**Выполнение**

**Часть 1**

#include <iostream>

using namespace std;

class OverflowException {

public:

    OverflowException() {

        cout << "#ERROR# stack overflow!" << endl;

    }

};

class NoElementsException {

public:

    NoElementsException() {

        cout << "#ERROR# Stack contains no elements!" << endl;

    }

};

template<typename T>

struct Node {

    T x;

    Node \*next;

};

template<typename T>

class AbstractStack {

protected:

    virtual int GetSize(void) const = 0;

    virtual void Push(const T &element) = 0;

    virtual void Pop(T &element) = 0;

    virtual void Peek(T &element) = 0;

};

template<typename T>

class Stack : public AbstractStack<T> {

private:

    Node<T> \*top;

    int size;

public:

    Stack();

    Stack(int size);

    Stack(const Stack &other);

    Stack(Stack &&other);

    ~Stack();

    Stack &operator=(const Stack &other);

    Stack &operator=(Stack &&other);

    friend ostream& operator<<(ostream &out, const Stack &rv){

        Node<T> \*cur = rv.top;

        while (cur!=NULL) {

            out << cur->x << " ";

            cur = cur->next;

        }

        return out;

    }

    int GetSize(void) const override;

    void Push(const T &element) override;

    void Pop(T &element) override;

    void Peek(T &element) override;

    bool EmptyStack(void) {return (top ? false : true);}

    void StackCopy(const Stack &other);

};

template<typename T>

Stack<T>::Stack() {

    Stack::size = 0;

    top = nullptr;

}

template<typename T>

Stack<T>::Stack(int size) {

    Stack::size = size;

    top = nullptr;

}

template<typename T>

Stack<T>::Stack(const Stack &other) {

    StackCopy(other);

}

template<typename T>

Stack<T>::Stack(Stack &&other) {

    StackCopy(other);

    other.top = nullptr;

}

template<typename T>

Stack<T>::~Stack(void) {

    while (top != NULL) {

        Node<T> \*cur = top->next;

        delete top;

        top = cur;

    }

}

template<typename T>

Stack<T> &Stack<T>::operator=(const Stack &other) {

    if (this == &other) {

        return \*this;

    }

    StackCopy(other);

    return \*this;

}

template<typename T>

Stack<T> &Stack<T>::operator=(Stack &&other) {

    if (this == &other) {

        return \*this;

    }

    StackCopy(other);

    other.top = nullptr;

    return \*this;

}

template<typename T>

void Stack<T>::StackCopy(const Stack &other) {

    top = NULL;

    size = other.size;

    Node<T> \*cur = other.top;

    while(cur!=NULL) {

        Push(cur->x);

        cur = cur->next;

    }

}

template<typename T>

int Stack<T>::GetSize(void) const {

    int count = 0;

    Node<T> \*cur = top;

    while (cur!= NULL) {

        cur = cur->next;

        count++;

    }

    return count;

}

template<typename T>

void Stack<T>::Push(const T &element) {

    try {

        if(GetSize() >= size) throw OverflowException();

    }

    catch (OverflowException& ex){return ;}

    Node<T> \*np = new Node<T>;

    np->x = element;

    np->next = top;

    top = np;

}

template<typename T>

void Stack<T>::Pop(T &element) {

    try {

        if (EmptyStack()) throw NoElementsException();

    }

    catch (NoElementsException& ex){return ;}

    element = top->x;

    Node<T> \*cur = top;

    top = top->next;

    delete cur;

}

template<typename T>

void Stack<T>::Peek(T &element) {

    try {

        if (EmptyStack()) throw NoElementsException();

    }

    catch (NoElementsException& ex){return ;}

    element = top->x;

}

int main(int argc, char \*\*argv) {

    Stack<double> sd(2);

    Stack<char> sc(6);

    for(int i=0; i<2; i++) {

        sd.Push(1.5);

    }

    cout << "Stack type <double> = " << sd << endl;

    sc.Push('c');

    sc.Push('e');

    sc.Push('n');

    sc.Push('o');

    sc.Push('G');

    sc.Push('L');

    cout << "Stack type <char> = " << sc << endl;

    char element;

    sc.Pop(element);

    cout << "\nPop() Stack<char> = " << element << endl;

    cout << "Stack <char> = " << sc << endl;

    return 0;

}

**Часть 2**

Стек на основе двунаправленного циклического списка

StackBasedOnBidirectionalCyclicLinkedList

#include <iostream>

using namespace std;

class OverflowException {

public:

    OverflowException() {

        cout << "#ERROR# stack overflow!" << endl;

    }

};

class NoElementsException {

public:

    NoElementsException() {

        cout << "#ERROR# Stack contains no elements!" << endl;

    }

};

template<typename T>

class AbstractStack {

public:

    virtual int GetSize(void) const {return 101;}

protected:

    virtual void Push(const T &element) = 0;

    virtual void Pop(T &element) = 0;

    virtual void Peek(T &element) = 0;

