МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №8

по дисциплине: Объектно-ориентированное программирование

тема: «**Создание шаблонов классов в C++»**

Выполнил: ст. группы ПВ-233

Ситников Алексей Павлович

Проверил:

Белгород 2025 г.

Вариант 3 (13)

Цель работы: Получение теоретических знаний о шаблонах классов в C++. Получение практических навыков по созданию классов-шаблонов C++.

Двусвязный список:

```
//
// Created by admin on 23.03.2025.
//
#include <iostream>

#ifndef UNTITLED11_DLIST_H
#define UNTITLED11_DLIST_H

template<typename T>
class Dlist;

template<typename T>
class Node {
    friend class Dlist<T>;
    T data;
    long int size;
    Node *nextLeft;
    Node *nextRight;
    Node(T data, Node* nextRight, Node *nextLeft) :data(data),
size(sizeof(T)), nextLeft(nextLeft), nextRight(nextRight) {

    }
    long int getSize(){
        return this->size;
    }
    T getData(){
        return this->data;
    }
    Node<T>* getNextLeft(){
        return this->nextLeft;
    }
    Node<T>* getNextRight(){
        return this->nextRight;
    }
    void setRight(Node* temp){
        this->nextRight = temp;
    }
    void setLeft(Node* temp){
        this->nextLeft = temp;
    }
};

template<typename T>
class Dlist{
    short error;
    int size;
    Node<T> *leftNode;
    Node<T> *rightNode;
    Node<T> *current;
public:
    Dlist() : error(0), size(0){
        this->leftNode = NULL;
```

```
        this->rightNode = NULL;
        this->current = NULL;
    }

    void appendLeft(T data){
        Node<T> *temp = new Node<T>(data, this->leftNode, NULL);
        temp->size = sizeof(T);
        if(temp == NULL){
            this->error = 1;
            return;
        }
        if(this->leftNode == NULL){
            this->current = temp;
            this->rightNode = temp;
        }
        else{
            this->leftNode->setLeft(temp);
        }
        this->size++;
        this->leftNode = temp;
    }

    void appendRight(T data){
        Node<T> *temp = new Node<T>(data, NULL, this->rightNode);
        temp->size = sizeof(T);
        if(temp == NULL){
            this->error = 1;
            return;
        }
        if(this->rightNode == NULL){
            this->current = temp;
            this->leftNode = temp;
        }
        else{
            this->rightNode->setRight(temp);
        }
        this->size++;

        this->rightNode = temp;
    }

    long int len(){
        return this->size;
    }

    ~Dlist(){
        while (true){
            if(this->leftNode == NULL){
                break;
            }
            Node<T> *temp = this->leftNode;
            this->leftNode = temp->getNextRight();
            delete temp;
        }
    }

    void setLeft(){
        this->current = this->leftNode;
    }

    void setRight(){
        this->current = this->rightNode;
    }
```

```
Node<T> *getCurrent(){
    return this->current;
}


int moveCurrentLeft(){
    if(this->current == NULL || this->current->getNextLeft() == NULL){
        return 1;
    }
    this->current = this->current->getNextLeft();
    return 0;
}


int moveCurrentRight(){
    if(this->current == NULL || this->current->getNextRight() == NULL){
        return 1;
    }
    this->current = this->current->getNextRight();
    return 0;
}


void deleteLeft(){
    if(this->leftNode == NULL){
        return;
    }
    Node<T> *temp = this->leftNode;
    this->leftNode = this->leftNode->getNextRight();
    if(this->leftNode != NULL){
        this->leftNode->setLeft(NULL);
    }else{
        this->rightNode = NULL;
    }
    if(this->current == temp){
        moveCurrentRight();
    }
    delete temp;
    this->size--;
}


void deleteRight(){
    if(this->rightNode == NULL){
        return;
    }
    Node<T> *temp = this->rightNode;
    this->rightNode = this->rightNode->getNextLeft();
    if(this->rightNode != NULL){
        this->rightNode->setRight(NULL);
    }else{
        this->leftNode = NULL;
    }
    if(this->current == temp){
        moveCurrentLeft();
    }
    delete temp;
    this->size--;
}


void deleteCurrent(){
    Node<T> *tempL = this->current->getNextLeft();
    Node<T> *tempR = this->current->getNextRight();
```

```
        Node<T> *tempD = this->current;
        if(tempL == NULL && tempR == NULL){
            delete tempD;
            this->leftNode = NULL;
            this->rightNode = NULL;
            this->current = NULL;
        }
        else{
            delete tempD;
            if(tempR != NULL) {
                tempR->setLeft(tempL);
                this->current = tempR;
            }
            if(tempL != NULL) {
                tempL->setRight(tempR);
                this->current = tempL;
            }
            if(tempR == NULL){
                this->rightNode = tempL;
            }
            if(tempL == NULL){
                this->leftNode = tempR;
            }
        }
        this->size--;
    }


    void appendCurrentLeft(T data) {
        if(this->current == NULL){
            return;
        }
        Node<T> *temp = new Node<T>(data, this->current, this->current-
>getNextLeft());
        temp->size = sizeof(T);
        if(this->current->getNextLeft() != NULL){
            this->current->getNextLeft()->setRight(temp);
        }
        this->current->setLeft(temp);
        this->size++;
    }


    void appendCurrentRight(T data) {
        if(this->current == NULL){
            return;
        }
        Node<T> *temp = new Node<T>(data, this->current->getNextRight(),
this->current);
        if(this->current->getNextRight() != NULL){
            this->current->getNextRight()->setLeft(temp);
        }
        this->current->setRight(temp);
        this->size++;
    }
    T getData(){
        return this->current->getData();
    }

    void creatFromArray(T *arr, int count){
        for(int i = 0; i < count; i++){
            appendRight(arr[i]);
        }
    }
```

```
    short getError(){
        return this->error;
    }

    Node<T> *getRight(){
        return this->rightNode;
    }
};



#endif //UNTITLED11_DLIST_H
```

