



## Advanced C++

### Gestion de la mémoire



Michel André  
michel@artworks.fr

## Erreurs communes

```
S* pS {new S[3]};  
  
// ...  
  
delete pS;
```

où est l'erreur ?

```
S* pS {new S};  
if (pS) {  
    pS->doIt();  
}  
else {  
    cout << "allocation failed";  
}
```

où est l'erreur ?

### Récupération d'un échec d'allocation

- Un échec d'allocation se traduit par l'émission d'une exception de type `bad_alloc`
  - `set_new_handler` permet de placer un hook global permettant d'en prendre connaissance

```
typedef void (*new_handler)();  
new_handler set_new_handler (new_handler new_p);
```

- la constante `nothrow` du type `nothrow_t` permet d'éviter la levée d'une exception

```
if (nullptr == new (nothrow) double[n])  
    cout << "echec d'allocation" << "\n";
```

© Artworks Tous droits d'utilisation et de reproduction réservés


3

### L'opérateur de "placement"

- Les opérateurs globaux `new` et `delete` répondent à des allocations de n'importe quelle taille
  - pas optimisés pour de grands nombres de petits objets ou de taille fixe
  - `new` ajoute un surcoût au bloc alloué pour que `delete` puisse travailler
- l'opérateur `new` "placement" permet de préciser l'adresse de création
  - `new(buffer) A(...)`
    - l'adresse spécifiée est passée au constructeur de la classe `A`
  - Usages courants
    - mémoire statique (système embarqué)
    - optimisation des systèmes à mémoire paginée

```
cout << "n ? => ";  
size_t n;  
cin >> n;  
A tab[n];
```

ne compile pas



© Artworks Tous droits d'utilisation et de reproduction réservés

4

## Utilisations spécifiques

```
S* pS{ static_cast<S*> (::operator new (sizeof S)) };
```

le constructeur n'est pas appelé

```
::operator delete (pS);
```

le destructeur n'est pas appelé

```
S s;  
// ...
```

appel au constructeur sans réservation

```
S* pS {new (&s) S};
```

```
pS->~S();
```

appel au destructeur sans récupération de mémoire

© Artworks Tous droits d'utilisation et de reproduction réservés

5

## Redéfinition de new et delete

```
struct C {  
    C(){cout << "ctor" << "\n";}  
    ~C(){cout << "dtor" << "\n";}  
  
    static void* operator new(size_t size){  
        cout << "op new, size: " << size << "\n";  
        return ::operator new(size);  
    }  
    static void* operator new[](size_t size){  
        cout << "op new[], size: " << size << "\n";  
        return ::operator new[](size);  
    }  
    static void operator delete(void* ptr){  
        cout << "op delete" << "\n";  
        ::operator delete(ptr);  
    }  
    static void operator delete[] (void* ptr){  
        cout << "op delete[]" << "\n";  
        ::operator delete[] (ptr);  
    }  
};
```

peut être != de sizeof(C)  
(sous-classe)

facultatif

© Artworks Tous droits d'utilisation et de reproduction réservés

6

## Usage transparent pour le client

```
int main() {  
    delete new C();  
    cout << "\n";  
    delete [] new C[2];  
}
```

```
op new, size: 1  
ctor  
dtor  
op delete
```

```
op new[], size: 6  
ctor  
ctor  
dtor  
dtor  
op delete[]
```

- opérateurs utiles pour créer un pool masqué au code client
  - recyclage d'objets

@ Artworks Tous droits d'utilisation et de reproduction réservés

7

## Memory Function Pairings

Allocator	Deallocator
aligned_alloc(), calloc(), malloc(), realloc()	free()
operator new()	operator delete()
operator new[]()	operator delete[]()
Member new()	Member delete()
Member new[]()	Member delete[]()
Placement new()	Destructor
alloca()	Function return

@ Artworks Tous droits d'utilisation et de reproduction réservés

8



## Advanced C++

### Gestion des ressources



Michel André  
michel@artworks.fr

## Gestion des ressources: le problème

```
class Wrappeur {
    struct Ressource {
        Ressource() {cout << "ctorRes" << " ";}
        ~Ressource() {cout << "dtorRes" << " ";}
    };
    Ressource* pRes;
public:
    Wrappeur() : pRes {new Ressource}{
        throw exception {"probleme"};
    }
    ~Wrappeur(){delete pRes;}
};
```

destructeur jamais appelé

```
try {
    Wrappeur w;
}
catch (const exception& ex){
    cout << ex.what() << '\n';
}
```

ctorRes probleme

### Déterminisme et robustesse

```
void f() {  
    database* pdb {open_database ("mydb")};  
    // ...  
    close_database (pdb);  
}
```

que se passe-t-il en cas de sortie prématurée ?



