

Programmation Avancée Les différents mécanismes des langages (dont C++) pour la généricité

Norme ISO

Raphaëlle Chaîne
raphaelle.chaine@liris.cnrs.fr
2020-2021

1

1

Effective C++

- Du nom des ouvrages de Scott Meyer
- Option de compilation `-Weffc++`

276

276

Gestion dynamique de la mémoire

277
277

277

Opérateurs de gestion dynamique de la mémoire

- Eviter les appels aux fonctions `malloc` et `free`
- Utilisation des opérateurs `new`, `delete`, `new[]` et `delete []`

```
int *p=new int; //allocation mémoire pour 1 int  
int *tab=new int[10]; //pour 10 int consécutifs
```

- `new` retourne l'adresse de la zone allouée
(levée de `bad_alloc` sinon)
- Avantage : Couplage avec une initialisation dépendant du
type alloué (constructeurs)

278
278

278

- Le ménage est à la charge du
programmeur

```
delete p;  
delete []tab;  
delete 0; //sans effets  
delete nullptr; //C++11
```
- Couplage avec l'appel au destructeur

279
279

279

- Pour un appel explicite à un constructeur sans
qu'il soit provoqué par une allocation de mémoire
– Il suffit de fournir l'adresse en paramètre
– Syntaxe: `new(addr) MyClass(arg1, ...);`
- Appel explicite au destructeur (non provoqué à
une désallocation)
– Syntaxe: `myObj.~MyClass();`
- Exemple

```
void *mem = malloc (sizeof(Complexe));  
Complexe * pz = new(mem) Complexe(4);  
...  
pz->~Complexe();  
free mem;
```

280
280

280

Allocation mémoire, Smart pointers

- Possibilité de définir notre propre politique de gestion de la mémoire en C++
- Principe
 - Redéfinir les opérateurs new et delete pour changer la stratégie d'allocation
- La stratégie d'allocation peut-être redéfinie à différents niveaux
 1. Surcharge des opérateurs globaux (impact sur toutes les allocations)
 2. Surcharge des opérateurs pour UNE classe seulement

281
281

281

- Intérêt
 - Optimisation
 - Zones de mémoire spécialisées
 - Mise en œuvre de « Memory Pools » dans lesquels on pourra allouer des blocs de taille fixe avec plus d'efficacité que malloc qui alloue des blocs de taille variable (fragmentation de la mémoire)
- Utilisation de ressources particulières

282
282

282

- Surcharges des opérateurs new et delete globaux :

```
void* operator new(size_t size);
void* operator new[](size_t size);
void operator delete(void* ptr);
void operator delete[](void* ptr);
```
 - Mais aussi :

```
void* operator new(size_t size,
                  void* ptr);
void* operator new[](size_t size,
                  void* ptr);
```
 - Possibilité d'opérateurs new paramétrés :

```
void* operator new(size_t size,
                  arguments arg);
```
- A l'appel :

```
new(arg) MyClass;
```

283
283

283

Allocateur d'une classe

- Surcharge de new et delete pour une classe (*)
- Fonctions membres static :

```
static void* Complexe::operator new(size_t size);
static void Complexe::operator delete(void* ptr);
static void* Complexe::operator new[ ](size_t size);
static void Complexe::operator delete[ ](void* ptr);
```
- Appelées pour allouer/désallouer des Complexes ou des objets de classes dérivées!!!

```
Complexe *p=new Complexe, *q=new Complexe[4];
delete p; delete [] q;
```

(*) indépendamment opérateur global

284
284

284

- Que penser de :

```
int tasComplexes=0;
void * Complexe::operator new(size_t size)
{ tasComplexes+=size;
  return ::new Complexe;
}
```
- Ou alors de :

```
int tasComplexes=0;
void * Complexe::operator new(size_t size)
{ tasComplexes+=size;
  return malloc(size);
}
```

```
Complexe * pz= new Complexe(2);
```

285
285

285

- Attention !

```
int tasComplexes=0;
void * Complexe::operator new(size_t size)
{ tasComplexes+=size;
  return ::new Complexe;
  //constructeur par défaut
  //appele automatiquement ici
} //constructeur par défaut
//appele automatiquement ici
```
- Ce qui est correct :

```
int tasComplexes=0;
void * Complexe::operator new(size_t size)
{ tasComplexes+=size;
  return malloc(size);
  //ou bien ::new char[size]
} //constructeur par défaut
//appele automatiquement ici
```

286
286

286

```

class Complexe
{
    static char pool_memoire[MAX_OCTETS];
    //Tableau statique dans lequel seront empilés
    //tous les Complexes alloués dynamiquement
    //Ce pool aurait également pu être créé dans le tas

    static char* place_libre;
    ...
    static void * operator new(size_t size)
    {
        if(place_libre + size - pool_memoire
            < MAX_OCTETS)
            return place_libre+=size;
        else ... bad_alloc par exemple
    }
    void operator delete(void *ptr){}
};

```

287

287

Redéfinition de new dans une classe

- Modification de la stratégie d'allocation pour tous les objets de la classe
- Indépendamment du conteneur pour lequel on alloue des objets...