Parser заголовочный:

```
//
// Created by admin on 10.04.2025.
//

#ifndef UNTITLED11_SYNTAXPARSER_H
#define UNTITLED11_SYNTAXPARSER_H
#include "Dlist.h"
#include <string>
#include <sstream>

struct Token;

enum TokenType_ {
    KEYWORD,
    IDENTIFIER,
    NUMBER,
    OPERATOR,
    DELIMITERS,
    STRINGLITERALS,
    COMMENTS,
    SEMICOLON,
    TYPE
};

class ParserSplit{
    int i; //строка
    int countIf;
    int countBegin;


public:
    ParserSplit() noexcept : i(1), countIf(0), countBegin(0){}
    void var(Dlist<Token> &list);
    void const_(Dlist<Token> &list);
    void begin(Dlist<Token> &list);
    void write(Dlist<Token> &list);
    void while_(Dlist<Token> &list);
    void if_(Dlist<Token> &list);
    void identifier(Dlist<Token> &list);
    void else_(Dlist<Token> &list);
    void end(Dlist<Token> &list);
    std::string tokenTypeToString(TokenType_ type);
};
```

```
class SyntaxParser{
public:
    void lexer(const std::string& code, Dlist<Token> &list);
    void parser(Dlist<Token> &list);
    int dataInArray(std::string value, Dlist<std::string> &arr);

};



#endif //UNTITLED11_SYNTAXPARSER_H
```

