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

TD N°5

Exercice 1: Liste chainée générique

On veut définir une liste chainée générique `List` dont les éléments sont triés par ordre croissant de manière à n'exposer à l'utilisateur que l'interface de manipulation.

1. Définir une structure `Cell` interne à la classe `List` permettant de représenter un élément de la liste, ainsi que les champs de `List` permettant de stocker la liste et de disposer d'un pointeur de lecture.
2. Définir deux constructeurs pour la structure `Cell`, `Cell(const T&)` et `Cell(T,Cell*)`.
3. Définir deux méthodes privées de `List` permettant d'insérer et effacer une `Cell` dans une liste chainée.
   (a) `Cell *CellInsert(Cell *prv, Cell *nxt, const T& v)` qui insére une nouvelle cellule contenant la valeur `v` entre les Cells `prv` et `nxt`, et retourne un pointeur vers la nouvelle cellule.
   (b) Cell *CellDelete(Cell *prv, Cell *cur, Cell *nxt) qui supprime la cellule `cur` placée entre `prv` et `nxt`, et retourne la première cellule non null dans la chaine modifiée.
4. Ajouter le constructeur par défaut.
5. Ajouter le destructeur.
6. Ajouter le constructeur par copie.
7. Ajouter la méthode permettant d'ajouter un élément à liste.
8. Ajouter la méthode permettant de retirer un élément à liste.
9. Ajouter les accesseurs permettant d'obtenir la valeur de l'éléments courant (= celle pointée par le pointeur de lecture) et la longueur de la liste.
10. Ajouter les méthodes permettant de naviguer le pointeur de lecture dans la liste (`Rewind`, `Next`, `Advance`, `isEOL`).
11. Ajouter la méthode qui permet de vérifier si un élément est dans la liste.
12. Surcharger l'opérateur `<<` pour la liste.

Solution:

1. Voir le code question suivante.
2. Voir le code :

```
template <class T> class List {
private:
  struct Cell {
    T     val;
    Cell  *nxt;
    explicit Cell(const T& v) : val(v), nxt(nullptr) {};
    Cell(const T& v, Cell *t) : val(v), nxt(t) {};
  };
  Cell  *first, *read;
public:
  void Add(const T& v);
  void Del(const T& v);
    ...
};
```

3. Voir le code (à ajouter dans la section privée) :
   (a) Voir le code (à ajouter dans la section privée) :

```
Cell *CellInsert(Cell *prv, Cell *nxt, const T& v) {
  Cell  *c = new Cell(v, nxt);
  assert(c != nullptr);
  if (prv != nullptr) prv->nxt = c;
  return c;
};
```

(b) Voir le code (à ajouter dans la section privée) :

```
Cell *CellDelete(Cell *prv, Cell *cur, Cell *nxt) {
  if (prv != nullptr) prv->nxt = nxt;
  delete cur;
  if (prv != nullptr) return prv;
  else return nxt;
}
```

4. Constructeur par défaut :

```
List() : first(nullptr), read(nullptr) {};
```

5. Destructeur :

```
List::~List() {
  Cell  *cur = first;
  while (cur != nullptr) {
    Cell *nxt = cur->nxt;
    delete cur;
    cur = nxt;
  };
  read = first = nullptr;
}
```

6. Constructeur par copie :

```
template <class T> List<T>::List(const List &L) {
    if (L.first == nullptr) first = nullptr;
    else {
      first = new Cell(L.first->val);
      assert(first != nullptr);
      Cell  *cur = first, *curc = L.first;
      while (curc->nxt != nullptr) {
        curc = curc->nxt;
        cur->nxt = new Cell(curc->val);
        cur = cur->nxt;
        assert(cur != nullptr);
      }
    }
    read = L.read;
}
```

7. Ajout d'un élément :

   Le code suivant suppose que l'opérateur `<` est défini pour le type `T`.

```
template <class T> void List<T>::Add(const T& v) {
    // liste vide
  if (first == nullptr) first = CellInsert(nullptr, nullptr, v);
    // insertion en 1ere position
  else if (first->val > v) first = CellInsert(nullptr, first, v);
  else { // insertion après un élèment plus petit ou à la fin
    Cell *cur = first, *nxt = first->nxt;
    while ((nxt != nullptr) && (nxt->val < v))
            { cur = nxt; nxt = nxt->nxt; }
    CellInsert(cur, nxt, v);
  }
}
```