288

288

Les allocateurs

- Définition :
Objets gérant l'allocation mémoire des conteneurs standard
- Utilité :
Contrôle de l'allocation des données par les conteneurs

289

289

```

• Exemple :
template <typename T>
class my_allocator
{
    ...
public :
    T *allocate (size_t num);
    // allocation memoire (sans initialisation)
    // default ::operator new char[num]
    void construct(T *p, const T & v);
    // default appel explicite constructeur copie
    // new(p) T(v)
    void deallocate(T *p, size_t num);
    // desallocation memoire sans appel destructeur
    // default ::operator delete
    void destroy(T * p);
    // default appel destructeur p->~T();
};

```

```

Puis :
Vector<int, my_allocator<int> > v;

```

290

290

```

Convention :
template< typename T >
class my_allocator
{
    typedef size_t    size_type;
    typedef T*        pointer;
    typedef const T*  const_pointer;
    typedef T         value_type;
    typedef T&        reference;
    typedef const T&  const_reference;
    ...
};

#include <memory>
...fournit allocator utilisé par défaut par la STL

```

291

291

Utilisation

```

allocator<double> alloc;
allocator<double>::pointer p;

p=alloc.allocate(2*sizeof(double));
double init=5.5; int i;
for (i=0; i<2; ++i)
    alloc.construct(p+i, init);
for (i=0; i<2; ++i)
    alloc.destroy(p+i);
alloc.deallocate(p, 2*sizeof(double));

```

292

292

- Gestion dynamique de mémoire
- Source de nombreux bugs
 - Codez avec soin vos constructeurs par copie (resp. par déplacement pour les capsules RAII)
 - et vos opérateurs d'affectation (copie ou déplacement)
- Comment ne plus imposer la désallocation à l'utilisateur?
 - Intérêt des objets créés sur la pile et des objets qui gèrent eux-mêmes leur propre mémoire (ex : conteneurs de la STL, capsules RAII)
 - Utilisation de pointeurs intelligents (*smart pointers*)
 - S'employant comme des pointeurs classiques
 - Chargés de libérer automatiquement les objets pointés quand plus nécessaires

293

293

Imaginons ce que pourrait être un pointeur intelligent?

- Un pointeur qui sait s'il est propriétaire ou pas
 - Un pointeur propriétaire libère l'espace mémoire pointé avant de disparaître, un pointeur non propriétaire non.
- Une implantation possible :
 - Un `smart_ptr` encapsule un pointeur

```
template <typename T>
class smart_ptr {
    T *pointeur;
    mutable bool propriétaire;
public:
    T& operator*() const; // dereferencement
    T* operator->() const;
    ...
};
```

294

294

Imaginons ce que pourrait être un pointeur intelligent?

- Questions qui se posent au programmeur:
 - Plusieurs pointeurs peuvent-ils pointer sur un même objet?
 - Si oui :
 - Qui est propriétaire?
 - Comment un pointeur non propriétaire peut-il s'assurer que ce sur quoi il pointe n'a pas déjà été libéré par son propriétaire?

295

295

Historique : template `auto_ptr` de la STL

- Permettait de fabriquer des pointeurs ... très moyennement intelligents (voire même débiles dans certains cas qui ne correspondaient pas à une bonne utilisation!!!)
- Objet encapsulant un pointeur
- C'était le dernier qui pointait qui devait se charger de la désallocation de l'objet pointé, à sa disparition...

