Modern C++ et gestion des ressources

Corentin Jabot

Mai 2015

C++ ?

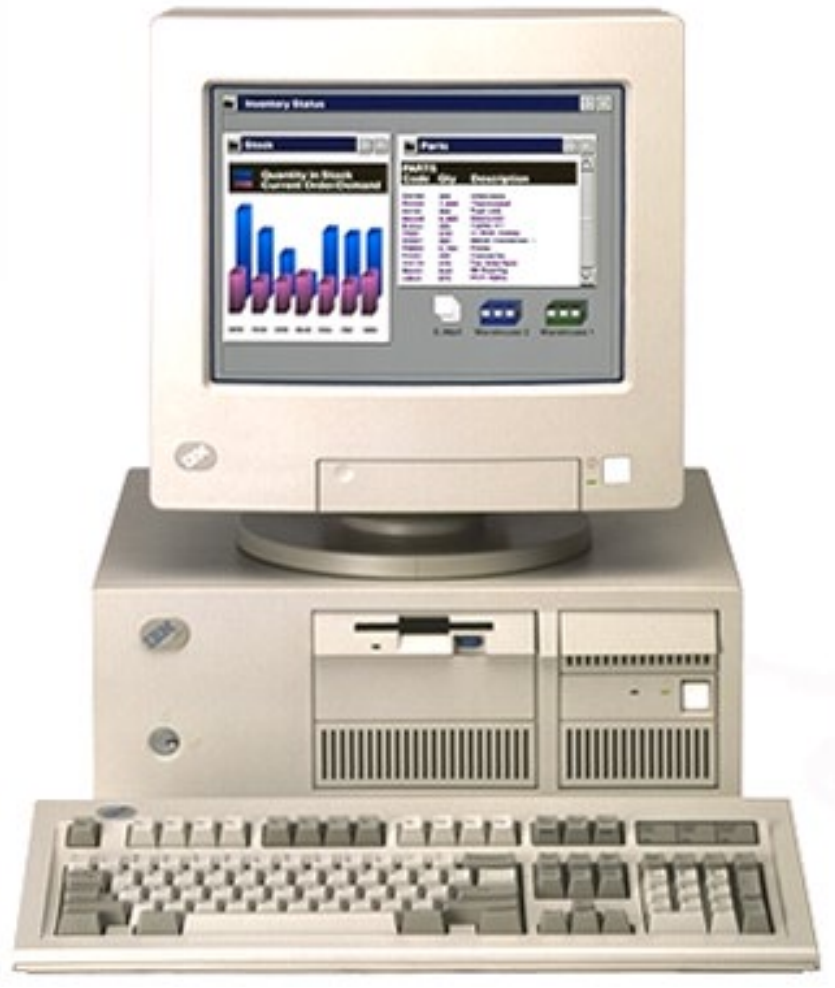
Bjarne Stroustrup

C++ ?

- 1983 : "C with classes"

C++ ?

- 1983 : "C with classes"
- 1989 : C89



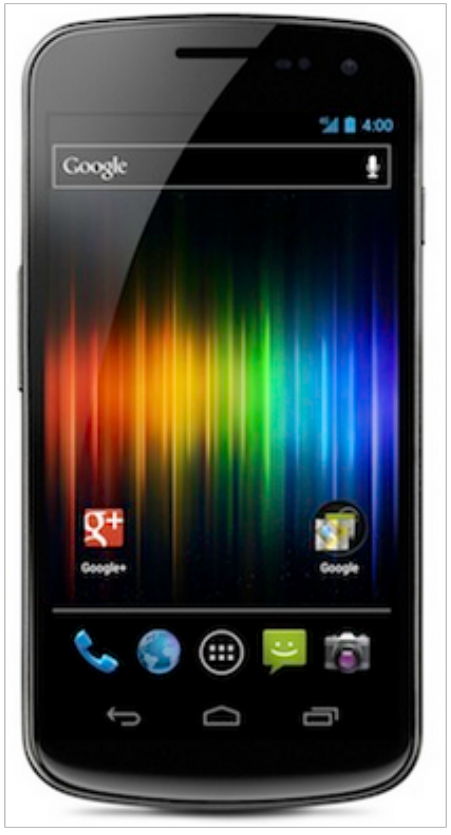
C++ ?

- 1983 : "C with classes"
- 1989 : C89
- 1998 : C++98

C++ ?

