IN204
Programmation Orientée Objet – Dérivation & Héritage des objets

Séance de Travaux Dirigés du 15 septembre 2020

B. Monsuez

Partie I – Création d'une classe dérivée

Dupliquez vos projets de compteurs en C++ que vous avez réalisé la semaine dernière. Pour ceux qui n'ont pas de projets compteurs fonctionnels, vous pouvez partir de la solution présente en ligne.

Question n° 1 : A partir de la classe `MyCounter` que vous avez développé la semaine, une nouvelle `MyBiDiCounter`. Cette classe doit ajouter deux méthodes à la classe de base :

Une première méthode `decrement` qui décrémente le compteur, cette méthode correspond au pseudo-code suivant :

```
decrement()
    si counter > 0
        counter <- counter - 1
    sinon
        counter = max;
```

Une seconde méthode `print` qui affiche l'état du compteur de la manière suivante.

```
print()
    affiche "Compteur : " counter "/" max (retour à la ligne)
```

Question n°2 : Tester votre nouveau compteur `MyBiDiCounter` en utilisant la fonction de test suivante.

```
void testMyBiDiCounter()
{
    MyBiDiCounter counterA;
    counterA.setMax(4);
    counterA.reset();
    counterA.print();
    for(int i=0; i < 6; i++)
    {
```

```
        counterA.increment();
        counterA.print();
    }
    for(int i=0; i < 6; i++)
    {
        counterA.decrement();
        counterA.print();
    }
}
```

Question n°3 : On avait créé des constructeurs par défaut pour la classe `MyCounter`. Est-il possible de les appeler pour créer la classe `MyBiDiCounter` ?

Partie II – Constructeurs & Arbres de Dérivation

Nous reprenons le code C++ de la partie précédente.

Question n°1 : Ajoutez à la classe `MyBiDiCounter` l'ensemble des constructeurs dont notamment :

- le constructeur par défaut,
- le constructeur de recopie,
- le constructeur spécifiant la valeur maximale,
- le constructeur spécifiant à la fois la valeur courante du compteur et la valeur maximale.

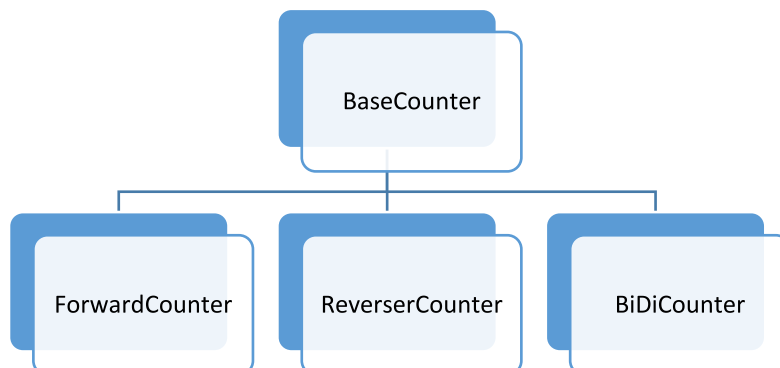
Modifier le code de la fonction `testMyBiDiCounter` pour appeler le bon constructeur.

Dans la suite, nous construisons un nouveau projet. Cependant, le code précédent sera encore utilisé dans la partie III du TD.

Question n°2 : La classe `MyBiDiCounter` ajoute la fonction `decrement` à la classe `MyCounter`. En fait, nous pouvons définir une famille de compteur :

- Le compteur `ForwardCounter` qui compte de 0 à max et repars à 0.
- Le compteur `BackwardCounter` qui compte max à 9 et repars à max.
- Et le compteur `BiDiCounter` qui peut incrémenter ou décrémenter le compteur interne.

Nous souhaitons partager le maximum de code entre ces différents compteurs. Une solution consiste à définir l'arbre de dérivation suivant :



et nous souhaitons factoriser le maximum de code entre les classes `ForwardCounter`, `ReverseCounter` & `BiDiCounter`, l'objectif étant que ces trois classes contiennent le minimum de code.

Question n°2.1 : Faites la liste des méthodes, champs pouvant être partagés et la liste des méthodes et champs propres à chacune des classes.

Question n°2.2 : Implanter la classe `BaseCounter`. On s'inspirera fortement de la classe `MyCounter` déjà définie.

