IN204
Programmation orientée objet – Introduction aux objets

Séance de Travaux Dirigés du 9 septembre 2020

B. Monsuez

Partie I – Création du compteur en C++

Question n° 0 : Lancer votre environnement de développement préféré.

Si vous n'avez pas d(environnement de développement, vous pouvez prendre soit monodevelop (https://www.monodevelop.com/) soit codeblocks (http://www.codeblocks.org/) sous Linux ou sous Windows. Vous pouvez prendre Visual Studio (https://visualstudio.microsoft.com/) sur Mac et sous Windows, XCode (https://developer.apple.com/xcode/) sur Mac.

Par la suite, il vous appartiendra de choisir l'environnement que vous préférez.

Question n°1 : Créer un ensemble de fichier counter.cpp et conter.hpp pour définir votre classe de base « MyCounter ». Au départ, seul le fichier counter.hpp contient le code suivant :

```
struct MyCounter
{
    unsigned counter;
    unsigned max;

    unsigned getCounter() const {
      return counter;
    }
    unsigned getMax() const {
      return max;
    }

    void increment() {
      counter ++;
      if(counter > max)
        counter = 0;
    }

    void reset() {
       counter = 0;
```

```
    }

    void set(unsigned value) {
      counter = (value <= max) ? value : counter;
    }

    void setMax(unsigned value) {
       if(counter >= value)
         counter = 0;
         max = value;;
    }

    MyCounter()
    {
        counter  = 0;
        max = value ;
    }
}
```

Question n°2 : Créer une fonction **useObjectA**() dans le fichier main.cpp reprenant le code présenté sur les transparents :

```
void useObjectA() {
    MyCounter Counter1;
    MyCounter Counter2;
    Counter1.setMax(2);
    Counter2.setMax(4);
    Counter1.reset();
    Counter2.reset();
    for(unsigned i = 0; i <= 5; i++) {
      std::cout
      << "Valeur des compteurs ("   << Counter1.counter
      << ", " << Counter2.counter   << ")" << std::endl;
        Counter1.increment();
        Counter2.increment();
    }
}
```

Question n°3 : Vérifier que le code de **useObjectA**() fonctionne correctement.

Question n°4 : Le code des fonctions membres a été écrit directement dans la **`struct`** `MyCounter`. Modifier le code pour séparer la déclaration des fonctions membres de leurs implantations. (cf. Annexe)

Partie II – Constructeurs en C++

Question n° 1 : Nous considérons les deux fonctions suivantes (C++):

```
void myfunctionA() {
  MyCounter counter;
  std::cout << counter.getCounter() << std::endl;
  std::cout << counter.getMax() << std::endl;
}

void myfunctionB() {
  MyCounter* counter = new MyCounter();
  std::cout << counter->getCounter() << std::endl;
  std::cout << counter->getMax() << std::endl;
  delete counter;
}
```

Exécuter les deux fonctions et expliquer pourquoi la valeur des champs est différente dans le cas où les objets sont alloués sur la pile ou dans le cas où sont alloués sur le tas.

Question n°2 : Que faut-il ajouter à la classe pour garantir que le comportement soit toujours correct ?

Question n° 3 : Dans la fonction **useObjectA**(), le programme initialise les différents champs des compteurs. Ceci n'est pas très élégant.

Proposer une solution pour créer des compteurs en passant en paramètre le nombre maximal d'éléments comptés à partir duquel le compteur revient à 0.

Simplifier le code de la fonction **useObjectA**() une fois que vous avez proposé ces solutions.

Question n° 4 : Proposer enfin un constructeur de recopie qui permet de créer un nouveau compteur étant la copie d'un compteur passé en paramètre.

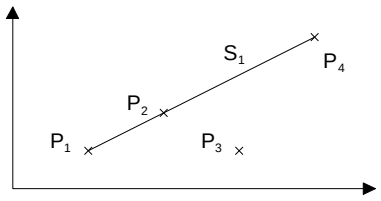
Partie III – Destructeurs en C++

Nous souhaitons que lorsque nous détruisons un compteur celui-ci signale sa destruction en affichant un message sur la console ainsi que sa valeur au moment de la destruction.

