Université de Reims Champagne-Ardenne
UFR Sciences Exactes et Naturelles
Licence Informatique
INFO0402: Programmation Orientée Objet

TD N°2

Exercice 1: Alignement

On veut mettre dans une structure deux booléens, un entier 16bit, un entier 32bit et un pointeur. Dans un premier temps, on supposera que le modèle de données est LLP64.

1. Rappeler ce qu'est l'alignement mémoire sur un ordinateur, et le rôle qu'il joue dans les transaction mémoire.
2. Donner la taille de la somme des champs composant cette structure.
3. Donner toutes les organisations possibles de champs qui permettent d'obtenir une structure sans trou.
4. Donner l'organisation des champs la pire possible (i.e. celle qui maximise la taille de la structure).
5. Dessiner la représentation mémoire d'un tableau de deux éléments dans les deux cas précédents.
6. La taille de cette structure peut-elle varier si on change le modèle de données ?
7. Si on reste sur un système 64bit, y-a-t-il d'autres cas où le changement de modèle de données change la taille de la structure ? Si oui, comment résoudre ce problème ?

Solution:

1. `&T % sizeof(T) = 0` pour tous les types élémentaires. Transaction mémoire sur des BITs entiers.
2. 2x8bit + 16bit + 32bit + 64bit = 128bit.
3. 1=8bit, 2=16bit 4=32bit 8=64bit,
   combinaison : 84211 84112 82114 81124, et idem avec le 8 à la fin plutôt qu'au début.
4. 18142 = 32 octets (4x8) = 256bit
5. faire le dessin
6. oui en ILP32, le pointeur passe en 32bit, structure sur 96bit au mieux. 14142 au pire = 20 octets (5x4) = 160bit au pire.
7. oui, avec les long/long long int.
   - Utiliser les types de stockage entier du c++11 pour garantir la longueur fixe,
   - Trier les champs par ordre décroissant de taille
   - Pour les pointeurs les placer entre les sizeof 8 et 4 afin de garantir la continuité lors des passages 64-32 bits.

Exercice 2: Fraction et opérateur

On veut définir une classe représentant une fraction (numérateur et dénominateur).

1. Définir la classe, et donner 3 façons de définir les constructeurs : constructeur dans le cas où le résultat de la fraction est un entier, constructeur par défaut (=0) : approche classique, avec constructeur délégué, avec initialisation par défaut.
2. La déclaration du constructeur/assignation par copie/déplacement et d'un destructeur est-elle nécessaire dans le cas de cette classe ? Si oui, les définir.
3. Définir les setters et getters appropriés. On fera en sorte qu'il ne soit jamais possible de créer une fraction avec une division par 0.
4. En supposant que l'on dispose d'une implémentation de l'algorithme d'Euclide, écrire une méthode pour réduire la fraction.
5. Écrire une méthode permettant d'inverser une fraction.

6. Définir une surcharge de l'opérateur `<<` sur le flux de sortie standard afin d'afficher une fraction. Si le dénominateur est 1, on affichera l'entier.
7. Définir l'opérateur qui permet d'additionner deux fractions. Le résultat devra être réduit.
8. On voudrait maintenant être en mesure d'ajouter un entier à une fraction. Que devons-nous définir ?
9. Donner l'ensemble des opérations qui sont effectuées par l'écriture `frac(4,3)+f2+3+f1` où `f1` et `f2` sont des fractions.
10. Définir l'opérateur qui permet de comparer deux fractions. Pourquoi la redéfinition de cet opérateur ne pose-t-elle pas de problème ?

Solution:

1. Classe et constructeurs

```
class Frac {
// version 1
private: int    num, den;
public:
  Frac(int n, int d) : num(n), den(d) {};
  Frac(int i) : num(i), den(1) {};
  Frac() : num(0), den(1) {};
```

```
// version 2
private: int    num, den;
public:
  Frac(int n, int d) : num(n), den(d) {};
  Frac(int i) : Frac(i,1) {};
  Frac() : Frac(0) {};
```

```
// version 3
private: int    num = 0, den = 1;
public:
  Frac(int n, int d) : num(n), den(d) {};
  Frac(int i) : num(i) {};
  Frac() = default;
```

2. non. Le comportement par défaut (=copie des champs) suffit.

