.
8. non, c'est l'équivalent de l'écriture `A a(12)`, mais peut provoquer une construction par copie.
9. non, une élision de copie est effectuée (la question précédente peut également être interprétée ainsi).

Exercice 4: Pile d'objets

On considère maintenant un objet de type `A` (possiblement beaucoup plus complexe) qui peut être copié et déplacé.

1. On suppose que l'objet `A` peut être copié et déplacé.
   (a) Quels sont les opérations supplémentaires qu'autorise `A` ?
   (b) en remarquant que le reste du code ne change pas, donner pour une `StaticStack` *(i)* les constructeurs par copie et par déplacement. *(ii)* l'assignation par copie et par déplacement.

(c) Donner pour une `DynamicStack`, *(i)* les constructeurs par copie et par déplacement, *(ii)* l'assignation par copie et par déplacement.
(d) pour une `SmartStack`, quelle est la différence avec un `DynamicStack`.
(e) quel est l'inconvénient de cette approche ?

2. On utilise l'allocation en place afin d'éviter les allocations par défaut inutile.
   (a) pourquoi n'est-ce pas possible avec `StaticStack` ? Dans les questions suivantes, on ne traitera que le cas `DynamicStack`.
   (b) donner la fonction qui permet d'effectuer en `C++`une allocation dynamique d'un espace mémoire sans lancer de constructeur.
   (c) donner les constructeurs standards.
   (d) donner le constructeur par copie et déplacement.
   (e) donner l'assignation par copie et déplacement.
   (f) donner le destructeur.
   (g) donner les optimisations des méthodes `Push`, `Top` et `Pop`.

Solution:

1. Objets copiable et déplaçable
   (a) Les constructeurs et assignations suivantes :

```
A(const A& v); // le constructeur par copie
A(A&& v);      // le constructeur par déplacement
A &operator=(const A& v) // l'assignation par copie
A &operator=(A&& v)      // l'assignation par déplacement
```

   Toutes constructions ou assignations à partir d'un `A` existant doit nécessairement faire appel à ces méthodes.
   (b) pour une `StaticStack`,
      i. constructeurs par copie et par déplacement

```
StaticStack(const StaticStack& orig) : pos(orig.pos) {
  for (size_t i = 0; i < orig.pos; i++)
    stack[i] = orig.stack[i];
  // le = ci-dessus utilise l'assignation par copie de A
}
StaticStack(StaticStack&& orig) : pos(orig.pos) {
  for (size_t i = 0; i < orig.pos; i++)
    stack[i] = std::move(orig.stack[i]);
  // le = ci-dessus utilise l'assignation par déplacement de A
}
```

      ii. assignation par copie et par déplacement

```
StaticStack& operator=(const StaticStack& orig) {
  if (this != &orig) {
    pos = orig.pos;
    for (size_t i = 0; i < orig.pos; i++)
      stack[i] = orig.stack[i];
    // le = ci-dessus utilise l'assignation par copie de A
  }
  return *this;
}
StaticStack& operator=(StaticStack&& orig) {
  if (this != &orig) {
    pos = orig.pos;
    for (size_t i = 0; i < orig.pos; i++)
            stack[i] = std::move(orig.stack[i]);
    // le = ci-dessus utilise l'assignation par déplacement de A
  }
  return *this;
}
```

(c) pour une `DynamicStack`,

i. constructeurs par copie et par déplacement

```
DynamicStack(const DynamicStack& orig) :
  stack(new A[orig.size]), size(orig.size), pos(orig.pos) {
  for (size_t i = 0; i < orig.pos; i++) stack[i] = orig.stack[i];
}
// move constructor
DynamicStack(DynamicStack&& orig) :
    size(orig.size), stack(orig.stack), pos(orig.pos) {
  // laisser size à 0 est génant dans ce cas
  orig.stack = nullptr;
  const_cast<size_t&>(orig.size) = 0; // using chuck norris to modify a const
  orig.pos = 0;
}
```

ii. assignation par copie et par déplacement

```
// copy assignation
DynamicStack& operator=(const DynamicStack& orig) {
  // pile d'accueil assez grande pour stocker les éléments de la pile copiée
  assert(orig.pos <= size);
  if (this != &orig) {
    for(int i=0;i<orig.pos;++i)
      stack[i]=orig.stack[i];
    pos = orig.pos;
  }
  return *this;
}
// move assignation
DynamicStack& operator=(DynamicStack&& orig) {
  assert(orig.pos <= size);
  if (this != &orig) {
    // if size was not constant: std::swap(size, orig.size);
    std::swap(const_cast<size_t&>(size), const_cast<size_t&>(orig.size));
    std::swap(pos, orig.pos);
    std::swap(stack, orig.stack);
    // no copy
  }
  return *this;
}
```

(d) idem pour un `SmartStack` (swap aussi défini pour un `unique_ptr`).