Question n° 1 : Définissez un destructeur en C++ qui effectue cette opération. Tester le code.

Partie VI – A effectuer en dehors du cours

Cette partie vise à définir des classes pour des objets graphiques. La plupart des bibliothèques d'interfaces graphiques sont écrites dans des langages orientés objets, qui sont particulièrement adaptés pour. Dans cette partie, nous définissons les classes de points, de segment et de droite dans le plan.



Question n° 1 En vous inspirant du code fourni, écrivez, compilez et mettez au point en C++

- Une classe Point, comprenant une abscisse entière et une ordonnée entière et donnant un accès à ces champs.
- Une classe Segment constituée de deux extrémités sous la forme de Point. Un segment est construit à l'aide de deux points et doit fournir les méthodes suivantes
    - une méthode pour tester si un point appartient au segment.
    - une méthode qui teste l'intersection du segment avec un autre segment. Cette méthode retourne vrai s'il existe une l'intersection ainsi que le point d'intersection faux, dans le cas contraire.
- Une classe Droite constituée d'un Point et d'un vecteur indiquant la direction, vecteur également constitué d'une partie abscisse et d'une partie ordonnée et doit fournir les méthodes suivantes
    - un constructeur à partir d'un segment,
    - un constructeur à partir d'un point et d'un angle
    - un constructeur à partir de deux points
    - une méthode qui teste l'intersection de la droite avec une autre droite. Cette méthode retourne vrai s'il existe une l'intersection ainsi que le point d'intersection faux, dans le cas contraire.
    - une méthode qui teste l'intersection de la droite avec un segment. Cette méthode retourne vrai s'il existe une l'intersection ainsi le point d'intersection faux, dans le cas contraire.

Question n° 2 : Ajouter à toutes ces classes des méthodes d'affichage pour le déboguage que l'on appellera `print`.

Question n° 3 : A la classe Droite

Ajouter une méthode `rotate90()` effectuant une rotation de 90 degrés à une droite.

Ajouter une méthode `move(int x, int y)` effectuant une translation à une droite.

Question n° 4 : Tester votre classe sur le code suivant :

```
#include <iostream>

int main() {
```

```
    Droite d1(Point(0,0), Point(4, 2));
    Droite d2(d1);
    d1.print();
    std::cout << std::endl;
    d2.print();
    std::cout << std::endl;
    d1.rotate90();
    d1.print();
    std::cout << std::endl;
    d2.print();
    return 0;
}
```

ANNEXE

Premiers éléments de syntaxe pour C++

Extensions de types

Les types **const**

C++ propose un nouveau qualificateur de type qui est introduit par le mot-clé **const**. Ce qualification est utilisée pour indiquer qu'un type est constant.

```
const int x = 3 ;       // Introduit une variable de type x qui ne peut pas être
                        // modifiée.
int z = 4 ;
const int* y = &z ;     // Introduit un pointeur sur une variable de type entier qui
                        // ne pourra pas être modifiée.
int const* m = &z ;     // Introduit un pointeur constant. C'est le pointeur qui ne
                        // peut pas être modifié et qui fera toujours référence à z.
                        // Par contre le contenu de z peut être modifié.
```

Les types référence

```
int& y = z;     // Le type y fait référence à la variable z. Cette référence ne
                // pourra jamais être modifiée.
```

Déclaration d'une fonction

Une fonction C++ reprend la syntaxe C. Cependant, il existe quelques extensions par rapport à une fonction C.

Le passage de paramètre par référence

En C++, il est possible de passer un paramètre par référence. Cela signifie que nous ne passons pas une valeur à la fonction mais une référence sur une variable existante et le paramètre va se comporter comme un alias de la variable existante.

```
void increment (int& value)
{
    value = value + 1 ;
}

...
int x = 3 ;
std::cout << x << std::endl;    // Affiche la valeur 3
increment(value);
std::cout << x << std::endl;   // Affiche la valeur 4
...

increment(3);                  // Erreur de compilation, doit faire référence
                               // à une variable.
```

La déclaration d'une classe

Une classe peut être déclarée soit par le mot clé class ou struct.

```
class A
{
    int x ; // Ce champ n'est pas visible de l'extérieur.
};
```

ou

```
struct B
{
    int x;  // Ce champ est visible de l'extérieur.
}
```

La seule différence entre **class** et **struct** est que les membres d'une classe définie par **struct** sont visibles de l'extérieur tandis que les membres d'une classe définies par class ne sont pas visibles de l'extérieur.

