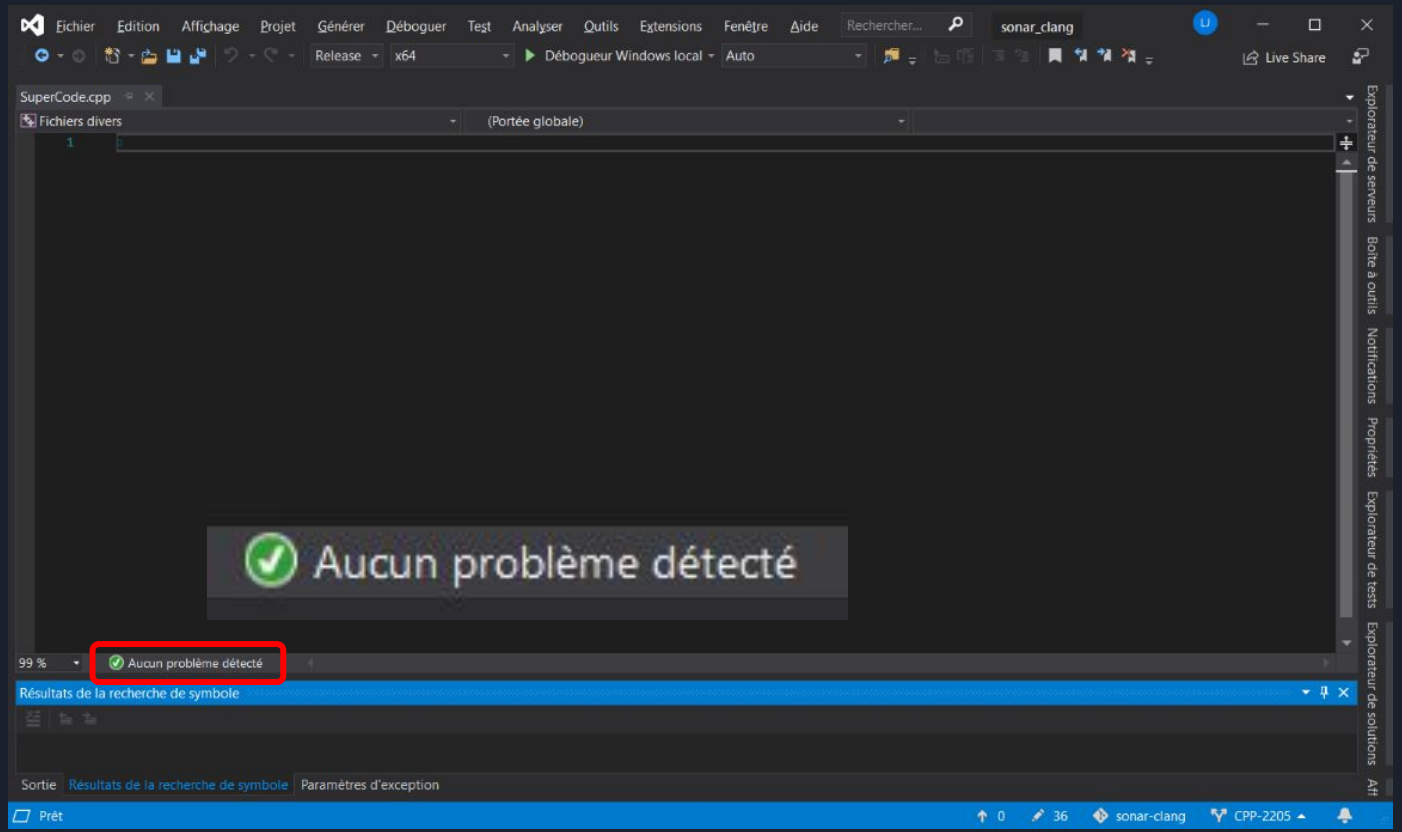
Élégance, style épuré et classe

Loïc Joly - loic.actarus.joly@gmail.com

Un extrait de code dont je suis fier



L'ensemble du code

```cpp
class Health {
public:
    Health(int maxHealth) :
        health(maxHealth),
        maxHealth(maxHealth)
    {}
    // <- Début du code dont je suis fier
    // <- Fin du code dont je suis fier
    void getDamage(int damages);
    void heal(int health);
    bool isAlive() const;
    int currentHealth() const;
private:
    int health;
    int maxHealth;
};
```