(e) l'allocation de la liste à la construction (avec un `new` ou en statique), lance le constructeur autant de fois que d'objet dans la liste.

2. Allocation en place

(a) L'allocation s'effectue en même temps que la structure : aucun moyen d'empêcher l'appel du constructeur par défaut de `A`.

(b) Utiliser la fonction suivante :

```
A* mallocA(size_t n)
    { return  reinterpret_cast<A*>(:: operator new(n * sizeof(A))); }
```

(c) constructeurs standards :

```
DynamicStack(size_t n) : size(n), stack(mallocA(n)), pos(0) {};
DynamicStack() : DynamicStack(defsize) {};
```

(d) constructeur par copie et déplacement :

```
// copy constructor
DynamicStack(const DynamicStack& orig) :
  stack(mallocA(orig.size)), size(orig.size), pos(orig.pos) {
  // seulement les données à copier
  for (size_t i = 0; i < orig.pos; i++)
    new(&stack[i]) A(orig.stack[i]); // Cc A
}
// move constructor
DynamicStack(DynamicStack&& orig)
    : size(orig.size), stack(orig.stack), pos(orig.pos) {
  // laisser size à 0 est génant dans ce cas
  orig.stack = nullptr;
  const_cast<size_t&>(orig.size) = 0; // using chuck norris to modify a const
  orig.pos = 0;
}
```

(e) assignation par copie et déplacement :

```
// copy assignation
DynamicStack& operator=(const DynamicStack& orig) {
  assert(orig.pos <= size);
  cout << "=c" << endl;
  if (this != &orig) {
        if (pos < orig.pos) { // this plus petit
            for (size_t i = 0; i < pos; i++)
                stack[i] = orig.stack[i]; // C=A
            for (size_t i = pos; i < orig.pos; i++)
                new(&stack[i]) A(orig.stack[i]); // Cc A
        } else { // this plus grand
            for (size_t i = 0; i < orig.pos; i++)
                stack[i] = orig.stack[i]; // C=A
            for (size_t i = orig.pos; i < pos; i++)
                stack[i].~A();    // destruction
        }
    pos = orig.pos;
  }
  return *this;
}
// move assignation
DynamicStack& operator=(DynamicStack&& orig) {
  assert(orig.pos <= size);
  if (this != &orig) {
    // if size was not constant: std::swap(size, orig.size);
    std::swap(const_cast<size_t&>(size), const_cast<size_t&>(orig.size));
    std::swap(pos, orig.pos);
    std::swap(stack, orig.stack);  // no copy
  }
  return *this;
}
```

(f) destructeur :

```
~DynamicStack() {
  if (stack) {
    /// manual cleaning of existing A
    for (size_t i = 0; i < pos; i++) stack[i].~A();
    ::operator delete(stack);
        stack = nullptr;
  }
};
```

(g) `Push`, `Top` et `Pop` :

```
// en poussant une valeur ou une constante
bool Push(const A &v) {
  if (isFull()) return false;
  else {
    /* allocation en place avec constructeur par copie */
    new(&stack[pos]) A(v); /* constructeur en place */
    ++pos;
    return true;
  }
}
// en poussant par déplacement
bool Push(A &&v) {
  if (isFull()) return false;
  else {
    /* allocation en place avec constructeur par copie */
    new(&stack[pos]) A(move(v)); /* constructeur en place */
    ++pos;
    return true;
  }
}


const A& Top(void) { /* fail is stack is empty */
  if (isEmpty()) throw 0;
  return stack[pos - 1];
}

// version 1
bool Pop(void) {
  if (isEmpty()) return false;
  --pos;
  stack[pos].~A();  /* destructeur en place */
  return true;
}
// version 2
A&& Pop(void) {
  if (isEmpty()) throw 0;
  --pos;
  return std::move(stack[pos]);
}
```