Deprecated in
C++ 11

296

296

Historique : template `auto_ptr` de la STL

- C'est le dernier qui pointait qui devait se charger de la désallocation de l'objet pointé, à sa disparition...
- Les pointeurs qui n'étaient plus propriétaires devenaient nuls :
 - il n'y a qu'un `auto_ptr` qui pointe sur une ressource allouée
 - les `auto_ptr` non nuls sont propriétaires

```
template <typename T>
class auto_ptr {
    T *pointeur;
public:
    T& operator*() const; // dereferencement
    T* operator->() const;
    ...
};
```

Deprecated in
C++ 11

297

297

- Utilisation
 - Un `auto_ptr` prend possession de l'objet pointé, lors d'une initialisation ou une affectation
 - Quand un `auto_ptr` propriétaire est détruit, l'objet pointé est détruit
 - Lors d'une affectation d'`auto_ptr`, la propriété est transférée car l'objet pointé ne peut avoir 2 propriétaires
 - Le pointeur qui n'est plus propriétaire est mis à nul.
- ```
auto_ptr<Complexe> pc1 (new Complexe(4,1.5));
auto_ptr<Complexe> pc2 (new Complexe(3));
auto_ptr<Complexe> pc3;
pc1->afficher();
pc1=pc2;
pc2->afficher();
pc3=pc2;
```

298

298

- Utilisation

```
auto_ptr<Complexe> pc1 (new Complexe(4,1.5));
auto_ptr<Complexe> pc2 (new Complexe(3));
auto_ptr<Complexe> pc3;
pc1->afficher();
pc1=pc2;
// destruction du Complexe pointé par pc1
// pc1 devient propriétaire de l'objet
// pointé par pc2
pc2->afficher(); //NON!
pc3=pc2; // Pas trop de sens, puisque pc2 nul
```

299

299

## Hiérarchie de classes

- Possibilité d'utiliser les `auto_ptr` au sein d'une hiérarchie de classe :  
Des `auto_ptr` instanciés avec des paramètres différents sont sans rapport entre eux, sauf en cas de polymorphisme ...  
`auto_ptr<Cadre> pc(new Cadre);`  
`auto_ptr<Employe> pe=pc;`

Possible grâce à une fonction membre d'affectation adhoc

```
template <typename T>
class auto_ptr {
...
template <typename Y>
auto_ptr<T> & operator=(auto_ptr<Y> & ptr);
...
};
```

300

300

## Limites et dangers des

### `auto_ptr`

- `pc3` ne peut plus prendre la possession de l'objet initialement pointé par `pc2` puisque `pc2` n'est plus propriétaire...
- Un `auto_ptr` n'est pas fait pour pointer sur une variable créée dans la pile
- Danger des `auto_ptr` passés par copie  

```
void f(auto_ptr<Employe> t)
{
// t, local à f, devient propriétaire de l'objet
// pointé par l'auto_ptr passé en paramètre
...
}
```

301

301

## Remplacement des `auto_ptr` par les `unique_ptr`

- Principe :
  - Garantit la propriété unique d'un objet à travers un `unique_ptr`
  - Deux `unique_ptr` ne peuvent pas pointer sur un même objet
  - Il est possible de transférer la propriété d'un objet d'un `unique_ptr` à un autre (le premier devient nul) avec `move`
  - La transmission de propriété doit être explicite

302

302

## Remplacement des `auto_ptr` par les `unique_ptr`

```
std::unique_ptr<LaClasse> p1(new LaClasse);
// p1 propriétaire
if (p1) p1->f();
{ std::unique_ptr<LaClasse>p2(std::move(p1));
// p2 devient propriétaire (donc p1 nul)
p2-> f();
p1 = std::move(p2);
// p1 reprend la propriété (donc p2 nul)
// avant que p2 ne disparaisse
}
p1->f(); #include<memory>
```

303

303

- Utilité pour de petits problèmes techniques
- Ex : Eviter de faire dépendre le nombre de blocs try catch d'appels à delete pour que les ressources soient libérées dans tout les cas

```
try(// 1 seul bloc grâce aux unique_ptr ou auto_ptr !
Fichier f1("nom_fichier1");
//peut lever une exception
unique_ptr<char> texte1(f1.contenuFichier());
//Alloc dyn pour une copie du texte dans la RAM
traitement(texte1.get()); //peut lever une exception
// alors que des ressources sont allouées
Fichier f2 ("nom_fichier2");
//peut lever une exception
unique_ptr<char> texte2 (f2.contenuFichier());
//Alloc dyn pour une copie du texte dans la RAM
traitement(texte2.get()); // peut lever une exception
// alors que d'autres ressources ont encore
// été allouées !
)
catch (exception1) & { ... }
catch (exception2) & { ... }
```

