МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

МИРЭА

Институт информационных технологий (ИТ)
Кафедра инструментального и прикладного программного обеспечения (ИППО)

Отчет по лабораторным работам

По дисциплине

«Объектно-ориентированное программирование»

Работу выполнил студент группы ИКБО-14-17: Сорокин А.А.

Работу проверила: Зорина Н.В.

Москва 2019

Лабораторная работа №1

1.1 Тема и цель работы

Тема: Создание многофайловых проектов.

Цель: освоить на практике создание многофайловых проектов в языке Си/Си++, познакомиться с директивами условной компиляции.

1.2 Теоретические сведения

1. **Условная компиляция (#ifdef, #ifndef, #else, #endif)**

Директивы условной компиляции препроцессора позволяют компилировать или пропускать часть программы в зависимости от выполнения некоторого условия. Условие может принимать одну из описываемых ниже форм.

```
1)#ifdef identifier
2) // код, находящийся здесь, компилируется, если  identifier уже был определен для препроцессора
3) // в команде #define.
4)#endif
5)#ifndef identifier
6) // код, находящийся здесь, компилируется, если identifier уже был определен для препроцессора
7) // в команде #define.
8)#endif
```

За любой из команд условной компиляции может следовать произвольное число строк, содержащих, возможно, команду вида #else и заканчивающихся #endif. Если проверяемое условие справедливо, то строки между #else и #endif игнорируются. Если же проверяемое условие не выполняется, то игнорируются все строки между проверкой и командой #else, а если ее нет, то командой #endif.

Пример:

```
1)#ifndef TestMode
2) #define TestMode
3)#endif
4)//+------------------------------------------------+
5)//| Script program start function                  |
6)//+------------------------------------------------+
 7)void OnStart()
8) {
9) #ifdef TestMode
10) Print("Test mode");
11) #else
12)Print("Normal mode");
13)#endif
14)}
```

В зависимости от типа программы и режима компиляции стандартные макросы определяются следующим образом:

Макрос __MQL4__ доступен при компиляции файла *.mq4, при компиляции *.mq5 доступен макрос __MQL5__.

Макрос _DEBUG доступен при компиляции под отладку.

Макрос _RELEASE доступен при компиляции не под отладку

Пример:

```
1) //| Script program start function void
OnStart()
2) {
3) #ifdef __MQL5__
4) #ifdef _DEBUG
5) Print("Hello from MQL5 compiler [DEBUG]");
6) #else
7) #ifdef _RELEASE
8) Print("Hello from MQL5 compiler [RELEASE]");
9) #endif
10) #endif
11) #else
12) #ifdef __MQL4__
13) #ifdef _DEBUG
14) Print("Hello from MQL4 compiler [DEBUG]");
15) #else
16) #ifdef _RELEASE
17) Print("Hello from MQL4 compiler [RELEASE]");
18) #endif
19) #endif
20) #endif
21) #endif
22) }
```

2. **Разработка многофайлового проекта:**

Современные программные проекты редко ограничиваются одним исходным файлом. Распределение исходного кода программы на несколько файлов имеет ряд существенных преимуществ перед однофайловыми проектами:

- Использование нескольких исходных файлов накладывает на репозиторий (рабочий каталог проекта) определенную логическую структуру.
- В однофайловых проектах любая модернизация исходного кода влечет повторную компиляцию всего проекта. В многофайловых проектах, напротив, достаточно откомпилировать только измененный файл, чтобы обновить проект. Это экономит массу времени.
- Многофайловые проекты позволяют реализовывать одну программу на разных языках программирования.
- Многофайловые проекты позволяют применять к различным частям программы разные лицензионные соглашения.

Обычно процесс сборки многофайлового проекта осуществляется по следующему алгоритму:

1. Создаются и подготавливаются исходные файлы. Здесь есть одно важное условие: каждый файл должен быть целостным, т.е. не должен содержать незавершенных конструкций. Если в рамках проекта предполагается создание исполняемой программы, то в одном из исходных файлов должна присутствовать функция main().
2. Создаются и подготавливаются заголовочные файлы. У заголовочных файлов особая роль: они устанавливают соглашения по использованию общих идентификаторов (имен) в различных частях программы. Если, например, функция func() реализована в файле a.c, а вызывается в файле b.c, то в оба файла требуется включить директивой #include заголовочный файл, содержащий объявление (прототип) нашей функции. Технически можно обойтись и без заголовочных файлов, но в этом случае функцию можно будет вызвать с произвольными аргументами, и компилятор, за отсутствием соглашений, не выведет ни одной ошибки. Подобный "слепой" подход потенциально опасен и в большинстве случаев свидетельствует о плохом стиле программирования.
3. Каждый исходный файл отдельно компилируется с опцией -c. В результате появляется набор объектных файлов.
4. Полученные объектные файлы соединяются компоновщиком в одну программу.

1.3 Основная часть

Задание

Напишите программу-калькулятор комплексных чисел. Для реализации необходимо разработать абстрактный тип данных – комплексное число. Программа должна реализовывать арифметические операции над комплексными числами. Программа должна быть представлена в виде многофайлового проекта, все прототипы функций, объявления структур должны быть вынесены в заголовочный файл с соответствующим названием. Всего должно быть три файла: файл с объявлениями, файл реализации и файл с функцией main(), демонстрирующий работу с новым типом данных. Файл с объявлениями должен называться Complex.h, файл с реализацией функций должен называться Complex.c, файл с функцией main() может называться main.c Программа должна обеспечивать удобный интерфейс пользователя для работы с ней.

Код

Complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H
#include <cmath>
#include "stdafx.h"
#include <iostream>

typedef struct Complex
{
	double real = 0;
	double imag = 0;
```

```
}Complex;

void print_menu(char select);
void print_error();
Complex toString(Complex a, Complex b);
Complex Sum(Complex a, Complex b);
Complex Substraction(Complex a, Complex b);
Complex Multiply(Complex a, Complex b);
Complex Division(Complex a, Complex b);

#endif // !COMPLEX_H
```

Complex.cpp

```
#include "stdafx.h"
#include "Complex.h"
using namespace std;



void print_error()
{
	cout << "Error, this symbol is not supported" << endl;
}

void print_menu(char select)
{
	cout << "Enter operation" << endl;
	cout << "------------------------" << endl;
	cout << "'+' Sum" << endl;
	cout << "'-' Substraction" << endl;
	cout << "'*' Multiplication" << endl;
	cout << "'/' Division" << endl;
	cout << "------------------------" << endl;
}

Complex toString (Complex a, Complex b)
{
	cout << "Real part is: " << a.real << endl;
	cout << "Imagine part is: " << b.imag << endl;
	cout << "Complex number is: (" << a.real << ';' << b.imag << "i)" << endl;
	return a, b;
}

Complex Sum(Complex a, Complex b)
{
	a.real = a.real + b.real;
	b.imag = a.imag + b.imag;
	return toString(a, b);
}

Complex Substraction(Complex a, Complex b)
{
	a.real = a.real - b.real;
	b.imag = a.imag - b.imag;
	return toString(a, b);
}
```

```
Complex Multiply(Complex a, Complex b)
{
	a.real = (a.real * b.real - a.imag * b.imag);
	b.imag = (a.real * b.imag + b.real * a.imag);
	return toString(a, b);
}

Complex Division(Complex a, Complex b)
{
	a.real = ((a.real * b.real + a.imag * b.imag) / (b.real*b.real +
b.imag*b.imag));
	b.imag = ((b.real * a.imag - a.real * b.imag) / (b.real*b.real +
b.imag*b.imag));
	return toString(a, b);
}
```

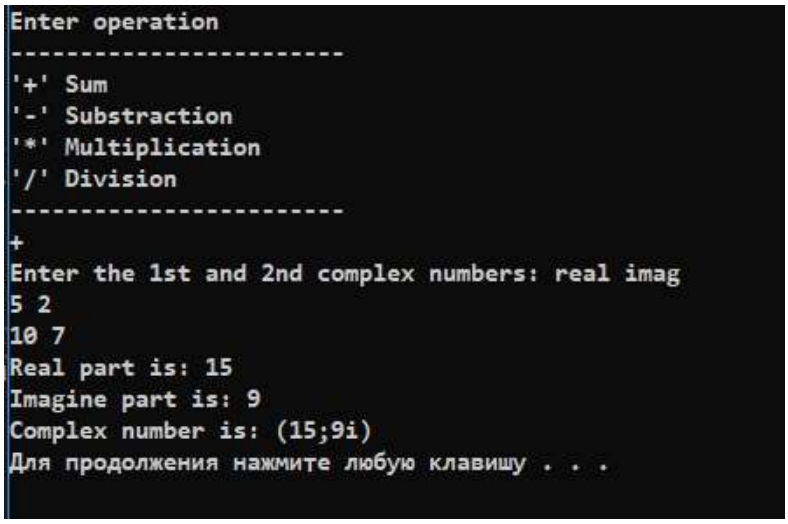
Main.cpp

```
#include "stdafx.h"
#include "Complex.h"
using namespace std;

int main()
{
	char select = ' ';
	Complex c1, c2, c3;

	print_menu(select);
	cin >> select;

	switch (select)
	{
	case '+':
		cout << "Enter the 1st and 2nd complex numbers: real imag" << endl;
		cin >> c1.real >> c1.imag >> c2.real >> c2.imag;
		c3 = Sum(c1, c2);
		break;

	case '-':
		cout << "Enter the 1st and 2nd complex numbers: real imag" << endl;
		cin >> c1.real >> c1.imag >> c2.real >> c2.imag;
		c3 = Substraction(c1, c2);
		break;

	case '*':
		cout << "Enter the 1st and 2nd complex numbers: real imag" << endl;
		cin >> c1.real >> c1.imag >> c2.real >> c2.imag;
		c3 = Multiply(c1, c2);
		break;

	case '/':
		cout << "Enter the 1st and 2nd complex numbers: real imag" << endl;
		cin >> c1.real >> c1.imag >> c2.real >> c2.imag;
		c3 = Division(c1, c2);
		break;

	default:
		print_error();
```

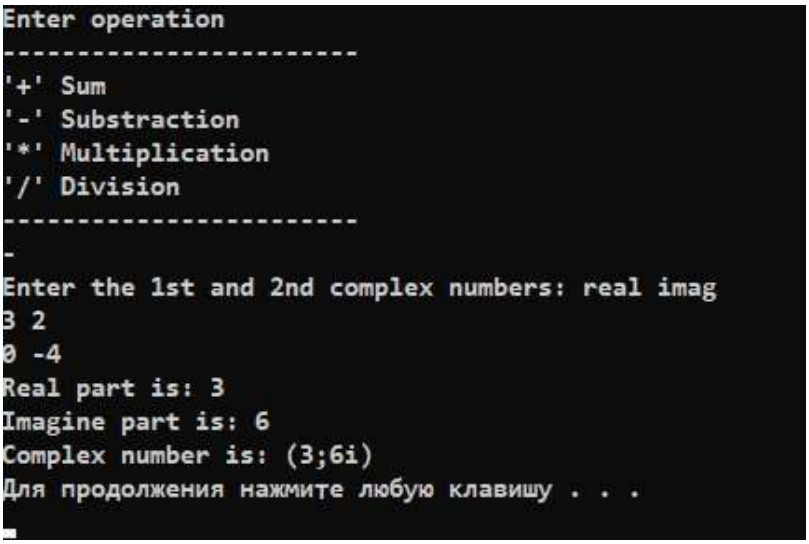
```
            break;
        }

    return 0;
}
```

Результат работы программы

1.



2.



3.

```
Enter operation
-----------------------
'+' Sum
'-' Substraction
'*' Multiplication
'/' Division
-----------------------
*
Enter the 1st and 2nd complex numbers: real imag
1 2
2 1
Real part is: 0
Imagine part is: 4
Complex number is: (0;4i)
Для продолжения нажмите любую клавишу . . .
```

4.