Du code dont je suis moins fier

```
class Button {
public:
  // A sprite can be owned by a button or in a global repository
  Button(Sprite *s, bool getOwnership) { ... }
  Button(Button const& b) { ... }
  Button(Button&& b) { ... }
  Button &operator=(Button const& b) { ... }
  ~Button() { ... }
private:
  Sprite* sprite;
  bool isSpriteOwned;
};
```

La simplicité est la clé de toute véritable élégance.

Coco Chanel

Conception d’une classe

L’éventuel héritage

Les fonctions… fonctionnelles 😉

- Spécifiques au domaine
- Très importantes

Les données membres

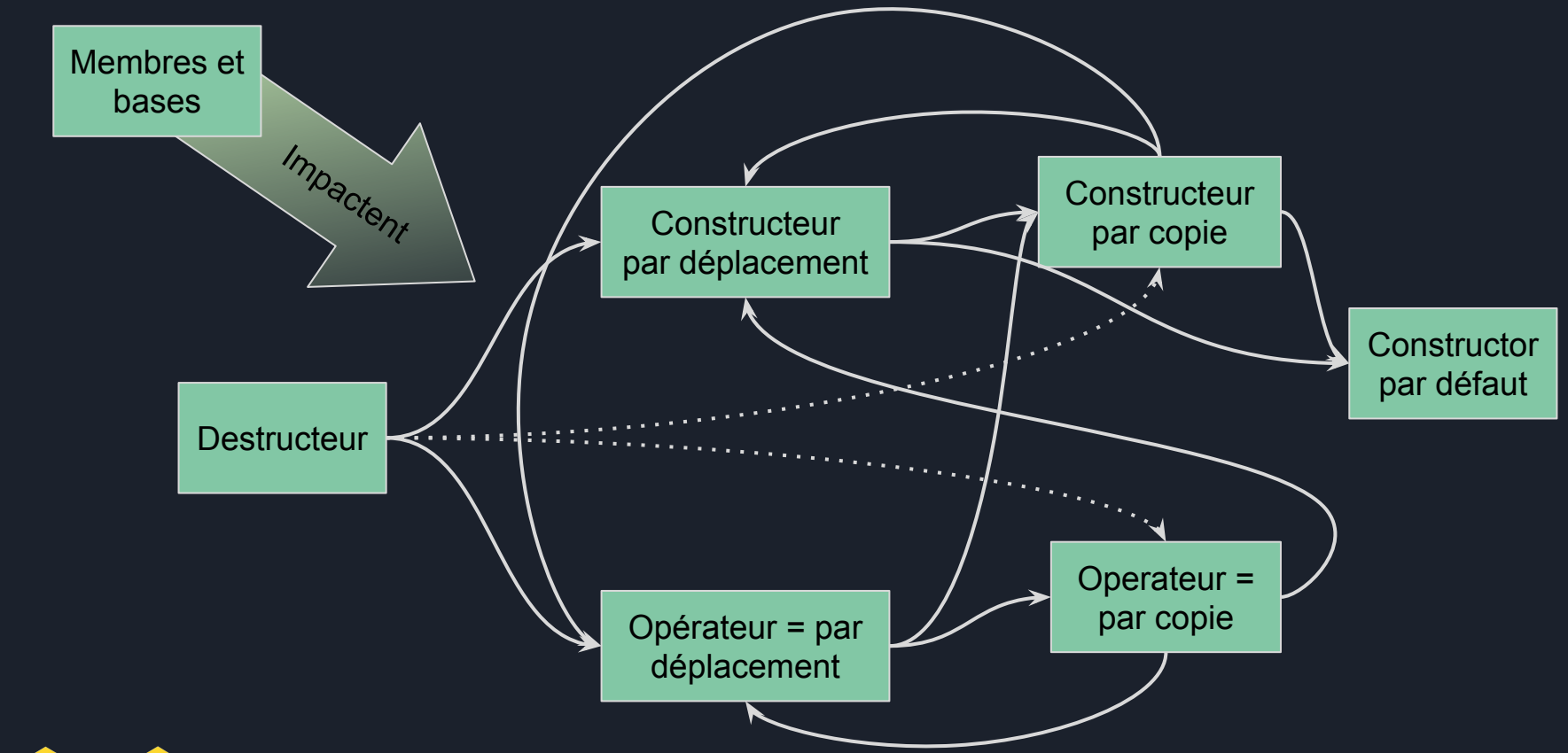
Hors sujet

Les fonctions utilitaires

- Constructeurs
  - Par défaut
  - Par copie
  - Par déplacement
- Destructeurs
- Operateurs =
  - Par copie
  - Par déplacement
- Comparaison

Comment faire un meuble ?

Comment faire un meuble ?



Les fonctions spéciales

2 règles de base

Règle des 5

Destructeur, constructeur par copie ou par déplacement, affectation par copie ou par déplacement : ces 5 opérations, toutes ou aucune tu considèreras !

Règle du 0

Et si tu peux te débrouiller pour en définir 0, profites-en !

Catégories de classes

Terminologie non officielle :

- Classes vides
- Agglomérats
- Sémantique de valeur
- Sémantique de référence
- Classes d'une hiérarchie polymorphe
- Classes de mécanisme

Classes vides

Caractéristiques

- Pas de données membre (sauf statique) => sans état
- Une seule valeur par type
- Taille non nulle
- Mais peut ne pas prendre de place (EBCO, `[[[no_unique_address]]]`)

Règle du 0 !

Exemples:

- Classes de traits `std::is_convertible`
- Tag pour la résolution de surcharge `piecewise_construct_t`
- Classe de policy `std::allocator`

Agglomérats

Règle du 0

Caractéristiques

- Des données groupées ensemble (valeurs)
- Pas vraiment de comportement
- Pas d'invariant

Composition

- Données membre publiques
- Quelques fonctions utilitaires
- Initialisation
  - Constructeur (souvent pas par défaut)
  - Initialiseur de membres
  - Aggrégat
- Comparaison ?

Agglomérat - Exemple 1

```
class CommitMetadata {
public:
    std::string message;
