Шаблоны:

Одиночка

```
template<typename T>

class Singleton

{

public:

static T *ptr;

protected:

Singleton();

public:

static T& instance()

{

returnptr?*ptr:*(ptr = new T);

}

private:

Singleton(Singleton<T>const&);

Singleton<T>& operator=(Singleton<T>const&);

};

template<T>

T* Singleton<T>::ptr=0;
```

Хранитель

```
template<typename T>
class Holder;
template<typename T>
classTrule {
private:
      T* ptr;
public:
      Trule(Holder<T>& h) {ptr = h.release();}
      ~Trule() {delete ptr;}
private:
      Trule(Trule<T>&);
      Trule<T>& operator =(Trule<T>&);
      friend class Holder<T>;
};
template<typename T>
class Holder {
private:
      T* ptr;
public:
      Holder() : ptr(0) {}
      explicit Holder(T* p) : ptr(p) {}
      ~Holder() {delete ptr;}
      T& operator *() const {return *ptr;}
      T&get() const {return *ptr;}
      T* operator ->() const {return ptr;}
      void exchange(Holder<T>& h);
Holder(Trule<T>const& t) {
      ptr = t.ptr;
      const_cast<Trule<T>&>(t).ptr = 0;
}
Holder<T>& operator =(Trule<T>const& t) {
      deleteptr;
      ptr = t.ptr;
      const_cast<Trule<T>&>(t).ptr = 0;
      return *this;
}
T* release() {
      T* p = ptr;
      ptr = 0;
      return p;
}
private:
      Holder(Holder<T>const&);
      Holder<T>& operator =(Holder<T>const&);
};
```

Паттерны:

Компоновщик

```
class Unit {
public:
	virtualintgetStrength() = 0;
	virtual void addUnit(Unit* p) {}
	virtual ~Unit() {}
};
class Archer: public Unit {
public:
	virtualintgetStrength() {return 1;}
};class Infantryman: public Unit {
public:
	virtualintgetStrength(){return 2;}
};
classCompositeUnit: public Unit {
public:
	intgetStrength() {
		int total = 0;
		for(inti=0; i<c.size(); ++i)
		total += c[i]->getStrength();
		return total;
	}
	voidaddUnit(Unit* p){c.push_back(p);}
	~CompositeUnit() {
		for(inti=0; i<c.size(); ++i)
		delete c[i];
	}
private:
	std::vector<Unit*> c;
};
CompositeUnit* createLegion() {
	CompositeUnit* legion = new CompositeUnit;
```

```
      for (inti=0; i<3000; ++i)
      legion->addUnit(new Infantryman);
      for (inti=0; i<1200; ++i)
      legion->addUnit(new Archer);
      return legion;
}
```

Абстрактнаяфабрика

```
class Infantryman{
public:
      virtual void info() = 0;
      virtual ~Infantryman() {}
};
class Archer{
public:
      virtual void info() = 0;
      virtual ~Archer() {}
};
classRomanInfantryman: public Infantryman{
public:
      void info() { cout<< "RomanInfantryman" <<endl;}
};
classRomanArcher: public Archer{
public:
      void info() { cout<< "RomanArcher" <<endl;}
};
classArmyFactory {
public:
      virtual Infantryman* createInfantryman() = 0;
      virtual Archer* createArcher() = 0;
      virtual ~ArmyFactory() {}
};
classRomanArmyFactory: public ArmyFactory {
public:
      Infantryman* createInfantryman() { return new RomanInfantryman; }
      Archer* createArcher() { return new RomanArcher; }
};
class Army {
public:
      ~Army() {
            inti;
```

```
        for(i=0; i<vi.size(); ++i) delete vi[i];

        for(i=0; i<va.size(); ++i) delete va[i];

    }

    void info() {

        inti;

        for(i=0; i<vi.size(); ++i) vi[i]->info();

        for(i=0; i<va.size(); ++i) va[i]->info();

    }

    vector<Infantryman*> vi;

    vector<Archer*>va;

};

class Game {

public:

    Army* createArmy(ArmyFactory& factory ) {

        Army* p = new Army;

        p->vi.push_back( factory.createInfantryman());

        p->va.push_back( factory.createArcher());

        return p;

    }

};

    int main(){

        Game game;

        RomanArmyFactoryra_factory;

        Army * ra = game.createArmy(ra_factory);

        cout<< "Roman army:" <<endl;

        ra->info();

    }
```

Адаптер

```
classFahrenheitSensor {
public:
      floatgetFahrenheitTemp() {float t = 32.0;return t;}
};
class Sensor {
public:
      virtual ~Sensor() {}
      virtual float getTemperature() = 0;
};
class Adapter : public Sensor {
public:
      Adapter(FahrenheitSensor* p ) : p_fsensor(p) {}
      ~Adapter() {delete p_fsensor;}
      floatgetTemperature() {
            return (p_fsensor->getFahrenheitTemp()-32.0)*5.0/9.0;
      }
private:
      FahrenheitSensor* p_fsensor;
};
```

