Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

Пермский национальный исследовательский политехнический университет

Электротехнический факультет

Кафедра информационных технологий и автоматизированных систем

**ОТЧЕТ**

**о работе по информатике**

Семестр: 2

На тему: «Классы и объекты. Инкапсуляция»

Выполнил студент ИВТ-22-2б:

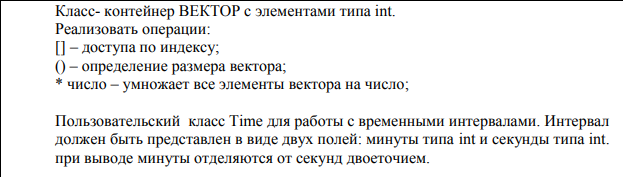
Захаров Дмитрий Сергеевич

Проверил доцент кафедры ИТАС:

Полякова Ольга Андреевна

Пермь 2023

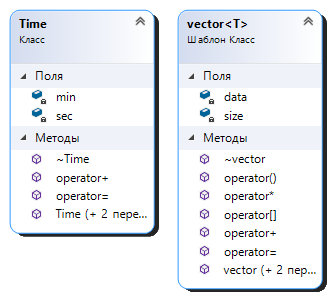
**1 Постановка задачи**

****

**2 Анализ задачи**

За основу берется 6 лабораторная работа, но в отличии от 6, в 7 класс Vectror является шаблонным.

**3 UML-диаграмма**

****

**4 Код программы**

#include <iostream>

#include "Time.h"

#include "vector.h"

using namespace std;

int main()

{

setlocale(LC\_ALL, "ru");

vector<int> a(5, 0);

cout << a << endl;

cin >> a;

cout << "Контейнер a: " << a << endl;

cout << "Размер контейнера а: " << a() << endl;

vector<int>b(10, 1);

b = a;

cout << "Контейнер b: " << b << endl;

vector<int> c(10, 2);

c = b \* 4;

cout << "Контейнер с(b\*4): " << c << endl;

cout << "Контейнер с + 4: " << c + 4 << endl;

Time t;

cin >> t;

cout << "Промежуток t: " << t << endl;

int tmp;

cout << "Введите количество секунд для сложения: ";

cin >> tmp;

cout << "t+tmp: " << t + tmp << endl;

Time s;

cin >> s;

cout << "Промежуток s: " << s << endl;

s = s + t;

cout << "Сложение двух промежутков: " << s << endl;

vector <Time> p(5, t);

cin >> p;

cout << "Контейнер р из временных промежутков: " << p << endl;

vector <Time> o(10, t);

cout << "Контейнер o, где каждая ячейка = промежутку t: " << o << endl;

o = p;

cout << "o=p: " << o << endl;

cout << "p[2]: " << p[2] << endl;

cout << "Размер: " << p() << endl;

o = p + t;

cout << "o+t: " << o;

return 0;

}

**Vector.h**

#pragma once

#include <iostream>

using namespace std;

template <class T>

class vector {

private:

int size;

T\* data;

public:

vector(int s, T k);

vector(const vector<T>& a);

~vector();

vector();

vector& operator=(const vector<T>& a);

T& operator[](int index);

vector operator\*(const T k);

vector operator+(const T k);

int operator()();

friend ostream& operator<< <>(ostream& out, const vector<T>& a);

friend istream& operator>> <>(istream& in, vector<T>& a);

};

template <class T>

vector<T> vector<T> :: operator+(const T k) {

vector<T> tmp(size, k);

for (int i = 0; i < size; i++)

tmp.data[i] = data[i] + k;

return tmp;

};

template <class T>

vector<T>::vector() {

size = 0;

data = 0;

}

template <class T>

vector<T>::vector(int s, T k) {

size = s;

data = new T[size];

for (int i = 0; i < size; i++)

data[i] = k;

};

template <class T>

vector<T>::vector(const vector<T>& a) {

size = a.size;

data = new T[size];

for (int i = 0; i < size; i++)

data[i] = a.data[i];

};

template <class T>

vector<T>::~vector() {

delete[] data;

data = 0;