Parser cpp:

```
//
// Created by admin on 10.04.2025.
//
#include "SyntaxParser.h"


struct Token {
    TokenType_ type;
    std::string value;
};

void SyntaxParser::lexer(const std::string& code, Dlist<Token> &list) {
    std::istringstream stream(code);
    std::string word;
    std::string keywords[] = {"program", "var", "begin", "end", "if", "then",
"else", "while", "do", "for", "to", "downto", "procedure", "function",
"array", "record", "case", "of", "repeat", "until", "with", "not", "and",
"or"};
    std::string operators[] = {"+", "-", "*", "/", ":=", "=", "<", ">", "<=",
">=", "<>", "and", "or", "not"};
    std::string limiters[] = {";", ",", ".", "(", ")", "[", "]"};
    std::string type[] = {"integer", "real", "char", "boolean", "string",
"array", "record", "file", "pointer", "set", "variant", "enumerated"};
    Dlist<std::string> keywordsDlist;
    Dlist<std::string> operatorsDlist;
    Dlist<std::string> limitersDlist;
    Dlist<std::string> typeDlist;
    keywordsDlist.creatFromArray(keywords, 24);
    operatorsDlist.creatFromArray(operators, 14);
    limitersDlist.creatFromArray(limiters, 7);
    typeDlist.creatFromArray(type, 12);

    while (stream >> word) {
        Token token;
        if (dataInArray(word, keywordsDlist)) {
            token.type = KEYWORD;

        } else if (std::isdigit(word[0])) {
            token.type = NUMBER;

        } else if (dataInArray(word, operatorsDlist)) {
            token.type = OPERATOR;

        } else if (dataInArray(word, limitersDlist)) {
            token.type = DELIMITERS;
```

```
        } else if (word == ";") {
            token.type = SEMICOLON;

        } else if(word[0] == '\'' ){
            std::string temp;
            while (true){
                stream >> temp;
                word+= ' ' + temp;
                if(word[word.size()-1] == '\''){
                    break;
                }
            }
            token.type = STRINGLITERALS;

        }
        else if(dataInArray(word,typeDlist)){
            token.type = TYPE;
        }
        else if(word == "//"){
            token.type = COMMENTS;
            std::string temp;
            while (true){
                if(word == "\n"){
                    break;
                }
                stream >> temp;
                word+=temp;
            }

        }
        else if(word == "{"){
            token.type = COMMENTS;
            std::string temp;
            while (true){
                stream >> temp;
                word+=temp;
                if(word == "}"){
                    break;
                }
            }

        }
        else if(word == "(*"){
            token.type = COMMENTS;
            std::string temp;
            while (true){
                stream >> temp;
                word+=temp;
                if(word == "*)"){
                    break;
                }
            }

        }
        else {
            token.type = IDENTIFIER;
        }
        token.value = word;
        list.appendLeft(token);
    }
}

int SyntaxParser::dataInArray(std::string value, Dlist<std::string> &arr){
```

```
    arr.setRight();
    while (true){
        if(value == arr.getData()){
            return 1;
        }
        if(arr.moveCurrentLeft()){
            return 0;
        }
    }
}

void SyntaxParser::parser(Dlist<Token> &list){
    list.setRight();
    ParserSplit p;
    bool conf = true;

    while (true){
        if(list.getData().value == "var"){
            p.var(list);
        }
        if(list.getData().value == "const"){
            p.const_(list);
        }
        if(list.getData().value == "begin"){
            p.begin(list);
        }
        if(list.getData().value == "write" || list.getData().value ==
"writeln" || list.getData().value == "readln" || list.getData().value ==
"assert"){
            p.write(list);
        }
        if(list.getData().value == "while"){
            p.while_(list);
        }
        if(list.getData().value == "if"){
            p.if_(list);
        }
        if(p.tokenTypeToString(list.getData().type) == "IDENTIFIER"){
            p.identifier(list);
        }
        if(list.getData().value == "else"){
            p.else_(list);
        }
        if(list.getData().value == "end"){
            p.end(list);
        }
        if(list.moveCurrentLeft()){
            break;
        }
    }
}

std::string ParserSplit::tokenTypeToString(TokenType_ type) {
    switch (type) {
        case KEYWORD: return "KEYWORD";
        case IDENTIFIER: return "IDENTIFIER";
        case NUMBER: return "NUMBER";
        case OPERATOR: return "OPERATOR";
        case DELIMITERS: return "DELIMITERS";
        case STRINGLITERALS: return "STRINGLITERALS";
        case COMMENTS: return "COMMENTS";
        case SEMICOLON: return "SEMICOLON";
        case TYPE: return "TYPE";
        default: return "UNKNOWN";
```

```
    }
}



void ParserSplit::var(Dlist<Token> &list){

    int flagDeclaration;
    int FlagInit;

    std::string t4 = list.getData().value;
    if(list.moveCurrentLeft()){
        std::cout << "not found end";
        exit(1);
    }

    std::string t = list.getData().value;
    if(list.moveCurrentLeft()){
        std::cout << "not found end";