304

304

## Smart pointers avec comptage de références

- Principe :
  - Garder pour chaque objet alloué le nombre de *smart\_pointers* qui pointent sur lui
  - Destruction et libération de l'objet alloué quand le compteur de pointeurs associé devient nul
- N'existaient pas encore dans la STL avant C++11

305

305

- Implantation
  - Cas où le compteur est inclus à l'intérieur des objets
  - Par exemple imposer que tous les objets pointés dérivent d'une classe de base *SmartObject* offrant un compteur
  - Quand un *SmartObject* est créé, son compteur est mis à 0
  - Quand un *smart\_pointer* stocke l'adresse d'un *SmartObject*, le compteur du *SmartObject* est incrémenté
  - Penser à permettre l'affectation d'un *smart\_pointer* à un autre au sein d'une hiérarchie de classe (*upcast*, *downcast*)

306

306

- Implantation
  - Cas où le compteur est inclus à l'intérieur des objets
  - Par exemple imposer que tous les objets pointés dérivent d'une classe de base offrant un compteur :

```
class SmartObject {
protected:
 mutable int compteur;
 void incrementer() const {compteur++;}
 void decrements() const {compteur--;}
public:
 SmartObject() compteur(0) {}
 virtual ~SmartObject {}
 friend template <class UnSmartObject>
 class SmartPtr;
};
```

307

307

```
template <typename UnSmartObject>
class SmartPtr {
private:
 UnSmartObject *ptr;
public:
 SmartPtr(UnSmartObject *p=0) :ptr(p)
 {if (ptr!=0) ptr->incremente();}
 ...
 UnSmartObject & operator *() const {return *ptr;}
 UnSmartObject * operator->() const {return ptr;}
};
```

```
template <typename UnAutreSO >
Smart_Ptr<UnSmartObject>& operator=(const Smart_Ptr<UnAutreSO>& sm)
{if (this!=&sm)
{if (ptr!=0)
{ptr->decremente();
if (ptr->compteur ==0) delete ptr;
}
ptr=sm.ptr;
if (ptr!=0) ptr->incremente();
}
}
```

```
~SmartPtr()
{if (ptr!=0)
{ptr->decremente();
if (ptr->compteur ==0) delete ptr;
}
}
```

308

308

```
class Employe : public SmartObject
{ virtual void presentation();...
};
class Cadre : public Employe
{...};
Puis
{SmartPtr<Employe> sme;
{ SmartPtr<Cadre> smc=new Cadre;
sme=smc;
smc->presentation();
}
sme->presentation();
}
Le polymorphisme est assuré!
```

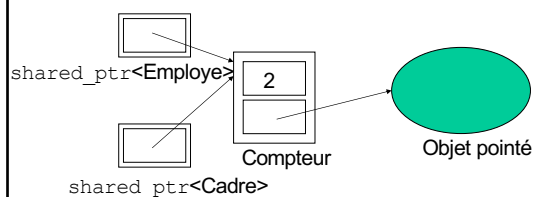
- Inconvénient de cette implantation :
  - Ne peut être utilisé sur les types primitifs

309

309

- Cas où le compteur est à l'extérieur des objets :

- Besoin d'une indirection supplémentaire!!!



- Ce type de pointeur est proposé dans la STL :  
`shared_ptr` (`#include <memory>`)

310

310

```
template <typename T>
struct Compteur {
 T *ptr;
 int compte;
 Compteur(T * p) : ptr(p), compte(0) {}
 ~Compteur() {delete ptr}
 int operator ++() {return ++compte;}
 int operator --() {return --compte;}
};
```

311

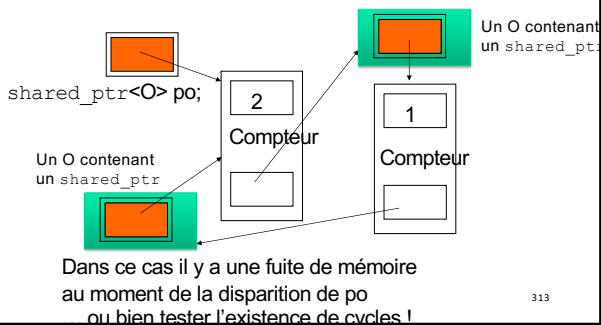
311

```
template <typename T>
class shared_ptr {
 ... Compteur<T> * cmpt;
public :
 shared_ptr (T * ptr =0) :
 cmpt(ptr?new Compteur(ptr):0)
 {if (cmpt)
 ++(*cmpt);
 }
 ~shared_ptr ()
 {if (ptr!=0)
 {--(*cmpt);
 if (cmpt->compte ==0) delete cmpt;
 }
 }
 ... Affectation, déréférencement, ...
};
```

312

312

- Limites des pointeurs intelligents avec compteur de référence :
- Problèmes des références circulaires ...

