Modern C++ et gestion des ressources

Objectifs du TD

Ce TD va vous amener à découvrir comment gérer la mémoire et les ressources système en C++11 grâce à la présentation de deux nouveaux concepts : La sémantique de mouvement, et les « smarts pointers »(pointeurs intelligents).

1 Rappels sur la « sémantique de valeur »

La classe `Point` introduite au TD2 est de retour !

Question 1

Pour chacune des expressions suivantes, quelles « fonctions membres spéciales »sont appellées ? (on appelle fonctions membres spéciales l'ensemble des constructeurs, destructeurs et opérateurs d'affectation.)

	`Point()`	`Point(int x, int y)`	`Point(const Point&)`	`Point::operator=(const Point & other)`

`Point a(0,1);`
`Point a = Point(0,1);`
`Point b = a;`
`b = a;`				

`Point a = {0, 1};`				

Question 2

```
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);
```

- Que fait la fonction `midpoint` ? Quelles sont les coordonées du point `c` ?
- Combien de copies de `a` et de `b` sont effectuées ? Combien de fois la valeur de retour est-elle copiée ?
- Changer la signature de la fonction pour réduire le nombre de copies.

2 La classe dynarray

La classe `std:::dynarray` est un conteneur (TAD) séquentiel (les objets sont alloués sur le tas et se suivent en mémoire), dont la taille est déterminée à l'instanciation de la classe et ne change pas pendant la durée de vie de l'objet. Elle s'utilise comme cela :

```
dynarray<std::string> bondMovies(24); // crée un tableau de 24 éléments, de taille fixe.
bondMovies[0] = "Dr. No"; //accède ou modifie un élément à l'index donné
```

Malheureusement, cette classe ne fait pas encore partie du Standard C++ et on vous a chargé de proposer une implémentation d'une partie de cette classe.

Question 3

Vous trouverez sur la page suivante une interface minimale pour cette classe. Vous remarquerez que les méthodes de la classe sont déclarées, mais les variables membres ont été oubliées! Complétez l'interface de la classe avec les variables membres nécessaires.

```
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 4

Proposez une implémentation du constructeur `dynarray<T>::dynarray(std::size_t)`. Quelles sont les mesures particulières à prendre lorsque de la mémoire est allouée sur le tas ? Implémentez ces mesures.

Règle de 3

Question 5

"La règle de 3" stipule que si un destructeur, un opérateur de copie ou un opérateur d'assignement est défini, alors les 2 autres devraient également être définis. Justifiez cette règle. Modifiez votre implémentation de la classe `dynarray` en fonction.

Question 6

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

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

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

Combien d'instances de `dynarray` sont créées dans cet extrait de code ? Combien d'instances de `std::string` ? Quels sont les problèmes de performances posés par ce code ? Proposez des solutions.

3 RAII - Ressource acquisition is initialization

Question 7

Citez des exemples de classes qui respectent le principe de RAII

Question 8

Proposez une classe `File` qui encapsule un descripteur de fichier POSIX (`FILE*`), en respectant le principe de RAII. Quelles sont les méthodes que vous devrez implémenter ?

Question 9

```
Shape* shapeFromName(const std::string & string ) {
	if(string == "square")
		return new Square;
	if(string == "triangle")
		return new Triangle;
	return 0;
}
void f() {
	Shape* shape = shapeFromName("triangle");
}
```

Identifiez les problèmes dans ce code.

Peut-on réécrire la fonction `shapeFromName` pour retourner une valeur plutôt qu'un pointeur ? une référence ? Justifiez.

4 Pointeurs intelligents

Question 10

Réécrivez la fonction `shapeFromName` pour utiliser un pointeur intelligent. Quel pointeur intelligent devez vous utiliser ?

Question 11

La fonction `shapeName` retourne le nom d'une forme. Complétez le code suivant, de sorte qu'il s'exécute correctement:

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

std:string shapeName(_____shape) {



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

Question 12

Voici l'interface (simplifiée), de la classe `std::unique_ptr`.

```
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> &&);

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

Cette classe comporte des opérateurs et des notations que vous n'avez jamais rencontré auparavant. De ce que vous connaissez du comportement des pointeurs et du fonctionnement de `std::unique_ptr`, pouvez vous déduire le fonctionnement de chaque méthode ?

Annexe : C++11

La première version de C++ est publiée en 1983. Mais ce n'est qu'en 1998 que C++ est standardisé dans une norme ISO connue sous le nom de « C++98 ». En 2011, une version majeure apporte un grand nombre de fonctionnalités qui changent la façon de développer en C++. La transformation est telle que les auteurs parlent souvent de « Modern C++ ». C++14 généralise certaines fonctionnalités de C++11.