```
Enter operation
-----------------------
'+' Sum
'-' Substraction
'*' Multiplication
'/' Division
-----------------------
/
Enter the 1st and 2nd complex numbers: real imag
2 1
1 2
Real part is: 0.8
Imagine part is: -0.12
Complex number is: (0.8;-0.12i)
Для продолжения нажмите любую клавишу . . .
```

1.4 Вывод

На практике освоены создание многофайловых проектов в языке Си/Си++, рассмотрены директивами условной компиляции.

Лабораторная работа №2

2.1 Тема и цель работы

Тема: Указатели на функции.

Цель: освоить на практике вызов функции с использованием указателей.

2.2 Теоретические сведения

Вызов функции – это сохранение состояния, передача аргументов и переход по адресу, где располагается функция. В языке программирования Си есть возможность создавать указатели на функции. Указатели на функции позволяют упростить решение многих задач. Совместно с указателями типа void* можно создавать функции общего назначения (например, сортировки и поиска).

Указатели позволяют создавать функции высших порядков (функции, принимающие в качестве аргументов функции): отображение, свёртки, фильтры и пр. Указатели на функции позволяют уменьшать сложность программ и создавать легко масштабируемые конструкции. Так же, как и другие переменные-указатели указатели на функции необходимо сначала объявлять, затем инициализировать и только потом использовать.

Проинициализированный указатель всегда содержит адрес в памяти, в случае указателя на функции такой указатель содержит адрес функции в памяти.

Синтаксис объявления указателя на функцию:
тип (*имя) (список параметров);
где список параметров может быть (тип аргумент1, …, тип аргументN) или пустой – void, тогда по умолчанию можно оставить пустые круглые скобки ()

Пример:

```
1) int (*pf)(int, char*);
```

Объявляем указатель на функцию с именем pf, такой указатель может использоваться только для функции, которая возвращает int и берет список параметров (int, char*);

Адресом функции является ее точка входа. Именно этот адрес используется при вызове функции. Так как указатель хранит адрес функции, то она может быть вызвана с помощью этого указателя. Он позволяет также передавать ее другим функциям в качестве аргумента.

Можно определять массив указателей на функции и затем их использовать:

```
1) void f1(int);
2) void f2(int);
3) void f3(int);
4) void (*fArray[3])(int)={f1, f2, f3};
```

В программе на Си адресом функции служит ее имя без скобок и аргументов (это похоже на адрес массива, который равен имени массива без индексов). Рассмотрим следующую программу, в которой сравниваются две строки, введенные пользователем. Обратите внимание на объявление функции check() и указатель p внутри main(). Указатель p, как вы увидите, является указателем на функцию.

Пример кода:

```
1) #include <stdio.h>
2) #include <string.h>
3) void check(char *a, char *b, a. int (*cmp)(const char *, const char *));
4) int main(void)
5) {
6) char s1[80], s2[80];
7) int (*p)(const char *, const char *);
8) /* указатель на функцию */
9) p = strcmp;
10) /* присваивает адрес функции strcmp указателю p */
11) printf("Введите две строки.\n");
12) gets(s1);
13) gets(s2);
14) check(s1, s2, p); /* Передает адрес функции strcmp i. посредством указателя p */
15) return 0;
16) }
17) void check(char *a, char *b, a. int (*cmp)(const char *, const char *))
18) {
19) printf("Проверка на совпадение.\n");
20) if(!(*cmp)(a, b)) printf("Равны");
21) else printf("Не равны"); }
```

Проанализируем эту программу подробно. В первую очередь рассмотрим объявление указателя p в main():

```
1) int (*p)(const char *, const char *);
```

Это объявление сообщает компилятору, что p — это указатель на функцию, имеющую два параметра типа const char * и возвращающую значение типа int. Скобки вокруг p необходимы для правильной интерпретации объявления компилятором. Подобная форма объявления используется также для указателей на любые другие функции, нужно лишь внести изменения в зависимости от возвращаемого типа и параметров функции.

Теперь рассмотрим функцию check(). В ней объявлены три параметра: два указателя на символьный тип (a и b) и указатель на функцию cmp. Обратите внимание на то, что указатель функции cmp объявлен в том же формате, что и p. Поэтому в cmp можно хранить значение указателя на функцию, имеющую два параметра типа const char * и возвращающую значение int. Как и в объявлении p, круглые скобки вокруг *cmp необходимы для правильной интерпретации этого объявления компилятором.

Вначале в программе указателю p присваивается адрес стандартной библиотечной функции strcmp(), которая сравнивает строки. Потом программа просит пользователя ввести две строки и передает указатели на них функции check(), которая их сравнивает. Внутри check() выражение (*cmp)(a, b) вызывает функцию strcmp(), на которую указывает

cmp, с аргументами a и b. Скобки вокруг *cmp обязательны. Существует и другой, более простой, способ вызова функции с помощью указателя cmp(a, b);

Однако первый способ используется чаще (и мы рекомендуем использовать именно его), потому что при втором способе вызова указатель cmp очень похож на имя функции, что может сбить с толку читающего программу. В то же время у первого способа записи есть свои преимущества, например, хорошо видно, что функция вызывается с помощью указателя на функцию, а не имени функции. Следует отметить, что первоначально в C был определен именно первый способ вызова. Вызов функции check() можно записать, используя непосредственно имя strcmp(): check(s1, s2, strcmp);

В этом случае вводить в программу дополнительный указатель p нет необходимости.

Рассмотрим еще пример использования указателя на функцию:

```
1) #include <conio.h>
2) int sum(int a, int b) {
3) return a + b;
4) }
5) void main () {
6) //Объявляем указатель на функцию
7) int (*fptr)(int, int) = NULL;
8) int result;
9) //Присваиваем указателю значение - адрес функции
10) //Это похоже на работу с массивом: операцию взятия адреса использовать не нужно
11) fptr = sum;
12) //Вызов осуществляется также, как и обычной функции
13) result = fptr(10, 40);
14) printf("%d", result);
15) }
```

Указатель на функцию может использоваться для реализации выбора вызова функции, например:

```
1) #include <stdio.h>
2) #include <stdlib.h>
3) float * f1();
4) float * f2();
5) int main () {
6) float *retvalue;
7) float *(*pf)(int xx);
8) if (<условие вызова функции f1 >)
9) pf = f1(); // pf инициализируется адресом функции f1
10) else
11) pf = f2(); // pf инициализируется адресом функции f2
12) ....
13) retvalue = pf();
14) return 0;
15) }
16) // определение функции f1()
17) float * f1(){
18) …
19) }
20) //определение функции f1()
21) float * f2(){
```

```
22) …
23) }
```

Так же возможно писать функции отображения.

Пример:

```
1) //Функция принимает массив, его размер и указатель на функцию,
2) //которая далее применяется ко всем элементам массива
3) void map(int *arr, unsigned size, int (*fun)(int)) {
4) unsigned i;
5) for (i = 0; i < size; i++) {
6) arr[i] = fun(arr[i]);
7) }
8) }
9) void main () {
10) int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11) unsigned i;
12) map(a, 10, deleteEven);
13) for (i = 0; i < 10; i++) {
14) printf("%d ", a[i]);
15) }
16) map(a, 10, dble);
17) printf("\n");
18) for (i = 0; i < 10; i++) {
19) printf("%d ", a[i]);
20) }
21) }
```

В этом примере мы создали функцию отображения, которая применяет ко всем элементам массива функцию, которая передаётся ей в качестве аргумента.

Когда мы вызываем функцию map, достаточно просто передавать имена функций (они подменяются указателями).

1.3 Основная часть

Задание

Напишите программу, которая вызывает различные виды функции в зависимости от заданного условия. Можно использовать примеры выше.

Код

Complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H
#include <cmath>
#include "stdafx.h"
#include <iostream>

typedef struct Complex
{
	double real = 0;
```

```
	double imag = 0;
}Complex;

void print_menu(char select);
void print_error();
Complex toString(Complex *a, Complex *b);
Complex Sum(Complex *a, Complex *b);
Complex Substraction(Complex *a, Complex *b);
Complex Multiply(Complex *a, Complex *b);
Complex Division(Complex *a, Complex *b);

	#endif // !COMPLEX_H
```

Complex.cpp

```
#include "stdafx.h"
#include "Complex.h"
using namespace std;



void print_error()
{
	cout << "Error, this symbol is not supported" << endl;
}

void print_menu(char select)
{
	cout << "Enter operation" << endl;
	cout << "------------------------" << endl;
	cout << "'+' Sum" << endl;
	cout << "'-' Substraction" << endl;
	cout << "'*' Multiplication" << endl;
	cout << "'/' Division" << endl;
	cout << "------------------------" << endl;
}

Complex toString (Complex *a, Complex *b)
{
	cout << "Real part is: " << a->real << endl;
	cout << "Imagine part is: " << b->imag << endl;
	cout << "Complex number is: (" << a->real << ';' << b->imag << "i)" << endl;
	return *a, *b;
}

Complex Sum(Complex *a, Complex *b)
{
	a->real = a->real + b->real;
	b->imag = a->imag + b->imag;
	return toString(a, b);
}

Complex Substraction(Complex *a, Complex *b)
{
	a->real = a->real - b->real;
	b->imag = a->imag - b->imag;
	return toString(a, b);
}

Complex Multiply(Complex *a, Complex *b)
```

```
{
	a->real = (a->real * b->real - a->imag * b->imag);
	b->imag = (a->real * b->imag + b->real * a->imag);
	return toString(a, b);
}

Complex Division(Complex *a, Complex *b)
{
	a->real = ((a->real * b->real + a->imag * b->imag) / (b->real*b->real + b->imag*b->imag));
	b->imag = ((b->real * a->imag - a->real * b->imag) / (b->real*b->real + b->imag*b->imag));
	return toString(a, b);
}
```

Main.cpp

```
#include "stdafx.h"
#include "Complex.h"
using namespace std;

int main()
{
	char select = ' ';
	Complex c1, c2, c3;

	print_menu(select);
	cin >> select;

	switch (select)
	{
	case '+':
		cout << "Enter the 1st and 2nd complex numbers: real imag" << endl;
		cin >> c1.real >> c1.imag >> c2.real >> c2.imag;
		c3 = Sum(&c1, &c2);
		break;

	case '-':
		cout << "Enter the 1st and 2nd complex numbers: real imag" << endl;
		cin >> c1.real >> c1.imag >> c2.real >> c2.imag;
		c3 = Substraction(&c1, &c2);
		break;

	case '*':
		cout << "Enter the 1st and 2nd complex numbers: real imag" << endl;
		cin >> c1.real >> c1.imag >> c2.real >> c2.imag;
		c3 = Multiply(&c1, &c2);
		break;

	case '/':
		cout << "Enter the 1st and 2nd complex numbers: real imag" << endl;
		cin >> c1.real >> c1.imag >> c2.real >> c2.imag;
		c3 = Division(&c1, &c2);
		break;

	default:
		print_error();
		break;
	}
```

```
    return 0;
}
```

Результат работы программы:

1.

```
Enter operation
-----------------------
'+' Sum
'-' Substraction
'*' Multiplication
'/' Division
-----------------------
+
Enter the 1st and 2nd complex numbers: real imag
5 2
10 7
Real part is: 15
Imagine part is: 9
Complex number is: (15;9i)
Для продолжения нажмите любую клавишу . . .
```

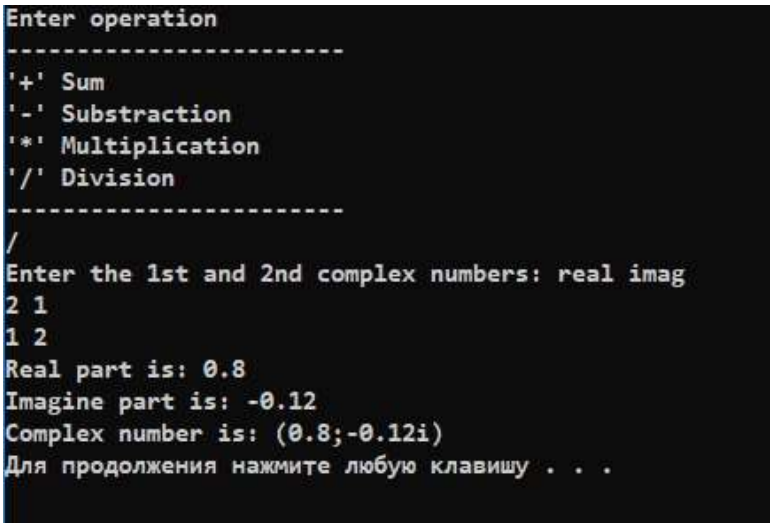
2.

```
Enter operation
-----------------------
'+' Sum
'-' Substraction
'*' Multiplication
'/' Division
-----------------------
-
Enter the 1st and 2nd complex numbers: real imag
3 2
0 -4
Real part is: 3
Imagine part is: 6
Complex number is: (3;6i)
Для продолжения нажмите любую клавишу . . .
```

3.

```
Enter operation
-----------------------
'+' Sum
'-' Substraction
'*' Multiplication
'/' Division
-----------------------
*
Enter the 1st and 2nd complex numbers: real imag
1 2
2 1
Real part is: 0
Imagine part is: 4
Complex number is: (0;4i)
Для продолжения нажмите любую клавишу . . .
```

4.



2.4 Вывод

На практике освоен вызов функции с использованием указателей.

Лабораторная работа №3

3.1 Тема и цель работы

Тема: Классы

Цель: знакомство с классами в языке C++.

3.2 Теоретические сведения

Класс — это производный структурированный тип, введенный программистом на основе уже существующих типов. Механизм классов позволяет создавать типы в полном соответствии с принципами абстракции данных, т.е. класс задает некоторую структурированную совокупность типизированных данных и позволяет определить набор операций над этими данными.

Простейшим образом класс можно определить с помощью конструкции:

ключ_класса имя_класса { список_компонентов };

где ключ_класса - одно из служебных слов class, struct, union; имя_класса - произвольно выбираемый идентификатор; список_компонентов - определения и описания типизированных данных и принадлежащих классу функций.

В проекте стандарта языка Си++ указано, что компонентами класса могут быть данные, функции, классы, перечисления, битовые поля, дружественные функции, дружественные классы и имена типов. Вначале для простоты будем считать, что компоненты класса — это типизированные данные (базовые и производные) и функции. Заключенный в фигурные скобки список компонентов называют телом класса. Телу класса предшествует заголовок. В простейшем случае заголовок класса включает ключ класса и его имя. Определение класса всегда заканчивается точкой с запятой. Принадлежащие классу функции мы будем называть методами класса или компонентными функциями. Данные класса - компонентными данными или элементами данных класса. В качестве ключа_класса можно использовать служебное слово struct, но класс отличается от обычного структурного типа, по крайней мере, возможностью включения компонентных функций. Например, следующая конструкция простейшим образом вводит класс "комплексное число":

```
1) struct complex1 // Вариант класса "комплексное число".
2) { double real; // Вещественная часть.
3) double imag; // Мнимая часть.
4) // Определить значение комплексного числа:
5) void define (double re = 0.0, double im = 0.0)
6) { real = re; imag = im; }
7) // Вывести на экран значение комплексного числа:
8) void display(void)
9) { cout << "real = " << real; cout << ", imag = " << imag;}
10) };
```

В отличие от структурного типа в класс (тип) complex1, кроме компонентных данных (real, imag), включены две компонентные функции define() и display(). Обратим еще раз внимание на тот факт, что класс (как и его частный случай - структура), введенный пользователем, обладает правами типа. Следовательно, можно определять и описывать объекты класса и создавать производные типы:

```
1)complex1 X1, X2, D;
2)// Три объекта класса complex1.
3)complex1 *point = &D;
4)// Указатель на объект класса complex1.
5)complex1 dim[8];
6)// Массив объектов класса complex1.
7) complex1 &Name = X2;
8) complex1 &Name = X2;
9) // Ссылка на объект класса //complex1.
```

и т.д.