    Author author;
    Date date;
    int bugId;
};

CommitMetadata c1{ "Correct nasty bug",
    currentUser, 15_d / June / 2019, 1234 };
CommitMetadata c2{ "Correct very nasty bug"};
CommitMetadata c3{ c2 };
```

Agglomérat - Exemple 2

```cpp
class CommitMetadata {
public:
    std::string message = "Default message";
    Author author;
    Date date = 1_d / January / 1970;
    int bugId = 0;
};

CommitMetadata c1{ "Correct nasty bug",
    currentUser, 15_d / June / 2019, 1234 };
CommitMetadata c2;
CommitMetadata c3{ c2 };
```

Agglomérat - Exemple 3

```cpp
class CommitMetadata {
public:
    CommitMetadata(std::string const& message, Author const& author,
        Date date, int bugId) :
        message(message), author(author), date(date), bugId(bugId)
    {}
    std::string message;
    Author author;
    Date date;
    int bugId;
};

CommitMetadata c1{ "Correct nasty bug",
    currentUser, 15_d / June / 2019, 1234 };
// CommitMetadata c2{ "Correct very nasty bug" };
CommitMetadata c3{ c1 };
```

Agglomérat - Exemple 4

```
struct CommitMetadata {
    CommitMetadata(std::string const& message, Author const& author,
        Date date, int bugId) :
        message(message), author(author), date(date), bugId(bugId) {
    }
    std::string message;
    Author author;
    Date date;
    int bugId;
};

CommitMetadata c1{ "Correct nasty bug",
    currentUser, 15_d / June / 2019, 1234 };
// CommitMetadata c2{ "Correct very nasty bug" };
CommitMetadata c3{ c1 };
```

Sémantique de valeur

Une instance représente une valeur qui existe en absolu

Ex: Une date, une chaîne de caractères, une longueur, un arbre binaire

Permet de réfléchir simplement

Tout dans le langage (et la bibliothèque) est prévu pour ça

Sémantique importante

- *Regular* ou *Semiregular*
  - Copyable
  - (Default constructible)
    +
  - EqualityComparable
- Avec la bonne sémantique

Sémantique de valeur

```
T b = a;
assert(a == b);
modifie(a);
// b n'a pas changé;
modifie(b);
assert(a == b);
```

Les deux représentent la même valeur

Copie et égalité sont liés

Mais que compare-t-on lors d'une égalité ?