};

template <class T>

vector<T>& vector<T>:: operator=(const vector<T>& a) {

if (this == &a)

return \*this;

size = a.size;

if (data != 0) delete[] data;

data = new T[size];

for (int i = 0; i < size; i++)

data[i] = a.data[i];

return \*this;

};

template <class T>

T& vector<T>:: operator[](int index) {

if (index < size || index>0) return data[index];

else cout << "Такого индекса нет!";

};

template <class T>

vector<T> vector<T>:: operator\*(const T k) {

vector<T> tmp(size, k);

for (int i = 0; i < size; i++)

tmp.data[i] += data[i] \* k;

return tmp;

};

template <class T>

int vector<T>:: operator()() {

return size;

};

template <class T>

ostream& operator<<(ostream& out, const vector<T>& a) {

for (int i = 0; i < a.size; i++)

out << a.data[i] << " ";

return out;

};

template <class T>

istream& operator>>(istream& in, vector<T>& a) {

for (int i = 0; i < a.size; i++)

in >> a.data[i];

return in;

};

**Time.h**

#pragma once

#include<iostream>

using namespace std;

class Time {

int min, sec;

public:

Time(void);

Time(int, int);

virtual ~Time() {}

Time(const Time&);

Time& operator=(const Time&);

Time operator+(Time);

friend Time operator+(Time, int);

friend ostream& operator<<(ostream& out, const Time& a);

friend istream& operator>>(istream& in, Time& a);

};

Time operator+(Time a, int n) {

int t = a.min \* 60 + a.sec + n;

a.min = t / 60;

a.sec = t % 60;

return a;

}

Time Time::operator+(Time a) {

int t = min \* 60 + sec;

int kt = a.min \* 60 + a.sec;

t += kt;

Time tmp(t / 60, t % 60);

return tmp;

};

ostream& operator<<(ostream& out, const Time& a) {

if (a.sec < 10 && a.sec >= 0) {

if (a.min < 10 && a.min >= 0)

out << '0' << a.min << ":0" << a.sec;

else out << a.min << ":0" << a.sec;

}

else if (a.min < 10 && a.min >= 0) {

out << '0' << a.min << ':' << a.sec;

}

if (a.min >= 10 && a.min < 60 && a.sec >= 10 && a.sec < 60)

out << a.min << ':' << a.sec;

if (a.min > 59 || a.sec > 59)

out << "Ошибка!";

return out;

};

istream& operator>>(istream& in, Time& a) {

cout << "Введите минуты: "; in >> a.min;

cout << "Введите секунды: "; in >> a.sec;

return in;

};

Time::Time() {

min = sec = 0;

};

Time::Time(int min, int sec) {

this->min = min;

this->sec = sec;

};

Time::Time(const Time& a) {

sec = a.sec;

min = a.min;

};

Time& Time:: operator=(const Time& a) {

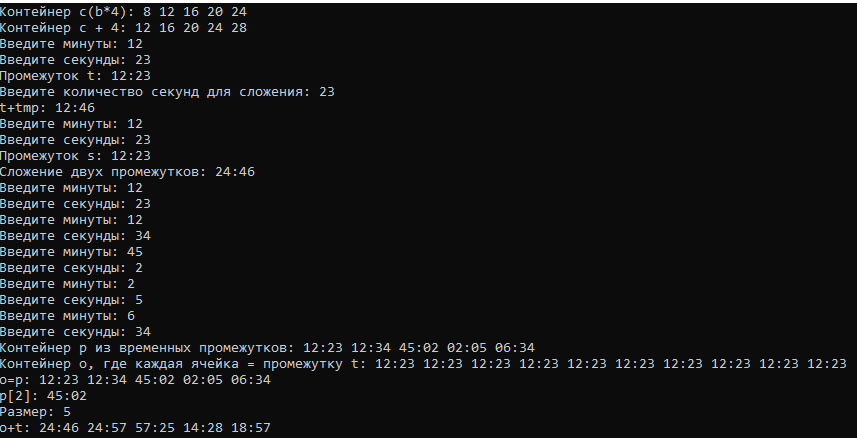
min = a.min;

sec = a.sec;

return \*this;

};

**5 Результаты работы программы**

****

**6 Ответы на вопросы**

1. В чем смысл использования шаблонов?