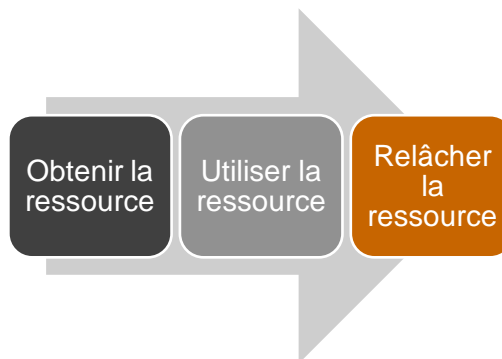
```
class DBConnection {  
    database* pdb;  
public:  
    explicit DBConnection (const char* name) :  
        pdb {open_database (name)}  
    {  
    }  
    ~DBConnection() {  
        close_database (pdb);  
    }  
}  
  
void f() {  
    DBConnection conn {"mydb"};  
    // ...  
}
```

© Artworks Tous droits d'utilisation et de reproduction réservés

11

### Resource Acquisition Is Initialization (RAII)

- Alias "Guard Idiom"
- Principe:
  - un objet détient la propriété d'un objet dynamique ou d'une ressource devant être libérée explicitement
    - construire l'objet lors de l'allocation de la ressource
    - libération de la ressource dans le destructeur de l'objet
- RAII s'applique à d'autres ressources que la mémoire



© Artworks Tous droits d'utilisation et de reproduction réservés

12

### L'opérateur ->

```
struct A {  
    void doThis() {cout << "doThis\n";}  
    void doThat() {cout << "doThat\n";}  
};  
  
class B {  
    A* pA;  
public:  
    B(A* pA) : pA {pA}{}  
    ~B() {delete pA;}  
    A* operator-> () {cout << "B::op->"; return pA;}  
};
```

```
B b {new A()};  
b->doThis();  
b->doThat();
```

```
B::op->doThis  
B::op->doThat
```

© Artworks Tous droits d'utilisation et de reproduction réservés

13

### L'opérateur -> et la généricité

```
template <typename T>  
class B {  
    T* pT;  
public:  
    B(T* pT) : pT{pT}{}  
    ~B() {delete pT;}  
    T* operator-> () {return pT;}  
};  
  
int main() {  
    B<A> b {new A()};  
    b->doThis();  
    b->doThat();  
}
```

wrappeur générique

© Artworks Tous droits d'utilisation et de reproduction réservés

14

## unique\_ptr<T>

```
int main() {
    unique_ptr<A> ap1 {new A};
}
```

construction  
destruction

```
template<class T, class D = default_delete<T>>
class unique_ptr {
public:
    typedef T element_type;
    explicit unique_ptr(T* p = nullptr);
    ~unique_ptr();
    void reset(T* p = 0);
    T* operator->() const;
    operator bool () const;
    bool operator! () const;
};
```

policy de destruction

```
unique_ptr<string> up {make_unique<string>(10, 'Z')};
```

© Artworks Tous droits d'utilisation et de reproduction réservés

15

## Custom deleter

- un deleter prend à sa charge la destruction de la ressource

```
template<typename T>
struct MyDeleter
{
    void operator()(T* t) const noexcept { delete t; }
};
```

stratégie de classe

```
unique_ptr<A, MyDeleter<A>> sp1{new A};
```

© Artworks Tous droits d'utilisation et de reproduction réservés

16



### Lazy Singleton avec `unique_ptr<T>`

```
class LazySingleton {
    LazySingleton() = default;
    ~LazySingleton() = default;
    LazySingleton(const LazySingleton& rhs) = delete;
    LazySingleton& operator= (const LazySingleton& rhs) = delete;
public:
    static LazySingleton& getInstance(){
        static unique_ptr<LazySingleton> instance;
        if (!instance)
            instance.reset(new LazySingleton());
        return *instance;
    }
    void doIt() { cout << "processing .....\\ndone\\n"; }

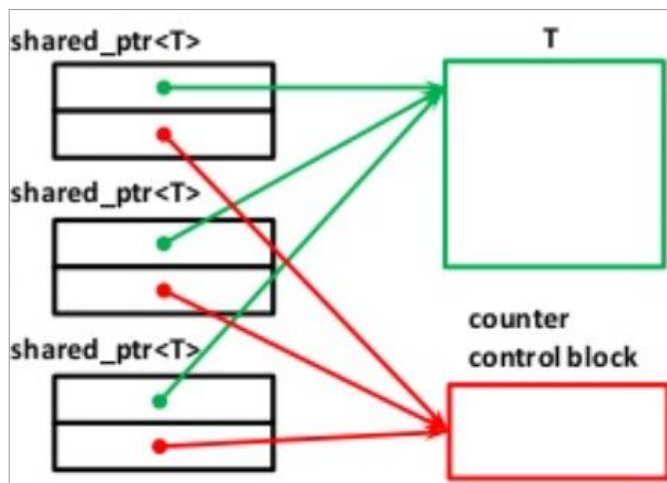
    friend default_delete<LazySingleton>;
};
```