Notez que le commité C++ publie un « Standard » (un document), et non un compilateur. Il faut souvent attendre plusieurs années pour que l'ensemble des fonctionnalités soient implémentées par les différents compliateurs C++ existants. Toutefois, Clang, GCC, et MSVC implémentent la majorité du standard C++11, y compris l'ensemble des fonctionnalités présentées en TD et dans ce document.

auto

auto est un mot clé utilisé en lieu et place d'un type, qui laisse la charge au compilateur de déterminer le type de la variable, en fonction de l'expression qui lui est affectée. On parle d'**inférence de type**.

```
auto a = 42; // a est de type int
auto b = "geronimo"; // b est de type const char*
a = b; //erreur : impossible de convertir un const char* en int
```

auto permet d'éviter les erreurs de conversion implicite, par exemple:

```
std::vector<Stuff> thingies;
auto size = thingies.size(); //size est du même type que std::vector::size(), à savoir... std::size_t !
int incorrectSize = thingies.size(); //conversion std::size_t -> int
```

Et simplifie l'écriture et la lecture du code. Considérez ces 2 façons de parcourir un vecteur, sans et avec **auto**:

```
for(std::vector<std::string>::iterator it = std::begin(vect); it != std::end(vect); ++it )
        std::cout << *it << std::endl;
```

```
for(auto it = std::begin(vect); it != std::end(vect); ++ it)
        std::cout << *it << std::endl;
```

Boucle for simplifiée pour les intervalles

Il existe une façon encore plus simple de parcourir les tableaux et les conteneurs:

```
std::vector<std::string> vect;
//[...]
for(const auto & elem : vect)
        std::cout << elem;
}
```

nullptr

En C++, la macro `NULL` correspond à la valeur entière `0`.

La fonction foo offre une surcharge pour **int** et **void***,

```
void foo(int i) {
        std::count << i << "est un int";
}
void foo(void* ptr) {
        std::count << ptr << "est un pointeur";
}
```

Que pensez-vous qu'affiche `foo(NULL);` ? "0 est un int" ... sans doute pas le résultat que vous cherchiez à obtenir !

Pour éviter ce problème il est recommendé de toujours utiliser `nullptr` à la place de `NULL` ou de `0` lorsque vous voulez représenter un pointeur nul. ici, on écrirait : `foo(nullptr);`

Initialisation uniforme et liste d'initialisation

C++11 offre une nouvelle syntaxe pour initialiser les variables.

Les déclarations suivantes sont équivalentes :

```
int x{42};
int x = {42};
int x(42);
int x = 42;
```

Cette syntaxe peut être utilisée pour initialiser des instances de classes, même lorsque la classe n'a pas de constructeur:

```
struct Truc {
        int a;
        std::string b;
};
Truc t{42, "universe"}; // l'ordre doit respecter l'ordre de déclaration des variables membres de Truc
```

Cette syntaxe permet également de remplir des tableaux, des conteneurs, ainsi que d'autres types de la STL

```
const char* alphabet[] = {"alpha", "beta", "charlie", "delta", nullptr}; //déjà possible en C !
std::vector<int> numbers = { 4, 8, 15, 16, 23, 42};
std::map<std::string,std::string> secretIdentities = {
        {"Bruce Wayne", "Batman"},
        {"Matt Murdock", "Daredevil"}
};
```

Création d'alias de type

le mot clé **using** peut être utilisé pour créer des alias de types afin de simplifier la lecture du code, ou de mieux montrer votre intention

```
using Username = std::string;
using Dictionary = std::map<std::string,std::string>;

Username myUserName{"toto"};
```

Aller plus loin

C++11 et C++14 offrent bien d'autres fonctionnalités. Lambdas, mots clés `final` et `override` pour l'héritage de classes, templates varidiques, héritage de construction, etc...

Il vous appartient d'approfondir vos connaissances dans les technologies qui vous intéressent. A cette fin, vous pouvez consulter les sites et ouvrages suivants:

- `http://fr.cppreference.com/w/cpp/language`
- `http://fr.wikipedia.org/wiki/C++11`
- `https://isocpp.org/get-started` (en anglais)

Références

B. Stroustrup, *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, 4 ed., 5 2013.

S. Meyers, *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, 1 ed., 12 2014.

B. Stroustrup, *The Design and Evolution of C++*. Addison-Wesley Professional, 1 ed., 4 1994.