Nom: _____(_____/100)

C0A—Contrôle final

Ceci constitue le contrôle final pour le cours ***INF737—Concepts OO avancés*** (C0A). Il est présumé, avant d'attaquer la présente, que vous avez pris une part active à la réalisation des travaux pratiques et que vous avez pris sur vous de faire les exercices qui, parmi la liste proposée, vous auront aidés à parfaire votre compréhension des éléments de matière qui vous semblaient un peu plus nébuleux.

Ce contrôle est récapitulatif, ce qui signifie qu'il couvre l'ensemble de la session. *Vous noterez qu'il est possible d'y amasser un petit peu plus que* `100` *points…*

Consignes

Les consignes pour ce contrôle sont :

- le droit aux notes de cours est accordé;
- l'envoi de notes écrites, par télépathie, via pigeons voyageurs, hiboux, ou à l'aide de tout autre mode de communication, est interdit;
- la calculatrice est bannie, de même que le téléphone cellulaire, l'ordinateur portatif, le *Walkie Talkie* et leurs différents dérivés;
- si vous bloquez sur une question, passez à la suivante—il est plus sage de revenir plus tard sur ce qui vous pose problème, s'il vous reste du temps;
- les questions de ce contrôle n'ont pas toutes le même poids—planifiez la manière dont vous investirez temps et efforts;
- prenez cet examen comme un petit bonheur, mais un bonheur sérieux (quand même!);
- répondez aux questions, simplement. Chaque réponse sera corrigée en fonction de l'énoncé de la question, alors assurez-vous de bien répondre à ce qui est demandé, pas à ce qui aurait peut-être pu l'être;
- dans au plus deux heures trente, une grande paix vous envahira. L'odeur d'une liberté durement gagnée?
- votre professeur vous adore (mais oui!);
- j'espère que ces quelques semaines vous auront été profitables. Moi, je vous ai trouvé fort sympathiques, et j'espère que votre carrière sera à la hauteur de vos attentes;
- répondez directement sur le document, aux endroits prévus à cet effet.

Durée limite:.....**`150` minutes**

Que de plaisir en perspective!

Q00 L'important, c'est la rose… (_____/15)

Votre entreprise a décidé d'attaquer un marché méconnu et plein de possibilités : le marché des jeux non violents et politiquement corrects. Raisonnant qu'il y a trop de joueurs dans le créneau de véhicules à haute vitesse ou dans le créneau des orques ignobles, il semble y avoir consensus autour de vous quant à l'intérêt d'approcher le marché du jeu vidéo sous un angle neuf.

Vous faites partie de l'équipe de *Berthe la Morue*[MD], histoire des plus excitantes dans laquelle une Morue nommée Berthe, végétarienne, ouverte à toutes les religions et à toutes les options politiques, doit se réaliser pleinement en tant que morue.

L'une des premières tâches dans lesquelles vous êtes impliqué(e) est celle de concevoir l'infrastructure de *MorueVille*[MD], cité cosmopolite où les poissons de tous les horizons coexistent en harmonie les uns avec les autres.

Pour le moment, *MorueVille*[MD] peut être vue comme un conteneur de poissons (classe `MorueVille`).

Note importante : *MorueVille*[MD] est un endroit où vivent les poissons mais ne doit pas être responsable de leur construction ou de leur destruction (c'est la responsabilité d'autres composants du jeu *Berthe la Morue*[MD]). Il est donc essentiel que les poissons y soient accédés indirectement plutôt que directement.

```
class Poisson
{
   // peu importe
public:
   virtual void Nager ();
   // peu importe
};
class Morue : public Poisson
{
   // peu importe
public:
   void Nager ();
   // peu importe
};
```

MorueVille[MD] est un endroit unique dans le jeu; en tout temps, il ne doit y en avoir qu'une seule instance dans une partie de *Berthe la Morue*[MD] donnée.

Étant donné les consignes qui précèdent, complétez le code ci-dessous, en indiquant clairement ce qui doit être placé dans le fichier d'en-tête `MorueVille.h` et ce qui doit être placé dans le fichier source `MorueVille.cpp`. Je m'attends à ce qu'une instance de `MorueVille` expose au minimum un `iterator`, un `const_iterator`, une méthode `push_back()` de même que les déclinaisons `const` et non `const` des méthodes `begin()` et `end()`.

```
#include "Poisson.h"
#include "Incopiable.h" // cadeau!
```

(*Q00 suite…*)

Q01 La valse des morues (_____/15)

Les poissons, comme tout le monde le sait, sont des êtres romantiques. Une scène marquante du jeu *Berthe la Morue*[MD] est celle de la *Ronde des amoureux*, où chaque poisson voudra trouver l'âme sœur pour contempler platoniquement avec elle ou avec lui les champs de corail.