8. Suppression d'un élément :

   Ce code suppose que l'opérateur `==` est défini pour le type `T`.

```
template <class T> void List<T>::Del(const T& v) {
  if (first == nullptr) return;
  if (first->val == v) first = CellDelete(nullptr, first, first->nxt); // debut
  else if (first->nxt != nullptr) {
    Cell *prv = first, *cur = first->nxt, *nxt = cur->nxt;
    while ((nxt != nullptr) && (cur->val < v)) {
      prv = cur; cur = nxt; nxt = nxt->nxt;
    }
    if (cur->val == v) CellDelete(prv, cur, nxt); // milieu
    else if ((nxt != nullptr) && (nxt->val == v))
                CellDelete(cur, nxt, nullptr);// fin
  }
}
```

9. Accesseur

```
const T& List::getVal(void)
    { if (read == nullptr) return T(); else return read->val; };
size_t List::Len(void) {
  size_t n = 0;
  Cell *cur = first;
  while (cur != nullptr) { n++; cur = cur->nxt; }
  return n;
}
```

10. Navigation du pointeur de lecture

```
// dans la classe
inline void Rewind(void) { read = first; };
inline void Next(void) { if (read != nullptr) read = read->nxt; };
inline size_t Advance(int p) {
  int n=0;
  while ((read != nullptr) && (p != n)) {
    n++;
    read = read->nxt;
  }
  return n; // retourne le nombre de déplacement effectif
};
inline bool isEOL(void)
    { return (read == nullptr ? true : false); }
```

11. Recherche d'un élément

```
template <class T> bool List<T>::find(const T& p) {
  Rewind();
  while ((isEOL()) && (read->val < p)) Next();
  if (isEOL()) return false;
  return (read->val == p);
}
```

12. Surcharge de l'opérateur `<<` pour `T` :

    Ce code suppose une surcharge de l'operateur `<<` pour le type `T`.

```
// dans la classe
template <class U> friend
    std::ostream& operator<<(std::ostream &out, const List<U> &a);
// définition externe
template <class T>
    std::ostream& operator<<(std::ostream &out, const List<T> &a) {
  // auto *cur = a.first;
  typename List<T>::Cell*   cur = a.first;
  while (cur != nullptr) {
    out << cur->val << " ";
    cur = cur->nxt;
  }
  return out;
}
```

Exercice 2: Liste chainée avec gestion des déplacements

On reprend la liste chainée générique `List` définie à la question précédent, et l'on veut ajouter la gestion des déplacements.

1. Gestion classique
   (a) Dans la structure `Cell`, ajouter les constructeurs par déplacement.
   (b) Ajouter une fonction `CellInsert` par déplacement.
   (c) Ajouter une méthode `void Add(T&&)`.
   (d) Ajouter le constructeur par déplacement.
2. Avec utilisation des références universelles
   (a) Dans la structure `Cell`, modifier les constructeurs par copie/déplacement en utilisant les références universelles.
   (b) Dans la structure `Cell`, modifier la fonction `CellInsert` afin de gérer la copie/déplacement en utilisant les références universelles.
   (c) Modifier la méthode `Add)` afin de gérer la copie/déplacement en utilisant les références universelles.

Solution:

1. Gestion classique
   (a) constructeurs par déplacement dans `Cell` :
   ```
   explicit Cell(T&& v) : val(move(v)), nxt(nullptr) {};
   Cell(T&& v, Cell *t) : val(move(v)), nxt(t) {};
   ```
   (b) fonction `CellInsert` par déplacement (= CellInsert avec move v lors de la transmission de paramètres) :
   ```
   Cell *CellInsert(Cell *prv, Cell *nxt, T&& v) {
     Cell  *c = new Cell(move(v), nxt);
     assert(c != nullptr);
     if (prv != nullptr) prv->nxt = c;
     return c;
   };
   ```
   (c) méthode `void Add(T&&)` (= Add avec move v lors de la transmission de paramètres) :

```
template <class T> void List<T>::Add(T&& v) {
    // liste vide
  if (first == nullptr) first = CellInsert(nullptr, nullptr, move(v));
    // insertion en 1ere position
  else if (v < first->val) first = CellInsert(nullptr, first, move(v));
  else { // insertion après un élèment plus petit ou à la fin
    Cell *cur = first, *nxt = first->nxt;
    while ((nxt != nullptr) && (nxt->val < v))
            { cur = nxt; nxt = nxt->nxt; }
    CellInsert(cur, nxt, move(v));
  }
}
```

(d) Constructeur par déplacement :

```
template <class T> List<T>::List(List &&L) : List() {
  swap(first, L.first);
  swap(read, L.read);
}
// code alternatif
template <class T> List<T>::List(List &&L) : first(L.first), read(L.read) {
  L.first = L.read = nullptr;
}
```

