Exercice 1 :

1)

```
template <class T> class List{
private:
    struct Cell{
        T val;
        Cell *nxt;
    };
};
```

2) Dans struct Cell :

```
explicit Cell(const T &v): val(v), nxt(nullptr) {}
Cell(const T &v, Cell *t) : val(v), nxt(t) {}
```

3) Dans la section private de List :

a)

```
Cell* CellInsert(Cell *prv, Cell *nxt, const T &v){
    Cell *c = new Cell(v,nxt);
    assert(c != nullptr);//A gerer
    if(prv != nullptr){
        prv->nxt = c;
    }
    return c;
}
```

b)

```
Cell* CellDelete(Cell *prv, Cell *crv, Cell *nxt){
    if(prv != nullptr){
        prv->nxt = nxt;
    }
    delete crv;
    crv = nullptr;
    if(prv != nullptr){
        return prv;
    }else{
        return nxt;
    }
}
```

4)

a) Dans la section private de List :

```
Cell *first;
Cell *rend;
```

b) Dans la section publique de List :

```
List(): first(nullptr), rend(nullptr) {}
```

5)

a) Dans la section publique de List :

```
~List();
```

b) A l'extérieure de List :

```
template<class T>List<T>::~List() {
    Cell *crv = first->nxt;
    while (crv != nullptr){
        nxt = crv->nxt;
        delete crv;
        crv = nxt;
    }
    *first = nullptr;
    *rend = nullptr;
}
```

6)

a) Dans la section publique de List :

```
List(const List &L) {}//Constructeur par copie
```

b) A l'extérieure de List :

```
template <class T>List<T>::List(const List &L) {
    if(L.first == nullptr){
        first = nullptr;
    }else{
        first = new Cell(L.first->val, nullptr);
        Cell *crv = first;
        Cell *crvc = L.first;
        while(crvc->nxt != nullptr){
            crvc = crvc->nxt;
            crv->nxt = new Cell(crvc->val, nullptr)
            crv = crv->nxt;
        }
        rend = first;
    }
}
```

7)

a) Dans la section publique de List :

```
void Add(const T &v);
```

b) A l'extérieure de List :

```
template <class T>void List<T>::Add(const T &v) {
    if(first == nullptr){
        first = CellInsert(nullptr, nullptr, v);
    }else if(first->val > v){
        first = CellInsert(nullptr, first, v);
    }else{
        Cell *crv = first;
        Cell *nxt = first->nxt;
        while ((nxt != nullptr) && (nxt->val < v)){
            crv = nxt;
            nxt = nxt->nxt;
        }
        CellInsert(crv, nxt, v);
    }
}
```

8)

a) Dans la section publique de List :

```
void Del(const T &v);
```

b) A l'extérieure de List :

```
template <class T>void List<T>::Del(const T &v) {
    if(first == nullptr){
        return;
    }
    if(first->val == v){
        first = CellDelete(nullptr, first, first->nxt);
    }else if(first->nxt != nullptr){
        Cell *prv = first;
        Cell *crv = first->nxt;
        Cell *nxt = crv->nxt;
        while ((nxt != nullptr) && (crv->val < v)){
            prv = crv;
            crv = nxt;
            nxt = nxt->nxt;
        }
        if(crv->val == v){
            CellDelete(prv, crv, nxt);
        }else if((nxt != nullptr) && (nxt->val == v)){
            CellDelete(crv, nxt, nullptr);
        }
    }
}
```

9) Dans la section publique de List :

```
bool getVal(T &v){
    if(rend == nullptr){
        return false
    }
    v = rend->val;
    return true;
}
int length(void){
    if(first == nullptr){
        return 0;
    }else{
        Cell *crv = first;
        Cell *nxt = first->nxt;
        int cpt = 0;
        while (nxt != nullptr){
            nxt = nxt->nxt;
            cpt++;
        }
        return cpt;
    }
}
```

10) Dans la section publique de List :

```
inline void Rewind(void){
    rend = first;
}
inline void Next(void){
    if(rend != nullptr){
        rend = rend->nxt;
    }
}
inline bool isEOL(void){
    return (rend == nullptr);
}
```

