Exercice 1 :

1) Adresse (&), &t%sizeof = 0, pour tous les types élémentaire. Toutes les transactions mémoires se font avec des bits entiers
2)

2 bool	2*1 octets	2 octets	a_1 et a_2
1 short int	1*2 octets	2 octets	b
1 int	1*4 octets	4 octets	c
1 int*	1*8 octets	8 octets	d

Total de 16 octets soit 128 bits.

3) Structure sans trous :

2	4	6	8	10	12	14	16
d				c		b	a_1 et a_2
d				c		a_1 et a_2	b
d				b	a_1 et a_2	c	
d				a_1 et a_2	b	c	
c		b	a_1 et a_2	d			
c		a_1 et a_2	b	d			
b	a_1 et a_2	c		d			
a_1 et a_2	b	c		d			

16 octets au mieux.

4) Pire cas possible :

1	2	4	6	8	10	12	14	16	17	18	20	22	24	26	28	30	32
a_1	vide				d				a_2	vide		c		b	vide		

32 octets au pire. Ici tjrs par paquet de 8 octets, ne peut pas être entre deux paquets.

5) Tableau de structure dans le meilleur cas et le pire :

d	c	b	a_1	a_2	16 octets	d	c	b	a_1	a_2	etc…

a_1	vide	d	a_2	c	b	vide	32 octets	a_1	vide	d	a_2	c	b	vide	etc…

6) Oui, en ILP32 le pointeur passe à 4 octets :

4	4	2	1	1
d	c	b	a_1	a_2

12 octets au mieux.

1	3	4	1	3	4	2	2
a_1	vide	d	a_2	vide	c	b	vide

20 octets au pire.

7) Oui, long / long long int dépend du modèle.
   Solution :
   - type de stockage en C_{11}++ (uint16_t, …)
   - trier les champs par ordre croissant.

Exercice 2 :

1)

```
class Frac{
private:int num, dnum;
//private:int num=0, dnum=1; //Pour les constructeur : valeur par defaut
public:
    //constructeur standart
    Frac(int n, int d): num(n), dnum(d){}
    Frac(int i): num(i), dnum(i) {}
    Frac(): num(0), dnum(1) {}
    //constructeur delegue
    /*  Frac(int n, int d): num(n), dnum(d){}
        Frac(int i): Frac(i,1) {}
        Frac(): Frac(0,1) {}
     */
    //constructeur valeur par defaut
    /*  Frac(int n, int d): num(n), dnum(d) {}
     *  Frac() = default;
    */
};
```

2)

Non, le comportement par défaut suffit, pour le destructeur rien de particulier à définir.

3)

Dans la class Frac :

```
void setNum(int n){num = n;}
void setDnum(int d) {if(d!=0){dnum=d;}} // assert(d!=0) {dnum=d;}
int getNum() {return num;}
int getDnum() {return dnum;}
```

4)

Dans la class Frac :

```
void reduce();
```

Dans le .cpp :

```
void Frac::reduce() {
    int pgcd = Euclide(num,dnum);
    if(pgcd > 1){
        num /= pgcd;
        dnum/= pgcd;
    }
}
```

5)

Dans la class Frac :

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

6)

Dans la class Frac :

```
friend ostream& operator << (ostream &os, const Frac &f){
    os << f.num;
    if(f.dnum != 0){
        os << "/" << f.dnum;
    }
    return os;
}
```

7)

Dans la class Frac :

```
friend Frac operator +(const Frac &f1, const Frac &f2){
    Frac f((f1.num * f2.dnum)+(f2.num * f1.dnum));
    f.reduce();
    return f;
}
```

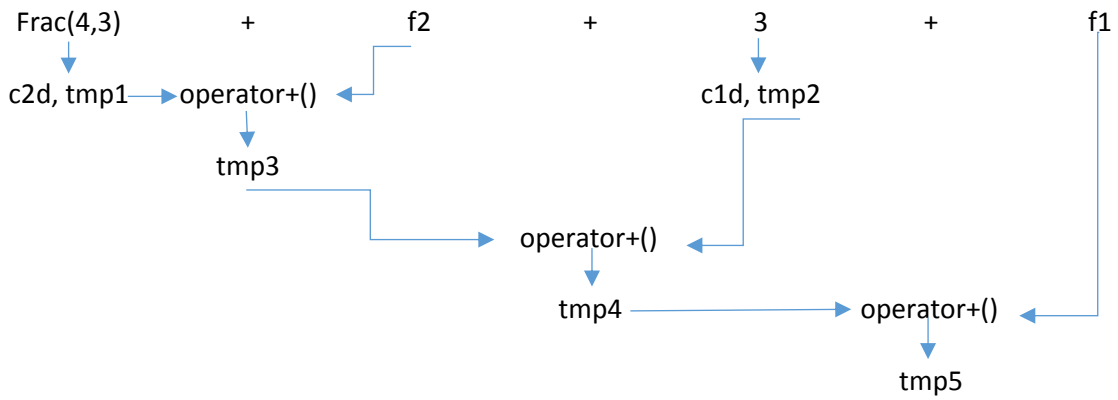
8)

Frac f1 ;

f2 = f1 + 5 ; -> On lance l'auto construction de Frac(5) et utilise operator+() ;

f3 = 5 + f2 ; -> Idem.

9)



c2d => construction avec 2 entiers.

c1d => construction avec 1 entier.

10)

Dans la class Frac :

```
friend bool operator ==(const Frac &f1, const Frac &f2){
    return f1.num*f2.dnum = f2.num*f1.dnum;
}
```

Note : std ::endl ➔endl : = \n + flush (vide le buffer d'affichage)

Exercice 3 :

1)

Dans le .h de la struct Vec2 :

```
struct Vec2{
    float u;
    float v;
    Vec2(float x,float y): u(x), v(y) {}
};
```

2)

Pas de cohérence souhaité sur les composantes du vecteur donc pas de souci, laisser l'accès libre.

3)

Cela retire le constructeur par défaut de Vec2, utiliser Vec2() = default; pour le rétablir.

4)

Non, la copie champs à champs suffit.

5)

Non, car cela retourne un float (objet temporaire de petite taille).

6)

Dans le .h de la struct Vec2 :

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

7)

Dans le .h de la struct Vec2 :

```
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));
}
```

8)

Dans le .h de la struct Vec2 :

```
struct Base2D{
    Vec2 v1;
    Vec2 v2;
    Base2D(const Vec2 &u1, const Vec2 &u2) : v1(u1), v2(u2) {}
};
```

9)

Ce constructeur fait un appelle explicite au constructeur par copie de Vec2.

10)

Dans la struct Base2D :

```
bool isOrtho() {
    return (v1*v2 == 0);
}
```

11)

Dans le .h de la struct Vec2 :

```
std::ostream &operator<<(std::ostream &os, const Vec2 &v){
    return os<<"("<<v.u<<","<<v.v<<")";
}
std::ostream &operator<<(std::ostream &os, const Base2D &b){
    return os<<"["<<b.v1<<","<<b.v2<<"]";
}
```

12)

Dans la struct :

```
Vec2 operator()(float x, float y){
    return x*v1 + y*v2;
}
```