3. setters et getters

```
  void setNum(int n) { num = n; }
  void setDen(int d) { assert(d != 0);  den = d; }
  int getNum() const { return num; }
  int getDen() const { return den; }
```

4. reduction de fraction

```
void reduce() {
  int  pgcd = Euclid(num,den);
  // printf("PGCD(%d,%d)=%d\n", num, den, pgcd);
  if (pgcd > 1) { num /= pgcd; den /= pgcd; }
}
```

5. inversion de la fraction

```
void inverse() {
  assert(num != 0);
  std::swap(num, den);
}
```

6. surcharge sur la sortie

```
friend ostream& operator<<(ostream& os, const Frac& f) {
  os << f.num;
  if (f.den != 1) os << "/" << f.den;
  return os;
}
```

7. addition de fraction

```
// noter que si operator+ n'était pas const, on ferait un get sur
// une fraction const, d'où la nécessité pour get d'être const
friend Frac operator+(const Frac &f1, const Frac &f2) {
  Frac   f(f1.num * f2.den + f2.num * f1.den, f1.den * f2.den);
  f.reduce();
  return f;
}
```

8. besoin de rien : car un entier peut être converti automatiquement en une fraction grâce au constructeur à un argument prenant en paramètre un entier.
9. 5 objets temporaires : x1 = Frac(4,3), x2 = x1+f2, x3=Frac(3), x4=x2+x3, x5=x4+f1.
   Noter qu'en surchargeant += pour (Frac,Frac) et (Frac,int) cette même opération pourrait être réalisée sans aucun objet temporaire en utilisant ces opérateurs.
10. faire la somme constructeur, les objets temporaires et les conversions.

```
// pas de souci car ne renvoie pas d'objet temporaire
bool operator==(const Frac &f1, const Frac &f2) {
  return f1.num * f2.den == f2.num * f1.den;
}
```

Exercice 3: Vecteurs et base

1. Définir une structure Vec2D (i.e. sans champs privés) qui permet de définir un vecteur 2D avec un constructeur à deux arguments.
2. Pourquoi une telle définition est-elle cohérente ?
3. Quel est l'effet de la définition du constructeur ?
4. A-t-on besoin de définir le constructeur/l'assignation par copie ? Même questions pour le destructeur et le constructeur/assignation par déplacement.
5. Y-a-t-il un inconvénient à redéfinir l'opérateur * afin que celui-ci réalise le produit scalaire ?
6. Définir une fonction permettant de calculer le produit scalaire.
7. Définir l'opérateur de produit avec un scalaire (flottant) et de somme de vecteur.
8. Définir maintenant une structure Base2D qui représente une base dans un espace 2D. on ajoutera un constructeur qui permet de créer une Base2D à partir de deux vecteurs.
9. Que peut-on dire sur le fonctionnement particulier de ce constructeur ?
10. Ajouter une méthode qui permet de savoir si une base est orthonormée.
11. Ajouter les surcharges de l'opérateur << sur la sortie standard afin d'afficher la valeur d'un vecteur et d'une base.
12. Surcharger l'opérateur parenthèse afin que, si b est une base, b(x,y) renvoie le vecteur de coordonnée (x,y) dans la base b.

Solution:

1. voir le code

```
struct Vec2 {
  float u, v;
  Vec2(float _u, float _v) : u(_u), v(_v) {}
};
```

2. il n'y a pas de cohérence particulière souhaitée entre les composantes des vecteurs ni sur le vecteur lui-même : donc on peut laisser libre l'accès aux champs. Faire des classes pour tout n'est pas nécessairement une bonne idée, mais peu être une règle de codage. Ici, un struct suffit.
3. ce constructeur retire le constructeur par défaut de la définition de la classe. Utiliser `Vec2 = default;` pour le rétablir.
4. non, car ils restent tous définis et correspondent à des copies champs à champs, ce qui est bien le comportement souhaité pour cette classe. Le destructeur ne fait rien. Le déplacement comme la copie.