- 1983 : "C with classes"
- 1989 : C89
- 1998 : C++98

- 2011 : C++11



C++ ?

- 1983 : "C with classes"
- 1989 : C89
- 1998 : C++98
- ...
- 2011 : C++11

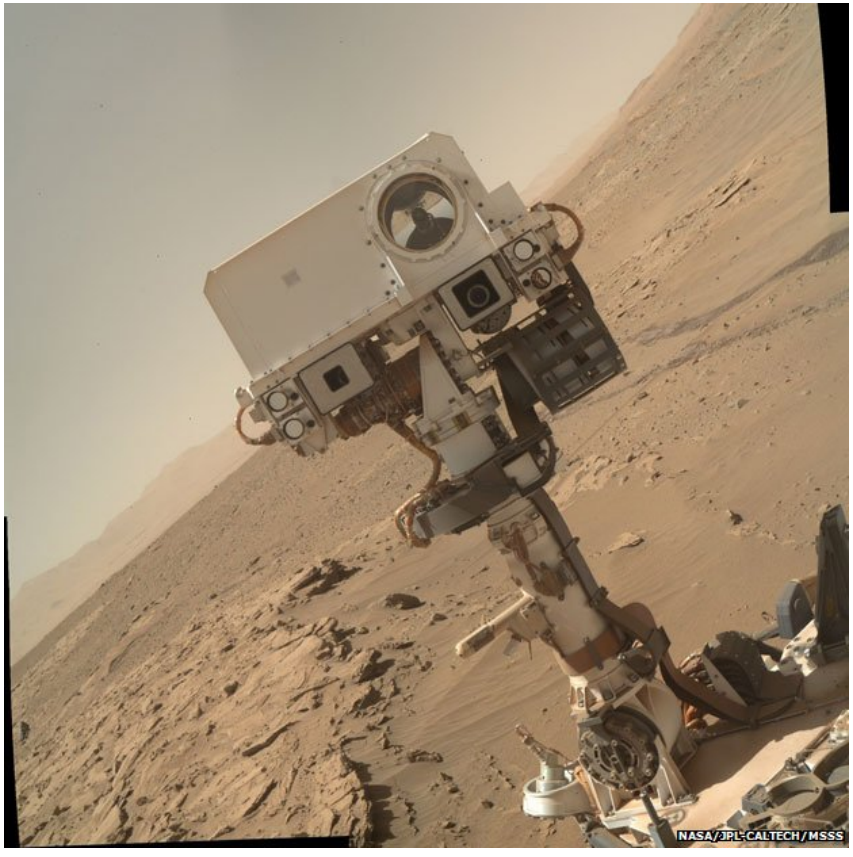
C++ ?

- Conçu pour les applications exigantes
- Facilités d'abstraction
- Mature, stable

- ... Compliqué
- ... "Not everything to everybody"

C++ ?

- Large Hadron Collider
- Photoshop
- Curiosity Rover
- Google (Back End)
- Java, PHP, Javascript, Web Renderer
- Jeux, VR



Evolution de C++

- C++11 est la première version majeure depuis C++98
- Toujours compatible avec le code écrit dans les années 90
- Aucune feature supprimée, mais de nouvelles bonnes pratiques
- Affinement de la même philosophie et des mêmes objectifs
- Nombreuses nouvelles fonctionnalités
- C++ continue d'evoluer : C++14, C++17, C++20...

Question Time

Question 1

Retour aux bases !

- `Point a(0,1);`

Retour aux bases !

- `Point a(0,1);`
  - `Point(int, int);`

Retour aux bases !

- `Point a(0,1);`
  - `Point(int, int);`
- `Point a = Point(0,1);`

Retour aux bases !

- `Point a(0,1);`
  - `Point(int, int);`
- `Point a = Point(0,1);`
  - `Point(int, int);`

Retour aux bases !

- `Point a(0,1);`
  - `Point(int, int);`
- `Point a = Point(0,1);`
  - `Point(int, int);`
  - `Point(const Point&)`

Retour aux bases !

- `Point a(0,1);`
  - `Point(int, int);`
- `Point a = Point(0,1);`
  - `Point(int, int);`
  - `Point(const Point&)`
- `Point b = a;`

Retour aux bases !

- `Point a(0,1);`
  - `Point(int, int);`
- `Point a = Point(0,1);`
  - `Point(int, int);`
  - `Point(const Point&)`
- `Point b = a;`
  - `Point(const Point &);`

Retour aux bases !