Итак, класс — это тип, введенный программистом. Каждый тип служит для определения объектов. Для описания объекта класса используется конструкция:

имя_класса имя_объекта;

В определяемые объекты класса входят данные (элементы), соответствующие компонентным данным класса. Компонентные функции класса позволяют обрабатывать данные конкретных объектов класса. Но в отличие от компонентных данных компонентные функции не тиражируются при создании конкретных объектов класса. Если перейти на уровень реализации, то место в памяти выделяется именно для элементов каждого объекта класса. Определение объекта класса предусматривает выделение участка памяти и деление этого участка на фрагменты, соответствующие отдельным элементам объекта, каждый из которых отображает отдельный компонент данных класса. Таким образом, и в объект X1, и в объект dim[3] класса complex1 входит по два элемента типа double, представляющих вещественные и мнимые части комплексных чисел.

На следующем шаге мы начнем рассматривать способы доступа к компонентам класса. С этого шага мы начнем рассматривать различные способы доступа к компонентам класса. Как только объект класса определен, появляется возможность обращаться к его компонентам. На этом шаге мы рассмотрим доступ с помощью квалифицированных имен. Такое имя имеет следующий формат:

имя_объекта.имя_класса::имя_компонента .

Имя класса с операцией уточнения области действия "::" обычно может быть опущено, и чаще всего для доступа к данным конкретного объекта заданного класса (как и в случае структур) используется уточненное имя:

имя_объекта.имя_элемента

При этом возможности те же, что и при работе с элементами структур. Например, можно явно присвоить значения элементам объектов класса complex1:

```
1) . . . . .
2) complex1 X1, X2, D;
3) // Три объекта класса complex1.
4) complex1 *point = &D;
5) // Указатель на объект класса complex1.
6) complex1 dim[8];
7) // Массив объектов класса complex1.
8) complex1 &Name = X2;
9) // Ссылка на объект класса complex1.
10) . . . . .
11) X1.real = dim[3].real = 1.24;
```

```
12) X1.imag = 2.3;
13) dim[3].imag = 0.0;
```

Уточненное имя принадлежащей классу (т.е. компонентной) функции

имя_объекта.обращание_к_компонентной_функции

обеспечивает вызов компонентной функции класса для обработки данных именно того объекта, имя которого использовано в уточненном имени. Например, можно таким образом определить значения компонентных данных для определенных на предыдущем шаге объектов класса complex1:

```
1) X1.define();
2) // Параметры выбираются по умолчанию:
3) // real == 0.0, imag == 0.0.
4) X2.define(4.3,20.0);
5) // Комплексное число 4.3 + i*20.0.
```

С помощью принадлежащей классу complex1 функции display() можно вывести на экран значения компонентных данных любого из объектов класса. Например, следующий вызов принадлежащей классу complex1 функции:

```
1) X2.display();
```

приведет к печати

```
1) real =4.3, imag = 20.0
```

На следующем шаге мы рассмотрим использование указателей для доступа к компонентам класса. На этом шаге мы рассмотрим использование указателей для доступа к компонентам класса.

Другой способ доступа к элементам объекта некоторого класса предусматривает явное использование указателя на объект класса и операции косвенного выбора компонента ('->'):

указателъ_на_объект_класса -> имя_элемента

Воспользовавшись указателем point, который адресует объект D класса complex1, можно следующим образом присвоить значения данным объекта D:

```
1) .. . . . .
2) complex1 X1, X2, D;
3) // Три объекта класса complex1.
4) complex1 *point = &D;
5) // Указатель на объект класса complex1.
6) complex1 dim[8];
7) // Массив объектов класса complex1.
8) complex1 &Name = X2;
9) // Ссылка на объект класса complex1.
10) . . . . .
11) point->real = 2.3;
12) // Присваивание значения элементу объекта D.
13) point->imag = 6.1;
14) // Присваивание значения элементу объекта D.
```

Указатель на объект класса позволяет вызывать принадлежащие классу функции для обработки данных того объекта, который адресуется указателем. Формат вызова функции:

указатель_на_объект_класса -> обращение_к_компонентной_функции.

Например, вызвать компонентную функцию display() для данных объекта D позволяет выражение

```
1) point->display();
```

3.3 Основная часть

Задание

Создать класс Complex, в котором реализовано комплексное число. В данном классе должны присутствовать методы, позволяющие рассчитать и вывеси модуль и аргументы данного числа.

Код

Complex.h

```
#include <cmath>
#include <iostream>
#include <string.h>

using namespace std;

void print_error();
void print_menu(char select);

class Complex
{
protected:
	double real;
	double imag;
public:
	Complex printComplex(Complex *a, Complex *b);
	Complex EnterComplex(Complex *a);
	Complex EnterComplex(Complex *a, Complex *b);
	Complex complexSum(Complex *a, Complex *b);
	Complex complexSubstraction(Complex *a, Complex *b);
	Complex complexMultiply(Complex *a, Complex *b);
	Complex complexDivision(Complex *a, Complex *b);
};

class ComplexNew : public Complex // наследование класса.
{
private:
	double mod;
public:
	double complexModulo(ComplexNew *a);

};
```

Complex.cpp

```
#include "stdafx.h"
#include "complex.h"

using namespace std;

void print_error()
{
	cout << "Error, this  symbol is not supported" << endl;

}

void print_menu(char select)
{
	cout << "Enter operation" << endl;
	cout << "------------------------" << endl;
	cout << "'+' Sum" << endl;
	cout << "'-' Substraction" << endl;
	cout << "'*' Division" << endl;
	cout << "'/' Multiplication" << endl;
	cout << "'M' or 'm' Modulo of complex number" << endl;
	cout << "------------------------" << endl;

}

// Определение методов класса COMPLEX.

Complex Complex::printComplex(Complex *a, Complex *b)
{
	cout << "Real: " << a->real << endl;
	cout << "Imagine: " << b->imag << endl;
	cout << "Complex number is: (" << a->real << ';' << b->imag << "i)" << endl;

	return *a, *b;
}

Complex Complex::EnterComplex(Complex *a)
{
	setlocale(LC_ALL, "ru");
	cout << "Enter complex number: real imag" << endl;
	cin >> a->real >> a->imag;

	return *a;
}

Complex Complex::EnterComplex(Complex *a, Complex *b)
{
	setlocale(LC_ALL, "ru");
	cout << "Enter first complex number: real imag " << endl;
	cin >> a->real >> a->imag;
	cout << "Enter second comlex number: real imag " << endl;
	cin >> b->real >> b->imag;

	return *a, *b;
}

Complex Complex::complexSum(Complex *a, Complex *b)
{
	a->real = a->real + b->real;
	b->imag = a->imag + b->imag;
```

```
	return printComplex(a, b);
}

Complex Complex::complexSubstraction(Complex *a, Complex *b)
{
	a->real = a->real - b->real;
	b->imag = a->imag - b->imag;
	return printComplex(a, b);
}

Complex Complex::complexMultiply(Complex *a, Complex *b)
{
	a->real = (a->real * b->real - a->imag * b->imag);
	b->imag = (a->real * b->imag + b->real * a->imag);
	return printComplex(a, b);
}

Complex Complex::complexDivision(Complex *a, Complex *b)
{
	a->real = ((a->real * b->real + a->imag * b->imag) / (b->real*b->real + b-
>imag*b->imag));
	b->imag = ((b->real * a->imag - a->real * b->imag) / (b->real*b->real + b-
>imag*b->imag));
	return printComplex(a, b);
}

// Определение методов наследственного класса.

double ComplexNew::complexModulo(ComplexNew *a)
{
	mod = sqrt(pow(a->real, 2) + pow(a->imag, 2));
	cout << "Modulo is = " << mod << endl;
	return mod;
}
```

Main.cpp

```
#include "stdafx.h"
#include "complex.h"

int main()
{
	setlocale(LC_ALL, "ru");
	char select = ' ';
	char ch;
	Complex *c1 = new Complex;
	Complex *c2 = new Complex;
	Complex *c3 = new Complex;
	ComplexNew *c = new ComplexNew;
	double *mod = new double;


	print_menu(select); // меню выбора операций
	do {
		cin >> select;
		switch (select)
		{
		case '+': //операция сложение
			c1, c2->EnterComplex(c1, c2);
			c3->complexSum(c1, c2);
```

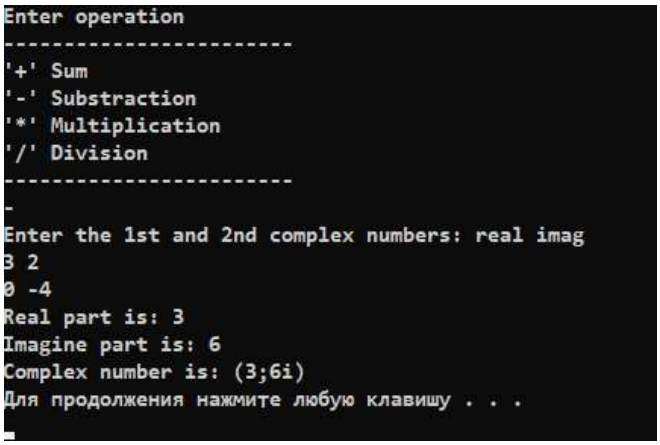
```
                break;

            case '-': // операция вычитание
                c1, c2->EnterComplex(c1, c2);
                c3->complexSubstraction(c1, c2);
                break;

            case '*': // операция умножение
                c1, c2->EnterComplex(c1, c2);
                c3->complexMultiply(c1, c2);
                break;

            case '/': // операция деление
                c1, c2->EnterComplex(c1, c2);
                c3->complexDivision(c1, c2);
                break;

            case 'M': // нахождение модуля комплексного числа
            case 'm':
                c->EnterComplex(c);
                c->complexModulo(c);
                break;

            default: // вызов ошибки в случае введения неверного символа.
                print_error();
                break;
            }
            cout << "\nПродолжать (y/n)? ";
            cin >> ch;
        } while (ch != 'n');
        system("pause");
        delete c1, c2, c3;
        delete c;
        delete mod;
}
```

Результат работы программы:

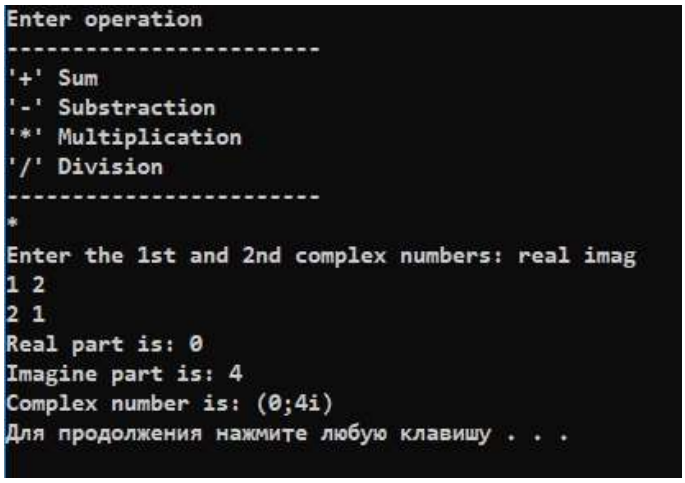
1.

```
Enter operation
-----------------------
'+' Sum
'-' Substraction
'*' Multiplication
'/' Division
-----------------------
+
Enter the 1st and 2nd complex numbers: real imag
5 2
10 7
Real part is: 15
Imagine part is: 9
Complex number is: (15;9i)
Для продолжения нажмите любую клавишу . . .
```

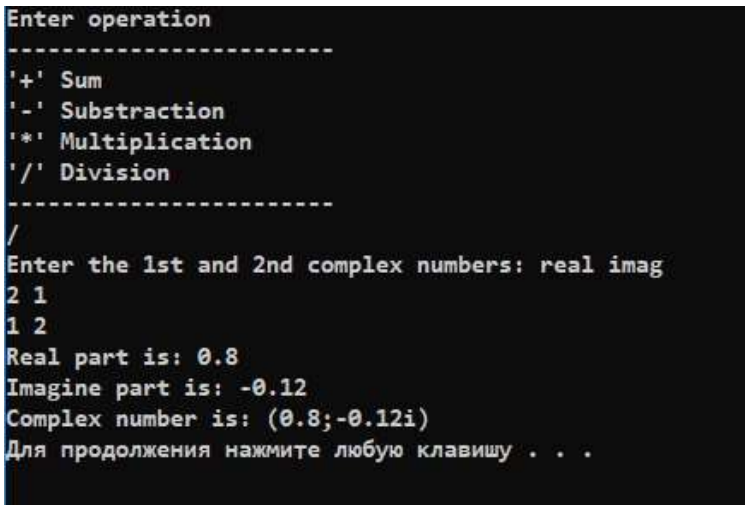
2.



3.



4.



3.4 Вывод

Рассмотрена работа с классами в языке C++.

Лабораторная работа №4

4.1 Тема и цель

Тема: Конструкторы и деструкторы.

Цель: Ознакомиться с конструкторами и деструкторами в языке C++.

4.2 Теоретические сведения

Конструкторы - это особые методы класса, используются для создания объектов и их инициализации, имеют имя, совпадающее с именем класса.