5. Il n'y en a pas (ne retourne pas de `Vec2` temporaire, mais un `float`).
6. Voir le code (fonction,externe à la classe) :

```
float operator*(const Vec2 &v1, const Vec2 &v2) {
  return v1.u * v2.u + v1.v * v2.v;
}
```

7. noter que toutes ces opérations génèrent des objets temporaires. Voir le code (fonctions, externes à la classe) :

```
inline Vec2 operator+(const Vec2 &v1, const Vec2 &v2) {
  return Vec2(v1.u+v2.u,v1.v+v2.v);
}
inline Vec2 operator*(const Vec2 &v, const float s) {
  return Vec2(s * v.u, s * v.v);
}
inline Vec2 operator*(const float s, const Vec2 &v) {
  return Vec2(s * v.u, s * v.v);
  // ou return v * s; (par symétrie)
}
```

8. voir le code

```
struct Base2D {
  Vec2  v1, v2;
  Base2D(const Vec2 &_v1, const Vec2 &_v2) : v1(_v1), v2(_v2) {}
};
```

9. il fait appel explicitement aux constructeurs de Vec2 dans la chaine d'initialisation.
10. voir le code (méthode, interne à a la classe)

```
bool isOrthonormal() { return (v1 * v2 == 0.f); }
```

11. voir le code (fonctions, externes aux classes Vec2 et Base2D)

```
ostream& operator<<(ostream& os, const Vec2& a) {
  return os << "(" << a.u << "," << a.v << ")";
}
ostream& operator<<(ostream& os, const Base2D& a) {
  return os << "(" << a.v1 << "," << a.v2 << ")";
}
```

remarquer que la surcharge pour les bases utilise la surcharge pour les vecteurs.

12. cette surcharge **DOIT** être placée dans la classe.

```
struct Base2D {
    ...
  Vec2 operator()(float x, float y) { return x*v1 + y*v2; }
};
// exemple
Vec2 p = b(3.f, 2.f);
cout << "p=" << p << endl;
```

Exercice 4: Classe `KeyId`

On veut définir une classe destinée à stocker un couple (Key,UniqueId) telle que UniqueId est un identificateur unique pour chacun des objets créés (y compris lorsqu'on le copie) et Key est une clef associée à cet identificateur. Dans un premier temps, on souhaite que l'UniqueId ne puisse pas être séparé de l'objet avec lequel il a été créé.

1. Comment faire en sorte que ces contraintes soient respectées lors de l'écriture de la classe ?
2. Déclarer la classe (constructeur à partir d'une clef, set/getter, constructeur et assignation par copie).
3. Définir le constructeur à partir d'une clef.
4. Définir les setter/getters.
5. Définir le constructeur par copie.

6. Définir l'assignation par copie.
7. Doit-on faire quelque chose pour le constructeur et l'assignation par déplacement ?
8. Quel est l'inconvénient d'une telle approche ?
9. Quel serait le sens d'une construction ou d'une assignation par déplacement ?

Solution:

1. définir un static qui va être utilisé pour produire les identificateurs uniques lors des constructions, et faire en sorte que toute opération de copie ne permette pas la copie de l'identifiant.
2. voir le code suivant : le question 2 demande juste la déclaration (prototypes), les suivantes le code.

```
class KeyId {                                                    // Q1
private:                                                         // Q1
    uint32_t            UniqueId;                                // Q1
    uint32_t            Key;                                     // Q1
    static uint32_t     LastId;                                  // Q1 + ci-après
public:                                                          // Q1
    KeyId(uint32_t k) : Key(k), UniqueId(LastId++) {}            // Q1+Q3
    inline uint32_t get_Key() const { return Key; }              // Q1+Q4
    inline uint32_t get_Id() const { return UniqueId; }          // Q1+Q4
    inline void set_Key(uint32_t k) { Key = k; }                 // Q1+Q4
    // Q3: pas de setter pour l'UniqueId car il ne doit pas changer
    KeyId(const KeyId &k) : Key(k.Key), UniqueId(LastId++) {}    // Q1+Q5
    KeyId& operator=(const KeyId &k) {                           // Q1+Q6
        if (this != &k) Key = k.Key;  // on conserve l'Id
        return *this;
    }
    // Q6: oui il faut faire quelque chose, sinon les objets temporaires
    //     peuvent créer et copie des UniqueId
    KeyId(KeyId &&k) = delete;                                   // Q7
    KeyId& operator=(KeyId &&k) = delete;                        // Q7
    friend ostream& operator<<(ostream& os, const KeyId& a);     // pour tests
};
```

```
uint32_t    KeyId::LastId = 0;                                   // Q1 !!!!
// déclaration externe à la classe
ostream& operator<<(ostream& os, const KeyId& a) {
  os << "(" << a.UniqueId << "," << a.Key << ")";
  return os;
}
```