        exit(1);
    }

    flagDeclaration = 2;
    if(list.getData().value == ":="){
        FlagInit = 1;
        while (true) {
            if (list.moveCurrentLeft()) {
                std::cout << "not found end";
                exit(1);
            }
            if (list.getData().value == ";") {
                if (FlagInit != 4) {
                    std::cout << "forgot `variable`" << ", line: " << i;
                    exit(1);
                }
                flagDeclaration = 0;
                FlagInit = 0;
                i++;
                break;
            }
            if(list.getData().value == "("){
                FlagInit = 3;
            }
            if(list.getData().value == ")") {
                FlagInit = 4;
            }
            if(tokenTypeToString(list.getData().type) == "IDENTIFIER"){
                FlagInit = 4;
            }
        }

    }
    else {
        while (true) {
            if (list.getData().value == ";") {
                if (flagDeclaration != 3) {
                    std::cout << "forgot `type`" << ", line: " << i;
                    exit(1);
                }

                flagDeclaration = 0;
                i++;
                break;
            }
```

```
            if (list.getData().value == ":") {
                if (flagDeclaration == 1) {
                    std::cout << "forgot `variable`" << ", line: " << i;
                    exit(1);

                }
                if (list.moveCurrentLeft()) {
                    std::cout << "not found end";
                    exit(1);

                }
                if (tokenTypeToString(list.getData().type) != "TYPE") {
                    std::cout << "forgot `type`" << ", line: " << i;
                    exit(1);

                }
                flagDeclaration = 3;
            } else if (list.getData().value == ",") {
                if (flagDeclaration == 1) {
                    std::cout << "forgot `variable`" << ", line: " << i;
                    exit(1);

                }
                flagDeclaration = 1;

            } else if (tokenTypeToString(list.getData().type) ==
"IDENTIFIER") {
                if (flagDeclaration == 2) {
                    std::cout << "forgot `,`" << ", line: " << i;
                    exit(1);
                }
                flagDeclaration = 2;
            }
            if (list.moveCurrentLeft()) {
                std::cout << "not found end";
                exit(1);
            }
        }
    }
}

void ParserSplit::const_(Dlist<Token> &list){
    if(list.moveCurrentLeft()){
        std::cout << "not found end";
        exit(1);

    }
    if(tokenTypeToString(list.getData().type) != "IDENTIFIER"){
        std::cout << "forgot `variable`" << ", line: " << i;
        exit(1);

    }
    if(list.moveCurrentLeft()){
        std::cout << "not found end";
        exit(1);

    }
    if(list.getData().value != "="){
        std::cout << "forgot `=`" << ", line: " << i;
        exit(1);

    }
    if(list.moveCurrentLeft()){
```

```
        std::cout << "not found end";
        exit(1);

    }
    if(tokenTypeToString(list.getData().type) != "IDENTIFIER" &&
tokenTypeToString(list.getData().type) != "NUMBER"){
        std::cout << "forgot `variable`" << ", line: " << i;
        exit(1);

    }
    if(list.moveCurrentLeft()){
        std::cout << "not found end";
        exit(1);

    }
    if(list.getData().value != ";"){
        std::cout << "forgot `;`" << ", line: " << i;
        exit(1);

    }
    i++;

}

void ParserSplit::begin(Dlist<Token> &list){
    countBegin += 1;
    i++;

}

void ParserSplit::write(Dlist<Token> &list){

    int flagIsAssert = 0;
    int flagIsWriteOrReading = 0;

    if(list.getData().value == "assert"){
        flagIsAssert = 1;

    }
    else{
        flagIsWriteOrReading = 1;
    }
    if(list.moveCurrentLeft()){
        exit(1);
    }
    if(list.getData().value != "("){
        std::cout << "forgot `(`" << ", line: " << i;
        exit(1);
    }

    while (true){

        if(list.moveCurrentLeft()){
            std::cout << "not found end";
            exit(1);

        }
        if(list.getData().value == ";"){
            if(flagIsWriteOrReading != 3){
                std::cout << "forgot `variable`" << ", line: " << i;
                exit(1);
            }
            flagIsWriteOrReading = 0;
            flagIsAssert = 0;
```

```
            i++;
            break;

        }

        if(tokenTypeToString(list.getData().type) == "IDENTIFIER" ||
tokenTypeToString(list.getData().type) == "STRINGLITERALS" ||
tokenTypeToString(list.getData().type) == "NUMBER"){

            if(flagIsWriteOrReading == 2){
                std::cout << "forgot `,`" << ", line: " << i;
                exit(1);
            }

            flagIsWriteOrReading = 2;

        }
        else if(tokenTypeToString(list.getData().type) == "OPERATOR" ||
list.getData().value == ","){

            if(flagIsWriteOrReading == 1){
                std::cout << "forgot `Variable`" << ", line: " << i;
                exit(1);

            }
            if(flagIsAssert == 1 && list.getData().value == ","){