Конструктор существует для любого класса, причем он может быть создан без явных указаний программиста. Таким образом, для классов complex1 и goods существуют автоматически созданные конструкторы.

По умолчанию формируются конструктор без параметров и конструктор копирования вида.

```
1) T::T(const T&)
2) //где T - имя класса. Например,
3) class F {
4) ...
5) public: F(const F&);
6) ...
7) }
```

Такой конструктор существует всегда. По умолчанию конструктор копирования создается общедоступным.

Сформулируем несколько правил использования конструкторов.

1. В классе может быть несколько конструкторов (перегрузка), но только один с значениями параметров по умолчанию.
2. Нельзя получить адрес конструктора.
3. Параметром конструктора не может быть его собственный класс, но может быть ссылка на него, как у конструктора копирования.
4. Конструктор нельзя вызывать как обычную компонентную функцию. Для явного вызова конструктора можно использовать две разные синтаксические формы:

```
1) имя_класса имя_объекта(фактические_параметры_конструктора);
2) имя_класса(фактические_параматры_конструктора);
```

Первая форма допускается только при непустом списке фактических параметров. Она предусматривает вызов конструктора при определении нового объекта данного класса:

```
1) complex SS(10.3,0.22); // SS.real == 10.3;
2) // SS.imag == 0.22;
3) complex EE(2.345); // EE.real == 2.345;
4) // По умолчанию EE.imag = 0.0.
5) complex DD() ; // Ошибка! Компилятор решит, что это
6) // прототип функции без параметров,
```

```
7) // возвращающей значение типа complex.
```

Вторая форма явного вызова конструктора приводит к созданию объекта, не имеющего имени. Созданный таким вызовом безымянный объект может использоваться в тех выражениях, где допустимо использование объекта данного класса. Например:

```
1) complex ZZ = complex(4.0,5.0);
```

Этим определением создается объект zz, которому присваивается значение безымянного объекта (с элементами real == 4.0, imag == 5.0), созданного за счет явного вызова конструктора.

Существуют два способа инициализации данных объекта с помощью конструкторов. Первый способ, а именно передача значений параметров в тело конструктора, который уже продемонстрирован на примерах. Второй способ предусматривает применение списка инициализаторов данных объекта. Этот список помещается между списком параметров и телом конструктора:

```
1.имя-класса(список_параметров):
список_инициализаторов_компонентных_данных
 {
тело_конструктора }
```

Каждый инициализатор списка относится к конкретному компоненту и имеет вид:

```
1) имя_компонента_данных (выражение)
```

Например:

```
1) class AZ
2) { int ii; float ee; char cc;
3) public:
4) AZ(int in, float en, char cn) : ii(5),
5) ee(ii * en + in), cc(cn) { }
6) };
7) AZ A(2,3.0 , 'd');
8) // Создается именованный объект A
9) // с компонентами A.ii == 5,
10) // A.ee == 17, A.cc == 'd'.
11) AZ X = AZ(0,2.0,'z');
12) // Создается безымянный объект, в
13) // котором ii == 5, ee == 10,
14) // cc == 'z', и копируется в объект X.
```

Итак, бывают:

- Конструктор пустой stroka(){}
- Конструктор с параметрами stroka(int);
- Инициализирующий конструктор stroka(int N = 80) : len(0)
- Конструктор копирования stroka (const stroka &obj)

Деструкторы

Заслуживает внимания компонентная функция ~stroka(). Это деструктор. Объясним его назначение

и рассмотрим его свойства.

Динамическое выделение памяти для объектов какого-либо класса создает необходимость в освобождении этой памяти при уничтожении объекта. Например, если объект некоторого класса формируется как локальный внутри блока, то целесообразно, чтобы при выходе из блока, когда уже объект перестает существовать, выделенная для него память была возвращена системе. Желательно, чтобы освобождение памяти происходило автоматически и не требовало вмешательства программиста. Такую возможность обеспечивает специальный компонент класса --- деструктор (разрушитель объектов) класса. Для него предусматривается стандартный формат:

1. ~имя_класса() { операторы_тела_деструктора };

Название деструктора в C++ всегда начинается с символа тильда '~', за которым без пробелов или других разделительных знаков помещается имя класса.

Основные правила использования деструкторов следующие:

1. У деструктора не может быть параметров (даже типа void).

2. Деструктор не имеет возвращаемого значения (даже типа void).

3. Вызов деструктора выполняется неявно, автоматически, как только объект класса уничтожается.

В нашем примере в теле деструктора только один оператор, освобождающий память, выделенную для символьного массива при создании объекта класса stroka.

4.3 Основная часть

Задание

Класс Complex из прошлой лабораторной работы дополнить всеми видами конструкторов и деструкторами.

Код

Complex.h

```
#include <cmath>
#include <iostream>
#include <string.h>

using namespace std;

void print_error();
void print_menu(char select);

class Complex
{
protected:
	double real;
	double imag;
public:
	Complex();
```

```
        ~Complex();
        Complex printComplex(Complex *a, Complex *b);
        Complex EnterComplex(Complex *a);
        Complex EnterComplex(Complex *a, Complex *b);
        Complex complexSum(Complex *a, Complex *b);
        Complex complexSubstraction(Complex *a, Complex *b);
        Complex complexMultiply(Complex *a, Complex *b);
        Complex complexDivision(Complex *a, Complex *b);

        double getreal() {}
        double getImag() {}
        void setReal(double real)
        {
                this->real = real;
        }
        void setImag(double imag)
        {
                this->imag = imag;
        }
};

class ComplexNew : public Complex // наследование класса.
{
private:
        double mod;
public:
        double complexModulo(ComplexNew *a);
        double getMod() {}
        double setMod(double mod)
        {
                this->mod = mod;
        }
        ComplexNew();
        ~ComplexNew();
};
```

Complex.cpp

```
#include "stdafx.h"
#include "complex.h"

using namespace std;

void print_error()
{
        cout << "Error, this  symbol is not supported" << endl;

}
void print_menu(char select)
{
        cout << "Enter operation" << endl;
        cout << "------------------------" << endl;
        cout << "'+' Sum" << endl;
        cout << "'-' Substraction" << endl;
        cout << "'*' Division" << endl;
        cout << "'/' Multiplication" << endl;
        cout << "'M' or 'm' Modulo of complex number" << endl;
        cout << "------------------------" << endl;
```

```
}

// Определение методов класса COMPLEX.

Complex::Complex() // Конструктор класса
{
	this->real = 0.0;
	this->imag = 0.0;
}
Complex::~Complex() // Деструктор класса Complex
{
}

Complex Complex::printComplex(Complex *a, Complex *b)
{
	cout << "Real: " << a->real << endl;
	cout << "Imagine: " << b->imag << endl;
	cout << "Complex number is: (" << a->real << ';' << b->imag << "i)" << endl;

	return *a, *b;
}

 Complex Complex::EnterComplex(Complex *a)
{
	setlocale(LC_ALL, "ru");
	cout << "Enter complex number: real imag" << endl;
	cin >> a->real >> a->imag;

	return *a;
}

 Complex Complex::EnterComplex(Complex *a, Complex *b)
 {
	 setlocale(LC_ALL, "ru");
	 cout << "Enter first complex number: real imag " << endl;
	 cin >> a->real >> a->imag;
	 cout << "Enter second comlex number: real imag " << endl;
	 cin >> b->real >> b->imag;

	 return *a, *b;
 }

Complex Complex::complexSum(Complex *a, Complex *b)
{
	a->real = a->real + b->real;
	b->imag = a->imag + b->imag;
	return printComplex(a,b);

}

Complex Complex::complexSubstraction(Complex *a, Complex *b)
{
	a->real = a->real - b->real;
	b->imag = a->imag - b->imag;
	return printComplex(a,b);
}

Complex Complex::complexMultiply(Complex *a, Complex *b)
{
	a->real = (a->real * b->real - a->imag * b->imag);
	b->imag = (a->real * b->imag + b->real * a->imag);
```

```
	return printComplex(a,b);
}

Complex Complex::complexDivision(Complex *a, Complex *b)
{
	a->real = ((a->real * b->real + a->imag * b->imag)/(b->real*b->real + b-
>imag*b->imag));
	b->imag = ((b->real * a->imag - a->real * b->imag) / (b->real*b->real + b-
>imag*b->imag));
	return printComplex(a, b);
}


// Определение методов наследственного класса.

ComplexNew::ComplexNew()
{
	this->mod = 0.0;
}
ComplexNew::~ComplexNew()
{
}

double ComplexNew::complexModulo(ComplexNew *a)
{
	mod = sqrt(pow(a->real, 2) + pow(a->imag, 2));
	cout << "Modulo is = " << mod << endl;
	return mod;
}
```

Main.cpp

```
#include "stdafx.h"
#include "complex.h"

int main()
{
	setlocale(LC_ALL, "ru");
	char select = ' ';
	char ch;
	Complex *c1 = new Complex;
	Complex *c2 = new Complex;
	Complex *c3 = new Complex;
	ComplexNew *c = new ComplexNew;
	double *mod = new double;


	print_menu(select); // меню выбора операций

	do {

	cin >> select;
	switch (select)
	{
	case '+': //операция сложение
		c1, c2->EnterComplex(c1, c2);
		c3->complexSum(c1, c2);
		break;
```

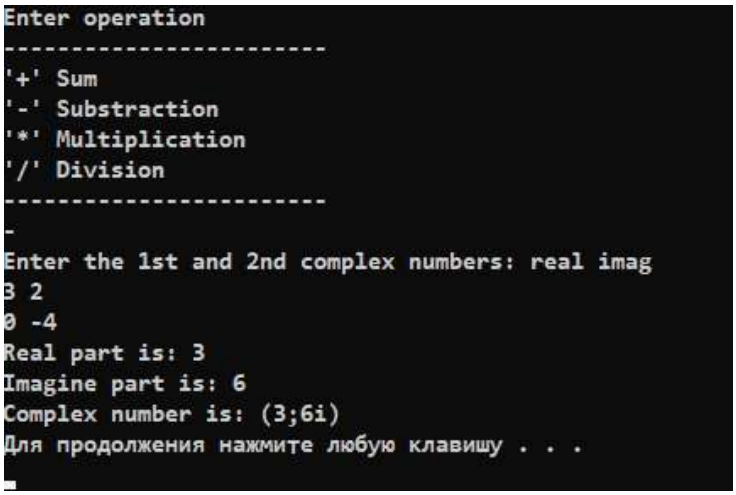
```
        case '-': // операция вычитание
               c1, c2->EnterComplex(c1, c2);
               c3->complexSubstraction(c1, c2);
               break;

        case '*': // операция умножение
               c1, c2->EnterComplex(c1, c2);
               c3->complexMultiply(c1, c2);
               break;

        case '/': // операция деление
               c1, c2->EnterComplex(c1, c2);
               c3->complexDivision(c1, c2);
               break;

        case 'M': // нахождение модуля комплексного числа
        case 'm':
               c->EnterComplex(c);
               c->complexModulo(c);
               break;

        default: // вызов ошибки в случае введения неверного символа.
               print_error();
               break;
               }
        cout << "\nПродолжать (y/n)? ";
        cin >> ch;
        } while (ch != 'n');
        system("pause");
        delete c1, c2, c3;
        delete c;
        delete mod;
}
```

Результат работы программы:

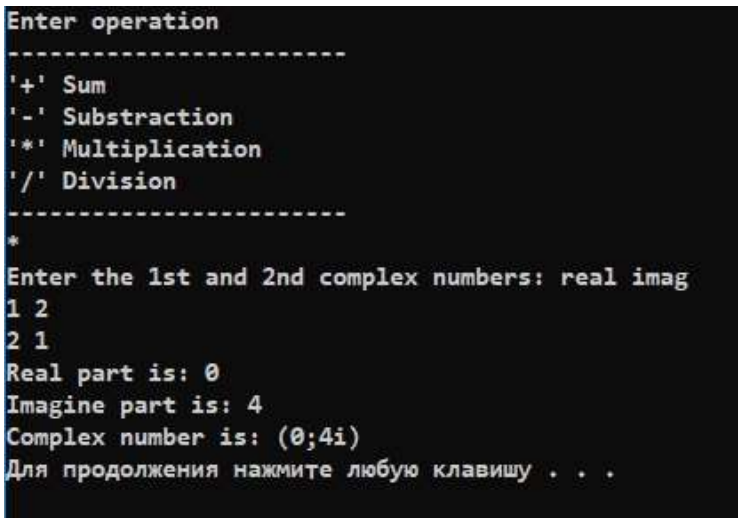
1.

```
Enter operation
-----------------------
'+' Sum
'-' Substraction
'*' Multiplication
'/' Division
-----------------------
+
Enter the 1st and 2nd complex numbers: real imag
5 2
10 7
Real part is: 15
Imagine part is: 9
Complex number is: (15;9i)
Для продолжения нажмите любую клавишу . . .
```

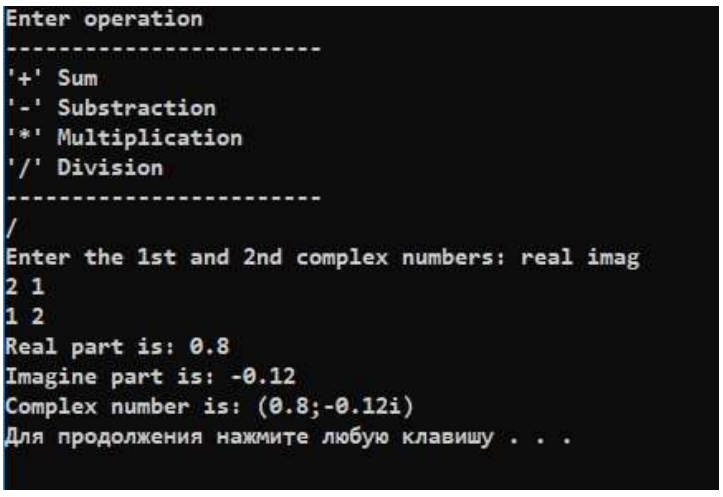
2.