Trouver l'âme sœur n'est pas chose facile. Heureusement, chaque poisson a un cœur d'or et, comme le veut l'adage selon lequel *qui se ressemble s'assemble*, dans le monde des poissons, plus le nombre de carats des cœurs de poissons est proche et plus leurs cœurs battent d'un seul rythme.

Évidemment, pour garder le tout à un niveau de rectitude politique acceptable, chaque poisson doit danser selon une stratégie qui lui est propre. Le résultat d'une danse sera un nombre entre `0` et `1` inclusivement qui, multiplié par le nombre de carats d'un cœur de poisson, déterminera sa capacité de séduction.

```
class Poisson
{
   // peu importe
public:
   typedef unsigned char carats_t;
   typedef float tauxseduction_t;
private:
   carats_t m_CoeurDOr;
public:
   carats_t GetCarats () const
   {
      return m_CoeurDOr;
   }
   virtual tauxseduction_t Danser () = 0;
   float GetScoreSeduction ()
   {
      return GetCarats () * Danser ();
   }
   // peu importe
};
```

Sachant que plus deux poissons distincts on une capacité de séduction proche l'une de l'autre et plus ils sont susceptibles d'être l'âme sœur l'une de l'autre et que `MorueVille` contient au moins deux poissons, écrivez la méthode `TrouverAmesSoeurs()` de `MorueVille` qui :

- prend chaque poisson de `MorueVille`;
- trouve son âme sœur;
- ajoute cette âme sœur à un vecteur de poissons (on veut qu'à la position `i` du vecteur d'âmes sœurs se trouve l'âme sœur du poisson à la position `i` dans `MorueVille`); et
- retourne le vecteur d'âmes sœurs.

Présumez que la méthode `MorueVille::TrouverAmesSoeurs()` est déjà déclarée et ne vous préoccupez que de sa définition. Votre solution doit au minimum utiliser l'algorithme `std::for_each()` et un foncteur (*suggestion : écrivez un foncteur qui trouve l'âme sœur d'un poisson*). Vous pouvez utiliser le code de votre réponse à la question `Q00`.

```
#include <cmath> // abs()
```

(*Q01 suite…*)

(*Q01 suite…*)

Q02 Quand le *bum* revient en ville (_____/10)

L'excitante trame de fond de *Berthe la Morue*[MD] veut qu'un événement perturbateur survienne éventuellement par l'introduction d'une barbotte mal élevée et au comportement impropre.

Une barbotte est une espèce de poisson mal léché

Cette étape est cruciale dans l'intrigue et représente un seuil de danger élevé pour la morale publique. Il importe donc de provoquer une sauvegarde subite de l'état du jeu avant que ne se produise cet événement.

Le jeu a évolué de manière telle que tous les éléments à sauvegarder sont dans une liste de `Serialisable` et que l'opérateur suivant :

```
#include <iosfwd>
// ... déclaration correcte de Serialisable, peu importe ce que c'est ...
std::ostream & operator<< (std::ostream &, const Serialisable &);
```

est défini correctement (vous pouvez prendre pour acquis que ça fonctionne et que c'est bien codé). La classe `Serialisable` est un type valeur enrobant (schéma de conception *Decorateur*) un objet accédé de manière indirecte et invoquant les méthodes requises pour que cet objet se projette correctement sur le flux.

Écrivez la fonction `SauvegardeRapide()` ci-dessous qui :

- prend en paramètre le nom du fichier dans lequel sauvegarder la partie (de type `std::string`) et une liste de `Serialisable`;
- ouvre un flux en sortie (format texte) correspondant au nom du fichier passé en paramètre;
- si le fichier est bien ouvert, y projette chaque élément de la liste séparés l'un de l'autre par un blanc et retourne `true`, sinon retourne `false`;
- vous n'avez pas le droit d'utiliser plus d'un `return` dans la fonction.

Nous voulons une fonction très efficace, donc prenez soin de bien choisir vos paramètres.

```
#include <string>
#include <iterator>
#include <algorithm>
#include <fstream>
#include <list>
#include <iostream>
#include "Serialisable.h" // Serialisable
```

(*Q02 suite…*)

Q03 Coraux et choix technologiques (_____/15)

La décoration de l'environnement chatoyant de *Berthe la Morue*[MD] repose fortement sur l'abondance de coraux. Une grande partie de ce *look* tient à la variété de couleurs, de textures et de lents mouvements des coraux (chaque type de corail oscillant à sa manière, comme tout le monde le sait), et vous envisagez mettre en place un système qui saupoudre des regroupements de coraux ici et là sur la surface du sol marin.