Шаблоны - это функции языка программирования, которые позволяют создать универсальный код, который может быть использован для разных типов данных.

Шаблоны позволяют создавать семейства функций и классов.

Семейство позволяет работать с разными типами данных по единой логике (алгоритму) - обобщённое программирование. В качестве параметра поступает тип данных.

Шаблоны вводятся для того, чтобы автоматизировать создание функций, обрабатывающих разнотипные данные.

1. Каковы синтаксис/семантика шаблонов функций?

template <параметры\_шаблона>

заголовок\_функции

{тело функции}

В коде:

template <typename T>

T find\_sum(T a, T b)

{

return a + b;

}

1. Каковы синтаксис/семантика шаблонов классов?

template <параметры шаблона>

class имя\_класса

{…};

В коде:

template <class T>

class Sum {

T a, b;

public:

Sum() { a = b = 0; }

Sum(T A, T B) { a = A; b = B; }

~Sum() {}

T find\_sum(T a, T b) { return a + b; }

};

1. Что такое параметры шаблона функции?

Можно считать, что параметры шаблона являются его формальными параметрами, а типы тех параметров, которые используются в конкретных обращениях к функции, служат фактическими параметрами шаблона. Именно по ним выполняется параметрическая настройка и с учетом этих типов генерируется конкретный текст определения функции.

template <typename T>

T find\_sum(T a, T b)

{

return a + b;

}

Если параметры будут типа double, то компилятор воспримет тип T как тип double, если типа int, то как int. В данном случае оба параметра должны быть одного типа.

1. Перечислите основные свойства параметров шаблона функции.

- Имена параметров шаблона должны быть уникальными во всем определении шаблона.

- Список параметров шаблона не может быть пустым, для того, чтобы компилятор мог инстанцировать шаблон.

- В списке параметров шаблона может быть несколько параметров, и каждому из них должно предшествовать ключевое слово class (или typename).

template<class type1, class type2>

- Недопустимо использовать в заголовке шаблона параметры с одинаковыми именами.

- Имя параметра шаблона имеет все права имени типа в определенной шаблоном функции.

1. Определение формальных параметров функции
2. Определение типа возвращаемого функцией значения
3. Определение любых объектов, локализованных в теле функции

- Все параметры шаблона функций должны быть обязательно использованы в спецификациях параметров определения функции. Таким образом, будет ошибочным такой шаблон:

template<class A, class B, class C>

B func(A n, C m) { B value; };

- Определённая с помощью шаблона функция может иметь любое количество непараметризованных формальных параметров. Может быть непараметризовано и возвращаемое функцией значение. Например, в следующей программе шаблон определяет семейство функций, каждая из которых подсчитывает количество нулевых элементов одномерного массива параметризованного типа:

template<class D>

long count0(int, D\*); //Прототип шаблона

int main()

{

int A[] = { 1, 0, 6, 0, 4, 10 };

int n = sizeof(A) / sizeof A[0];

cout << "\ncount0(n,A) = " << count0(n, A);

float X[] = { 10.0, 0.0, 3.3, 0.0, 2.1 };

n = sizeof(X) / sizeof X[0];

cout << "\ncount0(n,X) = " << count0(n, X);

return 0;

}

- В списке параметров прототипа шаблона имена параметров не обязаны совпадать с именами тех же параметров в определении шаблона.

При конкретизации шаблонного определения функции необходимо, чтобы при вызове функции типы фактических параметров, соответствующие одинаково параметризованным формальным параметрам, были одинаковыми. - Для определённого ниже шаблона функций с данным прототипом недопустимо использовать такое обращение к функции:

template<class E> void swap(E, E);

int n = 4;

double d = 4.3;

swap(n, d); // Ошибка в типах параметров

swap(double(n), d); // Правильные типы параметров

- Для правильного обращения к такой функции требуется явное приведение -типа одного из параметров.

- При использовании шаблонов функций возможна перегрузка как шаблонов, так и функций. Могут быть шаблоны с одинаковыми именами, но разными параметрами. Или с помощью шаблона может создаваться функция с таким же именем, что и явно определённая функция. В обоих случаях "распознавание" конкретного вызова выполняется по сигнатуре, т.е. по типам, порядку и количеству фактических параметров.