3.



4.



4.4 Вывод

Рассмотрены конструкторы и деструкторы в языке Си+

Лабораторная работа №5

5.1 Тема и цель.

Тема: Перегрузка операторов в языке C++.

Цель: изучение перегрузки операторов в языке C++ и использование перегруженных операторов на практике.

5.2 Теоретические сведения

Синтаксис перегрузки операторов очень похож на определение функции с именем operator@, где @ — это идентификатор оператора (например +, -, <<, >>). Рассмотрим простейший пример:

```
1) class Integer
2) {
3) private:
4) int value;
5) public:
6) Integer(int i): value(i)
7) {}
8) const Integer operator+(const Integer& rv) const {
9) return (value + rv.value);
10) }
11) };
```

В данном случае, оператор оформлен как член класса, аргумент определяет значение,
находящееся в правой части оператора. Вообще, существует два основных способа перегрузки операторов: глобальные функции, дружественные для класса, или подставляемые функции самого класса. Какой способ, для какого оператора лучше, рассмотрим в конце топика.

В большинстве случаев, операторы (кроме условных) возвращают объект, или ссылку на тип, к которому относятся его аргументы (если типы разные, то вы сами решаете как интерпретировать результат вычисления оператора).

2. Перегрузка унарных операторов

Рассмотрим примеры перегрузки унарных операторов для определенного выше класса Integer. Заодно определим их в виде дружественных функций и рассмотрим операторы декремента и инкремента:

```
1) class Integer
2) {
3) private:
4) int value;
5) public:
6) Integer(int i): value(i)
7) {}
8)
9) //унарный +
10) friend const Integer& operator+(const Integer& i);
11)
```

```
12) //унарный -
13) friend const Integer operator-(const Integer& i);
14)
15) //префиксный инкремент
16) friend const Integer& operator++(Integer& i);
17)
18) //постфиксный инкремент
19) friend const Integer operator++(Integer& i, int);
20)
21) //префиксный декремент
22) friend const Integer& operator--(Integer& i);
23)
24) //постфиксный декремент
25) friend const Integer operator--(Integer& i, int);
26) };
27) //унарный плюс ничего не делает.
28) const Integer& operator+(const Integer& i) {
29) return i.value;
30) }
31)
32) const Integer operator-(const Integer& i) {
33) return Integer(-i.value);
34) }
35)
36) //префиксная версия возвращает значение после инкремента
37) const Integer& operator++(Integer& i) {
38) i.value++;
39) return i;
40) }
41)
42) //постфиксная версия возвращает значение до инкремента
43) const Integer operator++(Integer& i, int) {
44) Integer oldValue(i.value);
45) i.value++;
46) return oldValue;
47) }
48)
49) //префиксная версия возвращает значение после декремента
50) const Integer& operator--(Integer& i) {
51) i.value--;
52) return i;
53) }
54)
55) //постфиксная версия возвращает значение до декремента
56) const Integer operator--(Integer& i, int) {
57) Integer oldValue(i.value);
58) i.value--;
59) return oldValue;
60) };
```

3. Бинарные операторы

Рассмотрим синтаксис перегрузки бинарных операторов. Перегрузим один оператор, который возвращает l-значение, один условный оператор и один оператор, создающий новое значение (определим их глобально):

```
1) class Integer
2) {
3) private:
4) int value;
5) public:
6) Integer(int i): value(i)
7) {}
8) friend const Integer operator+(const Integer& left, const
Integer& right);
9)
10) friend Integer& operator+=(Integer& left, const Integer& right);
11)
12) friend bool operator==(const Integer& left, const Integer&
right);
13) };
14)
15) const Integer operator+(const Integer& left, const Integer& right) {
16) return Integer(left.value + right.value);
17) }
18)
19) Integer& operator+=(Integer& left, const Integer& right) {
20) left.value += right.value;
21) return left;
22) }
23)
24) bool operator==(const Integer& left, const Integer& right) {
25) return left.value == right.value;
26) }
```

Во всех этих примерах операторы перегружаются для одного типа, однако, это необязательно. Можно, к примеру, перегрузить сложение нашего типа Integer и определенного по его подобию Float.

4. Особые операторы

В C++ есть операторы, обладающие специфическим синтаксисом и способом перегрузки.

Например оператор индексирования []. Он всегда определяется как член класса и, так как подразумевается поведение индексируемого объекта как массива, то ему следует возвращать ссылку

Оператор разыменования указателя

Перегрузка этих операторов может быть оправдана для классов умных указателей. Этот оператор обязательно определяется как функция класса, причём на него накладываются некоторые ограничения: он должен возвращать либо объект (или ссылку), либо указатель, позволяющий обратиться к объекту.

Оператор присваивания

Оператор присваивания обязательно определяется в виде функции класса, потому что он неразрывно связан с объектом, находящимся слева от "=". Определение оператора присваивания в глобальном виде сделало бы возможным переопределение стандартного поведения оператора "=".

5. Неперегружаемые операторы

Некоторые операторы в C++ не перегружаются в принципе. По всей видимости, это сделано из соображений безопасности.

```
1) class Integer
2) {
3) private:
4) int value;
5) public:
6) Integer(int i): value(i)
7) {}
8)
9) Integer& operator=(const Integer& right) {
10) //проверка на самоприсваивание
11) if (this == &right) {
12) return *this;
13) }
14) value = right.value;
15) return *this;
16) }
17) };
```

• Оператор выбора члена класса «.".
• Оператор разыменования указателя на член класса ".*"
• В C++ отсутствует оператор возведения в степень (как в Fortran) "**".
• Запрещено определять свои операторы (возможны проблемы с определением приоритетов).
• Нельзя изменять приоритеты операторов

5.3 Основная часть

Задание

Для класса Complex перегрузить операторы присваивания, инкремента, декремента, сравнения, ввода и вывода.

Код

Complex.h

```
#include <cmath>
#include <iostream>
#include <string.h>

using namespace std;

void print_error();
void print_menu(char select);
void entercomplex();

class ComplexMain
{
protected:
	double real;
	double imag;
public:
	ComplexMain() : real(0), imag(0)
	{}
```

```
	~ComplexMain();
	ComplexMain(double real, double imag);
};

class Complex : public ComplexMain
{
public:
	Complex operator =(const Complex & other);
	Complex operator +( Complex & other);
	Complex operator - (Complex & other);
	Complex operator * (Complex & other);
	Complex operator / (Complex & other);
	friend ostream & operator << (ostream & os,  Complex & other);
	friend istream & operator >> (istream & os,  Complex & other);

	/*Complex(double real, double imag) : real(real), imag(imag) {}

	Complex operator + (Complex *c1)
	{
		return Complex(this->real + c1->real, this->imag + c1->imag);
	}*/
};

class ComplexNew : public Complex // наследование класса.
{
private:
	double mod;
public:
	double complexModulo(ComplexNew a);
	ComplexNew();
	~ComplexNew();
};
```

Complex.cpp

```
#include "stdafx.h"
#include "complex.h"
using namespace std;
//////////////////////////////////////////////////////////////////////////
void print_error()
{
	cout << "Error, this  symbol is not supported" << endl;

}
void print_menu(char select)
{
	cout << "Enter operation" << endl;
	cout << "------------------------" << endl;
	cout << "'+' Sum" << endl;
	cout << "'-' Substraction" << endl;
	cout << "'*' Division" << endl;
	cout << "'/' Multiplication" << endl;
	cout << "'M' or 'm' Modulo of complex number" << endl;
	cout << "------------------------" << endl;

}
void entercomplex()
{
	setlocale(LC_ALL, "ru");
	cout << "Enter 1st and 2nd complex numbers, real imag: ";
```

```
}

ComplexMain::ComplexMain(double real, double imag)
{
	this->real = real;
	this->imag = imag;
}
ComplexMain::~ComplexMain() // Деструктор класса Complex
{
}

// Перегрузка операторов
Complex Complex::operator =(const Complex & other)
{
	this->real = other.real;
	this->imag = other.imag;
	return other;
}

Complex Complex::operator + ( Complex &other)
{
	Complex temp;
	temp.real = this->real + other.real;
	temp.imag = this->imag + other.imag;
	return temp;
}

Complex Complex::operator - (Complex & other)
{
	Complex temp;
	temp.real = this->real - other.real;
	temp.imag = this->imag - other.imag;
	return temp;
}

Complex Complex::operator * (Complex & other)
{
	Complex temp;
	temp.real = (this->real * other.real - this->imag * other.imag);
	temp.imag = (this->real * other.imag + other.real * this->imag);
	return temp;
}

Complex Complex::operator / (Complex & other)
{
	Complex temp;
	temp.real = ( (this->real*other.real + this->imag*other.imag) /
(other.real*other.real + other.imag*other.imag) );
	temp.imag = ( (this->imag*other.real - this->real*other.imag) /
(other.real*other.real + other.imag*other.imag) );
	return temp;
}

ostream& operator << (ostream & os, Complex &other)
{
	os << "Real: " << other.real << endl;
	os << "Imagine: " << other.imag << endl;
	os << "Complex number is: (" << other.real << ';' << other.imag << "i)" <<
endl;
	return os;
}
```

```
istream & operator >> (istream & os, Complex & other)
{
	os >> other.real >> other.imag;
	return os;
}

// Определение методов производного класса ComplexNew
ComplexNew::ComplexNew()
{
	this->mod = 0.0;
}
ComplexNew::~ComplexNew()
{
}

double ComplexNew::complexModulo(ComplexNew a)
{
	mod = sqrt(pow(a.real, 2) + pow(a.imag, 2));
	cout << "Modulo is: " << mod << endl;
	return mod;
}
```

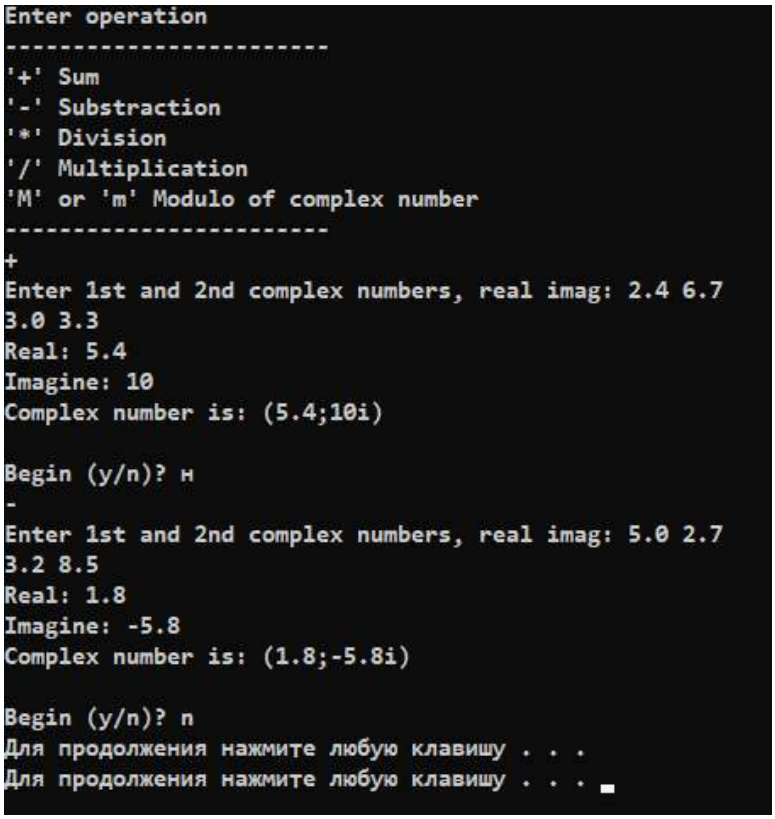
Main.cpp

```
#include "stdafx.h"
#include "complex.h"
using namespace std;

int main()
{

	setlocale(LC_ALL, "ru");
	char select = ' ';
	char ch;
	Complex c1, c2, c3;
	ComplexNew c;
	double *mod = new double;

	print_menu(select); // меню выбора операций
	do {

	cin >> select;
	switch (select)
	{
	case '+': //операция сложение
		entercomplex();
		cin >> c1 >> c2;
		c3 = c1 + c2;
		cout << c3;
		break;
	case '-': // операция вычитание
		entercomplex();
		cin >> c1 >> c2;
		c3 = c1 - c2;
		cout << c3;
		break;
	case '*': // операция умножение
		entercomplex();
		cin >> c1 >> c2;
```

```
                c3 = c1 * c2;
                cout << c3;
                break;
        case '/': // операция деление
                entercomplex();
                cin >> c1 >> c2;
                c3 = c1 / c2;
                cout << c3;
                break;
        case 'M': // нахождение модуля комплексного числа
        case 'm':
                cout << "Enter complex number, real imag: "; cin >> c;
                c.complexModulo(c);
                break;
        default: // вызов ошибки в случае введения неверного символа.
                print_error();
                break;
                }
        cout << "\nBegin (y/n)? ";
        cin >> ch;
        } while (ch != 'n');
        system("pause");
        delete mod;
}
```

Результат работы программы

1.