- `Point a(0,1);`
  - `Point(int, int);`
- `Point a = Point(0,1);`
  - `Point(int, int);`
  - `Point(const Point&)`
- `Point b = a;`
  - `Point(const Point &);`
- `b = a;`

Retour aux bases !

- `Point a(0,1);`
  - `Point(int, int);`
- `Point a = Point(0,1);`
  - `Point(int, int);`
  - `Point(const Point&)`
- `Point b = a;`
  - `Point(const Point &);`
- `b = a;`
  - `Point::operator=(const Point &);`

Retour aux bases !

- `Point a(0,1);`
  - `Point(int, int);`
- `Point a = Point(0,1);`
  - `Point(int, int);`
  - `Point(const Point&)`
- `Point b = a;`
  - `Point(const Point &);`
- `b = a;`
  - `Point::operator=(const Point &);`
- `Point a = {0, 1};`

Retour aux bases !

- `Point a(0,1);`
  - `Point(int, int);`
- `Point a = Point(0,1);`
  - `Point(int, int);`
  - `Point(const Point&)`
- `Point b = a;`
  - `Point(const Point &);`
- `b = a;`
  - `Point::operator=(const Point &);`
- `Point a = {0, 1};`
  - `Point(int, int);`

Question 2

Retour aux bases !

```
Point midpoint(Point a, Point b) {
        return Point (
                (a.x() + b.x()) / 2,
                (a.y() + b.y()) / 2
        );
}
Point a(0, 0);
Point b(84, 84);
Point c = midpoint(a, b);
```

Retour aux bases !

```
Point midpoint(const Point & a, const Point & b) {
        return Point (
                (a.x() + b.x()) / 2,
                (a.y() + b.y()) / 2
        );
}
Point a(0, 0);
Point b(84, 84);
Point c = midpoint(a, b);
```

std::dynarray

- (futur) conteneur standard

std::dynarray

- (futur) conteneur standard
- Allocation dynamique à la construction de l'objet

std::dynarray

- (futur) conteneur standard
- Allocation dynamique à la construction de l'objet
- Non redimensionnable

std::dynarray

- (futur) conteneur standard
- Allocation dynamique à la construction de l'objet
- Non redimensionnable
- Eléments alloués et initialisés à la construction de l'objet

std::dynarray

```
std::dynarray<std::string> bondMovies(24);
bondMovies[0] = "Dr. No";
//...
bondMovies[23] = "Spectre";

std::cout << bondMovies.at(0) << bondMovies.size();
```

std::dynarray

```
template <typename T>
class dynarray {
public:
        dynarray(std::size_t);
        std::size_t size() const;
        const T & operator[](std::size_t index) const;
        T & operator[](std::size_t index);
};
```

std::dynarray

```
template <typename T>
class dynarray {
public:
        dynarray(std::size_t);
        std::size_t size() const;
        const T & operator[](std::size_t index) const;
        T & operator[](std::size_t index);
private:
        //???????
};
```

Question 3

std::dynarray

```
template <typename T>
class dynarray {
public:
        dynarray(std::size_t);
        std::size_t size() const;
        const T & operator[](std::size_t index) const;
        T & operator[](std::size_t index);
private:
        T* m_data;
        std::size_t m_size;
};
```

Question 4

std::dynarray

```
template <typename T>
dynarray<T>::dynarray(std::size_t size)
        : m_data(new T[size])
        , m_size(size) {
}
```

std::dynarray

```
template <typename T>
class dynarray {
public:
        dynarray(std::size_t);
        ~dynarray();
        std::size_t size() const;
        const T & operator[](std::size_t index) const;
        T & operator[](std::size_t index);
private:
        T* m_data;
        std::size_t m_size;
};
```

std::dynarray

```
template <typename T>
dynarray<T>::dynarray(std::size_t size)
        : m_data(new T[size])
        , m_size(size) {
}

template <typename T>
dynarray<T>::~dynarray() {
        delete[] m_data;
}
```

Question 5

std::dynarray

Règle de 3

```
template <typename T>
class dynarray {
public:
        dynarray(std::size_t);
        ~dynarray();

        //...
};
```

std::dynarray

Règle de 3

```
template <typename T>
class dynarray {
public:
        dynarray(std::size_t);
        ~dynarray();

        dynarray(const dynarray<T> & other);

        //...
};
```

std::dynarray

Règle de 3

```
template <typename T>
class dynarray {
public:
        dynarray(std::size_t);
        ~dynarray();

        dynarray(const dynarray<T> & other);
        dynarray& operator=(const dynarray<T> & other);

        //...
};
```

