Rapport de MOOC
C++

Thomas Kowalski

Décembre 2018

1 Introduction

Le C++ est apparu pour la première fois en 1985.

Son principal attrait : son paradigme orienté objet et ses fonctionnalités modernes (sur lesquelles je reviendrai dans ce rapport), le tout dans un langage compilé permettant la gestion de mémoire bas-niveau (pas de *garbage collector*, allocation mémoire "manuelle" façon `malloc`...)

Ces traits en font, de nos jours, le langage de choix dans beaucoup d'applications critiques : des logiciels applicatifs de gestion d'entreprises aux sondes spatiales, en passant par des serveurs SQL, le C++ est très répandu dans le milieu professionnel.

Pour toutes ces raisons, et après une introduction au langage C en première année, et plusieurs applications de celui-ci pendant ma scolarité à l'ENSIIE (au cours des projets IPI, PIP, mais aussi dans certaines matières comme la l'OSS et la PRW, ou encore cette année l'ASS-COM et la MRO), j'ai été tenté d'apprendre le C++.

C'est pourquoi j'ai décidé de choisir le module LOA au S4. LOA demande cependant d'avoir suivi l'UE PAP au S3. Cette dernière est incompatible avec l'UE ASS-COM, que je souhaitais absolument faire (car elle me paraît importante pour comprendre comment fonctionnent *réellement* les langages et les logiciels). Mon suivi de l'UE MOOC s'explique donc grandement par ces raisons.

Dans tout le rapport, pour que les exemples que je donne soient plus concrets, je donnerais souvent des liens marqués Exécuter sur `repl.it`*. Il s'agit d'un site permettant de compiler et d'exécuter en ligne différents langages, dont le C++. En cliquant sur le lien, vous pourrez avoir une mise en application directe - et souvent plus développée - de mon exemple.*

2 Quelques points importants avant de commencer

Tirant ses fondements du langage C, le C++ reprend de nombreuses syntaxes caractéristiques de celui-ci.

```
int a;
float b;
int tab[5];

int func(int a, float b) { return a + b; }
```

Cependant, il apporte de nombreuses fonctionnalités nouvelles et améliorations, notamment le fait d'être orienté objet et d'offrir dans beaucoup de cas des syntaxes plus simples (et parfois plus claires) que le C.

On citera notamment :

Les namespaces Qui permettent de mieux "ranger" les différents objets (classes, fonctions, ...) du langage dans des espaces de noms : `std`, `Boost`... ;

De nouveaux mots-clés Comme `register` (pour indiquer au compilateur d'utiliser, si possible, un registre pour la variable), `inline` (pour indiquer au compilateur d'*inliner* la fonction si possible) qui permettent de mieux travailler son code sans pour autant devoir toucher à l'assembleur ;

Les références A mi-chemin entre variables et pointeurs, elles permettent de passer en argument à une fonction une variable sans la copier, et sans avoir la lourdeur d'un pointeur :
`void my_function(int a, int& b);`

Ceci étant dit, l'utilisation de toutes ces fonctionnalités n'est pas forcément aussi évidente que dans d'autres langages objet, c'est pourquoi je souhaite revenir sur quelques points. Cette liste est loin d'être exhaustive, mais elle montre quelques points caractéristiques.

2.1 Création de classe

Pour déclarer une classe, on utilise la syntaxe suivante :

```
class Ecole {
    // Par defaut, les membres sont "protected"
    // J'ai remarqué que beaucoup de programmeurs ajoutent un underscore à la fin des
noms des attributs de leurs classes
    std::string nom_;
    int annee_creation_;

    // En C++, le niveau d'accès des membres se fait grâce à des sections (et non pas à
des mots clés devant les membres, comme 'public', 'private'...)
```

```
        // Ici, on rentre dans une section privée. On y reste jusqu'à en changer
    manuellement
        private:
            void operation_dangereuse();

        // Section publique
        public:
            // Constructeur
            Ecole(std::string nom, const int annee_creation) :
                nom_(nom),
                annee_creation_(annee_creation)
                {}
    }
```

Il existe de nombreuses particularités liées à la syntaxe des classes, cependant beaucoup d'entre elles sont assez communes et simples. Je ne vais donc m'attarder que sur les caractéristiques ayant posé problème lors du suivi de ce MOOC.

2.2 Déclaration de fonctions *overridables*

Une fonctionnalité utile des langages objet est de pouvoir redéfinir une fonction dans une classe fille pour qu'elle agisse différemment de sa classe mère.

En Java, c'est d'ailleurs toujours le cas pour les fonctions `toString` et `equals` : le développeur réimplémente celles-ci pour remplacer les fonctions par défaut du *super-objet* `Object`, qui sont limitées et généralement inadaptées au besoin.

Obtenir le même comportement en C++ est tout à fait possible, cependant c'est un peu plus complexe.

Une méthode qui peut-être *overridée* par une classe fille doit être déclarée avec le mot-clé `virtual`. De plus, une méthode surchargée peut être marquée comme telle avec le mot-clé `override`, pour que le compilateur s'assure que

— La méthode existe dans la classe parente ;
— Cette dernière est virtuelle.

```
class Parent {
  public:
  virtual void print_virtual() const
    std::cout << "print_virtual: Appel depuis parent" << std::endl;
};

class Enfant : public Parent {
  public:
  void print_virtual() const
    std::cout << "print_virtual: Appel depuis enfant" << std::endl;
};

int main() {
  Parent *x = new Enfant();
  x->print_virtual();
}
```

Exécuter sur `repl.it`

2.3 Héritages *privés* et *protégés*

Le C++ propose plusieurs accessibilités lors de la dérivation d'une classe : `private`, `protected` et `public`. Je n'ai pas eu l'occasion de toutes les utiliser lors de ce MOOC, cependant, je connais leurs spécifications, qui sont intéressantes à savoir :

public La classe dérivée expose les mêmes attributs que la classe parente ;

protected Dans la classe dérivée, les attributs publics et protégés de la classe parente sont protégés ;

private Dans la classe dérivée, les attributs publics et protégés de la classe parente sont privés.

```
class Base {};
class Priv : private Base {};
class Prot : protected Base {};
class Pub  : public Base {};
```

3 Exercices sur LeetCode.com

M'intéressant également beaucoup à l'aspect algorithmique des programmes, et étant tenté par le C++ par ses performances supposées excellentes, j'ai été tenté de faire des exercices en ligne pour pouvoir comparer mes capacités en algorithmique avec celles d'autres personnes.

Ces exercices m'ont notamment permis de réaliser un fait intéressant : l'abstraction apportée par les classes apporte souvent, avec sa praticité, un encombrement certain et une perte de performances.

Résultat, entre un algorithme utilisant des algorithmes standard pour faire des calculs sur une chaîne de caractères et un algorithme agissant directement sur le tableau de caractères sous-jacent (obtenu avec `std::string::c_str`, le second est généralement bien meilleur.

En effet, et sur plusieurs exercices, la première implémentation permet d'atteindre le top 75% (à condition de s'y prendre correctement), la seconde permet généralement de se classer dans le top 3%.

Ici un exemple d'exercice parmi ceux que j'ai effectués sur `LeetCode`. Il s'agit de réimplémenter la fonction `strstr`, qui cherche la première occurrence d'une chaîne dans une autre et renvoie son indice (ou -1 en cas d'absence).

```
class Solution {
    public:
    /*         ||||| J'essaie autant que possible de faire attention au caractère constant
    /*         vvvvv de mes variables quand j'écris du code */
    int strStr(const string haystack, const string needle)
    {
        /* J'utilise intensivement auto, j'en reparle plus tard dans le rapport */
        auto needle_length = needle.length();
        if (needle_length == 0)
            return 0;

        auto haystack_length = haystack.length();
        if (haystack_length < needle_length)
            return -1;

        auto s = haystack.c_str();
        auto sub = needle.c_str();

        int j = 0;
        for (auto i = 0; i < haystack.length() - needle.length() + 1; i++)
        {
            if (haystack[i] == needle[0])
            {
                if (needle_length == 1)
                    return i;

                j = 1;
                while (haystack[i + j] == needle[j])
                {
                    if (j == needle.length() - 1)
                        return i;
                    j++;
                }
            }
        }

        return -1;
    }
};
```

Résultat : `Runtime: 4 ms, faster than 98.66% of C++ online submissions for Implement strStr().`

4 Lectures de blogs de professionnels

L'un de mes principaux objectifs, lors du suivi de ce MOOC, était de ne pas me limiter au contenu de celui-ci. Le C++ étant un langage en perpétuelle évolution, je trouvais intéressant de pouvoir lire les réflexions de professionnels l'utilisant dans leur métier afin de pouvoir m'en inspirer, que ce soit dans le cadre du MOOC ou plus tard dans ma vie.