default destruction policy

© Artworks Tous droits d'utilisation et de reproduction réservés

17

### Comptage de références

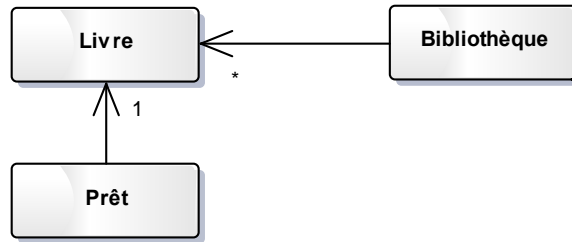


© Artworks Tous droits d'utilisation et de reproduction réservés

18

## **shared\_ptr<T>**

- la propriété est partagée par plusieurs instances de `shared_ptr`
  - l'objet embarqué n'est relâché que lorsque le dernier `shared_ptr` le référençant est détruit



© Artworks Tous droits d'utilisation et de reproduction réservés

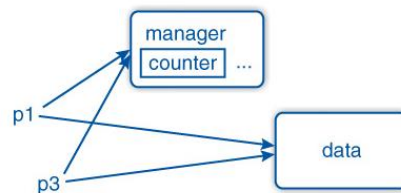
19

## **make\_shared <T>**

- alloue et initialise un objet sur le tas et retourne un `shared_ptr` qui pointe sur lui
  - `new` et `delete` sont totalement masqués

```

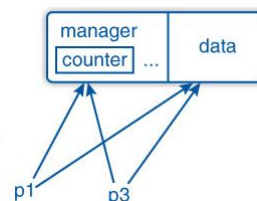
shared_ptr<string> p1 {
    new string {10, 'Z'}};
shared_ptr<string> p3 {p1};
    
```



```

shared_ptr<string> p1 {
    make_shared<string>(10, 'Z')};
shared_ptr<string> p3 {p1};
    
```

optimisé et exception safe



© Artworks Tous droits d'utilisation et de reproduction réservés

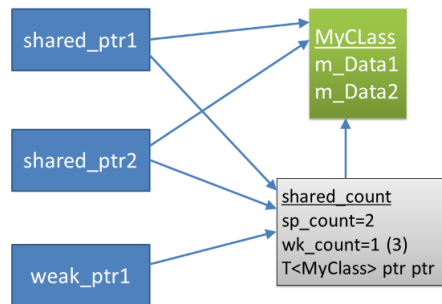
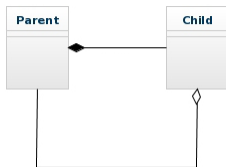
20

## **weak\_ptr<T>**

- Un weak\_ptr enveloppe une ressource **sans agir** sur son compteur de références
  - permet de **briser les dépendances circulaires**

```
template<class T>
class weak_ptr {
public:
    typedef T element_type;
    // ...

    long use_count() const;
    bool expired() const;
    shared_ptr<T> lock() const;
};
```



@ Artworks Tous droits d'utilisation et de reproduction réservés

21

## **weak\_ptr<T>: exemple d'usage**

```
weak_ptr<int> gw;

void f() {
    if (shared_ptr<int> spt {gw.lock()})
        cout << *spt << "\n";
    else
        cout << "gw is expired\n";
}

int main() {
    {
        auto sp {make_shared<int>(999)};
        gw = sp;
        f();
    }
    f();
}
```

maintien de la ressource

999  
gw is expired

@ Artworks Tous droits d'utilisation et de reproduction réservés

22

## unique\_ptr et tableaux dynamiques

C++ 17

- depuis C++ 17, shared\_ptr<T> gère les tableaux
  - même déclaration que pour unique\_ptr<T>

```
struct A {
    A(){ cout << "construction" << endl; }
    ~A(){ cout << "    destruction" << endl; }
};
```

```
int main(){
    {
        shared_ptr<A> ap1 {new A};
    }
    cout << endl;
    {
        shared_ptr<A[]> ap2 {new A[3]};
    }
}
```

```
construction
destruction

construction
construction
construction
destruction
destruction
destruction
```

@ Artworks Tous droits d'utilisation et de reproduction réservés

23

## RAII: Conclusion

- Dans d'autres langages, la seule ressource gérée par le ramasse-miette est la mémoire
  - C++ systématise l'usage RAII pour d'autres ressources:
    - mutex, streams, ...
    - finally est donc une instruction inutile
- les smart pointers utilisent RAII et proposent une interface legacy
- les fonctions make\_XXX permettent d'éviter une construction directe et optimisent vitesse et l'overhead; quelques limitations cependant:
  - problème si le deleter est redéfini
  - à éviter si les opérateurs new / delete sont redéfinis
    - taille du bloc ajoutée à celle de l'objet lors de l'appel
  - utiliser (...) au lieu de {...} pour invoquer le perfect forwarding
  - l'usage d'un weak\_ptr impose le maintien de l'objet, même s'il n'est plus partagé car il réside à côté du bloc de contrôle

@ Artworks Tous droits d'utilisation et de reproduction réservés

24