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

TD N°5

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);
```