                std::cout << "Cannot use `,`" << ", line: " << i;
                exit(1);

            }

            flagIsWriteOrReading = 1;

        }
        else if(list.getData().value == ")"){

            if(flagIsWriteOrReading != 2){
                std::cout << "forgot `Variable`" << ", line: " << i;
                exit(1);
            }

            flagIsWriteOrReading = 3;

        }

    }
}
void ParserSplit::while_(Dlist<Token> &list){

    int flagIsWhile = 1;
    int flagIsDo = 0;

    while (true){
        if(list.moveCurrentLeft()){
            std::cout << "not found end";
            exit(1);
        }

        if(list.getData().value == "do"){
            if(flagIsDo == 1 || flagIsWhile != 2){
                std::cout << "bad condition" << ", line: " << i;
                exit(1);
            }
            flagIsWhile = 0;
```

```
            flagIsDo = 0;
            i++;
            break;

        }

        if(list.getData().value == "("){
            flagIsDo = 1;
        }

        if(list.getData().value == ")"){
            if(flagIsDo == 0){
                std::cout << "forgot `(`" << ", line: " << i;
                exit(1);
            }

            flagIsDo = 0;

        }

        if(tokenTypeToString(list.getData().type) == "IDENTIFIER" ||
tokenTypeToString(list.getData().type) == "NUMBER"){
            if(flagIsWhile != 1){
                std::cout << "forgot operator" << ", line: " << i;
                exit(1);
            }
            flagIsWhile = 2;
        }

        if(tokenTypeToString(list.getData().type) == "OPERATOR"){
            if(flagIsWhile != 2){
                std::cout << "forgot variable" << ", line: " << i;
                exit(1);
            }
            flagIsWhile = 1;

        }
    }
}

void ParserSplit::if_(Dlist<Token> &list){

    int flagIsCorrectIf;
    int flagIsCorrectCondition = 0;
    countIf += 1;
    flagIsCorrectIf = 1;

    while (true){

        if(list.moveCurrentLeft()){
            std::cout << "not found end";
            exit(1);
        }

        if(list.getData().value == "then"){
            if(flagIsCorrectCondition == 1 || flagIsCorrectIf != 2){
                std::cout << "bad condition" << ", line: " << i;
                exit(1);
            }
            flagIsCorrectIf = 0;
            flagIsCorrectCondition = 0;
            i++;
            break;
        }
```

```
        if(list.getData().value == "("){
            flagIsCorrectCondition = 1;
        }

        if(list.getData().value == ")"){
            if(flagIsCorrectCondition == 0){
                std::cout << "forgot `(`" << ", line: " << i;
                exit(1);
            }

            flagIsCorrectCondition = 0;
        }

        if(tokenTypeToString(list.getData().type) == "IDENTIFIER" ||
tokenTypeToString(list.getData().type) == "NUMBER"){
            if(flagIsCorrectIf != 1){
                std::cout << "forgot operator" << ", line: " << i;
                exit(1);
            }
            flagIsCorrectIf = 2;
        }

        if(tokenTypeToString(list.getData().type) == "OPERATOR"){
            if(flagIsCorrectIf != 2){
                std::cout << "forgot variable" << ", line: " << i;
                exit(1);
            }
            flagIsCorrectIf = 1;
        }
    }
}

void ParserSplit::identifier(Dlist<Token> &list){

    int flagIsCorrectCondition = 0;

    if(list.moveCurrentLeft()){
        std::cout << "not found end";
        exit(1);
    }

    if(list.getData().value != ":="){
        std::cout << "forgot `:=`" << ", line: " << i;
        exit(1);
    }

    flagIsCorrectCondition = 1;

    while (true) {
        if (list.moveCurrentLeft()) {
            std::cout << "not found end";
            exit(1);
        }
        if(list.getData().value == ";"){
            if(flagIsCorrectCondition == 1){
                std::cout << "forgot `variable`" << ", line: " << i;
                exit(1);
            }
            i++;
            flagIsCorrectCondition = 0;
            break;
        }
```

```
        if(tokenTypeToString(list.getData().type) == "IDENTIFIER"){
            if(flagIsCorrectCondition != 1){
                std::cout << "forgot operator" << ", line: " << i;
                exit(1);
            }
            flagIsCorrectCondition = 2;
        }
        if(tokenTypeToString(list.getData().type) == "OPERATOR"){
            if(flagIsCorrectCondition != 2){
                std::cout << "forgot variable" << ", line: " << i;
                exit(1);
            }
            flagIsCorrectCondition = 1;
        }
    }
}

void ParserSplit::else_(Dlist<Token> &list){
    if(countIf<1){
        std::cout << "not found if from else" << ", line: " << i;
        exit(1);
    }
    countIf -= 1;
    i++;
}

void ParserSplit::end(Dlist<Token> &list){
    if(countBegin < 1){
        std::cout << "not found begin from end" << ", line: " << i;
        exit(1);
    }
    if (list.moveCurrentLeft()) {
        std::cout << "not found end";
        exit(1);
    }
    if(list.getData().value != ";" &&  list.getData().value != "."){
        std::cout << "not found ;" << ", line: " << i;
        exit(1);
    }
    countBegin--;
    i++;
}
```