Comportements des méthodes de copie générées par le compilateur

```
template <typename T>
dynarray<T>::dynarray(const dynarray<T> & other)
        : m_data(other.m_data)
        , m_size(other.m_size) {
}

void f() {
        dynarray<int> a(42);
        dynarray<int> b = a;
}
```

std::dynarray

Règle de 3

```
template <typename T>
dynarray<T>::dynarray(const dynarray<T> & other)
        : m_data(new T[other.m_size])
        , m_size(other.m_size) {

        for(std::size_t i = 0; i < m_size; ++i) {
                m_data[i] = other.m_data[i];
        }
}
```

std::dynarray
Règle de 3

```
template <typename T>
dynarray<T> & dynarray<T>::operator=(const dynarray<T> &
    other) {

        if(this == &other)
                return *this;

        delete[] m_data;

        m_data = new T[other.m_size];
        m_size = other.m_size;

        for(std::size_t i = 0; i < m_size; ++i) {
                m_data[i] = other.m_data[i];
        }

        return *this;
}
```

std::dynarray

Règle de 3

```
dynarray<int> a(1337);
dynarray<int> b(42);
//...
a = b;
```

On veut pouvoir empêcher la copie, ou l'assignement par copie de certains type....

```
dynarray& operator=(const dynarray<T> & other) = delete;
```

On veut pouvoir empêcher la copie, ou l'assignement par copie de certains type....

```
dynarray& operator=(const dynarray<T> & other) = delete;
```

... Ou demander explicitement au compilateur de générer un opérateur par défaut

```
Point(const Point<T> & other) = default;
```

std::dynarray
Règle de 3

```
template <typename T>
class dynarray {
public:
        dynarray(std::size_t);
        ~dynarray();
        dynarray(const dynarray<T> & other);
        dynarray& operator=(const dynarray<T> &) = delete;
        std::size_t size() const;
        const T & operator[](std::size_t index) const;
        T & operator[](std::size_t index);
private:
        T* m_data;
        std::size_t m_size;
};
```

Règle de 3

A retenir

- l'implémentation d'un destructeur, constructeur par copie ou opérateur d'assignement par copie, dénote d'une gestion non triviale d'une ou plusieurs variables membres.
- Règle de 3 : Définir
  - Destructeur
  - Constructeur par copie
  - Opérateur d'assignement par copie
- Forcer le compilateur à générer une méthode par défaut avec `= default;`
- Empêcher le compilateur de générer une méthode par défaut avec `= delete;`

Question 6

Retourner par valeur

```
dynarray<std::string> collectNames() {

    std::size_t size = 1'000'000;
    dynarray<std::string> names(size);
    //...
    return names;
}

void foo() {
    dynarray<std::string> names = collectNames();
}
```

Retourner par valeur

```
dynarray<std::string> collectNames() {

    std::size_t size = 1'000'000;
    dynarray<std::string> names(size);
    //...
    return names;
}

void foo() {
    dynarray<std::string> names = collectNames();
}
```

- 2 000 000 instances de `std::string` lors de la copie

Retourner par valeur

```
dynarray<std::string> collectNames() {

    std::size_t size = 1'000'000;
    dynarray<std::string> names(size);
    //...
    return names;
}

void foo() {
    dynarray<std::string> names = collectNames();
}
```

- 2 000 000 instances de `std::string` lors de la copie
- 3 000 0000 instances crées puis détruites dans la durée de vie du programme (en ignorant les optimisation du compilateur)

Retourner par valeur

```
dynarray<std::string> collectNames() {

    std::size_t size = 1'000'000;
    dynarray<std::string> names(size);
    //...
    return names;
}

void foo() {
    dynarray<std::string> names = collectNames();
}
```

- 2 000 000 instances de `std::string` lors de la copie
- 3 000 0000 instances crées puis détruites dans la durée de vie du programme (en ignorant les optimisation du compilateur)
- => Performances désastreuses

Retourner par valeur

```
dynarray<std::string>* collectNames() {

    std::size_t size = 1'000'000;
    dynarray<std::string>* names
        = new dynarray<std::string>(size);
    //...
    return names;
}

void foo() {
    dynarray<std::string>* names = collectNames();
    delete names;
}
```

Retourner par valeur

```
dynarray<std::string>* collectNames() {

    std::size_t size = 1'000'000;
    dynarray<std::string>* names
        = new dynarray<std::string>(size);
    //...
    return names;
}

void foo() {
    dynarray<std::string>* names = collectNames();
    delete names;
}
```

NOPE.

