

# Programmation multi- paradigmes en C++

Session #09 : Durée de vie des objets (RAII)

Julien Deantoni, Thomas Soucheyre,  
Stephane Janel, Ken-Patrick  
Lehrmann, Ludovic Tessier, Leandro  
Fontoura Cupertino, Kevin Yeung

# Agenda

## Session 9: Durée de vie des objets (RAII)

---

**01** Initialisation des objets

---

**02** Les pointeurs intelligents (smart pointers)

---

**03** Paradigmes modernes du C++

---

# 1.

# Initialisation des objets

# Initialisation des objets en C++

## Default & Zero initialization

### Default initialization

- Ne fait **rien** sur les types primitifs
  - Pointeurs, entiers, bool, etc. auront une valeur non définie
- Les tableaux initialisent leurs éléments par défaut
- Héritée du langage C
- Appelle le constructeur par défaut si explicitement déclaré

```
int i; // undefined value

bool *ptr; // undefined value

std::string s; // calls default
               constructor: empty string
```

### Zero initialization

- Convertit la valeur 0 pour le type en question (ou ses membres pour les classes)
  - *0, false, nullptr, 0.0*
- Les tableaux zero initialisent leurs éléments
- Invoqué par les variables statiques et globales et la **value initialization**

```
static int i; // zero-initialization
```

Pour en savoir plus:

[https://en.cppreference.com/w/cpp/language/default\\_initialization](https://en.cppreference.com/w/cpp/language/default_initialization)

[https://en.cppreference.com/w/cpp/language/zero\\_initialization](https://en.cppreference.com/w/cpp/language/zero_initialization)

# Initialisation des objets en C++

## Value initialization

### Value initialization

- Invoqué avec des {} ou ()
- Appelle soit:
  - Constructeur par défaut si explicitement **déclaré**
  - **Zero initialization** sinon
- À préférer pour éviter les mauvaises surprises
  - Pas de valeurs indéfinies en général

```
int j = int(); // 0
// Note: int j(); is not possible
// it declares a function j() returning an int

bool *pBool{}; // nullptr

std::string str{}; // default constructed: ""

struct IntString {
    int i;
    std::string s;
};
IntString obj2{}; // i = 0, s = ""
```

Pour en savoir plus:

[https://en.cppreference.com/w/cpp/language/value\\_initialization](https://en.cppreference.com/w/cpp/language/value_initialization)

# Initialisation des objets en C++

Comment définir ses classes pour se protéger des dangers

```
struct IntString {
    int i;
    std::string s;
};
IntString obj1; // i is undefined!

struct Bad {
    Bad() {}

    int i;
    std::string s;
};
Bad bad1; // BAD
Bad bad2{}; // BAD: i is still not initialized!
```

```
struct Good {
    Good() = default; // optional (Rule of Zero)

    int i{};
    std::string s;
};
Good good1; // OK
Good good2{}; // OK: same as above
```

Si la classe définit un constructeur par défaut, s'assurer qu'il  
**“value-initialize”** tous ses membres

# Initialisation des objets en C++

## Autres initialisations

### Aggregate initialization

- Pour les array, std::array et les structs "standards"
- Invoqué avec des {}

```
int a[]{1, 2, 3, 4};

int b[5] = {1, 2, 3, 4};
// last element (after 4) is zero-initialized

std::array<int, 4> arr{1, 2, 3, 4};
// last element (after 4) is zero-initialized

struct IntString {
    int i;
    std::string s;
};
IntString obj3{3, "str"};
IntString obj4{.i=4, .s="str2"}; // C++20
```

### Copy initialization

- Appelle le constructeur par copie de l'objet
- Attention: ce n'est pas une assignation ! (le *copy-assignment* n'est pas appelé dans l'exemple ci-dessous).

```
IntString l = IntString{};
IntString obj5(obj4);
```

Pour en savoir plus:

[https://en.cppreference.com/w/cpp/language/aggregate\\_initialization](https://en.cppreference.com/w/cpp/language/aggregate_initialization)

[https://en.cppreference.com/w/cpp/language/copy\\_initialization](https://en.cppreference.com/w/cpp/language/copy_initialization)

# 2.

## **Les pointeurs intelligents (smart pointers)**

# Les pointeurs intelligents (smart pointers)

Pourquoi ?