Déclaration des sections contenant des membres publics, privés et protégés

```
class A
{
    public:
        // Début d'une section de membres publics.
    private:
        // Début d'une section de membres privés.
    private:
        // Début d'une section de membres protégés.
};
```

Déclaration des champs du programme

Les champs sont déclarés comme dans les **struct** en C.

```
struct A
{
    unsigned fieldA;
    float fieldB;
    std::string fieldC;
    const char* fieldD;
};
```

Déclaration des fonctions membres

La déclaration d'une fonction membre peut se faire selon deux syntaxes :

- Soit le prototype de la fonction est fournie dans la classe, et le code est fournie en dehors de la classe,
- Soit le code de la fonction est directement intégrée dans la classe.

Syntaxe : Prototype dans la classe et code hors classe

Typiquement dans le fichier où vous déclarez la classe (typiquement le fichier .h ou .hpp), vous ajoutez les prototypes des fonctions :

```
class A
{
    public:
        int firstFunction() const;  // Le const indique que la fonction ne modifie
                                    // aucun champ dans l'objet.
        void secondFunction(int value);
};
```

Dans le fichier où vous implanteriez naturellement le code de la fonction, si c'était une fonction normale, vous spécifiez le code de la fonction comme suit.

```
int A::firstFunction() const
```

```
{
  // Code de la fonction
}

void A::secondFunction (int value)
{
  // Code de la fonction
}
```

Syntaxe : Code dans la classe

Il est possible de déclarer le code directement au sein de la classe :

```
class A
{
    public:
        int firstFunction() const
        {
            // Code de la fonction
        }
        void secondFunction(int value)
        {
            // Code de la fonction
        }
};
```

En fait ce code est équivalent à écrire :

```
class A
{
    public:
        int firstFunction() const;
        void secondFunction(int value);
};

inline int A::firstFunction() const
{
    ... // Code de la fonction
}

inline void A::secondFunction (int value)
{
    ... // Code de la fonction
}
```

En fait, l'instruction **inline**

```
inline int myFunction(int args)
{
    return args + 1 ;/ code de myFunction
}
```

indique au compilateur qu'en présence d'un appel fonctionnel, celui-ci peut choisir entre deux implantations :

- Soit l'implantation classique en effectuant un appel fonctionnel au code partagé de la fonction :

```
…. // code avant l'appel
    x = myFunction(3) ; // appel effectif de la fonction
…. // code après l'appel
```

- Ou alors en remplaçant l'appel de la fonction par le code de la fonction :

```
…. // code avant l'appel
…  x = 3 + 1 // insertion du code de myFunction !
…. // code après l'appel
```

Qui est nettement plus rapide que le code précédent, surtout qu'il peut être optimisé automatiquement en

```
…. // code avant l'appel
…  x = 4 // insertion du code de myFunction et simplification !
…. // code après l'appel
```

Affichage en C++

En C, nous avons la fonction printf pour afficher. Nous pouvons utiliser cette fonction en C++. Cependant C++ offre une autre méthode de lecture et d'écriture sur des flux.

Pour l'utiliser, nous devons charger les entêtes qui sont présents dans <iostream>.

```
#include<iostream> // Doit être spécifié pour accéder aux objets std
```

Nous avons deux objets `std::cout` et `std::cin` qui définisse le flux sortant de la console et le flux entrant de la console.

```
std::cout << x ; // Ecrit le contenu de la variable x vers la console.
std::cin >> x ;  // Lit le contenu de la variable x de l'entrée de la console.
```

Nous pouvons écrire plusieurs contenus à la fois en écrivant ainsi :

```
std::cout << "Les valeus (x,y) sont " << x // Ecrit le contenu de la variable x
          << y << std::endl;               // Ecrit le contenu de la variable y
                                           // ainsi qu'un saut de ligne (std::endl)
```