2.

```
Enter operation
-----------------------
'+' Sum
'-' Substraction
'*' Division
'/' Multiplication
'M' or 'm' Modulo of complex number
-----------------------
*
Enter 1st and 2nd complex numbers, real imag: 1 2 2 1
Real: 0
Imagine: 5
Complex number is: (0;5i)

Begin (y/n)? y
/
Enter 1st and 2nd complex numbers, real imag: 1 2 2 1
Real: 0.8
Imagine: 0.6
Complex number is: (0.8;0.6i)

Begin (y/n)? y
m
Enter complex number, real imag: 3 4
Modulo is: 5

Begin (y/n)? n
Для продолжения нажмите любую клавишу . . .
Для продолжения нажмите любую клавишу . . .
```

5.4 Вывод

Изучены перегрузка операторов в языке C++ и использование перегруженных операторов на практике

Лабораторная работа №6

6.1 Тема и цель

Тема: Наследование

Цель: Изучение наследования классов в языке C++.

6.2 Теоретические сведения

Наследование — это механизм создания нового класса на основе уже существующего. При этом к существующему классу могут быть добавлены новые элементы (данные и функции), либо существующие функции могут быть изменены. Основное назначение механизма наследования — повторное использование кодов, так как большинство используемых типов данных являются вариантами друг друга, и писать для каждого свой класс нецелесообразно.

Объекты разных классов и сами классы могут находиться в отношении наследования, при котором формируется иерархия объектов, соответствующая заранее предусмотренной иерархии классов.

Иерархия классов позволяет определять новые классы на основе уже имеющихся. Имеющиеся классы обычно называют базовыми(иногда порождающими), а новые классы, формируемые на основе базовых, – производными (порожденными, классами-потомками или наследниками).

Производные классы «получают наследство» – данные и методы своих базовых классов, и могут пополняться собственными компонентами (данными и собственными методами). Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах. Сообщение, обработку которого не могут выполнить методы производного класса, автоматически передается в базовый класс. Если для обработки сообщения нужны данные, отсутствующие в производном классе, то их пытаются отыскать автоматически в базовом классе.

При наследовании некоторые имена методов (функций-членов) и данных-членов базового класса могут быть по-новому определены в производном классе. В этом случае соответствующие компоненты базового класса становятся недоступными из производного класса. Для доступа из производного класса к компонентам базового класса, имена которых повторно определены в производном, используется операция разрешения контекста ::

Для порождения нового класса на основе существующего используется следующая общая форма:

```
class Имя : МодификаторДоступа ИмяБазовогоКласса
{ объявление_членов;};
```

При объявлении порождаемого класса МодификаторДоступа может принимать значения public, private, protected либо отсутствовать, по умолчанию используется значение private. В любом случае порожденный класс наследует все члены базового класса, но доступ имеет не ко всем. Ему доступны общие (public) члены базового класса и недоступны частные (private).
Для того, чтобы порожденный класс имел доступ к некоторым скрытым членам базового класса, в базовом классе их необходимо объявить со спецификацией доступа защищенные (protected). Члены класса с доступом protected видимы в пределах класса и в любом классе, порожденном из этого класса.

1. Модификатор доступа public – наследование

При общем наследовании порожденный класс имеет доступ к наследуемым членам базового класса с
видимостью public и protected. Члены базового класса с видимостью private – недоступны.

Спецификация доступа	Внутри класса	В порожденном классе	Вне класса
Private	+	-	-
Procected	+	+	-
Public	+	+	+

Общее наследование означает, что порожденный класс – это подтип базового класса. Таким образом, порожденный класс представляет собой модификацию базового класса, которая наследует общие и защищенные члены базового класса.
Пример:

```
1) class student {
2) protected:
3) char fac[20];
4) char spec[30];
5) char name[15];
6) public:
7) student(char *f, char *s, char *n);
8) void print();};
9) class grad_student : public student {
10) protected:
11) int year;
12) char work[30];
13) public:
14) grad_student(char *f, char *s, char *n, char *w, int y);
15) void print(); };
```

Порожденный класс наследует все данные класса student, имеет доступ к protected и public- членам базового класса. В новом классе добавлено два член-данных, и порожденный класс переопределяет функцию print().

```
1) student :: student(char *f, char *s, char *n) {
2) strcpy(fac, f);
3) strcpy(spec, s);
4) strcpy(name, n);
5) }
6) grad_student :: grad_student(char *f, char *s, char *n, char *w, int
y) :
7) student(f,s,n) {
8) year = y;
9) strcpy(work, w);
10) }
```

Конструктор для базового класса вызывается в списке инициализации.

Перегрузка функции print().

```
1) int main() {
2) system("chcp 1251");
3) system("cls");
4) student s("МТ", "АМСП", "Сидоров Иван");
5) grad_student stud("ПС", "УиТС", "Иванов Петр", "Метран", 2000);
6) student *p = &s;
7) p->print();
```

```
1) void student :: print() {
2) cout << endl << "fac: " << fac << " spec: " << spec
3) << " name: " << name;
4) }
5) void grad_student :: print() {
6) student :: print();
7) cout << " work: " << work << " year: " << year;
8) }
```

```
8) grad_student *gs = &stud;
9) student *m;
10) gs->print();
11) m = gs;
12) m->print();
13) cin.get();
14) return 0;
15) }
```

Указатель на порожденный класс может быть неявно передан в указатель на базовый класс. При этом переменная-указатель m на базовый класс может указывать на объекты как базового, так и порожденного класса.

Указатель на порожденный класс может указывать только на объекты порожденного класса. Неявные преобразования между порожденным и базовым классами называются предопределенными стандартными преобразованиями:

• объект порожденного класса неявно преобразуется к объекту базового класса.

• ссылка на порожденный класс неявно преобразуется к ссылке на базовый класс.

• указатель на порожденный класс неявно преобразуется к указателю на базовый класс.

2. Модификатор доступа private - наследование

Порожденный класс может быть базовым для следующего порождения. При порождении private наследуемые члены базового класса, объявленные как protected и public, становятся членами порожденного класса с видимостью private. При этом члены базового класса с видимостью public и protected становятся недоступными для дальнейших порождений. Цель такого порождения — скрыть классы или элементы классов от использования в дальнейших порождениях. При порождении private не выполняются предопределенные стандартные преобразования:

```
class grad_student : private student
{...};
int main() {...
grad_student *gs = &stud;
student *m;
gs->print();
m = gs; // ошибка
m->print();
cin.get();
return 0;
}
```

Однако порождение private позволяет отдельным элементам базового класса с видимостью public и protected сохранить свою видимость в порожденном классе. Для этого необходимо:

• в части protected порожденного класса указать те наследуемые члены базового класса с
видимостью protected, уточненные именем базового класса, для которых необходимо оставить видимость protected и в порожденном классе;

• в части public порожденного класса указать те наследуемые члены базового класса с видимостью public, уточненные именем базового класса, для которых необходимо оставить видимость public и в порожденном классе.

```
class X {
private:
int n;
protected:
int m;
char s;
public:
void func(int);
};
class Y : private X {
private:
...
protected:
...
X :: s;
public:
...
X :: func();
};
```

3. Модификатор доступа protected - наследование

Возможен и третий вариант наследования – с использованием модификатора доступа protected.

Доступ к элементам базового класса из производного класса, в зависимости от модификатора наследования:

Модификатор наследования -> Модификатор доступа	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	Нет доступа	Нет доступа	Нет доступа

4. Конструкторы и деструкторы при наследовании

Как базовый, так и производный классы могут иметь конструкторы и деструкторы. Если и у базового и у производного классов есть конструкторы и деструкторы, то конструкторы выполняются в порядке наследования, а деструкторы – в обратном порядке. То есть если A – базовый класс, B – производный из A, а C – производный из B (A-B-C), то при создании объекта класса C вызов конструкторов будет иметь следующий порядок:

- конструктор класса A
- конструктор класса B
- конструктор класса C.

Вызов деструкторов при удалении этого объекта произойдет в обратном порядке:

- деструктор класса C
- деструктор класса B
- деструктор класса A.

Поскольку базовый класс "не знает" о существовании производного класса, любая инициализация выполняется в нем независимо от производного класса, и, возможно, становится основой для инициализации, выполняемой в производном классе. Поскольку базовый класс лежит в основе производного, вызов деструктора базового класса раньше деструктора производного класса привел бы к преждевременному разрушению производного класса.

Конструкторы могут иметь параметры. При реализации наследования допускается передача параметров для конструкторов производного и базового класса. Если параметрами обладает только конструктор производного класса, то аргументы передаются обычным способом. При необходимости передать аргумент из производного класса конструктору родительского класса используется расширенная запись конструктора производного класса:

```
КонструкторПроизводногоКласса(СписокФормальныхАргументов)
: КонструкторБазовогоКласса(СписокФактическихАргументов)
{ // тело конструктора производного класса }
```

Для базового и производного классов допустимо использование одних и тех же аргументов. Возможно, списки аргументов конструкторов производного и базового классов будут различны.

Конструктор производного класса не должен использовать все аргументы, часть предназначены для передачи в базовый класс:

```
1) student :: student(char *f, char *s, char *n) {
2) strcpy(fac, f);
3) strcpy(spec, s);
4) strcpy(name, n);
5) }
6) grad_student :: grad_student(char *f, char *s, char *n, char *w, int
y) :
7) student(f,s,n) {
8) year = y;
9) strcpy(work, w);
10) }
```

В расширенной форме объявления конструктора производного класса описывается вызов конструктора базового класса.

6.3 Основная часть

Задание

1) Разработать иерархию классов «Студент» -> «Староста». Создать класс «Староста», производный от класса «Студент». Новый класс должен содержать несколько дополнительных методов и полей, расширяющих базовый класс.

2) Создать иерархию классов : линия - прямоугольник - пирамида. Все классы должны содержать методы для фиксации и получения значений всех координат, а производные классы методы вычисления площади (прямоугольник), площади поверхности и объема (пирамида)

3) Разработать класс для работы с многочленами. Создать производные классы от базового класса:

- класс, в котором хранятся в массиве степени при коэффициентах многочлена;
- класс, в котором многочлен так же хранится в виде отформатированной строки.

Код

Complex.h

```
#include <cmath>
#include <iostream>
#include <string.h>

using namespace std;

void print_error();
void print_menu(char select);
void entercomplex();

class ComplexMain
{
protected:
	double real;
	double imag;
public:
	ComplexMain() : real(0), imag(0)
	{}
	~ComplexMain();
	ComplexMain(double real, double imag);
};
```

```
class Complex : public ComplexMain
{
public:
	Complex operator =(const Complex & other);
	Complex operator +( Complex & other);
	Complex operator - (Complex & other);
	Complex operator * (Complex & other);
	Complex operator / (Complex & other);
	friend ostream & operator << (ostream & os,  Complex & other);
	friend istream & operator >> (istream & os,  Complex & other);

	/*Complex(double real, double imag) : real(real), imag(imag) {}

	Complex operator + (Complex *c1)
	{
		return Complex(this->real + c1->real, this->imag + c1->imag);
	}*/
};

class ComplexNew : public Complex // наследование класса.
{
private:
	double mod;
public:
	double complexModulo(ComplexNew a);
	ComplexNew();
	~ComplexNew();
};
```

Complex.cpp

```
#include "stdafx.h"
#include "complex.h"
using namespace std;
////////////////////////////////////////////////////////////////////////
void print_error()
{
	cout << "Error, this  symbol is not supported" << endl;

}
void print_menu(char select)
{
	cout << "Enter operation" << endl;
	cout << "------------------------" << endl;
	cout << "'+' Sum" << endl;
	cout << "'-' Substraction" << endl;
	cout << "'*' Division" << endl;
	cout << "'/' Multiplication" << endl;
	cout << "'M' or 'm' Modulo of complex number" << endl;
	cout << "------------------------" << endl;

}
void entercomplex()
{
	setlocale(LC_ALL, "ru");
	cout << "Enter 1st and 2nd complex numbers, real imag: ";
}
```

```
ComplexMain::ComplexMain(double real, double imag)
{
	this->real = real;
	this->imag = imag;
}
ComplexMain::~ComplexMain() // Деструктор класса Complex
{
}


// Перегрузка операторов
Complex Complex::operator =(const Complex & other)
{
	this->real = other.real;
	this->imag = other.imag;
	return other;
}

Complex Complex::operator + ( Complex &other)
{
	Complex temp;
	temp.real = this->real + other.real;
	temp.imag = this->imag + other.imag;
	return temp;
}

Complex Complex::operator - (Complex & other)
{
	Complex temp;
	temp.real = this->real - other.real;
	temp.imag = this->imag - other.imag;
	return temp;
}

Complex Complex::operator * (Complex & other)
{
	Complex temp;
	temp.real = (this->real * other.real - this->imag * other.imag);
	temp.imag = (this->real * other.imag + other.real * this->imag);
	return temp;
}

Complex Complex::operator / (Complex & other)
{
	Complex temp;
	temp.real = ( (this->real*other.real + this->imag*other.imag) /
(other.real*other.real + other.imag*other.imag) );
	temp.imag = ( (this->imag*other.real - this->real*other.imag) /
(other.real*other.real + other.imag*other.imag) );
	return temp;
}

ostream& operator << (ostream & os, Complex &other)
{
	os << "Real: " << other.real << endl;
	os << "Imagine: " << other.imag << endl;
	os << "Complex number is: (" << other.real << ';' << other.imag << "i)" <<
endl;
	return os;
}
```

```
istream & operator >> (istream & os, Complex & other)
{
	os >> other.real >> other.imag;
	return os;
}


// Определение методов производного класса ComplexNew
ComplexNew::ComplexNew()
{
	this->mod = 0.0;
}
ComplexNew::~ComplexNew()
{
}

double ComplexNew::complexModulo(ComplexNew a)
{
	mod = sqrt(pow(a.real, 2) + pow(a.imag, 2));
	cout << "Modulo is: " << mod << endl;
	return mod;
}
```