};

template<typename T>

class Stack : public AbstractStack<T> {

private:

    struct Node {

        T x;

        Node \*next;

        Node \*prev;

    };

    Node \*top;

    int size;

public:

    Stack();

    Stack(int size);

    Stack(const Stack &other);

    Stack(Stack &&other);

    ~Stack();

    Stack &operator=(const Stack &other);

    Stack &operator=(Stack &&other);

    friend ostream& operator<<(ostream &out, const Stack &rv){

        Node \*cur = rv.top;

        do {

            out << cur->x << " ";

            cur = cur->next;

        } while (cur != rv.top);

        return out;

    }

    int GetSize(void) const override;

    void Push(const T &element) override;

    void Pop(T &element) override;

    void Peek(T &element) override;

    bool EmptyStack(void) {return (top ? false : true);}

    void StackCopy(const Stack &other);

};

template<typename T>

Stack<T>::Stack() {

    Stack::size = 0;

    top = nullptr;

}

template<typename T>

Stack<T>::Stack(int size) {

    Stack::size = size;

    top = nullptr;

}

template<typename T>

Stack<T>::Stack(const Stack &other) {

    StackCopy(other);

}

template<typename T>

Stack<T>::Stack(Stack &&other) {

    StackCopy(other);

    other.top = nullptr;

}

template<typename T>

Stack<T>::~Stack(void) {

    T buf;

    while (top != NULL) {

        Pop(buf);

    }

}

template<typename T>

Stack<T> &Stack<T>::operator=(const Stack &other) {

    if (this == &other) {

        return \*this;

    }

    StackCopy(other);

    return \*this;

}

template<typename T>

Stack<T> &Stack<T>::operator=(Stack &&other) {

    if (this == &other) {

        return \*this;

    }

    StackCopy(other);

    other.top = nullptr;

    return \*this;

}

template<typename T>

void Stack<T>::StackCopy(const Stack &other) {

    top = NULL;

    size = other.size;

    Node \*cur = other.top;

    do {

        Push(cur->x);

        cur = cur->next;

    } while (cur != other.top);

}

template<typename T>

int Stack<T>::GetSize(void) const {

    if (top == NULL) return 0;

    int count = 0;

    Node \*cur = top;

    do {

        cur = cur->next;

        count++;

    } while (cur != top);

    return count;

}

template<typename T>

void Stack<T>::Push(const T &element) {

    try {

        if(GetSize() >= size) throw OverflowException();

    }

    catch (OverflowException& ex){return ;}

    Node \*np = new Node;

    np->x = element;

    if(top == NULL) {

        np->next = np;

        np->prev = np;

    }

    else {

        np->next = top;

        np->prev = top->prev;

        top->prev = np;

        np->prev->next = np;

    }

    top = np;

}

template<typename T>

void Stack<T>::Pop(T &element) {

    try {

        if (EmptyStack()) throw NoElementsException();

    }

    catch (NoElementsException& ex){return ;}

    element = top->x;

    Node \*pTop = top;

    if((top->next == top) || (top->prev == top)){

        top = nullptr;

    }

    else {

        top->prev->next = top->next;

        top->next->prev = top->prev;

        top = top->next;

    }

    delete pTop;

}

template<typename T>

void Stack<T>::Peek(T &element) {

    try {

        if (EmptyStack()) throw NoElementsException();

    }

    catch (NoElementsException& ex){return ;}

    element = top->x;

}

int main(int argc, char \*\*argv) {

    cout << "Creating Cyclic Stack type <int> SIZE=4\n" << endl;

    Stack<int> si(4);

    cout << "Using Pop() on empty Cyclic Stack type <int>" << endl;

    int elem;

    si.Pop(elem);

    cout << "\nFilling Cyclic Stack type <int> with 5 elements '4'" << endl;

    for(int i=0; i<5; i++) {

        si.Push(4);

    }

    cout << "Cyclic Stack type <int> = " << si << endl;

    Stack<int> si2(si);

    cout << "\nCopy of this Stack = " << si2 << endl;

    si2.Pop(elem);

    cout << "\nPop(): " << elem << " element from Copy of Stack" << endl;

    cout << "Current Copy of Stack = " << si2 << endl;

    si2.~Stack();

    Stack<char> sc(2);

    sc.Push('z');

    sc.Push('Q');

    cout << "\nCyclic Stack type <char> = " << sc << endl;

    AbstractStack<int> &sip = si;

    cout << "\n\nMessage from reference on Abstract class: " << sip.AbstractStack::GetSize() << endl;

    AbstractStack<int> \*Base = &si;

    cout << "\nMessage from pointer on Abstract class: " << Base->AbstractStack::GetSize() << "\n" << endl;

    si.~Stack();

    return 0;

}