2. Gestion avec les références universelles

(a) constructeurs par déplacement dans `Cell` :

```
template <class U> explicit Cell(U&& v)
    : val(forward<T>(v)), nxt(nullptr) {};
template <class U> Cell(U&& v, Cell *t) : val(forward<T>(v)), nxt(t) {};
```

(b) fonction `CellInsert` par déplacement (=template + CellInsert avec forward v)

```
template <class U> Cell *CellInsert(Cell *prv, Cell *nxt, U&& v) {
  Cell  *c = new Cell(forward<T>(v), nxt);
  assert(c != nullptr);
  if (prv != nullptr) prv->nxt = c;
  return c;
};
```

(c) méthode `void Add(T&&)` (=template + Add avec forward v) :

```
template <class T> template <class U> void List<T>::Add(U&& v) {
    // liste vide
  if (first == nullptr) first = CellInsert(nullptr, nullptr, forward<T>(v));
    // insertion en 1ere position
  else if (v < first->val) first = CellInsert(nullptr, first, forward<T>(v));
  else { // insertion après un élèment plus petit ou à la fin
    Cell *cur = first, *nxt = first->nxt;
    while ((nxt != nullptr) && (nxt->val < v))
            { cur = nxt; nxt = nxt->nxt; }
    CellInsert(cur, nxt, forward<T>(v));
  }
}
```

Exercice 3: Pile d'objets template

On reprend l'idée de la `StaticStack`, mais cette fois, le type de la pile et le nombre d'éléments dans la pile sont des arguments génériques.

1. Donner la définition de la classe.
2. Quelle est la différence avec le StaticStack non template ?
3. Écrire la surcharge de l'opérateur `<<`.
4. On voudrait écrire le constructeur par copie qui permet la copie d'un `StaticStack<T,size2>` dans un `StaticStack<T,size>` . Quel est le problème ?

5. L'écrire. On recopiera le maximum de données possible depuis la copie.

Solution:

1. définition de la classe :

```
template <class T,size_t size> class StaticStack {
    private:
    T     stack[size];
    size_t  pos;
    public: ...
};
```

2. La valeur des paramètres templates créée un nouveau type pour chaque combinaison de paramètres. Exemple : `StaticStack<int,4>` est un type différent de `StaticStack<int,10>`.

3. surcharge opérateur `<<`.

```
friend ostream& operator<< (ostream& stream, const StaticStack<T, size>& s) {
    stream << "[" << pos << "/" << size << "] {";
        if (pos != 0) {
            size_t i = s.pos;
            do { --i; stream << " " << s.stack[i]; } while (i != 0);
        }
        return stream << " }";
}
```

4. Les problèmes sont multiples.
   - le type `StaticStack<T,size2>` sera différent de `StaticStack<T,size>`. Donc, ce constructeur par copie sera lui aussi un template puisqu'il doit marcher pour des types quelconques.
   - afin de copier les données internes de la classe, ces classes doivent être friend. Sinon impossible d'accéder directement aux données du stack.
   - le friend template ne pouvant pas être des spécialisations partielles, il faut limiter les instanciations possible de la classe template.

5. constructeur par copie friend

```
// friend template complet
template <class T2, int size2> friend class StaticStack;
// version 1: partir d'une instanciation complète
// et limiter les instanciations avec static_assert
template <class T2, int size2> StaticStack(const StaticStack<T2,size2> &orig) :
pos(std::min(size,orig.pos)) {
    static_assert( std::is_same<T,T2>::value );
    cout << "Template Cc" << endl;
    for (int i = 0; i < pos; i++) stack[i] = orig.stack[i];  // C=A
}
// version 2: partir d'un spécialisation partielle
template <class T2, int size2> friend class StaticStack;
template <int size2> StaticStack(const StaticStack<T,size2> &orig) :
pos(std::min(size,orig.pos)) {
    for (int i = 0; i < pos; i++) stack[i] = orig.stack[i];  // C=A
}
```

Exercice 4: Allocateur

1. on considère le code d'un allocateur conforme au standard C++.

```
// Allocateur conforme au standard C++
template<typename T> class Allocator {
public :
    // typedefs
    typedef T value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;
public :
    // conversion d'allocateur vers un autre type
    template<typename U> struct rebind {
      typedef Allocator<U> other; };
public :
    // constructeur
    explicit Allocator();
    ~Allocator();
    // allocation mémoire
    inline pointer allocate(size_type cnt);
    inline void deallocate(pointer p, size_type);
    // taille
    inline size_type max_size() const;
    // construction/destruction
    inline void construct(pointer p, const T& t);
    inline void destroy(pointer p);
    // si un autre allocateur de ce type est
    // équivalent à cet allocateur
    inline bool operator==(Allocator const&);
    inline bool operator!=(Allocator const& a);
};
```

   (a) Rappeler le rôle des différentes méthodes d'un allocateur.