Main.cpp

```
#include "stdafx.h"
#include "complex.h"
using namespace std;

int main()
{
	setlocale(LC_ALL, "ru");
	char select = ' ';
	char ch;
	Complex c1, c2, c3;
	ComplexNew c;
	double *mod = new double;

	print_menu(select); // меню выбора операций
	do {

	cin >> select;
	switch (select)
	{
	case '+': //операция сложение
		entercomplex();
		cin >> c1 >> c2;
		c3 = c1 + c2;
		cout << c3;
		break;
	case '-': // операция вычитание
		entercomplex();
		cin >> c1 >> c2;
		c3 = c1 - c2;
		cout << c3;
		break;
	case '*': // операция умножение
		entercomplex();
```

```
            cin >> c1 >> c2;
            c3 = c1 * c2;
            cout << c3;
            break;
        case '/': // операция деление
            entercomplex();
            cin >> c1 >> c2;
            c3 = c1 / c2;
            cout << c3;
            break;
        case 'M': // нахождение модуля комплексного числа
        case 'm':
            cout << "Enter complex number, real imag: "; cin >> c;
            c.complexModulo(c);
            break;
        default: // вызов ошибки в случае введения неверного символа.
            print_error();
            break;
            }
        cout << "\nBegin (y/n)? ";
        cin >> ch;
        } while (ch != 'n');
        system("pause");
        delete mod;
}
```

Результат работы программы

1.

```
Enter operation
-----------------------
'+' Sum
'-' Substraction
'*' Division
'/' Multiplication
'M' or 'm' Modulo of complex number
-----------------------
+
Enter 1st and 2nd complex numbers, real imag: 2.4 6.7
3.0 3.3
Real: 5.4
Imagine: 10
Complex number is: (5.4;10i)

Begin (y/n)? н
-
Enter 1st and 2nd complex numbers, real imag: 5.0 2.7
3.2 8.5
Real: 1.8
Imagine: -5.8
Complex number is: (1.8;-5.8i)

Begin (y/n)? n
Для продолжения нажмите любую клавишу . . .
Для продолжения нажмите любую клавишу . . .
```

2.

```
Enter operation
-----------------------
'+' Sum
'-' Substraction
'*' Division
'/' Multiplication
'M' or 'm' Modulo of complex number
-----------------------
*
Enter 1st and 2nd complex numbers, real imag: 1 2 2 1
Real: 0
Imagine: 5
Complex number is: (0;5i)

Begin (y/n)? y
/
Enter 1st and 2nd complex numbers, real imag: 1 2 2 1
Real: 0.8
Imagine: 0.6
Complex number is: (0.8;0.6i)

Begin (y/n)? y
m
Enter complex number, real imag: 3 4
Modulo is: 5

Begin (y/n)? n
Для продолжения нажмите любую клавишу . . .
Для продолжения нажмите любую клавишу . . .
```

6.4 Вывод

Изучены наследования классов в языке C++.

Лабораторная работа №7

7.1 Тема и цель

Тема: Создание абстрактных классов

Цель: Целью данной лабораторной работы является изучение и создание абстрактных классов в языке C++.

7.2 Теоретические сведения

1. **Виртуальные функции**

Функция - метод класса может содержать спецификатор virtual. Такая функция называется виртуальной. Спецификатор virtual может быть использован только в объявлениях нестатических функций-членов класса.

Если некоторый класс содержит виртуальную функцию, а производный от него класс содержит функцию с тем же именем и типами формальных параметров, то обращение к этой функции для объекта производного класса вызывает функцию, определённую именно в производном классе.

Функция, определённая в производном классе, вызывается даже при доступе через указатель или ссылку на базовый класс. В таком случае говорят, что функция производного класса подменяет функцию базового класса. Если типы функций различны, то функции считаются разными, и механизм виртуальности не включается. Ошибкой является различие между функциями только в типе возвращаемого значения.

```
1) class Base
2) { public:
3) virtual void f1();
4) virtual void f2();
5) virtual void f3();
6) void f();
7) };
```

```
1) class Derived : public Base
2) { public:
3) void f1();
4) void f2(int);
5) char f3();
6) void f();
7) };
8) / Скрывает Base::f2()
9) // Ошибка – различие только в типе возвращаемого значения!
```

```
1) Derived *dp = new Derived;
2) Base *bp = dp;
3) // Преобразование указателя на производный класс в указатель на
базовый класс
4)
5) bp->f(); // Вызов Base::f
6) dp->f(); // Вызов Derived::f
7) dp->Base::f(); // Вызов Base::f
```

```
1) dp->f1(); // Вызов Derived::f1
2) bp->f1(); // Всё равно вызов Derived::f1!!!
```

```
3) bp->Base::f1();// Вызов Base::f1 (использование явного квалификатора
блокирует механизм виртуальности)
```

```
1) bp->f2();
2) dp->f2(0);
3) dp->f2();
4) dp->Base::f2();
5)
6) // Вызов Base::f2
7) // Вызов Derived::f2
8) // Ошибка, т.к. не
```

Виртуальную функцию можно использовать, даже если у её класса нет производных классов. Производный класс, который не нуждается в собственной версии виртуальной функции, не обязан её реализовывать.

Интерпретация вызова виртуальной функции зависит от типа объекта, для которого она вызывается, в то время как интерпретация вызова не виртуальной функции-метода класса зависит от типа указателя или ссылки, указывающей на этот объект.

Этот механизм делает производные классы и виртуальные функции ключевыми понятиями при разработке программ на C++. Базовый класс определяет интерфейс, для которого производные классы обеспечивают набор реализаций. Указатель на объект класса может передаваться в контекст, где известен интерфейс, определённый одним из его базовых классов, но этот производный класс неизвестен. Механизм виртуальных функций гарантирует, что этот объект всё равно будет обрабатываться функциями, определёнными для него, а не для базового класса.

Спецификатор virtual предполагает принадлежность функции классу, поэтому виртуальная функция не может быть ни глобальной функцией, ни статическим членом класса, поскольку вызов виртуальной функции нуждается в конкретном объекте для выяснения того, какую именно функцию следует вызывать.

Подменяющая функция в производном классе также считается виртуальной, даже при отсутствии спецификатора virtual.

Виртуальная функция может быть объявлена дружественной в другом классе. Такое поведение, когда функции базового класса подменяются функциями производного класса, независимо от типа указателя или ссылки, называется полиморфизмом. Тип, имеющий виртуальные функции, называется полиморфным типом. Для достижения полиморфного поведения в языке C++ вызываемые функции-члены класса должны быть виртуальными, и доступ к объектам должен осуществляться через ссылки или указатели. При непосредственных манипуляциях с объектом (без использования указателя или ссылки) его точный тип известен компилятору, и поэтому полиморфизм времени выполнения не требуется.

Для реализации механизма виртуальности используются таблицы виртуальных функций. Каждый объект класса, имеющего виртуальные функции, содержит таблицу виртуальных функций, в которой хранятся адреса виртуальных функций, определённых для того класса, к которому реально принадлежит объект. Поскольку при создании объекта его тип известен, компилятор может определить адреса виртуальных функций этого класса и записать их в таблицу виртуальных функций. При вызове виртуальной функции её адрес определяется не на этапе компиляции, а во время выполнения программы. Из таблицы виртуальных функций берётся элемент с определённым номером и вызывается функция, находящаяся по этому адресу. Таким образом, для вызова

виртуальной функции требуется одна дополнительная операция выбора значения из таблицы виртуальных функций.

Соображения эффективности ни в коем случае не должны считаться препятствием для использования виртуальных функций. Для реализации механизма виртуальности требуется количество памяти, равное размеру указателя, на каждую виртуальную функцию и одна операция выбора значения из памяти на каждый вызов виртуальной функции. Однако другие методы, реализующие подобный механизм, также потребуют накладных расходов. Например, при использовании поля типа также требуется память и время на выполнение проверок, причем если количество памяти невелико, то временные затраты, особенно при большом количестве производных классов, могут быть несопоставимо больше. Кроме того, механизм виртуальных функций позволяет легко модифицировать программы и добавлять новые классы.

7.3 Основная часть

Задание

Создать базовый абстрактный класс «Человек», который имеет нереализованную виртуальную функцию вывода информации на экран. Затем создать классы потомки «Ученик» и «Босс», унаследованные от него.

Код

Main.cpp

```
// vitrpers.cpp
// виртуальные функции и класс person
#include "stdafx.h"
#include <iostream>
using namespace std;
////////////////////////////////////////////
class Person //класс person
{
protected:
	char name[100];
	char surname[100];
public:
	void getName()
	{

		cout << " Введите имя и фамилию: "; cin >> name >> surname;

	}
	void putName()
	{

		cout << "\n Имя: " << name << ' ' << surname << endl;

	}
	virtual void getData() = 0; //чистые
	virtual bool isOutstanding() = 0; //виртуальные
                                                    //функции
};
////////////////////////////////////////////
class Student : public Person //класс student
{
private:
```

```
	float gpa; //средний балл
public:
	void getData() //запросить данные об ученике у пользователя
	{

		Person::getName();
		cout << " Средний балл ученика: "; cin >> gpa;

	}

	bool isOutstanding()
	{

		return (gpa > 3.5) ? true : false;

	}
};
////////////////////////////////////////////
class Professor : public Person //класс professor
{
private:
	int numPubs; //число публикаций
public:

	void getData() //запросить данные о педагоге у
	{ //пользователя

		Person::getName();
		cout << " Число публикаций: "; cin >> numPubs;

	}
	bool isOutstanding()
	{

		return (numPubs > 100) ? true : false;

	}
};
////////////////////////////////////////////
int main()
{
	setlocale(LC_ALL, "ru");
	Person* persPtr[100]; //массив указателей на person
	int n = 0; //число людей, внесенных в список char choice;
	char choice;

	do {

		cout << " Учащийся (s) или педагог (p): ";
		cin >> choice;
		if (choice == 's') //Занести нового ученика
			persPtr[n] = new Student; // в массив
		else //Занести нового
			persPtr[n] = new Professor; //педагога в массив
		persPtr[n++]->getData(); //Запрос данных о персоне
		cout << " Ввести еще персону (y/n)? "; //создать еще персону
		cin >> choice;

	} while (choice == 'y'); //цикл, пока ответ 'y'

	for (int j = 0; j<n; j++)
```

```
        {

                persPtr[j]->putName(); //Вывести все имена,
                if (persPtr[j]->isOutstanding()) //сообщать о
                        cout << " Это выдающийся человек!\n\n"; //выдающихся
                else { cout << " Это не выдающийся человек :(\n\n";  }

        }
        return 0;
} //Конец main()
```

Результат работы программы

```
Учащийся (s) или педагог (p): s
Введите имя и фамилию: Anton Sorokin
Средний балл ученика: 2.7
Ввести еще персону (y/n)? y
Учащийся (s) или педагог (p): p
Введите имя и фамилию: Mihail Trinozhenko
Число публикаций: 127
Ввести еще персону (y/n)? n

Имя: Anton Sorokin
Это не выдающийся человек :(


Имя: Mihail Trinozhenko
Это выдающийся человек!
```

7.4 Вывод

Изучение наследования классов в языке C++.

Лабораторная работа №8

8.1 Тема и цель

Тема: Библиотека STL

Цель: знакомство с библиотекой STL – стандартной библиотекой шаблонов - в языке C++, а также показать ее использование на примерах.

8.2 Теоретические сведения

Собственно, сам механизм шаблонов был встроен в компилятор C++ с целью дать возможность программистам C++ создавать эффективные и компактные библиотеки. Естественно, через некоторое время была создана одна из библиотек, которая впоследствии и стала стандартной частью C++. STL -- это самая эффективная библиотека для C++, существующая на сегодняшний день.

Сегодня существует великое множество реализаций стандартной библиотеки шаблонов, которые следуют стандарту, но при этом предлагают свои расширения, что является с одной стороны плюсом, но, с другой, не очень хорошо, поскольку не всегда можно использовать код повторно с другим компилятором. Поэтому я рекомендую вам все же оставаться в рамках стандарта, даже если вы в дальнейшем очень хорошо разберетесь с реализацией вашей библиотеки.