Retourner par valeur

```
collectNames(dynarray<std::string> & result) {

    std::size_t size = 1'000'000;
    dynarray<std::string> names;
    //...
    result = names;
}
```

Retourner par valeur

```
collectNames(dynarray<std::string> & result) {

    std::size_t size = 1'000'000;
    dynarray<std::string> names;
    //...
    result = names;
}

void foo() {
    dynarray<std::string> names(????);
    collectNames(names);
}
```

Retourner par valeur

```
collectNames(dynarray<std::string> & result) {

    std::size_t size = 1'000'000;
    dynarray<std::string> names;
    //...
    result = names; //Ne compile pas ( souvenez vous,
        operator= déclaré "=delete");
}
void foo() {
    dynarray<std::string> names(????);
    collectNames(names);
}
```

Retourner par valeur

- Retourner par pointeur : Pas élégant, peu compréhensible, très dangereux.

Retourner par valeur

- Retourner par pointeur : Pas élégant, peu compréhensible, très dangereux.
- Paramètre de sortie (référence) : pas toujours possible, pas intuitif non plus.

Retourner par valeur

- Retourner par pointeur : Pas élégant, peu compréhensible, très dangereux.
- Paramètre de sortie (référence) : pas toujours possible, pas intuitif non plus.
- Retourner par valeur : simple, mais création de copies...

Retourner par valeur

- Retourner par pointeur : Pas élégant, peu compréhensible, très dangereux.
- Paramètre de sortie (référence) : pas toujours possible, pas intuitif non plus.
- Retourner par valeur : simple, mais création de copies...

A moins que...

Sémantique de Déplacement (move)

- Beaucoup de copies sont suivies de destruction (retour de fonction)

Sémantique de Déplacement (move)

- Beaucoup de copies sont suivies de destruction (retour de fonction)
- Selon le niveau d'optimisation le compilateur peut éliminer certaines copies de valeur de retour (RVO)

Sémantique de Déplacement (move)

- Beaucoup de copies sont suivies de destruction (retour de fonction)
- Selon le niveau d'optimisation le compilateur peut éliminer certaines copies de valeur de retour (RVO)
- La majorité des classes encapsulent des données allouées dynamiquement

Sémantique de Déplacement (move)

- Solution: **déplacer** les variables membres dans le cas ou l'objet va être détruit (C++11)

Sémantique de Déplacement (move)

- Solution: **déplacer** les variables membres dans le cas ou l'objet va être détruit (C++11)
- Le compilateur sait détecter les variables sur le point d'être détruites

Sémantique de Déplacement (move)

- Solution: **déplacer** les variables membres dans le cas ou l'objet va être détruit (C++11)
- Le compilateur sait détecter les variables sur le point d'être détruites
- Il est possible de créer des Constructeur par copie et des opérateurs d'assignement spécifiques pour ce cas de figure.

Sémantique de Déplacement (move)

```
template <typename T>
class dynarray {
public:
        dynarray(std::size_t);
        ~dynarray();
        dynarray(const dynarray<T> & other);
        dynarray& operator=(const dynarray<T> &) = delete;


        //! Constructeur par déplacement
        dynarray(dynarray<T> && other);

        //! operateur d'assignement par déplacement
        dynarray& operator=(dynarray<T> && other) = delete;

        //..
};
```

Sémantique de Déplacement (move)

```
template <typename T>
dynarray<T>::dynarray(dynarray<T> && other)
        : m_data(other.m_data)
        , m_size(other.m_size) {

        other.m_data = 0;
}

template <typename T>
dynarray<T> & dynarray<T>::operator=(dynarray<T> && other) {

        std::swap(m_data, other.m_data);
        std::swap(m_size, other.m_size);
}
```

Sémantique de Déplacement (move)

A retenir

- En C++11, retourner de gros objets est performant et recommandé
- Le compilateur sait distinguer les variables qui peuvent être déplacées de celles qui doivent être copiées
- Déplacement = Optimisation de la copie
- 2 nouvelles "fonction spéciales" : La règle de 3 devient règle de 5.
- Introduction au sujet, à vous d'approfondir...

