Exercice 1 :

1)

a)

```
int a[4]; =>Tableau statique -> de taille statique ≠ global
```

Variable automatique associée à la portée de la variable (dans le stack).

b)

```
struct A {
    int a[4];
};
sizeof(A) = 4 * sizeof(int);
```

Dans une struct ou une class : intégré à sa taille, alloué ou désalloué avec l'objet associé.

2)

```
struct B {
    int *b; //sizeof(B) = sizeof(int*)
};
int main() {
    int *b1 = new int[10];
//Tableau de 10 int, avec le constructeur par défaut du au new.
    B b2;
    b2.b = new int[10];
//Tableau de 10 int, avec le constructeur par défaut du au new.
    delete[] b2.b;
    delete b1;
    //int *b3 = new int(10) => alloue 1 int initialisé à 10
    //Faire la différence entre () et []
    return 0;
}
```

Alloué explicitement et désalloué explicitement.

3)

Problèmes :

- On doit conserver un pointeur sur la mémoire alloué dynamiquement
  - ⇨ Lors d'une allocation dynamique dans une fonction qui ,e retourne pas le pointeur, alors tous les chemins doivent libérer la mémoire (y compris lors d'exception).
- Tout objet qui alloue de la mémoire doit prévoir sa désallocation
- Toute allocation dynamique peut échouer, il faut prévenir ces échecs.

4)

```
A* a = new A[10];
//Allocation mémoire puis
//le constructeur par défaut est lancé sur tous les éléments du tableau
delete[] a;
//Le destructeur par défaut est lancé sur chaque éléments du tableau puis
//désallocation de la mémoire
```

Exercice 2 :

1)

a)

```
class sStack{
private:
    static const size_t size = 10;
    int stack[size];
    size_t pos;
public:
};
```

b)

```
sizeof(sStack) = sizeof(size_t) + size*sizeof(int)
```

c) Dans la class sStack :

```
public:
    sStack(): pos(0){}
    ~sStack(){} //Ou ~sStack() = default;
  //Mémoire désalloué avec l'objet, car mémoire dans l'objet
```

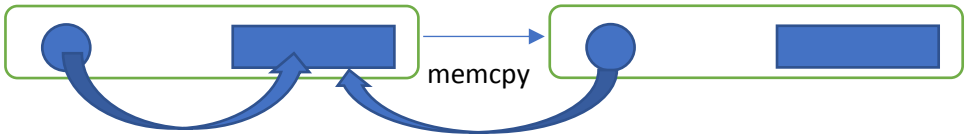
d) Dans la class sStack :

```
public:
    bool isEmpty()const{ return (pos == 0);}
    bool isFull()const{ return (pos == size);}
    size_t size()const {return size;}
    //Mettre const si la fonction ne modifie pas l'objet
    bool Push(const int v){
        if(isFull()){
            return false;
        }
        else{
            stack[pos++] = v;
            return true;
        }
    }
    bool Top(int &v){
        if(isEmpty()){
            return true;
        }else{
            v = stack[pos-1];
            return false;
        }
    }
    bool Pop(int &v){
        if(isEmpty()){
           return false;
        }else{
           v = stack[--pos];
           return true;
        }
    }
```

e) Dans la class sStack :

```
sStack(const sStack &arg){
    if(pos){
        memcpy(stack, arg.stack, arg.pos* sizeof(int));
    }
}
```

memcpy =>Que sur des BIT (faire très attention), copie brut des données, pointeur = même pointeur.



Utilisation de memcpy sur des pointeurs

f) Dans la class sStack :

```
sStack &operator=(const sStack &arg){
    if(&arg != this){
        pos = arg.pos;
        if(pos){
            memcpy(stack, arg.stack, arg.pos* sizeof(int));
        }
    }
    return *this;
}
```

2)

a)

```
class dStack{
private:
    static const size_t defsize = 4; //Taille par défaut
    const size_t size;
    int *stack;
    size_t pos;
public:
};
```

b)

```
sizeof(dStack) = 2*sizeof(size_t) + sizeof(int*)
```

c)

- Sa taille ne dépend pas de la mémoire allouée.
- La mémoire doit être allouée et désallouée explicitement.

d) Dans la class dStack :

```
public:
    dStack(size_t n): size(n), stack(new int[n]), pos(0) {}
    dStack(): dStack(defsize) {} //Constructeur délégué
    ~dStack(){
        delete[] stack;
        stack = nullptr;
    }
```

e) Dans la class dStack :

```
public:
    dStack(const dStack &arg): dStack(arg.size) {
        pos = arg.pos;
        if(pos){
            memcpy(stack, arg.stack, arg.pos* sizeof(int));
        }
    }
    dStack &operator=(const dStack( &arg)){
        assert(arg.pos<=size);
        if(this != &arg){
            pos = arg.pos;
            if(pos){
                memcpy(stack, arg.stack, arg.pos* sizeof(int));
            }
        }
        return *this;
    }
```