Il importe pour votre design qu'on puisse séparer la question de déterminer le lieu et la sorte de corail pour un regroupement donné (problème surtout statistique) de la question de construire des coraux d'un type donné. Vous envisagez d'éventuelles extensions à *Berthe la Morue*[MD] et souhaitez pouvoir ajouter des types de coraux dans le futur sans devoir recompiler le système tout entier (au pire, une édition des liens devrait suffire—cet hiver, on utilisera des bibliothèques à liens dynamiques, mais bon).

Proposez un design qui permette de respecter ces consignes. Indiquez clairement ce qui sera générique, ce qui sera polymorphique, ce qui sera abstrait et les schémas de conception que vous choisirez d'appliquer. La forme est libre (texte, schémas, code, mélange de tout ça, autre) mais plus votre proposition sera claire, opérationnelle et plus elle remplira le mandat donné ci-dessus et meilleure sera la note.

(*Q03 suite…*)

Q04 Problème de multiplication… (_____/15)

Le moteur de *Berthe la Morue*[MD] a d'importants besoins de performance et, en ce sens, met ses développeuses et ses développeurs face à des défis de design relevés.

Vient le moment de la fusion des composants du projet. Il se trouve que :

- certaines équipes de développement ont utilisé des stratégies reposant fortement sur le clonage et les interfaces abstraites (manipulées à travers des pointeurs) supportant une mécanique de clonage; alors que
- d'autres manipulent des conteneurs de pointeurs sur des types valeurs qui n'ont pas implémenté de mécanique de clonage.

Un constat s'impose : lorsque le besoin se fait sentir de dupliquer des objets, il faut parfois les cloner et parfois solliciter leur constructeur par copie.

La décision est prise de ne pas favoriser l'une ou l'autre des deux options, ce qui signifie qu'il devient important de pouvoir dupliquer chaque objet de la meilleure manière possible *pour lui*.

Implémentez une stratégie qui comprendra les éléments suivants :

- tout objet qu'il sera préférable de cloner sera dupliqué par invocation de sa méthode de clonage;
- tout objet qu'il sera préférable de copier sera dupliqué par invocation de son constructeur par copie;
- il est interdit d'ajouter un parent à quelque classe existante que ce soit;
- présumant les classes `RocheMarine` et `Poisson` proposées à droite, le programme principal ci-dessous trouvera *à la compilation* la meilleure stratégie de duplication à appliquer lors de chaque invocation de `Dupliquer()`.

```
#include <string>
class RocheMarine { /* ... */ };
struct Clonable
{
   virtual Clonable *Cloner () const = 0;
   virtual ~Clonable () {}
};
class Poisson : public Clonable
{
   std::string m_Nom;
public:
   Poisson (const std::string &Nom)
      : m_Nom (Nom)
   {
   }
   Poisson (const Poisson &p)
      : m_Nom (p.GetNom ())
   {
   }
   std::string GetNom () const
   {
      return m_Nom;
   }
   Poisson * Cloner () const
   {
      return new Poisson (*this);
   }
};
```

```
// ...inclusions pour RocheMarine et pour Poisson...
int main ()
{
   // compact parce que l'espace manque...
   Poisson *p = new Poisson ("Haddock"); Poisson *p2 = Dupliquer (p);
   RocheMarine *r = new RocheMarine; RocheMarine *r2 = Dupliquer (r);
   delete r2; delete r; delete p2; delete p;
}
```

Rédigez votre solution dans les pages qui suivent.

(*Q04 suite…*)

(*Q04 suite…*)

Q05 L'assaut des requins recyclables (_____/15)

Les requins assaillent périodiquement la paisible communauté de *MorueVille*MD avec des tracts de mauvais goût et des invitations aux actions impures. Évidemment, les poissons de *MorueVille*MD cherchent à chasser ces assaillants à l'aide de remontrances perspicaces.

Durant ces moments de stress, il arrive très fréquemment que des requins doivent être créés puis détruits puis recrées et une analyse du patron typique d'utilisation de la mémoire pour la classe `Requin` démontre qu'être capable de réutiliser rapidement un bloc de mémoire de taille `sizeof(Requin)` tout juste libérée entraînerait un gain de vitesse.