8. L'objet ne peut jamais être passé par copie ou retourné par copie, donc toujours passé par référence ou pointeur (interdit la production d'objet temporaire de type KeyId).
9. Il serait possible d'avoir une autre approche : celle d'autoriser les objets temporaires, et de voler les identifiants des rvalues. Souci : un cast en rvalue (std : :move) permettrait de voler un identifiant.

```
// le constructeur par déplacement vole la valeur de l'identifiant
KeyId(KeyId &&k) : Key(k.Key), UniqueId(k.UniqueId) {}
// l'assignation par déplacement vole la valeur de l'identifiat
KeyId& operator=(KeyId &&k) {
    if (this != &k) { Key=k.Key; UniqueId=k.UniqueId; };
    return *this;
}
```

Exercice 5: Littéraux

On veut définir une classe Angle tel qu'elle est utilisée pour stocker un angle en radian (unité par défaut).

1. écrire la classe Angle.
2. peut-on construire directement un Angle depuis un double (i.e. écrire Angle a = 0.4) ?

3. ajouter une surcharge de l'opérator << sur le flux de sortie standard afin que son unité soit affichée.
4. modifier la classe permette une conversion direct en double (i.e. sans utiliser de getter),
5. ajouter une méthode permettant d'obtenir l'angle en degré.
6. on veut utiliser les littéraux _deg et _rad afin d'initialiser un angle soit à partir de constante double en degré ou en radia. Écrire les fonctions nécessaires.
7. donner un code qui illustre les fonctionnalités décrites ci-dessus.

Solution:

1. classe `Angle` :

```
// must be before anything else
const double     Pi_d = M_PI;
const float      Pi = static_cast<float>(M_PI);
```

```
class Angle {
private:
  double value;   // stocké en radian
public:
  Angle(double v) : value(v) {}
  // conversion d'un Angle en double: pas besoin de getter! (question 4)
  operator double() const { return value; }
  // l'angle en degré (question 5)
  double deg() { return value * 180.0 / Pi_d; }
  // surcharge pour l'affichage (question 3)
  friend ostream& operator<<(ostream& os, const Angle& a);
};
```

2. le constructeur le permet directement.
3. surcharge de l'opérator <<

```
// question 3
ostream& operator<<(ostream& os, const Angle& a) {
  os << a.value << "rad";
  return os;
}
```

4. littéraux

```
// question 6
Angle operator"" _rad(long double x) {
    return Angle(static_cast<double>(x)); }
Angle operator"" _deg(long double x) {
    return Angle(static_cast<double>(x / 180.L*Pi_ld)); }
```

5. code d'illustration

```
// question 7
Angle a = Pi_d / 2.0;
cout << "a=" << a << endl;
Angle b = 180.0_deg;   // note 180_deg ne fonctionne pas car 180 est un entier
cout << "b=" << b << endl;
Angle c = 180.0_rad;
cout << "b=" << b << endl;
double V1 = a;
double V2 = 3.0 * a; // si conversion en float aussi définie, ceci devient ambigu
double V3 = a.deg() / 6;
cout << "V1=" << V1 << "␣V2=" << V2 << "␣V3=" << V3 << endl;
```