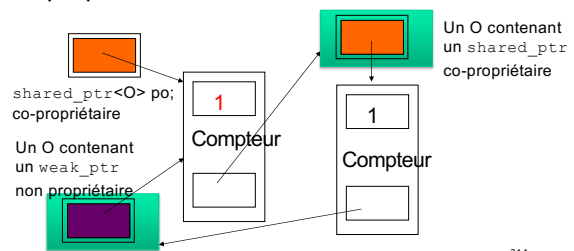


313

313

## Problème des cycles

- Il faudrait que dans le cycle un des pointeurs intelligent soit non propriétaire!



314

314

## Pointeurs non propriétaires weak\_ptr

- Principe :
  - Pointe sur un objet également géré par un ou plusieurs shared\_ptr
  - Utile pour briser les références circulaires et les ressources partagées
  - Les weak\_ptr modélisent la notion d'accès temporaire
    - S'ils sont non nuls et que la ressource pointée n'a pas été libérée, ils peuvent temporairement être convertis en shared\_ptr, le temps d'accéder à l'objet pointé
    - Pendant tout le temps où le shared\_ptr existe, le compteur de l'objet pointé ne peut être nul

315

## Pointeurs non propriétaires weak\_ptr

```
void f(std::weak_ptr<int> & wp)
{ if (std::shared_ptr sp=wp.lock())
 //Conversion en shared_ptr
 { std::cout<< *sp << std::endl; }
 else
 { std::cout <<"L'objet pointé par wp a expiré" << std::endl; }
}
```

- Le fait qu'un shared\_ptr soit créé le temps de la lecture est particulièrement utile dans un contexte multithread

316

316

## Pointeurs non propriétaires `weak_ptr`

```
int main()
{
 std::weak_ptr<int> wp;
 { std::shared_ptr sp
 =std::make_shared<int>(55);
 wp=sp;
 f(wp); // affiche 55
 }
 f(wp); // L'objet pointé a expiré
}
```

317

317

## Pointeurs non propriétaires `weak_ptr`

- `wp.lock()` échoue lorsque le compteur de `shared_ptr` est nul et que la ressource pointée a été libérée...
- Un truc qui m'a chiffonnée:
  - Cela signifie que le compteur (dans sa zone appelée bloc de contrôle) survit à la zone R libérée... puisqu'on consulte encore son compteur!!!!
  - Qui libère ce bloc de contrôle?

318

318

## Pointeurs non propriétaires `weak_ptr`

- Le bloc de contrôle est partagé par des `shared_ptr` et des `weak_ptr`
- Il contient un second compteur relatif à lui même (nombre de `shared_ptr` et de `weak_ptr` pointant sur le bloc de contrôle)
- Libération du bloc de contrôle quand ce second compteur s'annule.

319

319

## Smart\_pointers initialement proposés dans Boost

- `shared_ptr<T>`
  - « pointer to T using a reference count to determine when the object is no longer needed. `shared_ptr` is the generic, most versatile smart pointer »
- `scoped_ptr<T>`
  - « a pointer to a T automatically deleted when it goes out of scope. **No assignment possible**, but no performance penalties compared to "raw" pointers »
- `intrusive_ptr<T>`
  - « another reference counting pointer. It provides better performance than `shared_ptr`, but requires the type T to provide its own reference counting mechanism »
- `weak_ptr<T>`
  - « a weak pointer, working in conjunction with `shared_ptr` to avoid circular references »

in C++ 11

Remplacé par  
`unique_ptr` in C++11

in C++11

320

- `shared_array<T>`
  - « like `shared_ptr`, but access syntax is for an Array of T »
- `scoped_array<T>`
  - « like `scoped_ptr`, but access syntax is for an Array of T »
- Possibilité d'utiliser des fonctions pour générer certains pointeurs intelligents :
  - `make_shared` pour les `shared_ptr<T>`
  - `make_unique` pour les `unique_ptr<T>`

Ex : `std::make_shared<int>(12)`

in C++ 11

in C++ 14

321

321

## Règle de programmation

- Désormais vous ne devriez plus avoir besoin de recourir directement à `delete`
  - Utilisation de pointeurs intelligents pour prendre la valeur de retour d'un `new`
  - Faire attention à la formation éventuelle de cycles....

322

322