Un collègue vous présente le concept de `Moton`. Un `Moton` de *`T`* exposera :

- une méthode `Allouer()` sans paramètres qui retournera un *`T`*`*`; et
- une méthode `Liberer()` qui prendra un `void*` en paramètre.

```
// ... inclusions et tout le tralala
class Requin
{
   // ...
public:
   Requin ()  { /* ... */ }
   ~Requin () { /* ... */ }
   void * operator new (size_t n)
   {
      return Moton<Requin>::Get().Allouer();
   }
   void operator delete (void *p)
   {
      Moton<Requin>::Get().Liberer(p);
   }
};
```

La particularité d'un `Moton` de *`T`* est qu'il s'agira d'un singleton qui recyclera la mémoire libérée sur une pile : `Liberer()` empilera le pointeur libéré plutôt que de le libérer et `Allouer()` recyclera si possible la mémoire la plus récemment libérée plutôt que de procéder à une allocation dynamique de mémoire.

Complétez la classe `Moton<`*`T`*`>` ci-dessous pour que :

- sa méthode `Vider()` libère toute la mémoire encore sur la pile `m_Recyclage`;
- sa méthode `Allouer()` recycle le plus récent bloc libéré ou alloue un bloc de `sizeof(`*`T`*`)` octets en mémoire à l'aide de `std::malloc()`, levant un `std::bad_alloc` si `std::malloc()` retourne un pointeur nul; et
- sa méthode `Liberer()` recycle le pointeur libéré en le plaçant sur la pile `m_Recyclage`.

La classe **`std::stack`** offre les méthodes **`push()`** pour empiler un élément, **`pop()`** pour dépiler un élément (***prudence*** *: `pop()` ne retourne pas l'élément dépilé*) et **`top()`** pour obtenir une référence sur l'élément au-dessus de la pile sans le dépiler.

Sa méthode **`empty()`** retourne `true` seulement si la pile est vide.

Rédigez votre solution dans les pages qui suivent.

(*Q05 suite…*)

```
// Inclusion des bibliothèques
#include <new>
#include <iostream>
#include <algorithm>
#include <stack>

// Déclaration de Moton<T>
template <class T>
   class Moton
   {
      std::stack <T *> m_Recyclage;
      static Moton Singleton;
      Moton () { }
      void Vider ();
   public:
      static Moton &Get () { return Singleton; }
      T * Allouer ();
      void Liberer (void *);
      ~Moton () { Vider (); }
   };

// Définition du singleton
template <class T>
   Moton<T> Moton<T>::Singleton;

// Complétez les définitions
template <class T>
   void Moton<T>::Vider ()
   {
      // Votre code suit...









   }
```

(*Q05 suite…*)

```
template <class T>
   T * Moton<T>::Allouer ()
   {
      // Votre code suit...









   }

template <class T>
   void Moton<T>::Liberer (void *p)
   {
      // Votre code suit...









   }
```

Q06 Des questions existentielles, rien de moins (_____/10)

Q06.0—**Pour 6 points**, indiquez pour chacun des deux exemples de code ci-dessous si la mention `throw()` est judicieuse ou non. Si elle n'est pas judicieuse, alors expliquez pourquoi.

Exemple A

```
class X
{
   int m_Val;
public:
   // ...
   const X & operator= (const X &x) throw()
   {
      m_Val = x.m_Val;
      return *this;
   }
};
```

Exemple B

```
template <class T>
   class X
   {
      T m_Val;
   public:
      // ...
      const X & operator= (const X &x) throw()
      {
         m_Val = x.m_Val;
         return *this;
      }
   };
```

Réponse : _____

_____

_____

_____

_____

_____

Q06.1—**Pour 4 points**, proposez une stratégie d'affectation à la fois générale et sécuritaire et appuyez votre affirmation par du code (prenez l'un ou l'autre des exemples de la question Q06.0 comme base de travail).

Q07	**Pour le plaisir**	**(_____ /10)**

La version réseau de *Berthe la Morue*[MD] est en production. Un mécanisme est requis pour faire en sorte que dès qu'un avatar (poisson manipulé par une joueuse ou par un joueur) atteindra le grade de *Fonctionnaire* (titre le plus convoité qui soit à *MorueVille*[MD]), toutes les autres joueuses et tous les autres joueurs en soient informés.

Proposez un schéma de conception pour réaliser cette tâche. Expliquez en quoi votre proposition solutionnera ce problème puis donnez-en au moins un avantage et un inconvénient.