Parmi ces blogs, on retrouve principalement *FluentCPP* (du français Jonathan Boccara) et `bfilipek.com` (du polonais Bartlomiej Filipek).

Étant donné le grand nombre d'articles lus, je ne restituerai pas le contenu en intégralité, j'ai cependant noté quelques points qui m'ont semblé intéressants et que je tente d'appliquer autant que possible.

4.1 La convention *auto-to-stick*

Sûrement l'une des lectures qui m'ont le plus marqué. En C++, la règle générale veut que l'on déclare les variables de la façon suivante :

```
// Déclaration
type variable; /* Constructeur par défaut */

// Déclaration et affectation
type variable = ...; /* Appel du constructeur de copie ou de déplacement s'il existe */

// Déclaration et construction
type variable(parametre1, parametre2, ...); /* Appel explicite du constructeur */
```

La convention *auto-to-stick*, s'accorde avec l'idée générale que, de plus en plus, on tend à utiliser le mot-clé `auto` plutôt qu'à donner explicitement le type des variables, notamment lors de l'appel à une fonction au type de retour compliqué :

```
// Version 1 (verbeuse)
std::pair<std::optional<int>, std::optional<int>> resultat = ma_fonction();

// Version 2 (plus simple, compatible avec les changements de type de ma_fonction)
auto resultat = ma_fonction();
```

Cette tendance pousse à écrire de plus en plus de lignes dont le type va être à droite (plutôt qu'historiquement à gauche). Pour conserver cette convention, on peut donc écrire :

```
// nom à gauche type à droite
//   vvvvvvvvv   vvvvvvvvvvv
auto ma_string = std::string("Thomas");
```

Bien qu'elle soit à première vue moins évidente, le résultat est dans les faits le même. De plus, l'utilisation des *move semantics* de plus en plus répandue permet d'avoir une efficacité égale à la convention historique.

Cette convention s'applique également aux fonctions :

```
auto main(const int argc, const char** argv) -> int
{
    return 0;
}
```

Est tout à fait correct est C++17. En C++20, on peut même faire

```
auto main()
{
    return 0; /* La fonction renvoie un int */
}
```

Les compilateurs compatibles (malheureusement pas encore très répandus) feront automatiquement l'inférence de type et comprendront que le type de retour de `main` est `int`.

Évidemment, une tentative du type

```
auto fonction()
{
    if (test())
```

```
        return 0; /* La fonction renvoie un int */
    else
        return "Thomas"; /* La fonction renvoie un const char* */
}
```

Mènera à une erreur de compilation.

4.2 L'utilisation de `std::optional`

Le C++17 ajoute une fonctionnalité intéressante, que j'ai déjà eu l'occasion d'utiliser beaucoup en IPF en langage OCaml. Elle permet, entre autres, de répondre à une question simple - qui est d'ailleurs souvent source d'interrogations dans certains algorithmes standard : que renvoyer si la fonction échoue ?

On peut penser à beaucoup de méthodes :
— Passer un pointeur vers la sortie et renvoyer un code d'erreur ou 0 ;
— Renvoyer une valeur spécifiée si la fonction échoue - mais quid de cette valeur ? Quel entier renvoyer en cas d'échec de conversion d'une chaîne en entier ? ;
— Lever une exception - mais l'utilisation des exceptions semble un peu découragée en C++, pour des raisons de performances.

`std::optional` répond à cette problématique grâce à une classe `optional<T>`. Le fonctionnement classique est d'avoir une fonction qui renvoie un `std::optional`, d'évaluer son retour en tant que booléen (`if (retour)`), et s'il est évalué à `true`, alors déréférencer le `std::optional` (`auto vrai_resultat = *result;` ; l'opérateur de déréférencement personnalisé est implémenté).

```
std::optional valopt = container.get(key);
if (valopt)
    // value found at key
    auto val = *valopt;
else
    // value not found
```

Ce fonctionnement a plusieurs avantages :
— Il est très performant ;
— Il permet de gérer les erreurs simples simplement ;
— Il est pratique à utiliser.
`repl.it` n'étant pas compatible C++17, je donne un exemple ici.

```
std::optional<int> to_int(const std::string s)
{
    const auto result = atoi(s.c_str());
    if (!result)
            return {};
    else
            return std::optional<int>(result);
}

int main()
{
    const auto test = to_int("1234");
    if (test)
            std::cout << "i = " << *test << std::endl;
    else
            std::cout << "Conversion impossible ou i = 0" << std::endl;
}
```

4.3 *Template Metaprogramming* (TMP)

Le *template meta-programming* (calcul de valeurs et de résultats constants à la compilation grâce aux *templates*) est une pratique encore peu fréquente mais qui a tendance à se généraliser. Se rapprochant beaucoup de la programmation fonctionnelle, elle utilise les *templates* pour spécifier des valeurs et des constantes, qui se basent sur ce qui est à constant à la compilation en C(++) : les types. Et en particulier, les types `struct`.

```
template<int n>
struct Int {
  /* const permet d'assurer au compilateur que la structure est constante */
  /* static permet de lui assurer qu'elle ne dépend pas de l'instance de s<int n> (et
donc qu'elle est constante à la compilation) */
  static const int val = n;
};
```

Cette expression est totalement déterminée à la compilation (à condition, évidemment, que `n` soit, lui aussi, déterminé à la compilation).

On peut alors complexifier la chose, tant que l'on travaille avec des *constexpr* (expressions constantes à la compilation).

```
template<typename a, typename b>
struct Produit {
    static const int val = a::val * b::val;
};
```

On peut alors calculer le produit de deux entiers à la compilation.

```
typedef Int<3> trois;
typedef Int<4> quatre;
typedef Produit<trois, quatre> produit;

std::cout << produit::val << std::endl;
```

Évidemment, le cas posé ici est relativement inutile. L'exemple (un peu) plus utile, souvent donné lors de l'introduction au TMP est le calcul d'une valeur de factorielle à la compilation.

```
template<unsigned int n>
struct fac {
  static const unsigned int val = n * fac<n - 1>::val;
};

template<>
struct fac<0> {
  static const unsigned int val = 1;
};

int main() {
  const int n = 10;
  std::cout << n << "! = " << fac<n>::val << std::endl;
}
```

Exécuter sur `repl.it`

On peut cependant aller plus loin, avec le dernier exemple que je présenterai : le calcul de π à la compilation.

```
/* Série de Riemann des 1/n^2 jusqu'à n */
template<unsigned long n>
struct riem {
  static const double val;
};
/* Initialisastion de la valeur pour n général, à savoir 1 / n^2 + riem<n-1> */
template<unsigned long n>
const double riem<n>::val = riem<n-1>::val + 1 / (double)n / (double)n;

/* Cas de base pour la série de Riemann, à savoir n = 1 */
template<>
struct riem<1> {
  static const double val;
};
/* Initialisation de la valeur du cas de base */
const double riem<1>::val = 1;

/* Structure "pi", qui renferme le résultat du calcul */
template<unsigned long n>
struct pi {
  static const double val;
};

/* Initialisation de la veur de pi pour un certain n */
template<unsigned long n>
const double pi<n>::val = sqrt(6 * riem<n>::val);
```

Le résultat est bien celui attendu (à la précision due aux 899 itérations et à celle des `double` près) :

```
Pi = 3.14053
```

Exécuter sur `repl.it`

Conclusion Le TMP est une technique qui me semble très puissante. J'ai notamment commencé à implémenter les listes chaînées (comme en OCaml), cependant j'ai dû faire face au problème principal de cette technique : le temps. En effet, le TMP n'étant pas une réelle *feature* du langage (en tout cas pas pour le moment), les options de débogage sont très réduites et la syntaxe rendant le tout peu lisible n'aide pas. J'ai cependant aimé faire ce premier pas dans ce monde.