Подписчик-издатель

```
class Observer {

public:

virtual void update(int value) = 0;

};

class Subject {

intm_value;

vectorm_views;

public:

void attach(Observer *obs) { m_views.push_back(obs); }

voidset_val(int value) { m_value = value; notify(); }

void notify() {

for (inti = 0; i<m_views.size(); ++i)

m_views[i]->update(m_value);

}

};

classDivObserver: public Observer {

intm_div;

public:

DivObserver(Subject *model, int div) {

model->attach(this);

m_div = div;

}

void update(int v) { cout<< v / m_div<< 'n'; }

};

int main() {

Subject subj;

DivObserverdivObs1(&subj, 4);

DivObserverdivObs2(&subj, 3);

subj.set_val(14);

}
```

Мост

```
classDrawingAPI {
public:
      virtual void drawCircle(double x, double y, double radius) = 0;
      virtual ~DrawingAPI() {}
};
class DrawingAPI1: public DrawingAPI {
public:
      DrawingAPI1() {}
      virtual ~DrawingAPI1() {}
      voiddrawCircle(double x, double y, double radius) {
            printf("nAPI1 at %f:%f %fn", x, y, radius);
      }
};
class Shape {
public:
      virtual void draw()= 0;
      virtual void resizeByPercentage(double pct) = 0;
      virtual ~Shape() {
      }
};
classCircleShape: public Shape {
public:
CircleShape(double x, double y, double radius, DrawingAPI&drawingAPI) :
x(x), y(y), radius(radius), drawingAPI(drawingAPI) {}
virtual ~CircleShape() {}
void draw() { drawingAPI.drawCircle(x, y, radius); }
voidresizeByPercentage(double pct) { radius *= pct; }
private:
      double x, y, radius;
      DrawingAPI&drawingAPI;
};
DrawingAPI1 api1;
```

```
CircleShapec1(1, 2, 3, api1);
Shape* shapes[1];
shapes[0] = &c1;
shapes[0]->resizeByPercentage(2.5);
shapes[0]->draw();
```

Библиотечные шаблонные классы

Вектор

```
class ArrayBase {
public:
        ArrayBase () { }
        virtual ~ArrayBase () { }

        intgetsize() const {
                return _size;
        }


protected:
        int _size;
        staticconstintDefaultSize = 20;
};


template<typename T>
classArrayT : publicArrayBase {
public:

        // Конструкторы
        ArrayT( int n = DefaultSize );
        ArrayT( const T*, intsz );
        ArrayT( constArrayT<T>& );

        // Деструктор
        ~ArrayT ( );

        // Копированиемассива
        ArrayT<T>&operator= ( constArrayT<T>& );

        // Перегрузкаоператоровсравнения
        booloperator == ( constArrayT<T>& ) const;

        // Перегрузкаоператора []
        T&operator [] ( int n ) {
                if (n<_size)
                        return _arr[n];
                else
                        throwRangeError(1);
        };
        const T&operator [] ( int n ) const {
                if (n<_size)
                        return _arr[n];
                else
                        throwRangeError(1);
        };
private:
        T* _arr;
        voidarr_resize( int );
};

template<typename T>
voidArrayT<T>::arr_resize( int n ) {
        ArrayT<T> *wrk = newArrayT<T>(n);
        if (wrk == nullptr) {
                throwMemoryError(0);
        }
        inti, s1;
        if (_size > n)
                s1 = n;
        else
                s1 = _size;

        for (i=0;i<s1;++i) {
```

```
        wrk->_arr[i] = _arr[i];
    }

    this->~ArrayT<T>(   );
    this->_arr = wrk->_arr;
    this->_size = wrk->_size;
}

template<typename T>
boolArrayT<T>::operator== ( constArrayT<T>&to_cmp) const {
    bool res = true;
    if (to_cpy._size != _arr._size)
        res = false;
    else {
        inti;
        for (i=0;i<to_cpy._size;i++) {
            if (_arr[i] != to_cpy[i])
                res = false;
        }
    }
    return res;
}


    // Конструкторы
template<typename T>
ArrayT<T>::ArrayT ( int n = DefaultSize )
{
    if (n <= 0) {
        throwRangeError(3);
    }
    if ( (_arr = new (std::nothrow) T[n]) == nullptr ) {
        throwMemoryError(0);
    }
    inti;
    for (i=0;i<n;++i)
        _arr[i] = T();
    _size = n;
};

template<typename T>
ArrayT<T>::ArrayT ( const T* t_arr, intsz )
{
    _size = sz;
    int i;

    if ( (_arr = new (std::nothrow) T[_size]) == nullptr) {
        throwMemoryError(0);
    }

    for (i=0;i<sz;++i) {
            _arr[i] = t_arr[i];
    }
};

template<typename T>
ArrayT<T>::ArrayT ( constArrayT<T>&t_arr )
{
    _size = t_arr.getsize();
    inti;

    if ( (_arr = new (std::nothrow) T[_size]) == nullptr ) {
        throwMemoryError(1);
    }

    for (i=0;i<_size;++i) {
        _arr[i] = t_arr[i];
    }
};
```

```
       // Деструктор
template<typename T>
ArrayT<T>::~ArrayT ( )
{
       delete [] _arr;
};
```