Начнем рассмотрение с краткого обзора основных коллекций. Каждая STL коллекция имеет собственный набор шаблонных параметров, который необходим ей для того, чтобы на базе шаблона реализовать тот или иной класс, максимально приспособленный для решения конкретных задач. Какой тип коллекции вы будете использовать, зависит от ваших задач, поэтому необходимо знать их внутреннее устройство для наиболее эффективного использования. Рассмотрим наиболее часто используемые типы коллекций. Реально в STL существует несколько большее количество коллекций, но, как показывает практика, нельзя объять необъятное сразу. Поэтому, для начала, рассмотрим наиболее популярные из них, которые с большой вероятностью могут встретиться в чужом коде. Тем более, что этих коллекций более чем достаточно для того, чтобы решить 99% реально возникающих задач.

vector - коллекция элементов T, сохраненных в массиве, увеличиваемом по мере необходимости. Для того, чтобы начать использование данной коллекции, включите #include <vector>.

list - коллекция элементов T, сохраненных, как двунаправленный связанный список. Для того, чтобы начать использование данной коллекции, включите #include <list>.

map - это коллекция, сохраняющая пары значений pair<const Key, T>. Эта коллекция предназначена для быстрого поиска значения T по ключу const Key. В качестве ключа может быть использовано все, что угодно, например, строка или int но при этом необходимо помнить, что главной особенностью ключа является возможность применить к нему операцию сравнения. Быстрый поиск значения по ключу осуществляется благодаря тому, что пары хранятся в отсортированном виде. Эта коллекция имеет соответственно и недостаток - скорость вставки новой пары обратно пропорциональна количеству элементов, сохраненных в коллекции, поскольку просто добавить новое значение в конец коллекции не получится. Еще одна важная вещь, которую необходимо помнить при использовании данной коллекции - ключ должен быть уникальным. Для того, чтобы начать использование данной коллекции, включите #include <map>. Если вы хотите

использовать данную коллекцию, чтобы избежать дубликатов, то вы избежите их только по ключу.

set — это коллекция уникальных значений const Key - каждое из которых является также и ключом - то есть, проще говоря, это отсортированная коллекция, предназначенная для быстрого поиска необходимого значения. К ключу предъявляются те же требования, что и в случае ключа для map. Естественно, использовать ее для этой цели нет смысла, если вы хотите сохранить в ней простые типы данных, по меньшей мере вам необходимо определить свой класс, хранящий пару ключ - значение и определяющий операцию сравнения по ключу. Очень удобно использовать данную коллекцию, если вы хотите избежать повторного сохранения одного и того же значения. Для того, чтобы начать использование данной коллекции, включите #include <set>.

multimap — это модифицированный map, в котором отсутствует требования уникальности ключа - то есть, если вы произведете поиск по ключу, то вам вернется не одно значение, а набор значений, сохраненных с данным ключом. Для того, чтобы начать использование данной коллекции включите #include <map>.

multiset - то же самое относится и к этой коллекции, требования уникальности ключа в ней не существует, что приводит к возможности хранения дубликатов значений. Тем не менее, существует возможность быстрого нахождения значений по ключу в случае, если вы определили свой класс. Поскольку все значения в map и set хранятся в отсортированном виде, то получается, что в этих коллекциях мы можем очень быстро отыскать необходимое нам значение по ключу, но при этом операция вставки нового элемента T будет стоить нам несколько дороже, чем, например, в vector. Для того, чтобы начать использование данной коллекции, включите #include <set>.

STL Строки

Не существует серьезной библиотеки, которая бы не включала в себя свой класс для представления строк или даже несколько подобных классов. STL - строки поддерживают как формат ASCII, так и формат Unicode.

string - представляет из себя коллекцию, хранящую символы char в формате ASCII. Для того, чтобы использовать данную коллекцию, вам необходимо включить #include <string>.

wstring — это коллекция для хранения двухбайтных символов wchar_t, которые используются для представления всего набора символов в формате Unicode. Для того, чтобы использовать данную коллекцию, вам необходимо включить #include <xstring>.
Используются для организации сохранения простых типов данных в STL строки в стиле C++.

Практическое знакомство с STL мы начнем именно с этого класса. Ниже приведена простая программа, демонстрирующая возможности использования строковых потоков:

```
1) //stl.cpp: Defines the entry point for the console application
2) #include "stdafx.h"
3) #include <iostream>
4) #include <strstream>
5) #include <string>
6) using namespace std;
7) int _tmain (int argc, _TCHAR* argv [])
8) {
9) strstream xstr;
10) for (int i = 0; i < 10; i++)
```

```
11) {
12) xstr << "Demo " << i << endl;
13) }
14) cout << xstr.str ();
15) string str;
16) str.assign (xstr.str (), xstr.pcount ());
17) cout << str.c_str ();
18) return 0;
19) }
```

Строковый поток - это просто буфер, в конце которого установлен нуль терминатор, поэтому мы наблюдаем в конце строки мусор при первой распечатке, то есть реальный конец строки определен не посредством нуль терминатора, а с помощью счетчика, и его размер мы можем получить с помощью метода: pcount ().

Далее мы производим копирование содержимого буфера в строку и печатаем строку второй раз. На этот раз она печатается без мусора.

Основные методы, которые присутствуют почти во всех STL коллекциях, приведены ниже.

empty - определяет, является ли коллекция пустой.

size - определяет размер коллекции.

begin - возвращает прямой итератор, указывающий на начало коллекции.

end - возвращает прямой итератор, указывающий на конец коллекции. При этом надо учесть, что реально он не указывает на ее последний элемент, а указывает на воображаемый несуществующий элемент, следующий за последним.

rbegin - возвращает обратный итератор, указывающий на начало коллекции.

rend - возвращает обратный итератор, указывающий на конец коллекции. При этом надо учесть, что реально он не указывает на ее последний элемент, а указывает на воображаемый несуществующий элемент, следующий за последним.

clear - удаляет все элементы коллекции, при этом, если в вашей коллекции сохранены указатели, то вы должны не забыть удалить все элементы вручную посредством вызова delete для каждого указателя.

erase - удаляет элемент или несколько элементов из коллекции.

capacity - вместимость коллекции определяет реальный размер - то есть размер буфера коллекции, а не то, сколько в нем хранится элементов. Когда вы создаете коллекцию, то выделяется некоторое количество памяти. Как только размер буфера оказывается меньшим, чем размер, необходимый для хранения всех элементов коллекции, происходит выделение памяти для нового буфера, а все элементы старого копируются в новый буфер. При этом размер нового буфера будет в два раза большим, чем размер буфера, выделенного перед этим - такая стратегия позволяет уменьшить количество операций перераспределения памяти, но при этом очень расточительно расходуется память. Причем в некоторых реализациях STL первое выделение памяти происходит не в конструкторе, а как ни странно, при добавлении первого элемента коллекции. Фрагмент программы ниже демонстрирует, что размер и вместимость коллекции - две разные сущности:

```
1) vector<int> vec;
2) cout << "Real size of array in vector: " << vec.capacity () << endl;
3) for (int j = 0; j < 10; j++)
4) {
5) vec.push_back (10);
6) }
```

```
7) cout << "Real size of array in vector: " << vec.capacity () << endl;
8) return 0;
```

8.3 Основная часть

Задание

Используйте шаблон контейнера list для организации двусвязного списка данных для хранения данных объектов класса Complex.

Код

Complex.h

```
#include <cmath>
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <list>
using namespace std;

void print_error();
void print_menu(char select);
void entercomplex();

class ComplexMain
{
protected:
	double real;
	double imag;
public:
	ComplexMain() : real(0), imag(0)
	{}
	~ComplexMain();
	ComplexMain(double real, double imag);
};



class Complex : public ComplexMain
{
public:
	Complex operator = (const Complex & other);
	Complex operator + (const Complex & other);
	Complex operator - (const Complex & other);
	Complex operator * (const Complex & other);
	Complex operator / (const Complex & other);
	friend ostream & operator << (ostream & os, const Complex & other);
	friend istream & operator >> (istream & os, Complex & other);
};



class ComplexNew : public Complex // наследование класса.
{
private:
	double mod;
public:
	double complexModulo(ComplexNew a);
```

```
	ComplexNew();
	~ComplexNew();
};
```

Complex.cpp

```
#include "stdafx.h"
#include "complex.h"
using namespace std;
////////////////////////////////////////////////////////////////////////
void print_error()
{
	cout << "Error, this  symbol is not supported" << endl;

}
void print_menu(char select)
{
	cout << "Enter operation" << endl;
	cout << "-----------------------" << endl;
	cout << "'+' Sum" << endl;
	cout << "'-' Substraction" << endl;
	cout << "'*' Division" << endl;
	cout << "'/' Multiplication" << endl;
	cout << "'M' or 'm' Modulo of complex number" << endl;
	cout << "-----------------------" << endl;

}
void entercomplex()
{
	setlocale(LC_ALL, "ru");
	cout << "Enter 1st and 2nd complex numbers, real imag: ";
}

ComplexMain::ComplexMain(double real, double imag)
{
	this->real = real;
	this->imag = imag;
}
ComplexMain::~ComplexMain() // Деструктор класса Complex
{
}


// Перегрузка операторов
Complex Complex::operator = (const Complex & other)
{
	this->real = other.real;
	this->imag = other.imag;
	return other;
}

Complex Complex::operator + (const Complex &other)
{
	Complex temp;
	temp.real = this->real + other.real;
	temp.imag = this->imag + other.imag;
	return temp;
}

Complex Complex::operator - (const Complex & other)
```

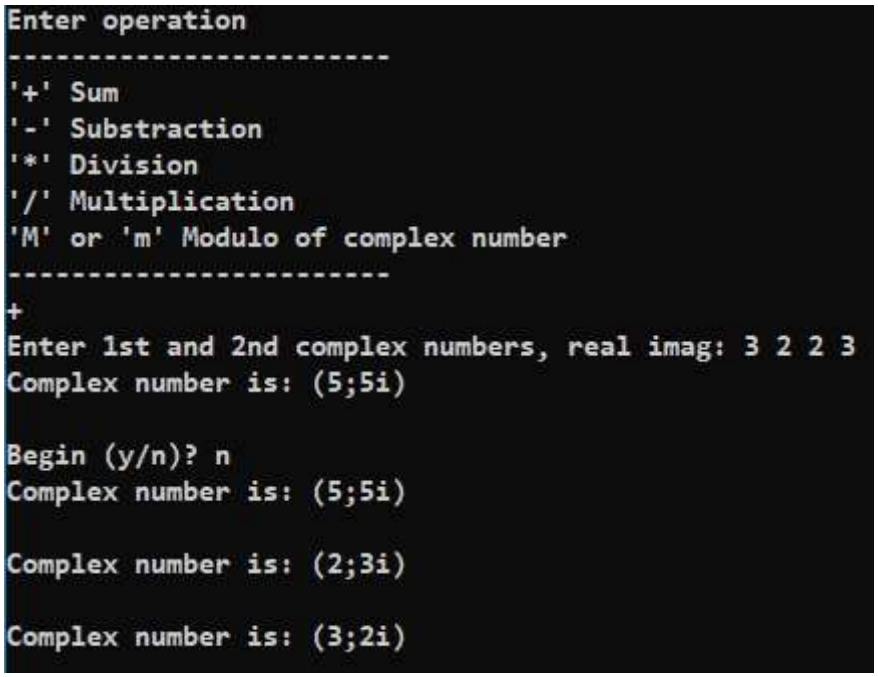
```
{
	Complex temp;
	temp.real = this->real - other.real;
	temp.imag = this->imag - other.imag;
	return temp;
}

Complex Complex::operator * (const Complex & other)
{
	Complex temp;
	temp.real = (this->real * other.real - this->imag * other.imag);
	temp.imag = (this->real * other.imag + other.real * this->imag);
	return temp;
}

Complex Complex::operator / (const Complex & other)
{
	Complex temp;
	temp.real = ( (this->real*other.real + this->imag*other.imag) /
(other.real*other.real + other.imag*other.imag) );
	temp.imag = ( (this->imag*other.real - this->real*other.imag) /
(other.real*other.real + other.imag*other.imag) );
	return temp;
}

ostream& operator <<(ostream & os, const Complex &other)
{

	os << "Complex number is: (" << other.real << ';' << other.imag << "i)" <<
endl;
	return os;
}


istream & operator >> (istream & os, Complex & other)
{
	os >> other.real >> other.imag;
	return os;
}



// Определение методов производного класса ComplexNew
ComplexNew::ComplexNew()
{
	this->mod = 0.0;
}
ComplexNew::~ComplexNew()
{
}

double ComplexNew::complexModulo(ComplexNew a)
{
	mod = sqrt(pow(a.real, 2) + pow(a.imag, 2));
	cout << "Modulo is: " << mod << endl;
	return mod;
}
```

Main.cpp

```
#include "stdafx.h"
#include "complex.h"


using namespace std;

template <typename T>
void PrintList(const list<T> &list)
{
	for (auto i = list.cbegin(); i != list.cend(); i++)
	{
		cout << "\t" << *i;
	}
}

int main()
{
	char select = ' ';
	char ch;
	double *mod = new double;

	list<Complex> myList;
	auto it = myList.begin();
	Complex c1, c2, c3;
	ComplexNew c;
	ofstream of;

	print_menu(select); // меню выбора операций
	do {

	cin >> select;
	switch (select)
	{
	case '+': //операция сложение

	{
		entercomplex();
		cin >> c1 >> c2;
		c3 = c1 + c2;
		cout << c3;
		myList.push_front(c3);
		myList.push_back(c2);
		myList.push_back(c1);

	}
		break;
	case '-': // операция вычитание
	{

		entercomplex();
		cin >> c1 >> c2;
		c3 = c1 - c2;
		cout << c3;
		myList.push_front(c3);
		myList.push_back(c2);
		myList.push_back(c1);

	}
```

```
            break;
        case '*': // операция умножение

            entercomplex();
            cin >> c1 >> c2;
            c3 = c1 * c2;
            cout << c3;
            myList.push_front(c3);
            myList.push_back(c2);
            myList.push_back(c1);
            break;
        case '/': // операция деление

            entercomplex();
            cin >> c1 >> c2;
            c3 = c1 / c2;
            cout << c3;
            myList.push_front(c3);
            myList.push_back(c2);
            myList.push_back(c1);

            break;
        case 'M': // нахождение модуля комплексного числа
        case 'm':
            cout << "Enter complex number, real imag: "; cin >> c;
            c.complexModulo(c);
            break;
        default: // вызов ошибки в случае введения неверного символа.
            print_error();
            break;
            }
        cout << "\nBegin (y/n)? ";
        cin >> ch;
        } while (ch != 'n');


        for (auto i = myList.begin(); i != myList.end(); i++)
        {
            cout << *i << endl;
        }

        system("pause");
        delete mod;
}
```

Результат работы программы



8.4 Вывод

Произведено знакомство с библиотекой STL – стандартной библиотекой шаблонов – в языке C++, а также показано ее использование на примерах.