4.4 *Lambda-expressions*

Une fonctionnalité de plus en plus répandue dans les langages modernes est les *lambda expressions*. Plus ou moins versatiles, elles permettent de déclarer des fonctions dans d'autres fonctions, et de manière anonyme (ou pas).

L'utilisation des *lambdas* a beaucoup d'avantages :

— Un code plus lisible ;
— Moins de déclaration de fonctions, donc un code moins long et éparpillé ;
— Éviter l'utilisation des "foncteurs" (classes implémentant l'opérateur `()`), qui peuvent rapidement rendre le code inutilement complexe.

La syntaxe de déclaration des *lambdas* est similaire à celle des fonctions, avec un point en plus : la capture de l'environnement. Lorsque l'on souhaite utiliser une variable déclarée hors de la *lambda*, on doit la "capturer". Si on tente de déclarer cette fonction :

```
auto a = 5;
auto equals_a = [](int x) { return x == a; };
```

On obtient le message suivant à la compilation :

```
main.cpp: In lambda function:
main.cpp:24:43: error: 'a' is not captured
   auto equals_a = [](int x) { return x == a; };
```

La solution est alors cette syntaxe :

```
auto equals_a = [a](int x) { return x == a; };
```

`a` est alors "capturée" et utilisable dans la fonction.

Point important : lorsque l'on capture une variable, c'est sa valeur qui est capturée, et non pas son adresse. Par exemple, le code suivant :

```
auto a = 5;
auto equals_a = [a](int x) { return x == a; };
a = 4;
std::cout << equals_a(4) << std::endl;
```

Affiche 0. Si l'on souhaite accéder à la valeur de `a` et si elle est modifiée plus loin dans le programme, on doit obligatoirement utiliser un pointeur.

```
int a = 5;
int* c = &a;
auto equals_a_ptr = [c](int x) { return x == *c; };
a = 4;
int x = 4;
std::cout << "By pointer: " << x << " == a : " << equals_a_ptr(4) << std::endl;
```

Exécuter sur `repl.it`

5 Quelques points intéressants du cours

En lisant le cours, quelques points m'ont semblé intéressants (et pas évidents).

5.1 Création de *callbacks*

Il est commun (notamment lors de la création d'interfaces graphiques) de vouloir exécuter une certaine fonction lorsque quelque chose se passe. On appelle ces fonctions des *callbacks*.

Le concept de *callback* ne date pas du C++ (il est tout à fait utilisable en C), mais le C++ facilite grandement son utilisation, et de deux façons :

— D'une part, l'utilisation des *templates* permet de ne pas se soucier du type des *callbacks* tant que l'on sait qu'ils sont compatibles (par exemple, impossible d'utiliser un *callback* renvoyant un `double` dans une fonction où l'on veut obtenir un `int`) ;

— D'autre part, si l'on souhaite manipuler les *callbacks* dans le code (ou même les déclarer à l'aide de *lambda*), on peut le faire grâce au mot clé `auto`.

```
int main(void)
{
    // Fonctionne tres bien si fonction_callback et autre_fonction_callback ont le meme
type
    auto ma_fonction = fonction_callback;
    if (selectionner_autre_fonction)
        ma_fonction = [] (const int x) { return 2 * x;} ;

    // Applique ma_fonction à chaque element de c
    std::for_each(c.begin(), c.end(), ma_fonction);

    // ...
}
```

Exécuter sur `repl.it`

6 Conversion vers un type dérivé

Il m'est déjà arrivé par le passé d'avoir besoin d'une collection d'objets (disons de `Vehicule`) et d'avoir besoin de traiter chaque objet en fonction de son type. Cette opération, simple en Java, l'est beaucoup moins en C++. J'ai appris comment la gérer.

L'idée est la suivante : lorsque l'on déclare un objet comme étant de type `Vehicule`, le compilateur le considère comme prenant la mémoire (et ayant accès à la mémoire) d'un objet `Vehicule`. Si, en réalité, notre `Vehicule` est un `Avion`, alors la zone spécifique au type `Avion` qui a été allouée sera inutilisée.

Obtenir les résultats que l'on souhaite, dans ce cas, demande l'utilisation de pointeurs. En effet, on ne perd alors pas d'information sur le type :

```
// C'est un pointeur Vehicule mais qui pointe vers un Avion
Vehicule* v = make_avion("Cesna");
// Appelle description de avion (sauf si elle n'est pas virtual dans Vehicule)
v->description();
// Ne compile pas, car v n'a pas de méthode "voler"
v->voler();
// Fonctionne, car v est de type avion
dynamic_cast<Avion*>(v)->voler();
// Core dump, car, dynamic_cast renvoie un pointeur nul
dynamic_cast<Avion*>(new Voiture("Peugot"))->voler();
```

Exécuter sur `repl.it`

6.1 Le *design pattern Iterator*

L'*Iterator* est un patron de conception très répandu en programmation orientée objet. Il permet d'itérer simplement et avec une syntaxe standard sur les collections (tableaux, listes, *maps*...) Ses implémentations et son fonctionnement diffèrent, cependant, en fonction du langage.

En C++, l'*idiom* générale est la suivante :

```
std::vector<int> v = {1, 2, 3, 4, 5};
for(auto it = v.begin(); it != v.end(); it++)
  std::cout << *it << std::endl;
```

En effet, les méthodes `begin` et `end` renvoient toutes deux un *iterator*, respectivement pointant sur le début du vecteur et sur la "juste après fin" de celui-ci.

La grande puissance de l'*iterator* est sa généricité : quelle que soit la structure de donnée intrinsèque au *container*, on gère l'itération de la même façon, avec la même syntaxe `it != end, it++, *it`.

La déclaration d'un *iterator* en C++ est un peu particulière. En effet, il existe plusieurs types d'*iterators* :

Input Iterator qui permet d'obtenir des données depuis le *container* ;

Output Iterator qui permet d'envoyer des données dans le *container* ;

Forward Iterator qui ne permet d'aller que "vers l'avant" dans le *container* (et qui garantir l'ordre de parcours) ;

Bidirectional Iterator qui permet d'aller dans les deux sens dans le *container* ;

Random Access Iterator qui permete d'aller n'importe où dans le *container* (il implémente `operator+=` et `operator-=` en plus des `operator++` et `operator-`.

L'autre principal intérêt (et objectif) de l'*iterator* est d'être compatible avec l'ensemble des algorithmes de la STL, dès lors qu'il est du bon type (*input*, *output*...).

Par exemple, cela signifie que l'on peut utiliser `std::transform` avec une structure de données personnalisées, dès lors qu'elle fournit les *iterators* nécessaires (exemple disponible sur `repl.it`).

Pour créer son propre *iterator*, et depuis C++17, on doit simplement indiquer le type d'itérateur. Comme tout doit être connu à la compilation, on indique ces informations dans la définition de la classe, à l'aide d'attributs aux noms donnés.

```
template<typename Value>
/* ... */
class LinkedListIterator
{
  public:
    // Permet de définir l'iterator comme étant un Input Iterator
    using iterator_category = std::input_iterator_tag;
    // Définit le type de données que renvoie l'iterator lorsqu'il est déréférencé
    using value_type = Value;
    // De mÃªme pour le type pointeur
    using pointer = Value *;
    // Et pour les références
    using reference = Value *;
}
```

Une fois ces valeurs indiquées, le compilateur (et les éventuels algorithmes qu'on appelle dans le programme) peuvent s'assurer à la compilation que le type d'*iterator* fourni est bien le bon et que les algorithmes pourront effectuer les modifications qu'ils veulent comme ils le veulent sur les *containers*.