Множество

```
classBaseSet
{
    public:
        BaseSet(){}
        virtual ~BaseSet(){}
};

template<typenameT>
classSet: publicBaseSet
{
    public:
        Set();
        Set(int, ...);
        Set(constSet<T>&);

        Set<T>Cross(constSet<T>&) const;
        Set<T>Cross(constT&) const;
        constSet<T>& operator *= (constSet<T>&);
        constSet<T>& operator *= (constT&);
        Set<T> operator * (constSet<T>&);

        Set<T>Combine(constSet<T>&) const;
        Set<T>Combine(constT&) const;
        constSet<T>& operator += (constSet<T>&);
        constSet<T>& operator += (constT&);
        Set<T> operator + (constSet<T>&);

        Set<T>Difference(constSet<T>&) const;
        Set<T>Difference(constT&) const;
        constSet<T>& operator -= (constSet<T>&);
        constSet<T>& operator -= (constT&);
        Set<T> operator - (constSet<T>&);

        bool Inside(constT&) const;
        bool Inside(constSet<T>&) const;

        bool Equal (constSet<T>&) const;
        bool operator == (constSet<T>&) const;
        bool operator != (constSet<T>&) const;
        bool operator ! () const;

        Set<T>& operator = (constSet<T>&);
        operatorint() const;
        int Size() const;

        void Clear();

    private:
        Array<T> set;
};

// setDef.h

#include"set.h"

template<typenameT>
Set<T>::Set() {}

template<typenameT>
Set<T>::Set(intcount, ...) : set(Array<T>())
{
    int *ptr = &count;
    ptr++;

    T *cur_ptr = (T*)(ptr);
    for (inti = 0; i<count; i++, cur_ptr++)
        set.Add(*cur_ptr);
```

```
}

template<typenameT>
Set<T>::Set(constSet<T>&array) : set(Array<T>(array.set)) {}

template<typenameT>
intSet<T>::Size() const
{
       returnset.Length();
}

template<typenameT>
voidSet<T>::Clear()
{
       for (inti = Size(); i> 0; i--)
              set.Del(0);
}

template<typenameT>
boolSet<T>::Inside(constT&elem) const
{
       returnset.Search(elem) != NO_RESULTS;
}

template<typenameT>
boolSet<T>::Inside(constSet<T>&elements) const
{
       bool result = true;

       for (inti = 0; i<elements.Size() && result; i++)
              result = Inside<T>(elements[i]);

       return result;
}

template<typenameT>
Set<T>Set<T>::Difference(constT&elem) const
{
       Set<T>temp(*this);
       temp -= elem;

       return temp;
}

template<typenameT>
Set<T>Set<T>::Difference(constSet<T>&elements) const
{
       Set<T>temp(*this);
       for (inti = 0; i<elements.Size(); i++)
              temp -= elements.set[i];

       return temp;
}

template<typenameT>
constSet<T>&Set<T>::operator -= (constT&elem)
{
       intpos = set.Search(elem);
       if(pos != NO_RESULTS)
              set.Del(pos);

       return *this;
}

template<typenameT>
constSet<T>&Set<T>::operator -= (constSet<T>&elements)
{
       for (inti = 0; i<elements.Size(); i++)
              *this -= elements.set[i];
```

```
        return *this;
}

template<typenameT>
Set<T>Set<T>::operator - (constSet<T>&elements)
{
        Set<T>temp(*this);
        temp -= elements;

        return temp;
}

template<typenameT>
Set<T>Set<T>::Cross(constSet<T>&elements) const
{
        Set<T>temp(*this);
        temp *= elements;

        return temp;
}

template<typenameT>
Set<T>Set<T>::Cross(constT&elem) const
{
        Set<T>temp(*this);
        temp *= elem;

        return temp;
}

template<typenameT>
constSet<T>&Set<T>::operator *= (constSet<T>&elements)
{
        if (!elements)
                Clear();

        for (inti = 0; i<set.Length(); i++)
                if(!Inside(elements.set[i]))
                        *this -= set[i--];

        return *this;
}

template<typenameT>
constSet<T>&Set<T>::operator *= (constT&elem)
{
        if (!Inside(elem) && Size())
                *this -= set[0];

        return *this;
}

template<typenameT>
Set<T>Set<T>::operator * (constSet<T>&elements)
{
        Set<T>temp(*this);
        temp *= elements;

        return temp;
}

template<typenameT>
Set<T>Set<T>::Combine(constT&elem) const
{
        Set<T>temp(*this);

        return temp += elem;
}
```

```
template<typenameT>
Set<T>Set<T>::Combine(constSet<T>&elements) const
{
	Set<T>temp(*this);

	for (inti = 0; i<elements.Size(); i++)
		temp += elements.set[i];

	return temp;
}

template<typenameT>
constSet<T>&Set<T>::operator += (constSet<T>&elements)
{
	for (inti = 0; i<elements.Size(); i++)
		*this += elements.set[i];

	return *this;
}

template<typenameT>
constSet<T>&Set<T>::operator += (constT&elem)
{
	if (!Inside(elem))
		set.Add(elem);

	return *this;
}

template<typenameT>
Set<T>Set<T>::operator + (constSet<T>&elements)
{
	Set<T>temp(*this);
	temp += elements;

	return temp;
}

template<typenameT>
boolSet<T>::Equal (constSet<T>&elements) const
{
	bool flag = Inside(elements);
	return flag &&elements.Inside(this);
}


template<typenameT>
boolSet<T>::operator == (constSet<T>&elements) const
{
	return Equal(elements);
}

template<typenameT>
boolSet<T>::operator != (constSet<T>&elements) const
{
	return !Equal(elements);
}

template<typenameT>
boolSet<T>::operator ! () const
{
	return !Size();
}

template<typenameT>
Set<T>::operatorint() const
{
	return Size();
}
```

```
template<typenameT>
Set<T>&Set<T>::operator = (constSet<T>&elements)
{
        Clear();
        for (inti = 0; i<elements.Size(); i++)
                *this += elements.set[i];

        return *this;
}

#endif
```