1. Как записывать параметр шаблона?

- Каждому из разных типов параметров должно предшествовать ключевое слово class или typename. Пример: template<class type1, class type2>

- Все параметры шаблона функций должны быть обязательно использованы в спецификациях параметров определения функции.

- Недопустимо использовать в заголовке шаблона параметры с одинаковыми именами, то есть ошибочен такой заголовок:

template<class T, class T, class T>

1. Можно ли перегружать параметризованные функции?

Да, можно. Главное, чтобы функции либо возвращали разное значение, либо имели разное количество параметров, либо имели разные типы параметров.

template <typename T>

void print(T value) { cout << "Шаблон" << value; } // Шаблон

template <>

void print(int value) { cout << "Специализация для int" << value; } // ничего не возвращает

int print(double value) { return value; } // возвращает double

1. Перечислите основные свойства параметризованных классов.

- Компонентные функции параметризованного класса автоматически параметризованы.

- Дружественные функции, которые описываются в параметризованном классе, не являются автоматически параметризованными, то есть по умолчанию такие функции являются дружественными для всех классов, которые организуются по шаблону.

- Если дружественная функция содержит в своём описании параметр параметризованного класса, то каждый класс, организованный по шаблону, имеет собственную параметризованную функцию.

- В параметризованном классе нельзя определить дружественные параметризованные классы.

- Шаблоны могут быть базовыми классами. Производными классами от такого класса могут быть обычными или шаблонными. Шаблоны могут наследоваться от обычных или от шаблонных классов.

- Шаблоны функций-членов нельзя описывать как виртуальные.

- Локальные (вложенные) классы не могут содержать шаблоны в качестве своих элементов.

- Определённые пользователем имена в описании шаблона как идентификаторы переменных. Чтобы имя рассматривалось как идентификатор типа, оно должно быть определено через ключевое слово typename.

- Если компонентная функция описывается вне шаблона класса:

template <typename T>

class MyClass

{

T value;

void Datasize(T);

};

template <typename T>

void MyClass <T> ::Datasize(T value)

{

cout << sizeof(value);

}

1. Все ли компонентные функции параметризованного класса являются параметризованными?

Компонентные функции параметризованного класса автоматически являются параметризованными. Их не обязательно объявлять как параметризованные с помощью template.

1. Являются ли дружественные функции, описанные в параметризованном классе, параметризованными?

В рамках параметризованного класса нельзя определить friend-шаблоны (дружественные параметризованные классы).

1. Могут ли шаблоны классов содержать виртуальные компонентные функции?

Шаблоны функций, которые являются членами классов, нельзя описывать как virtual.

1. Как определяются компонентные функции параметризованных классов вне определения шаблона класса?

- Определение должно начинаться с ключевого слова template, за которым следует такой же список\_параметров\_типов в угловых скобках, какой указан в определении шаблона класса.

- За именем\_класса, предшествующим операции области видимости (::), должен следовать список\_имен\_параметров шаблона.

template<список\_типов>тип\_возвр\_значения имя\_класса<список\_имен\_ параметров> : : имя\_функции(список\_параметров){ . . . }

template <typename T>

class MyClass

{

T value;

void Datasize(T);

};

template <typename T>

void MyClass <T> ::Datasize(T value)

{

cout << sizeof(value);

}

1. Что такое инстанцирование шаблона?

Процесс генерации компилятором определения конкретного класса по шаблону класса и аргументам шаблона называется инстанцированием шаблона.

1. На каком этапе происходит генерирование определения класса по шаблону?

При включении шаблона класса в программу никакие классы на самом деле не генерируются до тех пор, пока не будет создан экземпляр шаблонного класса, в котором вместо параметра шаблона указывается конкретный тип. Экземпляр создается либо объявлением объекта, либо объявлением указателя на инстанцированный шаблонный тип с присваиванием ему адреса с помощью операции new.

Point <int> a(13, 15);

Point <float>\* pa = new Point<float>(10.1, 0.55);