   (b) A quoi sert la section `typedef` de la définition de l'allocateur ?

   (c) A quoi sert la redéfinition de l'opérateur == et != ?

   (d) Donner une implémentation des méthodes pour une utilisation sans pool de mémoire.

   (e) Utiliser cet allocateur pour définir une classe DynamicStack avec un type et un allocateur générique.

   (f) Donner un exemple d'utilisation de cette classe.

2. On voudrait maintenant des politiques afin de séparer la façon de gérer les objets (trait de l'objet) de la manière d'allouer la mémoire (politique d'allocation).

   (a) Soit le code suivant pour un trait d'objet et pour la politique d'allocation standard. Donner le code de l'allocateur qui utise le trait d'objet et la politique d'allocation.

```
template<typename T> class ObjectTraits {
public :
  inline explicit ObjectTraits() {}
  inline ~ObjectTraits() {}
  inline explicit ObjectTraits(ObjectTraits  const&) {}
  inline T* address(T& r) { return &r; }
  inline T const* address(T const& r)
        { return &r; }
  inline static void construct(T* p, const T& t)
        { new(p) T(t); }
  inline static void destroy(T* p)
        { p->~T(); }
};
```

```
template<typename T> class StandardAllocPolicy {
public:
  typedef T value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef std::size_t size_type;
  typedef std::ptrdiff_t difference_type;
public:
    inline explicit StandardAllocPolicy() {}
    inline ~StandardAllocPolicy() {}
    inline explicit StandardAllocPolicy
         (StandardAllocPolicy const&) {}
    inline pointer allocate(size_type cnt) {
      return reinterpret_cast<pointer>
         (::operator new(cnt * sizeof (T))); }
    inline void deallocate(pointer p)
      { ::operator delete(p); }
    inline size_type max_size() const { return
        std::numeric_limits<size_type>::max()
        / sizeof(T); }
};
```

   (b) Que faudrait-il faire pour modifier le trait d'objet afin que celui-ci soit copiable, déplaçable et constructible en place ?

3. On s'intéresse à un allocateur qui utilise un pool de mémoire.
   (a) Définir une fonction de gestion de pool mémoire (elle allouera un bloc et distribuera la mémoire demandée, et enregistrera les désallocations). Par simplicité, on ne gérera pas la réallocation des blocs désalloués.
   (b) Définir la politique d'allocation associée.
   (c) Donner un exemple d'utilisation.
4. Définir un gestionnaire simple d'un pool de mémoire.
5. Définir l'allocateur qui utilise ce pool de mémoire.

Solution:

1. Allocateur standard
   (a) allocate=allouer la mémoire seulement, dellocate = desallouer la mémoire seulement, construct = construction en place (=sur une zone allouée), destroy = destruction en place (=sans désallouer).
   (b) Elle permet de donner en interne à la classe des noms, et comme ils sont publiques, d'accèder aux types utilisés dans la définition de la classe.

   Exemple : `Allocator<T>::value_type` permet de récupérer le type de `T`.

   Utilise que un `type template P = Allocator<T>` lors d'une instanciation et que l'on veut récupérer le type de stocké par l'allocateur.
   (c) A vérifier si un autre allocateur peut être utilisé à la place de l'allocateur courant.
   (d) Voir le code suivant :

```
inline explicit Allocator() {}
inline ~Allocator() {}
inline pointer allocate(size_type cnt)
    { return reinterpret_cast<pointer>(::operator new(cnt * sizeof (T))); }
inline void deallocate(pointer p, size_type)
    { ::operator delete(p); }
inline size_type max_size() const
    { return std::numeric_limits<size_type>::max() / sizeof(T); }
inline void construct(pointer p, const T& t) { new(p) T(t); }
inline void destroy(pointer p) { p->~T(); }
inline bool operator==(Allocator const&) { return true; }
inline bool operator!=(Allocator const& a) { return !operator==(a); }
```