`std::vector::capacity()` ?

Uniquement les membres qui comptent vraiment, liés à la valeur

Après la copie, a et b vivent leur vie chacun de leur côté

Pas de partage d'état

Être identique implique qui si on applique les mêmes opérations, on a le même comportement

`std::vector::capacity()` ?

Toujours pas !

Sémantique de valeur - Opérations possibles

- Copie, déplacement, affectation par copie ou déplacement, swap
- Constructeur par défaut ?
- Comparaison
  - Seulement si coût raisonnable
  - Ex : Regex

Ne veut pas dire écrire manuellement

- Relation d'ordre
  - Seulement si coût raisonnable
  - Et si un ordre canonique
- Données membres
  - Privées
  - Invariants à assurer

Eux, on doit les écrire...

Règle du 0... Ou des 5

Digression : Et si on en faisait plus ?

Mieux que des types de base

- Contrôle des conversions
- Typage plus fort
- Pas de pénalité de perfs

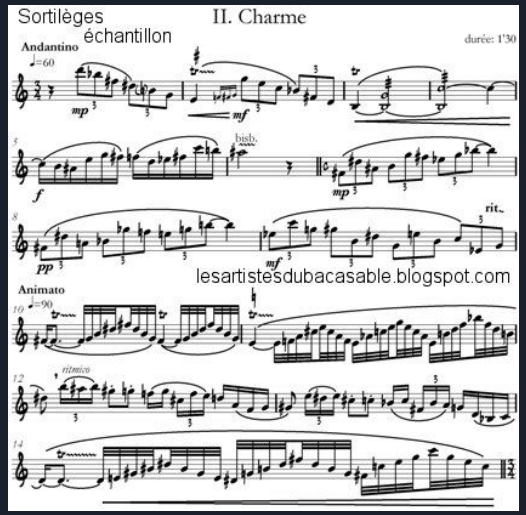
Définir des types numériques spécialisés : Types forts

Quelques points de design

- Unités SI
- Aller plus loin que les unités
- Espace vectoriel vs espace affine
  - Durée vs instant
  - Vecteur vs point
  - Taille vs adresse

Quizz

Une classe modélisant une partition musicale a-t-elle une sémantique de valeur ?



Ce code est-il correct (pensez aux fonctions spéciales)

```
class PeerToPeer {
public:
    PeerToPeer(string const &request);
    void refreshList();
    void selectActiveServer();
    IP activeServer();
private:
    vector<IP> serversWithFile;
    vector<IP>::iterator activeServer;
};
```

Sémantique de référence

- Référence des données existant par ailleurs
  - Valeurs ou entités
- Dangereuse à utiliser
  - Null reference
  - Dangling reference (attention aux temporaires)
  - Accès hors bornes
  - Sûre lors d'une descente d'un arbre d'appels
- Parfois plus performant

Exemples :

- `T*, T&`
- Itérateurs
- `std::weak_pointer`
- `std::string_view`
- `std::span`

Toujours *Regular* ou *Semiregular*

Ce code est-il correct ?

```
string getName() { return "Loïc"; }

bool containsO() {
    auto p = getName().c_str();
    while (*p != 0) {
        if (*p == 'o') {
            return true;
        }
        ++p;
    }
    return false;
}
```

L’utilisation d’aiguille dans display peut-elle être dangereuse ?

```
bool
g(vector<string> const &botteDeFoin) {
    auto aiguille = find(
        botteDeFoin.begin(),
        botteDeFoin.end(),
        "Aiguille");
    display(aiguille);
}
```

Sémantique de ~~valeur~~ *référence*

~~Les deux représentent la même valeur~~

Les deux référencent le même objet

```
T b = a;
assert(a == b);
modifie(a);
// b n'a pas changé;
```

Après la copie, a et b vivent leur vie chacun de leur côté

Ils restent liés, formant une structure de données émergente

Et d'ailleurs, qu'est-ce que ça veut dire ?