Question n°2.3 : Implanter les classes `ForwardCounter`, `BackwardCounter` et `BiDiCounter` qui héritent chacune de la classe `BaseCounter`.

Question n°3 : Tester le comportement de vos compteurs à partir du code suivant

```
void testFamilyOfCounters()
{
    ForwardCounter incCounter(0, 4);
    BackwardCounter decCounter(0, 3);
    BiDiCounter biDiCounter(0, 5);
```

```
    for(int i=0; i < 6; i++)
    {
        incCounter.increment();
        incCounter.print();
        decCounter.decrement();
        decCounter.print();
        biDiCounter.increment();
        biDiCounter.print();
    }
    for(int i=0; i < 6; i++)
    {
        biDiCounter.decrement();
        biDiCounter.print();
    }
}
```

Partie III – Surcharge & Masquage

Nous repartons du code de `MyBiDiCounter` tel défini à la fin de la question 1 de la partie II.

Question n°1 : Ajouter à la classe `MyBiDiCounter` une nouvelle méthode :

```
increment(unsigned value)
    si counter + value <= max
        counter <- counter + value
    sinon
        counter = counter + value mod max
```

Question n°2 : Tester le bon fonctionnement de cette classe à partir du code suivant :

```
void testNewIncMethod() {
    MyBiDiCounter bidiCounter1(0, 5);
    for(unsigned i = 0; i <= 5; i++)
    {
        bidiCounter1.increment(5);
        bidiCounter1.print();
    }
}
```

Question n°3 : Tester le code suivant.

```
void testOldIncMethod() {
    MyBiDiCounter bidiCounter1(0, 5);
    for(unsigned i = 0; i <= 5; i++)
    {
        bidiCounter1.increment();
        bidiCounter1.print();
    }
}
```

Expliquer pourquoi cela ne fonctionne pas ? Proposer une modification de l'appel pour que cela puisse fonctionner.

Question n°4 : Modifier la classe MyBiDiCounter de manière à ce que les deux méthodes soient accessibles, à la fois la méthode `increment()` et la méthode `increment(unsigned)`.

Tester ensuite que le code initial de la fonction `testOldIncMethod()`.

Partie IV – Surcharge & Masquage

Question n°1 : Nous souhaitons redéfinir dans une classe MyAdvCounter qui dérive de la classe MyCounter une nouvelle méthode increment() en remplacement de la méthode actuelle dont le comportement est le suivant :

```
increment()
    si counter <= max
        counter <- counter + 1
    sinon
        counter = max
```

Ie. le compteur ne revient pas à zéro et reste à `max` une fois la valeur max atteinte.

Question n°2 : Tester le bon fonctionnement de la méthode à partir du code suivant et vérifier que le comportement est conforme

```
void testMyAdvCounter()
{
    MyAdvCounter incCounter(0, 4);
    for(int i=0; i < 6; i++)
    {
        incCounter.increment();
        incCounter.print();
    }
}
```

Question n°3 : Nous créons la fonction suivante :

```
void testCounter(MyCounter& unCompteur)
{
    for(int i=0; i < 6; i++)
    {
        unCompteur.increment();
        unCompteur.print();
    }
}
```

Tester la méthode en passant à la fonction testCounter un compteur de type « MyCounter » et un compteur de type « MyAdvCounter ». Expliquer le comportement de la fonction pour chacun des types de compteur.

Partie V – Destructeurs

Question n°1 : En partant du code des classes `BaseCounter`, `ForwardCounter`, `BackwardCounter` et `BiDiCounter`, ajouter à chacune de ses classes un destructeur qui affiche simplement le message « Destruction : » suivi du *Nom de la Classe*.

Question n°2 : Tester le fonctionnement du destructeur à partir de la fonction :

```
void testFamilyOfCounters()
{
    ForwardCounter incCounter(0, 4);
    BackwardCounter decCounter(0, 3);
    BiDiCounter biDiCounter(0, 5);
    for(int i=0; i < 6; i++)
    {
        incCounter.increment();
        incCounter.print();
        decCounter.decrement();
        decCounter.print();
        biDiCounter.increment();
```

```
        biDiCounter.print();
    }
    for(int i=0; i < 6; i++)
    {
        biDiCounter.decrement();
        biDiCounter.print();
    }
}
```