11) Trivial

12)

a) Dans la section publique de List :

```
template <class U>friend ostream &operator<<(ostream &out, const List<U>
&a);
```

b) A l'extérieure de List :

```
template <class T> ostream &operator<<(ostream &out, const List<T> &a){
    typename List<T>::Cell *crv = a.first;
    while (crv != nullptr){
        out << crv->val << " ";
        crv = crv->nxt;
    }
    return out;
}
```

Exercice 2 :

1)

a) Dans la struct Cell :

```
explicit  Cell(T &&v): val(move(v)), nxt(nullptr) {}
Cell(T &&v, Cell *t): val(move(v)), nxt(t) {}
```

b) Dans la struct Cell :

```
Cell* CellInsert(Cell *prv, Cell *nxt, T &&v){
    Cell *c = new Cell(move(v), nxt);
    assert(c != nullptr);//A gerer
    if(prv != nullptr){
        prv->nxt = c;
    }
    return c;
}
```

c) Dans la struct Cell :

```
void Add(T &&v){
    if(first == nullptr){
        first = CellInsert(nullptr, nullptr, move(v));
    }else if(first->val > v){
        first = CellInsert(nullptr, first, move(v));
    }else{
        Cell *crv = first;
        Cell *nxt = first->nxt;
        while ((nxt != nullptr) && (nxt->val < v)){
            crv = nxt;
            nxt = nxt->nxt;
        }
        CellInsert(crv, nxt, move(v));
    }
}
```

d) Dans la section publique de List :

```
List(List &&L): List(){
    std::swap(first, L.first);
    swap(rend, L.rend);
}
```

2)

a) Dans la struct Cell :

```
template <class U> explicit Cell(U &&v): val(forward<T>(v)), nxt(nullptr){}
```

b) Dans la struct Cell :

```
template <class U> Cell* CellInsert(Cell *prv, Cell *nxt, U &&v){
    Cell *c = new Cell(forward<T>(v), nxt);
    assert(c != nullptr);//A gerer
    if(prv != nullptr){
        prv->nxt = c;
    }
    return c;
}
```

c) Dans la struct Cell :

```
template <class U>void Add(U &&v){
    if(first == nullptr){
        first = CellInsert(nullptr, nullptr, forward<T>(v);
    }else if(first->val > v){
        first = CellInsert(nullptr, first, forward<T>(v);
    }else{
        Cell *crv = first;
        Cell *nxt = first->nxt;
        while ((nxt != nullptr) && (nxt->val < v)){
            crv = nxt;
            nxt = nxt->nxt;
        }
        CellInsert(crv, nxt, move(v))forward<T>(v)
}
```

Exercice 3 :

1)

```
template <class T, size_t size> class StaticStack{
private:
    T stack[size];
    size_t pos;
public:
};
```

2)

```
StaticStack<int,10>
```

3) Dans la section publique de StaticStack :

```
friend ostream &operator<<(ostream &os, const StaticStack<T,size> &s){
    for(const T &x : s.stack){
        cout << x << " ";
    }
    return os;
}
```

4)

```
StaticStack<int,2>;

StaticStack<int,4>;
```

Le type de S1 est différent de S2 car la taille fait partie du type. Afin de copier les données interne, ces class doivent être friend.

Les friend template ne peuvent pas être des spécialisations partielles, il faut limiter les instanciations possibles des template.

5) friend template complet :

```
template <class T2, size_t size2> friend StaticStack;
```

a) Version 1 :

```
template <class T2, size_t size2> StaticStack(const StaticStack<T2, size2>
&a): pos(min(size, size.pos)){
    static_assert(is_name<T, T2>::value);
    for(int i=0; i<pos; ++i){
        stack[i] = a.stack[i];
    }
}
```

b) Version 2 :

```
template <size_t size2>StaticStack(const StaticStack<T,size2> &a):
pos(min(size, size.pos)){
    static_assert(is_name<T, T2>::value);
    for(int i=0; i<pos; ++i){
        stack[i] = a.stack[i];
    }
}
```