- *Pointe sur le même objet*
- *Pointe sur des objets ayant la même valeur*

Sémantique de référence : Règle du 0 ?

C’est rare d’en définir nous-même

- Et dans ce cas, peut-être la règle des 5

On en utilise souvent des pré-existants

- Et dans ce cas, la classe hébergeant peut utiliser la règle du 0
- La sémantique de référence est souvent contagieuse

Sémantique de référence : Bonnes pratiques

Éviter de stocker trop longtemps des références

- Capture par référence dans une lambda
- Membre de type référence
- ...

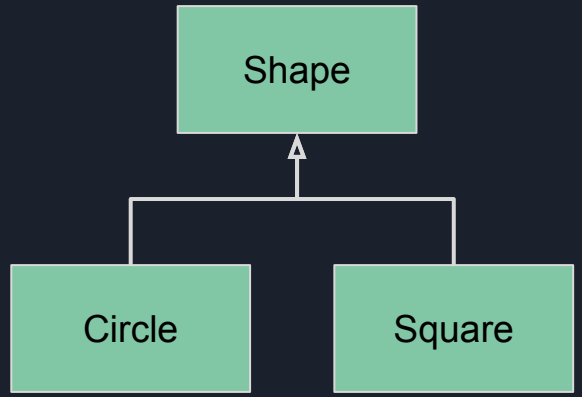
Parfois simplicité > performances

Sûr si :

- L'objet référencé est une variable locale non aliasée
- On passe la référence en descendant la pile d'appel des fonctions

Classe d’une hiérarchie polymorphe

- Souvent sur-utilisé
- Le destructeur doit être virtuel ⇒ pas de règle du 0 pour la classe de base
- Pousse vers une sémantique de référence
  - Généralement, la copie ne fait pas de sens
  - Si elle en fait : Fonction virtuelle `clone`
  - Vers `std::polymorphic_value` ?



Hiérarchie typique

```cpp
class Base {
public:
    virtual ~Base() = default;
    Base(Base const&) = delete;
    Base& operator=(Base const&) = delete;
    virtual void doSomething() = 0;
};

class Derived : public Base {
public:
    void doSomething() override;
};
```

Clarifie qu'il ne sert qu'à avoir du polymorphisme à la destruction

Inutile d'empêcher constructeur par déplacement et affectation par déplacement, ils sont déjà exclus

Copie interdite

Pas obligatoire, mais souvent une bonne idée

Ici, règle du 0

Remplace virtual dans une classe dérivée

Hiérarchie typique

Suffit à empêcher copie et déplacement

```cpp
class Base {
public:
    virtual ~Base() = default;
    Base& operator=(Base &&) = delete;
    virtual void doSomething() = 0;
};

class Derived : public Base {
public:
    void doSomething() override;
};
```

Classes de mécanismes

- Ce qui compte, c’est ce que fait la classe, pas ce qu’elle vaut
- Associée à la notion de ressource
- Importance du RAII

Exemples :

- `std::thread`
- `std::cout`
- `std::mutex`
- `std::scoped_lock`
- `ip::tcp::socket`

Classes de mécanismes

Mécanismes de base

- RAII ⇒ Règle des 5
- Déplaçable uniquement
  - Souvent suffisant
  - Souvent difficile de faire plus
- Copiable ?
  - Généralement pas de sens

Mécanismes avancés

- Une seule ressource par mécanisme
  - On combine des mécanismes de base déjà domptés
- La règle du 0 redevient applicable

Élégance, style épuré et classe

Conclusion

Conclusion

- Avant d’écrire une classe, il faut avoir une vision claire du rôle qu’elle va tenir
- Les fonctions spéciales de classe sont complexes
- Devoir en écrire est très rare

Écrivez du code clair, épuré, incisif : du code élégant !

La simplicité est la sophistication suprême

Léonard de Vinci