TD3 : Classe et objet (1/2)

Exercice 1 : Conception et spécificateur

On souhaite créer un type de donnée pour représenter une personne. Cette personne est identifiée par son nom, son prénom et son numéro de sécurité sociale (numéro unique). On veut pouvoir saisir et afficher ces informations.

a. Ecrivez, en C++, la classe Personne.

```
#include <string>
#include <iostream>
using namespace std;

class Personne {
  public:
    string nom, prenom;
    unsigned int numero;

    void saisir () {
      cout << "Saisir le nom: ";
      cin >> nom;
      cout << "Saisir le prénom: ";
      cin >> prenom;
      cout << "Saisir le numéro de sécu: ";
      cin >> numero;
    }

    void afficher () const {
      cout <<"Nom: "<<nom<<"\nPrénom: "<<prenom<<"\nNuméro: "<<numero<<endl;
    }
};
```

b. Donner le programme principal, en C++, permettant de saisir puis d'afficher les informations d'une personne.

```
int main () {
  Personne patient;
  patient.saisir();
  patient.afficher();
  return 0;
}
```

c. En programmant la procédure membre d'affichage, vous avez dû choisir sous quel format la personne est affichée, et en particulier quel(s) caractère(s) de séparation utiliser entre les différentes données membres à afficher. Ecrire une deuxième version de cette procédure, utilisant le principe de surcharge, permettant de paramétrer le(s) caractère(s) utilisé(s).

```
class Personne {
  [...]
  void afficher () const {
      cout <<"Nom: "<<nom<<"\nPrénom: "<<prenom<<"\nNuméro: "<<numero<<endl;
  }

  void afficher (string sep) const {
      cout <<"Nom: "<<nom<<sep<<"Prénom: "<<prenom<<sep<<"Numéro: "<<numero<<endl;
  }
```

```
   [...]
};
```

d. Modifier cette procédure pour qu'elle prenne une valeur de paramètre par défaut. Voyez-vous un problème ?

```
  void afficher (string sep="\n") const {
      cout <<"Nom: "<<nom<<sep<<"Prénom: "<<prenom<<sep<<"Numéro: "<<numero<<endl;
  }
```

Un problème d'ambiguité apparaît. En effet, le compilateur ne pourra plus savoir si l'instruction `patient.afficher()` fait référence à la première version (sans paramètre) ou bien à la deuxième version (avec paramètre mais utilisant sa valeur par défaut). Cela produit une erreur à la compilation de cette instruction (appel ambigu).

e. Que doit-on faire pour interdire aux utilisateurs de la classe de manipuler les données membres directement?

Il faut spécifier les données membres comme privés à la classe. Cela se fait en déplaçant les trois données dans un spécificateur private (ne pas parler de protected, vu dans une prochaine UE).

```
class Personne {
  public:
    void saisir () {...}
    void afficher () {...}

  private:
    string nom, prenom;
    unsigned int numero;
};
```

f. Afin de permettre leur manipulation, écrivez des procédures/fonctions dont le rôle est de lire et modifier les données membres. Quels sont les avantages et inconvénients de cette conception ?

```
class Personne {
  private:
    string nom, prenom;
    unsigned int numero;

  public:
    void saisir () {...}
    void afficher () {...}
    string getNom() const {return nom;}
    void setNom(const string n) {nom = n;}
    string getPrenom() const {return prenom;}
    void setPrenom(const string p) {prenom = p;}
    unsigned int getNumero() const {return numero;}
    void setNumero(const unsigned int n) {numero = n;}
};
```

Ces fonctions sont appellées les mutateurs (set) et accesseurs (get) de la classe. Ceci a l'avantage de contrôler comment les données membres sont manipulées depuis l'extérieur (indépendamment du choix de la structure de donnée) et de faire des vérifications si besoin (ex. validité de l'indice lors de l'accès à un tableau membre). Par contre, il faut penser à les écrire et cela demande de les appeler à chaque manipulation des données membres (ce qui peut allourdir les appels).

Remarque : les instructions `nom = n;` et `prenom = p;` fonctionnent correctement car l'opérateur d'affectation = est surchargé dans la classe C++ string.

g. Ecrivez un programme principal qui modifie le nom d'une personne après saisie.

```
int main () {
  Personne patient;
  patient.saisir();
  string nouveauNom;
  cout << "Saisir le nouveau nom: ";
  cin >> nouveauNom;
  patient.setNom(nouveauNom);
  return 0;
}
```

Exercice 2 : Surcharge d'opérateurs

On souhaite pouvoir faire des opérations mathématiques sur des points 2D. Une solution serait de définir des fonctions membres telles que **`void`** `additionnerPoints(`**`const`** `Point2D & p)`. Une autre solution consiste à définir les opérateurs élémentaires pour un point 2D de telle façon à ce que l'on puisse exécuter des instructions telles que `pt3=pt1+pt2;`.

a. Ecrire la classe Point2D avec les fonctionnalités nécessaires pour additioner, soustraire et tester l'égalité de deux points.

```
class Point2D {
  public:
    float x,y;
    Point2D () {x = 0.0; y = 0.0;}

    Point2D operator + (const Point2D & p) const {
      Point2D resultat;
      resultat.x = x + p.x;
      resultat.y = y + p.y;
      return resultat;
    }

    Point2D operator - (const Point2D & p) const {
      Point2D resultat;
      resultat.x = x - p.x;
      resultat.y = y - p.y;
      return resultat;
    }

    bool operator == (const Point2D & p) const {
      return x == p.x && y == p.y;
    }
};
```

b. Ajouter les fonctionnalités d'affectation et d'incrément préfixé (ajout de 1 à chaque coordonnée lorsque l'on fait `++pt`).

```
class Point2D {
  [...]
  Point2D& operator = (const Point2D & p) {
    x = p.x;
    y = p.y;
    return *this;
```

```
  }

  Point2D& operator ++ () {
    x++;
    y++;
    return *this;
  }
};
```

Pour information, l'entête de la version postfixe est : `Point2D operator ++ (int);`
Le type de retour n'est pas une référence car l'ancienne valeur est retournée. Le paramètre int n'est pas utilisé.

c. Ajouter les fonctionnalités de symétrie centrale par rapport à l'origine définie par l'opérateur unaire « - », et le test si le point est situé à l'origine avec l'opérateur « ! ».

```
class Point2D {
  [...]
  Point2D operator - () const {
    return Point2D(-x,-y);
  }

  bool operator ! () const {
    return x==0.0 && y==0.0;
  }
};
```

d. Ajouter les fonctionnalités d'affichage et de lecture des coordonnées du point suivant le format `( x , y )`.

```
class Point2D {
  [...]
  friend std::ostream& operator << (std::ostream& flux, const Point2D& p) {
    flux << "( " << p.x << " , " << p.y << " )";
    return flux;
  }

  friend std::istream& operator >> (std::istream& flux, Point2D& p) {
    flux >> p.x >> p.y;
    return flux;
  }
};
```

Comme ces opérateurs sont globaux, il faut les déclarer comme amis de la classe Point2D afin de pouvoir accéder à ses données membres (ici les données sont publiques donc ça n'aurait pas posé de problème, mais c'est une bonne habitude à prendre). On rend le flux pour pouvoir enchaîner les appels (ex. `cout << pt1 << pt2;` et `cin >> pt1 >> pt2;`).

Dans la lecture du flux, il n'est pas utile de spécifier la lecture des caractères déliminateurs, cin cherche automatiquement les premières valeurs des types demandés (ici des float, et donc ne tient pas compte de ce qui n'est pas un float, comme des parenthèses).