Un exemple très complet est disponible sur `repl.it` : Exécuter sur `repl.it`

6.2 Le patron de conception *Singleton*

Le *Singleton* est un patron de conception très utilisé en programmation orienté objet. En effet, il résout un problème souvent rencontré : celui d'une classe ne devant avoir qu'une instance.

```
class DBConnection
{
  private:
    static DBConnection* instance;
    DBConnection()
    {
      std::cout << "Building DBConnection" << std::endl;
    }

  public:
    static DBConnection* get_instance()
    {
      if (!instance)
        instance = new DBConnection();

      return instance;
    }

    void execute(std::string sql)
    {
      std::cout << "Executing '" << sql << "'" << std::endl;
    }
};
DBConnection *DBConnection::instance = 0;

int main() {
  // Ne fonctionne pas (erreur à la compilation car constructeur privé)
  // DBConnection connection;

  // Calls the constructor (because it's the first time)
  DBConnection* connection = DBConnection::get_instance();
  connection->execute("SELECT * FROM Users");

  // Doesn't call the constructor a second time
  DBConnection* connection2 = DBConnection::get_instance();
  connection2->execute("SELECT Email FROM Users WHERE Username = 'Thomas'");
}
```

6.3 Autres

Le cours revenait sur d'autres points intéressants en programmation objet, notamment certains *design patterns* tels que l'*Adapter* ou encore le *Command*.

Je ne les détaillerais pas plus, dans la mesure où ils sont déjà au programme d'ILO en première année et puisque des exemples d'implémentations foisonnent sur Internet pour chacun.

7 Le projet : rendu 3D avec *raytracing*

7.1 Introduction

L'objectif de ce projet (inspiré grandement du projet no 4 de l'UE PAP) est de créer en C++ un programme capable d'effectuer un rendu 3D d'une scène en utilisant la méthode du tracé de rayons (en utilisant `libpng` pour la sortie).

Celle-ci se base sur les lois de l'optique et permet d'obtenir des rendus extrêmement fidèles à la réalité (à condition bien sûr de prendre en compte toutes les lois, à savoir réflexion, réfraction...) mais est en pratique extrêmement lente et n'a jamais jusqu'ici été utilisée pour du rendu en temps réel (jeux-vidéos...).

7.2 Présentation du modèle

Pour la première version de ce projet (et probablement pour ce que j'aurai le temps de faire avant le rendu), je me suis basé sur un modèle simple :

— Uniquement des sphères aux couleurs personnalisables ;
— De la réflexion possible sur les sphères ;
— Autant de sources lumineuses que l'on souhaite (chaque sphère peut émettre de la lumière) ;
— Une caméra placée où l'on veut mais avec un angle défini (bien que cela fasse partie de mes projets d'évolution pour la présentation orale).

7.3 Présentation de la solution

Pour effectuer le rendu, j'ai mis en place diverses classes :

Vector3f Un vecteur 3D avec des coordonnées flottantes, c'est également la structure de données utilisée pour stocker les couleurs ;

Ray3f Un rayon 3D, à savoir une origine (vecteur) et une direction (vecteur unitaire) ;

Sphere Une sphère, avec un rayon et une origine, et des données visuelles : couleur, lumière émise, taux de réflexion...

Scene Une scène, avec un ensemble de `Sphere`s et un emplacement pour la caméra.

7.4 Algorithme de rendu

L'algorithme de rendu est relativement simple et se base sur le fait que les chemins suivis par les rayons lumineux est le même quel que soit le sens dans lequel vont les rayons (comme vu en CPGE : Principe de Fermat / Principe du retour inverse de la lumière).

Émettre une (pseudo-) infinité de chaque sphère émettrice de lumière prendrait énormément de temps, pour un gaspillage de calculs non négligeable : seule une petite partie des rayons se retrouve à rentrer dans le "capteur" de notre caméra.

Grâce au principe de Fermat, nous pouvons "tirer" un rayon pour chaque pixel de l'image rendue directement depuis la caméra, plutôt que de tirer un grand nombre de rayons depuis la source lumineuse en espérant qu'ils arrivent sur la caméra, sans pour autant perdre en information.

Une fois un rayon tiré, on calcule d'abord l'objet le plus proche sur le chemin du rayon (s'il y en a un). Cette donnée nous permettra de connaître la couleur "de base" (à savoir la couleur de surface de la sphère).

Alors, on calcule le rayon entre ce point et chaque source de lumière afin de savoir si un objet se trouve entre l'objet et la source. Le cas échéant, le point est à l'ombre pour cette source ; autrement, le point est éclairé.

Lorsqu'on trouve un point d'intersection entre rayon et objet, on peut appliquer d'autres lois pour avoir un rendu plus réaliste : la réflexion (si le milieu est réfléchissant), la réfraction (si le milieu transmet le rayon). Cependant, le support des sphères translucides n'est pas à l'ordre de ce projet pour le moment.

7.5 Philosophie générale de l'implémentation

L'objectif de cette implémentation est avant tout de mettre en application ma formation en C++. Cependant, cela n'est pas toujours compatible avec l'idée de faire un code aussi performant que possible : on l'a vu notamment avec les exercices de LeetCode.

J'ai donc dû faire un choix, et ce fut de rester autant que faire se peut dans une optique orientée objet. Je n'utiliserai donc pas de tableaux mais plutôt des *std : :vector*. De même, plutôt que, pour une fonction renvoyant deux valeurs optionnelles, renvoyer un booléen et passer deux pointeurs en argument, j'utiliserai un type complexe *std : :optional<std : :pair<float, float»*. L'utilisation de celui-ci aura, au moins en version non optimisée du programme compilé, une incidence nette sur les performances à cause des différentes indirections

ajoutées, cependant il permettra aussi et avant tout de conserver un code non seulement lisible, mais aussi représentatif des *idioms* C++ courants (et de ne pas faire du C déguisé en C++).

Il est également conseillé (bien que ça ne soit pas propre au C++) d'être toujours aussi propre que possible dans la déclaration des fonctions, et d'utiliser notamment `const` autant que possible, ce que j'ai fait. J'ai aussi remplacé partout où c'était possible les appels par valeur par des appels par référence (remplacement inoffensifs si les classes sont bien écrites et que les paramètres sont marqués avec `const`, justement), ce qui permet d'économiser de nombreuses copies mémoire.

Enfin, après avoir bien avancé dans le projet, et avec toujours ces idées en tête, j'ai eu l'occasion de tester le rendu en activant l'option *-O3* de G++. Celle-ci permet d'activer le maximum d'optimisations dans le code et améliore grandement la performance de l'algorithme. Finalement, le temps de calcul total peut être largement satisfaisant (moins de trois secondes pour l'image d'exemple donnée plus bas).

7.6 Choix de l'implémentation

En plus des choix discutés ci-dessus, mon implémentation a quelques particularités :

— Elle se limite au rendu de boules (car c'est une forme "parfaite" et facile à implémenter) ;
— Elle ne permet pas la définition d'une source lumineuse unique mais utilise plutôt les sphères comme source, permettant ainsi à l'utilisateur d'avoir autant de sources qu'il le souhaite ;
— Elle ne permet pas la définition d'un sol, car la définition d'un sol reste arbitraire : je préfère laisser à l'utilisateur le choix de mettre une boule de taille assez grande pour donner l'impression que c'est localement un plan ;
— Elle ne permet pas l'utilisation de boules transparentes (tout simplement parce que je n'ai pas eu le temps de me pencher sur la question) et ne gère donc que le tracé d'ombres et la réflexion.

7.7 Difficultés rencontrées

7.7.1 Utilisation de `libpng`

Le sujet original de monsieur Torri interdisait l'utilisation de toute bibliothèque externe, à part `libpng`. Si je n'ai pas souhaité m'imposer cette restriction, je n'ai pas pour autant eu besoin d'utiliser quoi que ce soit d'autre. L'utilisation de `libpng`, pour l'exportation, m'a cependant posé problème.

Je n'avais jusqu'alors dans ma vie jamais utilisé de "vraie" bibliothèque C, et force fut de constater que ce n'était pas évident. J'ai lu la documentation (seule l'écriture de fichiers PNG m'intéressait, cependant une lecture des principes généraux du format PNG a été nécessaire pour pouvoir saisir la partie écriture) et ai fini par écrire un code fonctionnel pour écrire une matrice de pixels.