(e) classe template DynamicStack

```
template <class T, class Alloc> class DynamicStack {
private:
    T*              stack;
    size_t          size, pos;
public:
    DynamicStack(size_t n) : size(n), pos(0) {
        Alloc   mem;
        stack = mem.allocate(n);
    }
    // copy constructor
    DynamicStack(const DynamicStack& s) : size(s.size), pos(s.pos) {
        Alloc   mem;
        stack = mem.allocate(size);
        for (size_t i = 0; i < s.pos; i++)
            mem.construct(&stack[i],s.stack[i]);
    }
    // destructor
    ~DynamicStack() {
        if (stack) {
            Alloc   mem;
            for (size_t i = 0; i < pos; i++) mem.destroy(&stack[i]);
            mem.deallocate(stack);
        }
    }
    inline bool isEmpty() const { return (pos == 0); }
    inline bool isFull()  const { return (pos == size); }
    bool Push(const A &v) {
        if (isFull()) return false;
        else {
            Alloc   mem;
            mem.construct(&stack[pos],v);
            ++pos;
            return true;
        }
    }
    bool Pop(void) {
        Alloc   mem;
        if (isEmpty()) return false;
        --pos;
        mem.destroy(&stack[pos]);
        return true;
    }
};
```

(f) Par exemple :

```
DynamicStack<A, Allocator<A>>     v(10);
v.Push(5);
```

2. Politique d'allocation

   (a) code de l'allocateur

```
template<typename T,
        typename Policy = StandardAllocPolicy<T>,
        typename Traits = ObjectTraits<T>
>
class Allocator : public Policy, public Traits {
public :
    using size_type         = typename Policy::size_type;
    using difference_type   = typename Policy::difference_type;
    using pointer           = typename Policy::pointer;
    using const_pointer     = typename Policy::const_pointer;
    using reference         = typename Policy::reference;
    using const_reference   = typename Policy::const_reference;
    using value_type        = typename Policy::value_type;
    inline explicit Allocator() {}
    inline ~Allocator() {}
    inline Allocator(Allocator const& rhs):Traits(rhs), Policy(rhs) {}
    template <typename U, typename P, typename T2>
    inline Allocator(Allocator<U, P, T2> const& rhs):Traits(rhs), Policy(rhs) {}
};
```

   (b) Trait d'objet

```
template<typename T> class ObjectTraits {
    public :
    inline explicit ObjectTraits() {}
    inline ~ObjectTraits() {}
    inline explicit ObjectTraits(ObjectTraits  const&) {}
    inline T* address(T& r) { return &r; }
    inline T const* address(T const& r) { return &r; }
    inline static void construct(T* p, const T& t) { new(p) T(t); }
    inline static void construct(T* p, T&& t) { new(p) T(std::move(t)); }
    template<typename... Args> inline static
        void construct(T* p, Args... args) { new(p) T(args...); }
    inline static void construct(T* p, const T& t) { new(p) T(t); }
    inline static void destroy(T* p) { p->~T(); }
};
```

3. Pool d'allocation

   (a) code du pool de mémoire

```
template <class T> class MemoryPool {
private:
    T       *pool = nullptr;
    size_t  size = 0, pos = 0;
public:
    bool init(size_t n) {
        if (size == 0) {
            size = n;
            pool = reinterpret_cast<T*>(::operator new(n*sizeof(T)));
            return true;
        } else return false;
    }
    bool release() {
        if (pos != 0) return false;
        else {
            delete [] pool;     pool = nullptr;
            return true;
        }
    }
```

```
    MemoryPool() = default;
    MemoryPool(const MemoryPool&) = delete;
    ~MemoryPool() {}
    T *allocate(size_t n) { // alloue un bloc
        if (pool == nullptr) return nullptr;
        if (pos+n > size) return nullptr; // pool full
        T* addr = &(pool[pos]);
        pos += n;
        return addr;
    }
    void deallocate(T *p) { // desalloue
        if (pool == nullptr) return nullptr;
        assert(p>pool);
        size_t  n = size_t(p-pool);
        assert(n < size);
        pos -= n; // memoire non récupérée
    }
};
```

(b) code de l'allocateur basé sur le pool

```
template<typename T> class PoolAllocPolicy {
private:
    static MemoryPool<T>   pool;
public :
    typedef T value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;
public :
    inline explicit PoolAllocPolicy() {}
    inline ~PoolAllocPolicy() {}
    inline explicit PoolAllocPolicy(PoolAllocPolicy const&) {}
    inline pointer allocate(size_type cnt) {
        return pool.allocate(cnt/sizeof(T));
    }
    inline void deallocate(pointer p) {
        pool.deallocate(p);
    }
    inline size_type max_size() const
    { return std::numeric_limits<size_type>::max() / sizeof(T); }
    static bool init(size_t n) { return pool.init(n); }
    static bool release() { return pool.release(); }
};
```

(c) utilisation

```
PoolAllocPolicy<A>::init(100);
{
    DynamicStack<A, Allocator<A, PoolAllocPolicy<A>,ObjectTraits<A>>>  v(10);
    v.Push(5);
}
PoolAllocPolicy<A>::init(100);
```