main:

```
#include <iostream>

#include <windows.h>

#include "Dlist.h"
#include "SyntaxParser.h"



struct Token {
    TokenType_ type;
    std::string value;
};

int main() {
    SetConsoleOutputCP(CP_UTF8);
    Dlist<Token> list;
```

```
    std::string code = R"(
        const eps = 0.0001 ;

var a , b : real ;
begin
  write ( ' Введите числа a и b (a<b) : ' ) ;
  readln ( a , b ) ;
  assert ( a < b ) ;

  var fa := sin ( a ) ;
  var fb := sin ( b ) ;
  assert ( fb * fa < 0 ) ;

  while ( b - a ) > eps do
  begin
    var x := ( b + a ) / 2 ;
    var fx := sin ( x ) ;
    if fa * fx <= 0 then
      b := x ;
    else
    begin
      a := x ;
      fa := fx ;
    end ;
  end ;
  writeln ( ' Корень функции на [a,b] равен ' , ( b + a ) / 2 ) ;
end .
    )";
    SyntaxParser p;
    p.lexer(code, list);
    p.parser(list);
    std::cout << "OK" << std::endl;

    return 0;
}
```

Вывод программы:

```
:

C:\Users\admin\CLionProjects\untitled11\cmake-build-debug\untitled11.exe
OK

Process finished with exit code 0
```

Сделаем ошибку в коде:

```
C:\Users\admin\CLionProjects\untitled11\cmake-build-debug\untitled11.exe
forgot `,`, line: 22
Process finished with exit code 1
```

```
C:\Users\admin\CLionProjects\untitled11\cmake-build-debug\untitled11.exe
forgot operator, line: 10
Process finished with exit code 1
```

Вывод: в ходе лабораторной работы я научился создавать шаблонные классы.