Список

```
class List
{
private:
    struct ELEMENT {
        charval[21];
        ELEMENT *next, *prev;
    };
    ELEMENT *head, *curr;

public:
    List();
    ~List();

    int Add(constchar*);
    int Del();
    int Get(char*);

    intMoveHead();
    intMoveNext();
    intMovePrev();

    intisHead();
    intisTail();

    void Sort(int);
};

//List.cpp
#include "List.h"
#include <iostream>
#include <stdlib.h>
usingnamespacestd;

List::List()
{
    head=curr=NULL;
}

List::~List()
{
    while(head){
        curr=head;
        head=head->next;
        free(curr);
    }
    head=curr=NULL;

}
int List::Add(constchar*val)
{
  ELEMENT *tmp=(ELEMENT *)malloc(sizeof(ELEMENT));
  if(!tmp)return0;
  if(!head){
    head=tmp; head->prev=NULL;
  }else{
    if(!curr)curr=head;
    while(curr->next)curr=curr->next;
    curr->next=tmp;
    tmp->prev=curr;
    tmp->next=NULL;
  }
  strcpy(tmp->val, val);curr=tmp;
  return1;
}

int List::Del()
{
  if(curr==NULL)return0;
  ELEMENT *tmp=curr->prev;
  if(!tmp){
    head=head->next;if(head) head->prev=NULL;
  }else{
    tmp->next=curr->next;
    if(curr->next)curr->next->prev=tmp;
  }
  free(curr);curr=tmp;
  return1;
}

int List::Get(char*val)
{
 if(curr==NULL)return0;
  strcmp(val,curr->val);
  return1;
}

int List::MoveHead()
{
  curr= head;
  if(head ==NULL)return0;
  return1;
}


int List::MoveNext()
{
  if((curr==NULL)||(curr->next==NULL))return0;
  curr=curr->next;
  return1;
}


int List::MovePrev()
{
  if((curr==NULL)||(curr->prev==NULL))return0;
  curr=curr->prev;
  return1;
}



int List::isHead()
{
    return(curr->prev==NULL);
}

int List::isTail()
{
    return(curr->next==NULL);
}


void List::Sort(inttt)
{
    int flag=1;
    while( flag){
        flag=0;
        curr= head;
        while(curr->next){
            intpr=0;
            if((strcmp(curr->val, curr->next->val)>0)&&(tt==0))pr=1;
            if((strcmp(curr->val, curr->next->val)<0)&&(tt==1))pr=1;
            if(pr==1){
                chartmp[21];
                strcpy(tmp, curr->val);
                strcpy(curr->val, curr->next->val);
                strcpy(curr->next->val,tmp);
                flag=1;
            }
            curr=curr->next;
        }
    }
    curr=head;
}
```