Université de Lyon – Université Claude Bernard Lyon 1
Master d'Informatique

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

Norme ISO

Raphaëlle Chaine
raphaelle.chaine@liris.cnrs.fr
2024-2025

Gestion dynamique de la mémoire

Opérateurs de gestion dynamique de la mémoire

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

```
LaClasse *p=new LaClasse; //Allocation mémoire pour 1
                          //laClasse + appel Constructeur
LaClasse *tab=new LaClasse[10]; //pour 10
                                //LaClasse consécutifs
```

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

- Le ménage est à la charge du programmeur

```
delete p;
delete []tab;

delete nullptr; // sans effets
delete 0; //Avant C++11
```

- Couplage avec l'appel au destructeur

- 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 pour un tableau :

```
void *mem=malloc(4*sizeof(Complexe));
for(int i=0;i<4;i++)
   new(mem+i*sizeof(Complexe)) Complexe(i);
….
for(int i=0;i<4;i++) {
  Complexe *pz
  =(Complexe*)mem+i*sizeof(Complexe);
   pz->~Complexe();}
free mem;
```


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

- 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

- 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;
```


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

- Que penser de :

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

- Ou alors de :

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

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


- Attention !

```
int tasComplexes=0;
void * Complexe::operator new(size_t size)
  { tasComplexe+=size;
    return ::new Complexe;
    //constructeur par defaut
    //appele automatiquement ici
  } //constructeur par defaut
    //appele automatiquement ici
```

- Ce qui est correct :

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



```
class Complexe
  {
    static char pool_memoire[MAX_OCTETS];
     //Tableau statique dans lequel seront empilés
     //tous les Complexes alloués dynamiquement
     //Ce pool aurait egalement pu etre cree 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){}
  };
```

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…

301

301

Les allocateurs

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

302

302

- Exemple :

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

Puis :

```
Vector<int, my_allocator<int> > v;
```

303

303

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

304

304

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));
```

305

305

- 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 pour pointer dans le tas
    - Chargés de libérer automatiquement les objets pointés quand plus nécessaires

306

306

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_pointer « pourrait » encapsuler un pointeur

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

307

307

Imaginons ce que pourrait être un pointeur intelligent?

- Questions qui se posent au programmeur:
  - Autorise-t-il plusieurs pointeurs à pointer sur un même objet?
  - Si NON :
    - Comment céder sa place à un autre pointeur?
  - Si OUI :
    - Qui est propriétaire?
    - Quand un pointeur propriétaire disparait, s'il est seul à pointer il libère la place de l'objet pointé ou bien il transmet la propriété!!!
    - Sinon comment un pointeur non propriétaire peut-il s'assurer que ce sur quoi il pointe n'a pas déjà été libéré par son propriétaire?

308

308

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 car la propriété se transmet trop facilement!!!)
- Objet encapsulant un pointeur
- C'était le dernier qui pointait qui devait se charger de la désallocation de l'objet pointé, à sa disparition…
- Transmission de propriété par operator =

Deprecated in C++ 11

309

309

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…
- **Un `auto_ptr` qui s'est fait voler la propriété devient nul :**
  - Un seul `auto_ptr` autoriisé à pointer 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;
 operator = … // transmission de propriété
  …
};
```

Deprecated in C++ 11

310

310

- Utilisation
  - Un `auto_ptr` prend possession de l'objet pointé, lors d'une initialisation par copie ou une affectation

```
template <typename T>
class auto_ptr { …
 auto_ptr(auto_ptr<T> & p);
 auto_ptr<T> &operator=(auto_ptr<T> & p);
};
```

  - 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));
pc1->afficher();
pc1=pc2;
pc2->afficher();
```

311

311

- 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
```

312

312

Hiérarchie de classes

- Possibilité d'utiliser les `auto_ptr` au sein d'une hiérarchie de classe :
  Des `auto_ptr` templatés avec des types différents peuvent avoir un rapport entre eux, 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);
…
};
```

313

313

Limites et dangers des `auto_ptr`

- 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
...
// et l'objet est donc détruit au sortir de f!!!
}
```

Sémantiquement ce n'est pas très judicieux de transférer la propriété par un constructeur par copie!!!

314

314

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 (pas de constructeur par copie)**
  - La transmission de propriété d'un `unique_ptr` à un autre doit être explicite
    - En utilisant le constructeur par déplacement
    - Et l'opérateur `move`
    - Le premier `unique_ptr` devient nul

315

315

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 proprietaire (et p1 devient nul)
  p2-> f();
  p1 = std::move(p2);
  // p1 reprend la propriété (donc p2 nul)
  // avant que p2 ne disparaisse
}
p1->f();
auto p3 = std::make_unique<double>(3.14);
```

#include<memory>

316

316

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

```
try{ // 1 seul bloc grâce aux unique_ptr ou auto_ptr !
 Fichier f1("nom_fichier1");
    //peut lever une exception fichier
 unique_ptr<char> texte1(f1.contenuFichier());
    //Alloc dyn pour une copie du texte dans la RAM
 traitement(texte1.get());
    //peut lever une exceptionTraitement
 Fichier f2 ("nom_fichier2");
    //peut lever une exception fichier
 unique_ptr<char> texte2 (f2.contenuFichier());
    //Alloc dyn pour une copie du texte dans la RAM
 traitement(texte2.get()); // peut lever une
    // exceptionTraitement
}
catch (exceptionTraitement) &
{ //où a-t-elle été levée?… Quelle mémoire faut-il
libérer? Les unique_ptr s'en occupent :-)}
```