**En C++ moderne, on n'utilise plus (en général) de pointeurs bruts pour la gestion de la propriété d'une ressource :**

- Transfert de propriété complexe
- Gestion de la fin de vie (libération de la ressource) difficile  
→ Sources de bugs et de failles de sécurité

- On préfère utiliser des wrappers **RAII** (voir cours précédents) définis par la librairie standard (dans le header `<memory>`):
  - `std::unique_ptr<T>`
  - `std::shared_ptr<T>` (et son ami `std::weak_ptr<T>`)
- Ils permettent au client de **s'affranchir** des difficultés précédentes et de **simplifier** le code

# Les smart pointers

[std::unique\\_ptr](#) ([documentation](#))

- C'est celui que l'on utilisera la plupart du temps
- La ressource est contenue dans une unique instance (d'où le nom)
- `std::unique_ptr` n'est pas copiable (pour garantir l'unicité de la gestion de la ressource)
  - Mais il est "*moveable*" (voir cours sur les *move semantics*) ce qui permet de transférer la responsabilité de la destruction de la ressource
- On utilise `std::make_unique` pour instancier la ressource sous-jacente

```
{  
    // allocation on the heap  
    std::unique_ptr<int> intPtr = std::make_unique<int>(5);  
  
    if (intPtr) { // has value?  
        int val = intPtr.get();  
    }  
} // destruction of the object and memory released  
  
std::unique_ptr<int> emptyIntPtr;
```

# Les smart pointers

[std::shared\\_ptr \(documentation\)](#)

- À utiliser lorsque l'on veut plusieurs instances qui partagent la propriété d'une ressource
- Implémenté avec un système de comptage de références, la ressource ne sera libérée que lorsque tous les *std::shared\_ptr* associés sont détruits
- *std::shared\_ptr* est **copiable**
  - Mais il est plus difficile de savoir quand la ressource sera effectivement libérée
- On utilise *std::make\_shared* pour instancier la ressource sous-jacente

```
{
    // allocation on the heap
    std::shared_ptr<int> intPtr = std::make_shared<int>(5);
    std::shared_ptr<int> copyPtr = intPtr; // underlying 5 is
    shared

    if (copyPtr) { // has value?
        int val = copyPtr.get(); // 5
    }
} // destruction of all objects holding the resource and
// memory released

std::shared_ptr<int> emptyIntPtr;
```

# Les smart pointers

[std::weak\\_ptr \(documentation\)](#)

- Associé à un `std::shared_ptr`, il permet de:
  - Contenir une référence en "lecture seule" de l'objet pointé
  - Casser les dépendences cycliques
- Doit être au préalable converti en `std::shared_ptr` pour pouvoir accéder à la ressource sous-jacente
  - Il dispose d'outils pour vérifier si la ressource pointée est toujours disponible
    - `lock()` pour créer un `std::shared_ptr` si la ressource existe
    - `expired()` pour vérifier si la ressource existe

```
auto intPtr = std::make_shared<int>(5);
std::weak_ptr<int> weakPtr = intPtr;

if (auto newSharedPtr = weakPtr.lock()) {
    std::cout << "value is " << *newSharedPtr << '\n';
} else {
    std::cout << "underlying resource has expired" << '\n';
}
```

# 3.

## Paradigmes modernes du C++

# Des smart pointers aux smart objects

## La vie d'un objet C++: le RAII

- Les smart pointers utilisent le concept de **RAII** (Resource Acquisition Is Initialization)
- C'est un des points d'ancrage du C++ qui permet de **lier l'utilisation d'une ressource** à la **durée de vie** d'un **objet**
  - La ressource est acquise à la construction de l'objet...
  - et sera libérée à sa destruction

```
volatile int g_i = 0;
std::mutex g_mutex;

void safe_increment() {
    std::lock_guard<std::mutex> lock(g_mutex);
    ++g_i;
    // g_mutex is automatically released when
    'lock' goes out of scope
}
```

```
class IntPtr {
public:
    IntPtr(int i) : _p(new int(i)) {}

    IntPtr(const IntPtr &) = delete;
    IntPtr& operator=(const IntPtr &) = delete;

    ~IntPtr() { delete _p; }

private:
    int *_p = nullptr;
};
```

# Composition ou héritage ?

La vie d'un objet C++: le RAII

- C++ est **multi-paradigme**:
  - Il est très souvent possible d'utiliser de la composition au lieu de l'héritage
  - Cela permet de:
    - **Simplifier** le design
    - **Réutiliser** les classes plus facilement
    - S'affranchir de problèmes liés à l'héritage (la **copie** par exemple\*)
    - Rendre le programme plus **performant** (« vous ne payez pas pour ce que vous n'utilisez pas »)

```
struct Animal {
    // virtual = 0 to make pure virtual method
    virtual std::string name() const = 0;
};

struct Snake : public Animal {
    std::string name() const override { return "snake"; }
};
```

```
struct Animal {
    std::string name() const;
};

struct Snake {
    std::string name() const { return _animal.name(); }

    Animal _animal;
};
```

\*Copie et héritage ne font pas bon ménage:

<https://stackoverflow.com/a/14461532/15438837>

# Fin de la session