Je me suis ensuite heurté à un problème d'encodage : mon format de données (des tableaux de tableaux de *char*) n'était pas celui attendu par `libpng`, car le *bit depth* que j'utilisais n'était pas celui que j'avais prévu. En résultait une image inattendue, avec deux fois l'image attendue, mais pas les bonnes couleurs ni le bon format.

Et si la résolution d'un bug est rapide quand on connaît parfaitement les outils avec lesquels l'on travaille, le faire en étant débutant est tout de suite plus difficile.

J'ai finalement réussi à résoudre le problème et à obtenir le rendu attendu.

FIGURE 1 – Un exemple d'image incorrecte

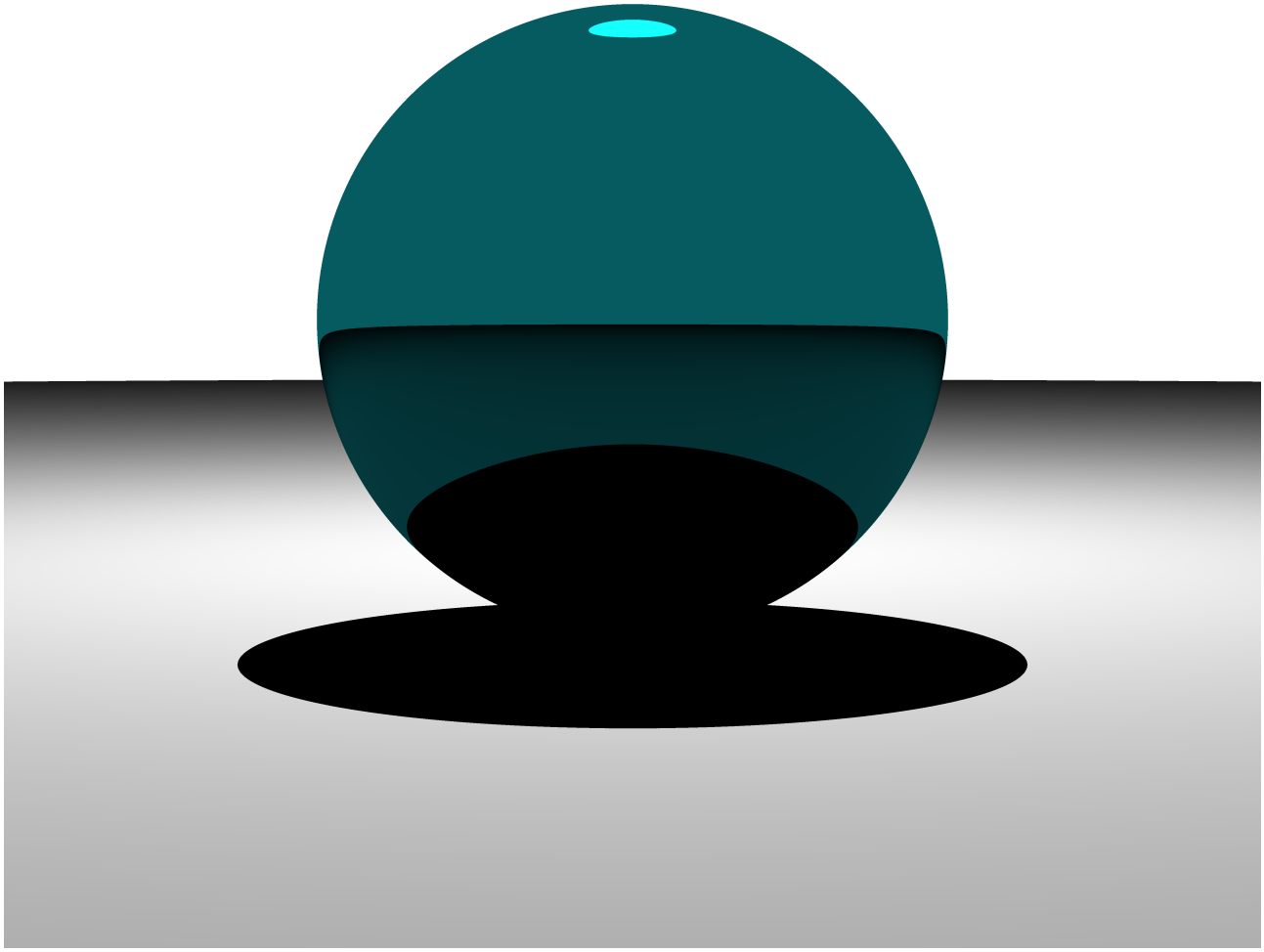

FIGURE 2 – Exemple de rendu simple

7.7.2 Implémentation du *raytracing*

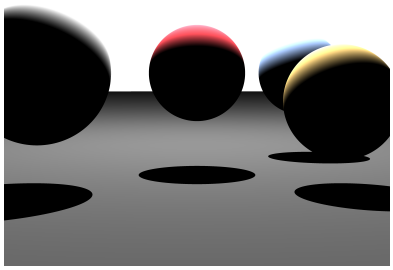

FIGURE 3 – Rendu sans réflexion

Bien que le concept de *raytracing* soit simple à comprendre et à imaginer, et bien que j'aie eu comme tout le monde, les acquis nécessaires en physique en classes préparatoires, créer l'algorithme de rendu a été une réelle difficulté. La raison à cela ? La modélisation mathématique.

Je n'avais jamais auparavant dans ma vie programmé ce genre de choses et j'ai été assez surpris de voir la complexité des algorithmes à implémenter, et la difficulté de leur débogage, due au manque de transparence d'une formule mathématique, une fois transposée dans un code informatique.

C'est honnêtement un de mes plus grands regrets concernant ce projet. Je trouvais le sujet extrêmement intéressant, et bien qu'il l'ait été, j'en retiens plus de longues heures de frustration et de progression à tâtons qu'une réelle satisfaction comme lorsque j'ai programmé mon projet d'ordonnancement (voir section suivante), qui avait un rendu certes moins impressionnant, mais une programmation beaucoup moins compliquée et des résultats tout aussi satisfaisants.