3)

a)

Permet d'associer l'allocation mémoire à la durée de vie de l'objet (unique_ptr).

b)

```
class vStack {
private:
    static const size_t defsize = 4;
    unique_ptr<int[]> stack;
    const size_t size;
    size_t pos;
public:
};
```

c) Dans la class vStack :

```
public:
    vStack(size_t n): size(n), stack(new int[n]), pos(0) {}
    vStack(): vStack(defsize) {} //Constructeur délégué
    ~vStack(){} //Ou : ~vStack() = default;
```

d) Dans la class vStack :

```
public:
    vStack(const vStack &arg): vStack(arg.size) {
        pos = arg.pos;
        if(pos){
            memcpy(stack.get(), arg.stack.get(), arg.pos* sizeof(int));
        }
    }
    vStack &operator=(const vStack( &arg)){
        assert(arg.pos<=size);
        if(this != &arg){
            pos = arg.pos;
            if(pos){
                memcpy(stack.get(), arg.stack.get(), arg.pos* sizeof(int));
            }
        }
        return *this;
    }
```

=>Utilisation du get() dans le memcpy.

Exercice 3 :

1)

Une variable automatique ne peut pas être déplacée, cependant sa valeur peut être copié.

2)

Uniquement la propriété d'un pointeur (ou d'une ressource) qui pointe sur une zone allouée dynamiquement (ou danger si elle est dans le stack). La propriété s'acquiert par copie du pointeur, le propriétaire initial ne doit plus pouvoir accéder au pointeur déplacé (perte de propriété).

3)

Les objets temporaire (ou non nommés) typiquement des rvalues.

4)

La qualification && indique qu'un paramètre est une rvalues.

5)

Pour un objet non temporaire, marquer qu'il peut être déplacé en le transformant en rvalues, et donc que cet objet est en fin de vie. Ce qu'implique de ne pas réutiliser après le déplacement sinon pour le réaffecter.

6)

Un déplacement doit toujours laisser un objet dans un état cohérent, ou du-moins assez cohérent afin que le destructeur fonctionne correctement.

7)

Au moment de sa création (constructeur par déplacement) ou lors d'une assignation (assignation par déplacement)

8)

Non, équivalent de A a(12) .

9)

Non, une élision de copie est effectuée.

Exercice 4 :

1)

a)

A(A &&v) => Constructeur par déplacement.

A &operator=(A &&v) => Assignation par déplacement.

b) Dans la class sStack :

```
sStack(const sStack &arg): pos(arg.pos){
    for(size_t i = 0; i < arg.pos; ++i){
        stack[i] = arg.stack[i];
    }
}
sStack(sStack &&arg): pos(arg.pos) {
    for(size_t i = 0; i < arg.pos; ++i){
        stack[i] = move(arg.pos);
    }
 }
 sStack &operator=(const sStack &arg){
     if(this != &arg){
         pos = arg.pos;
         for(size_t i = 0; i < arg.pos; ++i){
             stack[i] = arg.stack[i];
         }
     }
     return *this;
 }
 sStack &operator=(sStack &&arg){
     if(this != &arg){
         pos = arg.pos;
         for(size_t i = 0; i < arg.pos; ++i){
             stack[i] = move(arg.stack[i]);
         }
     }
     return *this;
 }
```

c) Dans la class dStack :

```
dStack(const dStack &arg): stack(new A[arg.size]), size(arg.size),
pos(arg.pos) {
    for(size_t i = 0; i < arg.pos; ++i){
        stack[i] = arg.stack[i];
    }
}
dStack(dStack &&arg): stack(arg.stack), size(arg.size), pos(arg.pos) {
    arg.stack = nullptr;//Pointeur originel mi a null
    const_cast<size_t &>(arg.size) = 0;
    arg.pos = 0;
}
dStack &operator=(dStack &&arg){
    assert(arg.pos <= size);
    if(this != &arg){
        swap(const_cast<size_t &>(size), const_cast<size_t &>(arg.size));
        swap(pos, arg.pos);
        swap(stack, arg.stack);
    }
    return *this;
}
dStack &operator=(const dStack &arg){
    //...Standart...//
}
```

d)

Aucun swap est définie pour unique_ptr.

e)

Liste allouée, plus, construction dans le new ou en statique.

2)

a)

Construction en même temps que l'allocation de l'objet.

b)

```
A *malloc_A(size_t n){
    return reinterpret_cast<A*>(::operator new(n* sizeof(A)));
}
```

c) Dans la class dStack :

```
dStack(size_t n): size(n), stack(malloc_A(n)), pos(0) {}
```

d) Dans la class dStack :

```
dStack(const dStack &arg): stack(malloc_A(n)), size(arg.size),
pos(arg.pos){
    for(size_t i = 0; i < arg.pos; ++i){
        new(&stack[i].A(arg.stack[i]));
    }
}
```