317

317

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 (mais présents dans boost)

318

318

- 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 récupère l'adresse d'un `SmartObject`, le compteur du `SmartObject` est incrementé
  - Penser à permettre l'affectation d'un `Smart_pointer` à un autre au sein d'une hiérarchie de classe (*upcast*, *downcast*)

319

- 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 decrementer() const {compteur--;}
public :
  SmartObject() compteur(0) {}
  virtual ~SmartObject {}
friend template <class UnSmartObject>
class SmartPtr;
};
```


320

```
template <typename UnSmartObject>
class SmartPtr {
private :
  UnSmartObject *ptr;
public :
   SmartPtr(UnSmartObject *p=0) :ptr(p)
         // initialisation sur une adresse classique
       {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;
       }
    }
};
```

Code à mettre également dans le constructeur par copie

321

```
class Employe : public SmartObject
  { virtual void presentation();…
  };
class Cadre : public Employe
  {…};
```

Puis

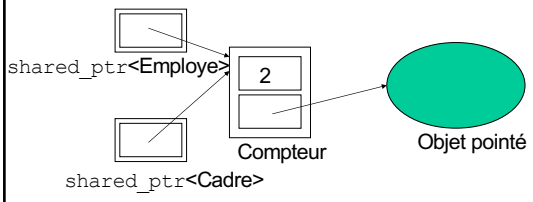
```
{
  SmartPtr<Employe> sme;
  {
    SmartPtr<Cadre> smc=new Cadre;
    sme=smc;
    smc->presentation();
  }
  sme->presentation();
}
```

Le polymorphisme est assuré!

- `intrusive_ptr` introduits par Boost
- Inconvénient de cette implantation :
  - Ne peut être utilisé sur les types primitifs

322

- Cas où le compteur est à l'extérieur des objets :
  - Besoin d'une indirection supplémentaire!!!



- Ce type de pointeur est proposé depuis C++ 11: `shared_ptr` (#include <memory>)

323

```
template <typename T>
struct Compteur {
        T *ptr;
        int compte;
        Compteur(T * p) : ptr(p), compte(1) {}
        // p doit etre non nul
        ~Compteur() {delete ptr}
        int operator ++() {return ++compte;}
        int operator --() {return --compte;}
};
```


324

```
template <typename T>
class shared_ptr {
  …
  Compteur<T> * cmpt;
 public :
  shared_ptr (T * ptr =0) :
     cmpt(ptr?new Compteur(ptr):0) {}
  shared_ptr (const shared_ptr<T> & sp);
     cmpt(sp.cmpt) {if (cmpt)
                      ++(*cmpt);}

  ~shared_ptr ()
    {if (cmpt!=0)
      {--(*cmpt);
       if (cmpt->compte =0) delete cmpt;
      }
    }

  … Affectation, déréférencement, …
};
```

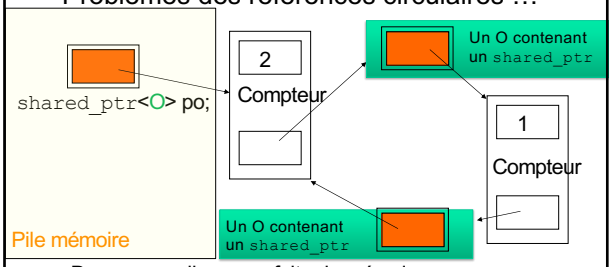
En pratique, pour créer l'objet pointé (au lieu du new) on utilise :

```
std::shared_ptr<LaClasse> p =
std::make_shared<LaClasse>();
```

325

325

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



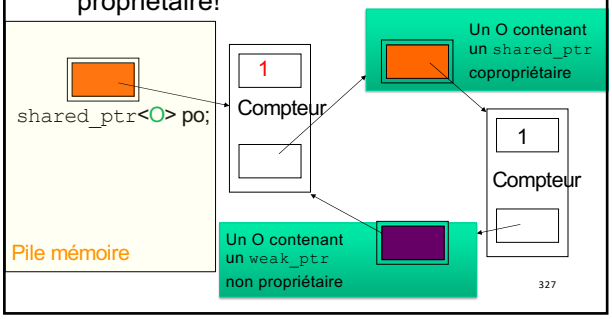
Dans ce cas il y a une fuite de mémoire
au moment de la disparition de po
… ou bien tester l'existence de cycles !

326

326

Problème des cycles

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



327

327

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 (fonction `lock`)
    - Pendant tout le temps où le `shared_ptr` existe, le compteur de l'objet pointé ne peut être nul

328

328

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

329

329

Pointeurs non propriétaires `weak_ptr`

```
int main()
{
 std::weak_ptr<int> wp;
 { std::shared_ptr<int> sp
       =std::make_shared<int>(55);
  // std::shared_ptr<int> sp(new int(55));
   wp=sp;
   f(wp); // affiche 55
 }
 f(wp); // L'objet pointé a expiré,
 //lock dans f pour s'en apercevoir
}
```

330

330

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?

331

331

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.

332

332

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

333

- `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 en allouant simultanément l'objet pointé :
  - 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

334

334

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….

335

335