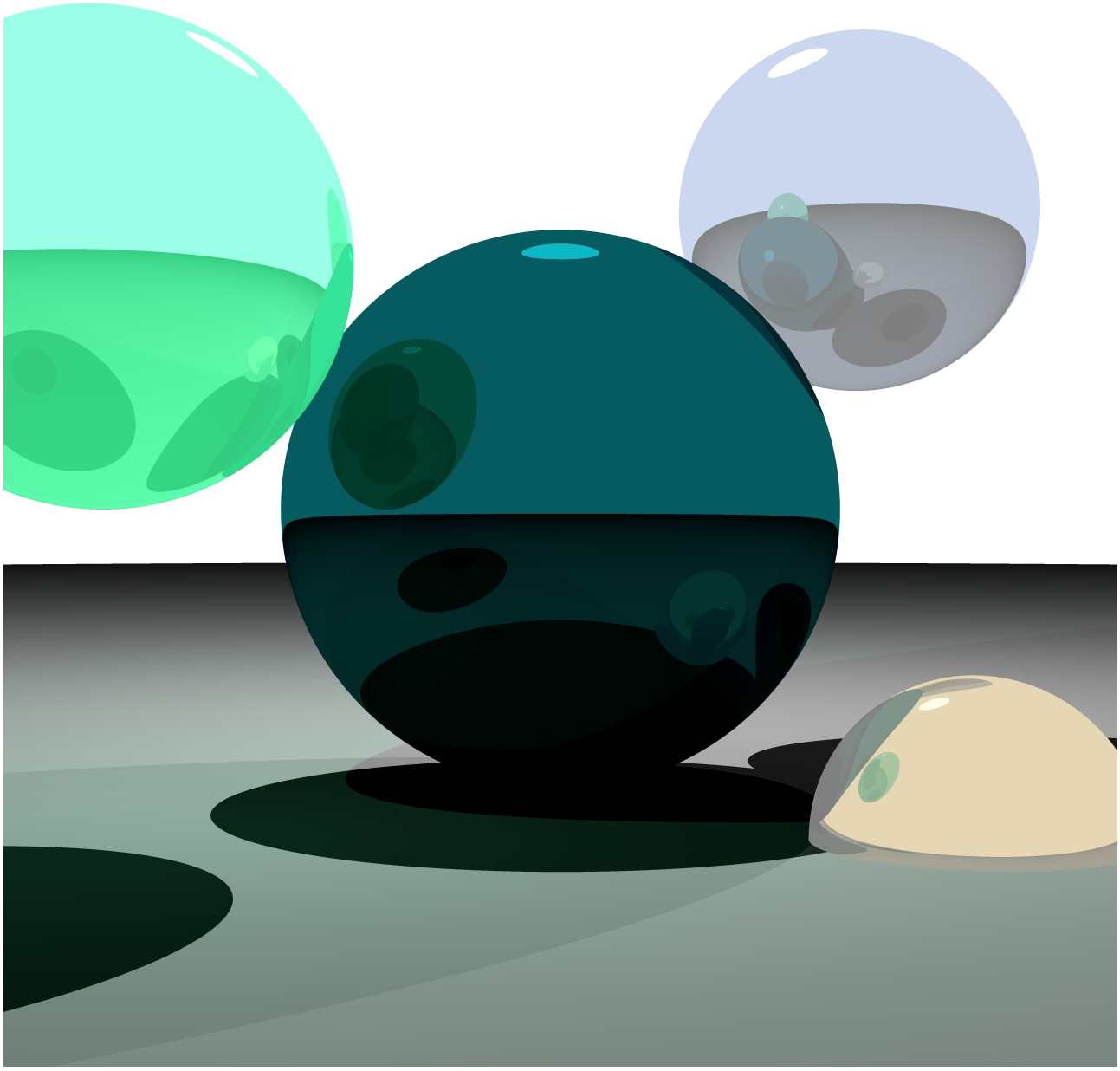

FIGURE 4 – Exemple de rendu plus complexe avec plusieurs sphères, dont une qui émet de la lumière (à gauche)

8 Résolution d'un problème d'ordonnancement

Lors de ma deuxième année de classes préparatoires, j'ai eu l'occasion de travailler, lors de mon TIPE, sur un problème d'ordonnancement. Celui-ci, une version particulière du problème *open-shop scheduling*, m'a fortement intéressé et mon année d'étude m'a permis d'arriver à un générateur de solutions efficace et de fonctionnement (relativement) simple.

Celui-ci tire ses origines de mon objectif initial, à savoir appliquer l'algorithme du recuit simulé, et du fait que j'aie conçu un algorithme génétique au cours de l'année. Séparément, un problème de complexité empêchait mon implémentation du recuit simulé d'être efficace (en plus d'être particulièrement compliquée dans son implémentation) ; de l'autre côté, les résultats de l'algorithme génétique étaient clairement en-deçà des résultats attendus. C'est pourquoi j'ai eu l'idée de rassembler les deux dans un algorithme qui utilise des chromosomes pour représenter la solution et utilise l'algorithme de Metropolis pour l'optimiser.

Problème Soient n élèves et p examinateurs. On souhaite que chaque élève passe un oral avec chaque examinateur. Trouver une solution en minimisant le temps total (*makespan*).

Le modèle utilisé dans ma solution est simple. On a d'un côté un chromosome de p éléments représentant l'ordre dans lequel les oraux des jurys sont assignés. De l'autre, on a p chromosomes dont les éléments donnent l'ordre d'assignation de chaque élève avec le jury p.

Pour assigner chaque élève à un jury, on utilise un algorithme glouton simple.

Pour mon TIPE, j'avais implémenté et testé l'algorithme en Python. Je souhaitais le réimplémenter en C++, pour plusieurs raisons, l'une d'elles étant un espoir de meilleures performances, et une comparaison possible des deux langages en termes de temps de calcul.

La sortie est la suivante : elle se compose du temps d'optimisation (166ms) et du *makespan* de sortie (29 unités de temps).

Les quelques dernières lignes permettent de connaître l'emploi du temps (chaque élève a un numéro, et chaque ligne représente la *timeline* d'un jury).

```
Optimization time: 166
Serialization time: 1
29
2 1 0 3
1 3 2 0
2 0 1 3
1 2 3 0
1 3 0 2
---333-000222111
33220011
000111222333
-------------3333222211110000
-------------00000333332222211111
```

Les résultats en temps de calcul sont cependant... surprenants. Alors que la version Python ne prend que 4s pour 100 exécutions (génération de solution et optimisation), ma version C++ en prend... près de 20. En activant toutes les optimisations de G++ (avec `-O3`), le temps d'exécution passe à 3.5s, donc un peu mieux que la version Python.

Je dois reconnaître un certain désarroi à la vue de ces résultats. De plus, j'ai commencé ce projet au début du cours de C++ et je n'avais que peu d'expérience dans le domaine (moins que maintenant, en tout cas), c'est pourquoi je n'ai pas plus cherché que ça à améliorer les résultats, mais plutôt à continuer de suivre le cours.

8.1 Difficultés rencontrées

Lors de la réalisation de ce projet, plus tôt dans le suivi du MOOC, j'ai rencontré quelques soucis auxquels je ne m'attendais pas forcément, mais principalement un qui m'a pris du temps à diagnostiquer.

8.1.1 Core dump sur une fonctionnalité de la STL

Dans une des premières versions de l'algorithme, j'utilisais pour générer des nombres aléatoires un Mersenne Twister Engine :

```
// La distribution dans laquelle on veut nos résultats
auto distribution = std::uniform_int_distribution<int>(1,6);
// Notre seed
std::mt19937::result_type seed = time(0);
// Générateur
auto engine = std::mt19937(seed);
// Foncteur dice_rand
auto dice_rand = std::bind(distribution, engine);
```

Cependant, lors de mes tests, mon programme provoquait constamment des *core dumps*. Trouver l'origine de ceux-ci m'a été assez long - ne soupçonnant pas un bug dans la STL - même avec ValGrind, et quand j'ai finalement testé le programme en utilisant la fonction C `rand` :

```
return min + (rand() % (max - min));
```

Tous les problèmes ont cessé. Bien que les valeurs obtenues ainsi ne soient pas *vraiment* aléatoires avec cette technique, c'est finalement celle que j'ai conservée dans la version finale.

9 Conclusion

Je considère le suivi de ce MOOC comme un réelle chance. En effet, bien que j'aie eu par le passé différentes occasions de découvrir le C++ (que ce soit en suivant des cours de mon côté, ou avec l'ILO en première année), je n'avais jamais réussi à aller plus loin que les bases, y compris l'année dernière où on nous avait rapidement présenté la syntaxe et les concepts, mais sans pour autant réellement s'y plonger.

Suivre ce MOOC m'a permis d'énormément m'améliorer dans ce langage dont je n'ai pourtant vu que la pointe de l'Iceberg. Grâce à la lecture indépendante des blogs dédiés au langage, j'ai également pu me familiariser avec le "C++ de la vraie vie" (voir en pratique comment imiter des fonctionnalités des dernières versions quand on ne les a pas encore, en apprendre plus sur des cas concrets d'utilisation de certains algorithmes standard, connaître quelques *dos* et *donts* - par exemple l'erreur commune d'utiliser `std::transform` pour appeler une fonction sur chaque élément d'un *container* alors que `std::for_each` est moins connue mais beaucoup plus adaptée...

Les deux projets que j'ai réalisés m'ont également été d'une grande aide dans ma formation. Bien que je regrette finalement un peu l'ampleur du projet de *raytracing* qui m'a pris beaucoup de temps pendant lequel j'aurais parfois préféré me documenter en lisant d'autres articles, et la difficulté générale du problème (qui était avant tout une difficulté dans la modélisation), je suis extrêmement satisfait des résultats.

Je me sens désormais totalement prêt pour entamer le semestre 4 et son UE "Langages Objets Avancé".