Gestion des ressources

Gestion des ressources

- Une ressource désigne tout ce qui peut être mis à disposition du programme par le système d'exploitation ou le matériel sous-jacent.

Gestion des ressources

- Une ressource désigne tout ce qui peut être mis à disposition du programme par le système d'exploitation ou le matériel sous-jacent.
- Acquérir / Libérer

Gestion des ressources

- Une ressource désigne tout ce qui peut être mis à disposition du programme par le système d'exploitation ou le matériel sous-jacent.
- Acquérir / Libérer
- Exemples:
  - Fichier `open / close`
  - Bloc mémoire **`new / delete`**
  - Socket réseau `connect / close`
  - Le bras du robot Curiosity, une fenêtre d'application, une imprimante, etc

Gestion des ressources

- Une ressource désigne tout ce qui peut être mis à disposition du programme par le système d'exploitation ou le matériel sous-jacent.
- Acquérir / Libérer
- Exemples:
  - Fichier `open / close`
  - Bloc mémoire **`new / delete`**
  - Socket réseau `connect / close`
  - Le bras du robot Curiosity, une fenêtre d'application, une imprimante, etc
- Les ressources sont rares

Gestion des ressources

- Une ressource désigne tout ce qui peut être mis à disposition du programme par le système d'exploitation ou le matériel sous-jacent.
- Acquérir / Libérer
- Exemples:
  - Fichier `open / close`
  - Bloc mémoire **`new / delete`**
  - Socket réseau `connect / close`
  - Le bras du robot Curiosity, une fenêtre d'application, une imprimante, etc
- Les ressources sont rares
- Les ressources acquise doivent toujours être libérées!

Gestion des ressources

```
bool fileStartsWithA(const char* filename) {
    FILE* file = open(filename, "r");
    if(!file)
     return false;

    unsigned char buffer;
    if(fread(&buffer, 1, 1, file) == 0 )
     return false;

    if(buffer == 'A')
        return true;

    close(file);
    return false;
}
```

Resource Acquisition Is Initialization

l'Acquisition d'une Ressource est une Initialisation

- Technique de gestion des ressources en C++
- Mise au point en 1984
- Reprise récemment par Rust & D
- La ressource est acquise dans le constructeur, et libérée dans le destructeur
- Généralisation : la ressource est libérée dans le destructeur

Resource Acquisition Is Initialization

l'Acquisition d'une Ressource est une Initialisation

La classe `dynarray` est un bon exemple de RAII

```
template <typename T>
dynarray<T>::dynarray(std::size_t size)
    : m_data(new T[size])
    , m_size(size) {
}

template <typename T>
dynarray<T>::~dynarray() {
    delete[] m_data;
}
```

Question 7

Resource Acquisition Is Initialization
l'Acquisition d'une Ressource est une Initialisation

Les classes de la librairies standard implémentent également ce principe:

- `std::vector`
- `std::string`
- `fstream`

Question 8

Resource Acquisition Is Initialization

l'Acquisition d'une Ressource est une Initialisation

```
class File {
    FILE* handle;

public:
    File(const char* filename, const char* mode) {
       handle = open(filename, mode);
    }

    ~File() {
        close(handle);
    }

    File(const File & other) = delete;
    File & operator=(const File & file) = delete;
};
```

Question 9

Resource Acquisition Is Initialization

l'Acquisition d'une Ressource est une Initialisation

```
Shape* shapeFromName(const std::string & name ) {
    if(name == "square")
        return new Square;
    if(name == "triangle")
        return new Triangle;
    return nullptr;
}

void f() {
    Shape* shape = shapeFromName("triangle");
}
```

Resource Acquisition Is Initialization

l'Acquisition d'une Ressource est une Initialisation

```
Shape* shapeFromName(const std::string & name ) {
    if(name == "square")
        return new Square;
    if(name == "triangle")
        return new Triangle;
    return nullptr;
}

void f() {
    Shape* shape = shapeFromName("triangle");
    delete shape;
}
```