10 Progression

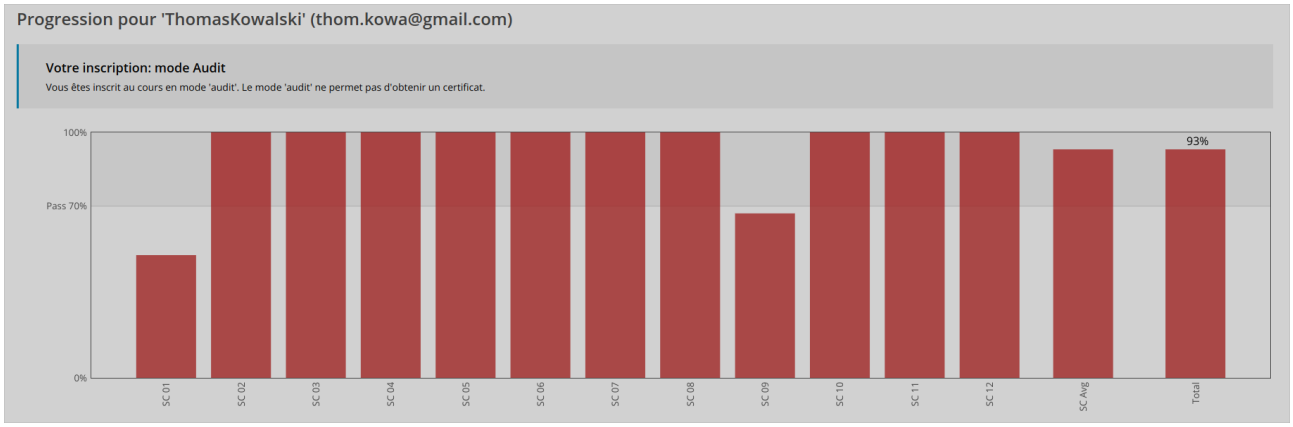

FIGURE 5 – Cours Débutant

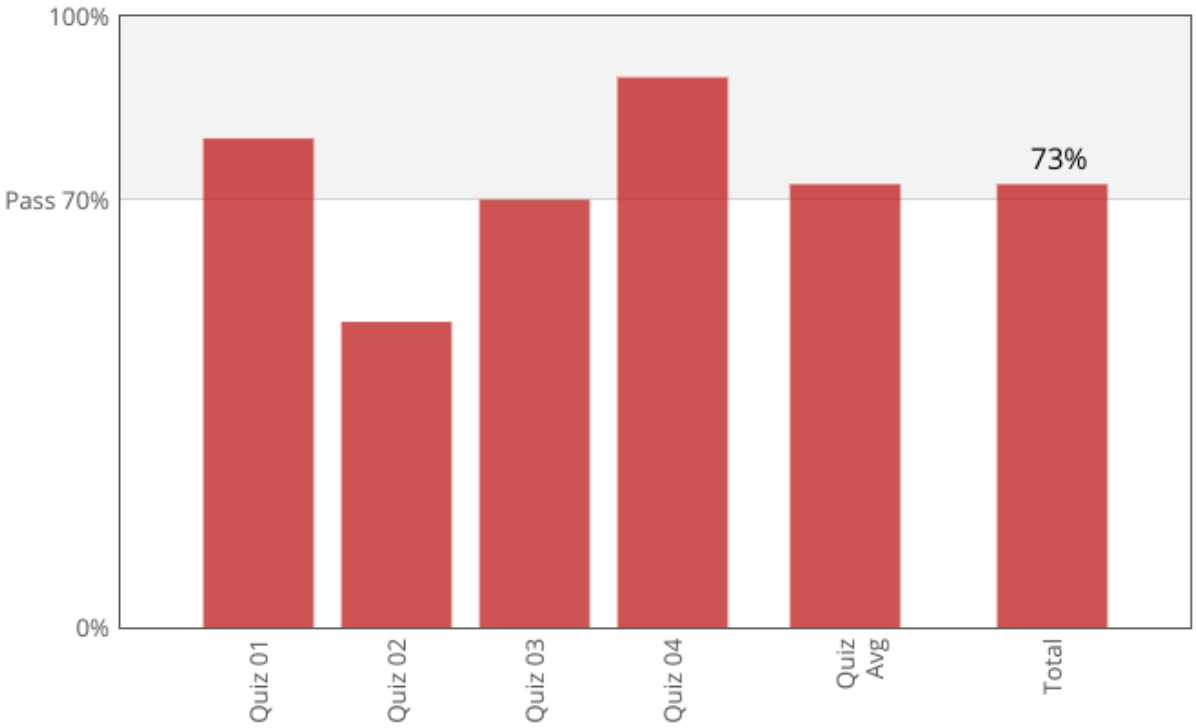

FIGURE 6 – Cours Avancé

Comme je l'ai précisé dans le mail, il y a un bug sur edX qui m'empêche d'accéder à un graphe pour mes résultats du deuxième cours, je vous transmets donc des captures d'écran des différentes pages.

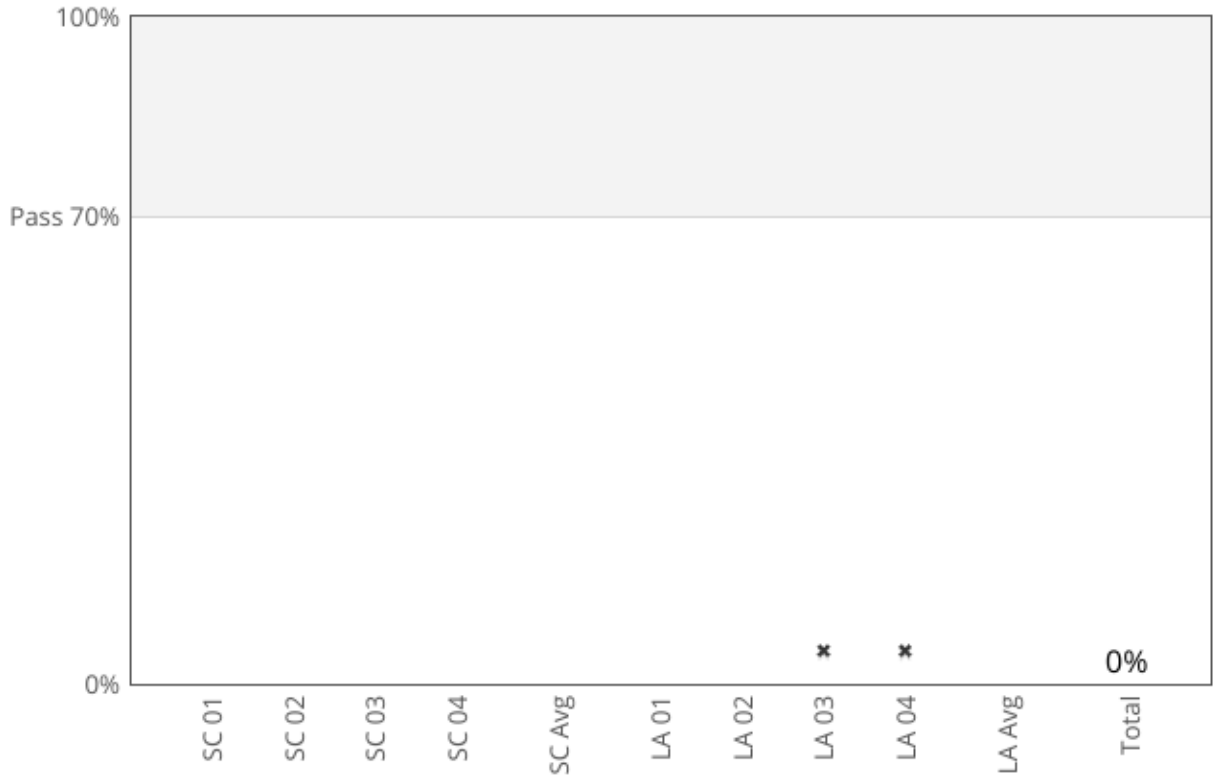

FIGURE 7 – Cours Intermédiaire

Et les captures d'écran (qui comptabilisent un total de 26 points sur 30) :

Self-Check Questions

Ajouter cette page aux favoris

Operators

1/1 point (graded)

Enter the operator for dereference and pointer use.

*

Soumettre Vous avez utilisé 1 essai sur 1

Afficher La Réponse

Reference Types

1/1 point (graded)

Which of the following is a valid way to create an alias for a variable in C++? Note that data types are excluded for simplicity.

- &refVariable = variable
- *refVariable = variable
- refVariable = &variable
- refVariable = *variable

Soumettre Vous avez utilisé 2 essais sur 2

Afficher La Réponse

Passing Reference Types

1/1 point (graded)

What is the side-effect of passing a reference type into a function?

- ○ You have to know the data type before passing a reference type
- ◉ Modifying the reference in the function modifies the value in memory ✔
- ○ Modifying the reference in the function only modifies a copy of the value
- ○ You must use the ref keyword in the function prototype

Soumettre | Vous avez utilisé 1 essais sur 2 | Enregistrer | Afficher La Réponse

Pointer Variables

1/1 point (graded)

Why must you provide a data type for a pointer variable even though pointer variables only store memory addresses?

- ◉ The complier needs to know how much to request for any value that will be stored in that memory location ✔
- ○ The compiler will create a temporary value of that type and store it in the memory location to prevent it being overwritten
- ○ It isn't necessary
- ○ It let's other programmers know what data to store in that pointer

Pointer Variable

0/1 point (graded)

Enter the proper name for your pointer variable you created in step 4 of the lab.

pNum ✖

Soumettre | Vous avez utilisé 1 essais sur 2 | Enregistrer

Pass By Value

1/1 point (graded)

What is the value of num1 after the call to PassByValue() returns?

3 ✔

3

Soumettre | Vous avez utilisé 1 essai sur 1 | Afficher La Réponse

Pass By Ref

1/1 point (graded)

What is the value stored in pNum after the call to PassByRef()?

50 ✔

50

Soumettre | Vous avez utilisé 1 essai sur 1 | Afficher La Réponse

Pointer Error

1/1 point (graded)

What error was or warning resulted from attempting to perform step 15, passing a pointer to PassByValue()?

NOTE: if you are not using Visual Studio, your compiler may generate a slightly different message but the context will be the same.

- ◉ argument of type int* is incompatible with parameter of type int ✔
- ○ a stack overflow error was generated
- ○ argument of type string is incompatible with parameter of type int
- ○ No errors resulted

Separate Files

1/1 point (graded)

Select the two extensions used when separating a C++ class into two files.

- [x] .cpp
- [ ] .c1
- [ ] .c2
- [x] .h
- [ ] .h1

✔

Note: Make sure you select all of the correct options—there may be more than one!

Soumettre | Vous avez utilisé 1 essai sur 1 | Afficher La Réponse

Header Files

1/1 point (graded)

The header file contains the _____?

- ○ Function implementation
- ◉ Function prototypes ✔
- ○ Link to all files used in the application

warnings and your class names in the .cpp file are underlined with red squiggly lines.

What is the most likely cause?

- You forgot to use the .cpp extension on the code file
- You forgot to include the header file reference in your implementation file ✔
- You need to have the header and implementation files both open in the compiler
- Your implementation file is too large

Soumettre | Vous avez utilisé 1 essai sur 1 | Afficher La Réponse

Constructors

1/1 point (graded)

Select two features of a constructor.

- [x] Has the same name as the class
- [ ] Starts with a tilde (~)
- [ ] Returns void
- [x] Has no return type
- [ ] Starts with an asterisk (*)

✔

Class Files

1/1 point (graded)

What line of code was necessary for you to instantiate a Student object in main()??

- import "Student.h"
- #import "Student.h"
- #include "Student.h" ✔
- #include "Student.cpp"

Soumettre | Vous avez utilisé 1 essai sur 1 | Afficher La Réponse

Class Scope

1/1 point (graded)

What is output to the console when the SitInClass() method is called on a Teacher object?

Sitting at front of class ✔

Soumettre | Vous avez utilisé 1 essai sur 1 | Afficher La Réponse

Class Instantiation

1/1 point (graded)

Student stud1;

What would be the value of the member variables of a Student object if you instantation stud1 with the above code?

- All values will be null
- Values will be initialized to default values for the data types ✔
- All values will be initialized to blank spaces
- All values will be initialized to zero

Soumettre | Vous avez utilisé 1 essai sur 1 | Afficher La Réponse

Inheritance

0/1 point (graded)

Select the currently supported types of inheritance in C++: Choose three.

- [x] public
- [ ] friend
- [ ] private
- [x] protected
- [x] internal

✖

Note: Make sure you select three!

Soumettre | Vous avez utilisé 2 essais sur 2 | Afficher La Réponse

Inherit from a Base Class

1/1 point (graded)

Which of the following code samples correctly inherits the Person class?

- (•) class Student : public Person ✔
- ( ) class Student inherits Person

Abstract Classes

1/1 point (graded)

What is one reason for creating abstract classes?

- ◉ To prevent the class from being instantiated in code ✔
- ○ To create a class that can only be inherited by other abstract classes
- ○ To force inherited classes to use a different name than the abstract class
- ○ To make all functions in the inherited class virtual

Soumettre | Vous avez utilisé 1 essais sur 2 | Enregistrer | Afficher La Réponse

Friend Function

1/1 point (graded)

True or false, a friend function is not a member of the class, but can access all members of the class.

- ◉ True ✔
- ○ False

The Protected Keyword

0/1 point (graded)

In your implementation of the Person base class, which attributes does the derived class Student have access to?
Select all that apply.

- [ ] first_name (public)
- [x] phone (protected)
- [ ] age (private)
- [ ] last_name (public)

✖

Soumettre | Vous avez utilisé 1 essai sur 1 | Afficher La Réponse

Multiple Choice

1/1 point (graded)

What would happen if you had forgotten to implement the OutputIdentity() function in the Student
but later tried to invoke OutputIdentity() from a Student object?

- ○ The program would run as intended
- ◉ The compiler would display an error and would not build ✔
- ○ The function would run Person::OutputIdentity because the base class's function was never overloaded

- [x] cin
- [ ] stdin
- [x] cout
- [ ] stdout
- [x] cerr
- [ ] stderr

✔

Soumettre | Vous avez utilisé 1 essais sur 2 | Enregistrer | Afficher La Réponse

Operator Functions

1/1 point (graded)

Assuming you have an int variable named s, what is wrong with this line of code? std::cin <<s

- ○ You can't input values to cin that are variables
- ○ You need to declare cin as type int first
- ○ You can't assign data from cin to a variable
- ◉ The operator function should be >> ✔

0/1 point (graded)

Consider the following code snippet:

```
std::string s;

std::cin >> s;
```

If you entered "This is a test string" at the console, what will the variable s contain?

- T
- This ✔
- This is a test string

Soumettre | Vous avez utilisé 1 essai sur 1 | Afficher La Réponse

Streams and Classes

0/1 point (graded)

What must you do to have istream or ostream input or output object types?

- Make your class inherit from istream and ostream
- Override the << and >> operators in your class
- Implement the iStream interface in your class
- Ensure you have placed the #include preprocessor directive in your class header file ✔

Type in the preprocessor directive that allows you to use setw() in your code.

#include <iomanip> ✔

Soumettre

Afficher La Réponse

Processing Files

1/1 point (graded)

The C++ Standard Library defines three stream-based classes for reading and writing data in files. The first two are ifstream and ofstream.

What is the name of thre third class that allows read and write?

fstream ✔

Soumettre Vous avez utilisé 1 essais sur 2

Enregistrer

Afficher La Réponse

File Open Modes

1/1 point (graded)

Which mode will cause the contents of an existing file to be overwritten?

- ios_base::ate
- ios_base::out
- ios_base::trunc ✔
- ios_base::over

0/1 point (graded)

In your implementation of Lab 4, which answer best represents the code used to open donation_record.txt?

- std::fstream file1("donation_total.txt", std::ios_base::trunc); ✖
- std::ofstream file1("donation_total.txt");
- std::fstream file1("donation_total.txt", std::ios_base::app);
- std::ifstream file1("donation_total.txt", std::ios_base::trunc);

Soumettre | Vous avez utilisé 1 essai sur 1 | Afficher La Réponse

Multiple Choice

1/1 point (graded)

Which is the correct way to extract a donation amount from a string with the format "[Name] [Amount]"?
Assume that stream is a stringstream object that was initialized as such: std::stringstream stream(str);

- stream >> amount;
- stream >> name >> amount; ✔
- name >> amount >> stream;
- name << amount << stream;