Smart pointers
Pointeurs intelligents

- Classes qui permettent la gestion automatique de la durée de vie d'une ressource allouée sur le tas
- Basées sur le principe de RAII
- header `<memory>`
  - `std::unique_ptr` : Non copiable
  - `std::shared_ptr` : Partagé, copiable (l'objet est détruit lorsque toutes les copies sont détruites)
  - `std::weak_ptr` : Construit à partir d'un `std::shared_ptr`, permet d'"observer" un pointeur sans influencer sur sa durée de vie

Smart pointers
Pointeurs intelligents

```
{
    std::unique_ptr<Shape> shape(new Triangle);
    std::cout << shape->name() << std::endl;
}
```

Smart pointers
Pointeurs intelligents

```
{
    std::unique_ptr<Shape> shape(new Triangle);
    std::cout << shape->name() << std::endl;
}

{
    std::shared_ptr<Shape> shape(new Triangle);
    std::shared_ptr<Shape> copy = shape;
}
```

Smart pointers
Pointeurs intelligents

```
{
    std::unique_ptr<Shape> shape
        = std::make_unique<Triangle>();

    std::cout << shape->name() << std::endl;
}

{
    std::shared_ptr<Shape> shape
        = std::make_shared<Triangle>();

    std::shared_ptr<Shape> copy = shape;
}
```

Smart pointers
Pointeurs intelligents

```
{
    auto shape = std::make_unique<Triangle>();
    std::cout << shape->name() << std::endl;
}

{
    auto shape = std::make_shared<Triangle>();
    std::shared_ptr<Shape> copy = shape;
}
```

Smart pointers
Pointeurs intelligents

Principales fonctionnalités:

```
std::unique_ptr<Shape> shape = std::make_unique<Triangle>();

//accès aux membres de l'objet géré par le pointeur
std::cout << shape->name() << std::endl;

//Récupération du pointeur
Shape& ptr = *ptr;
Shape* ptr2 = ptr.get();

//tester si le pointeur est nul
if(!shape) {
// le pointeur est null
}
```

Question 10

Smart pointers

```
std::unique_ptr<Shape>
shapeFromName(const std::string & string ) {
    if(string == "square")
        return make_unique<Square>();
    if(string == "triangle")
        return make_unique<Triangle>();
    return std::unique_ptr<Shape>();
}
```

Smart pointers

```
std::unique_ptr<Shape>
shapeFromName(const std::string & string ) {
    if(string == "square")
        return make_unique<Square>();
    if(string == "triangle")
        return make_unique<Triangle>();
    return std::unique_ptr<Shape>();
}

void f() {
    std::unique_ptr<Shape> shape = shapeFromName("triangle");
}
```

Question 11

Smart pointers

```
std::__________<Shape> shapeFromName(const std::string &);

std:string shapeName(__________shape) {
    return shape->name();
}
void f() {
    __________ shape = shapeFromName("hello");
    std:: cout << shapeName(__________) << std::endl;
}
```

Smart pointers

```
std::unique_ptr<Shape>
shapeFromName(const std::string &);

std:string shapeName(const Shape* shape) {
    if(!shape)
        return "";
    return shape->name();
}
void f() {
    std::unique_ptr<Shape> shape
            = shapeFromName("hello");

    std::cout << shapeName(shape.get()) << std::endl;
}
```

Question 12

Smart pointers

```
template <typename T>
class std::unique_ptr {
    T* ptr;
public:
    unique_ptr(T*);
    ~unique_ptr();

    unique_ptr(const unique_ptr<T> &) = delete;
    unique_ptr(const unique_ptr<T> && ) = default;

    unique_ptr<T>& operator=(const unique_ptr<T> &) = delete;
    unique_ptr<T>& operator=(unique_ptr<T> && ) = default;

    T* operator->() const;
    operator bool() const;
    T & operator*() const;
    T* get() const;
};
```

Conclusion

- Retournez de préférence par valeur
- Passez les paramètes d'entrée par référence constante
- Utilisez le principe de RAII pour gérer les ressources système.
- Pensez à la règle de 3 (et 5)
- Préférez ne redéfinir aucun destructeur / opérateur ou constructeur de copie : **règle de 0**

Conclusion

- Evitez d'utiliser `new` et `delete`
- Utilisez les smart pointers pour gérer la durée de vie des objets alloués dynamiquement
- Préférez `std::make_unique` et `std::make_shared` pour créer des pointeurs intelligents.
- N'utilisez de smarts pointers que lorsque vous avez besoin de gérer la durée de vie de l'objet. Continuez d'utiliser des pointeurs et des références dans les autres cas.