**Московский государственный технический**

**университет им. Н.Э. Баумана**

Факультет «Информатика и системы управления»

Кафедра ИУ5 «Системы обработки информации и управления»

Курс «Программирование на основе классов и шаблонов»

Отчет по лабораторной работе №4

«Разработка класса множество»

|  |  |  |
| --- | --- | --- |
| Выполнил: |  | Проверил: |
| студент группы ИУ5-22 |  | преподаватель каф. ИУ5 |
| Барышников Михаил |  | Козлов А.Д. |
| Подпись и дата: |  | Подпись и дата: |

Москва, 2020 г.

**Задача**

Разработать класс множество (Bunch) для выполнения операций над множествами (+, -, \*, ==, !=) и функцию **main( )** для его тестирования.

Класс вектор должен быть динамическим массивом, размер которого может автоматически изменяться в процессе выполнения программы. Добавление элементов производится в конец.

В качестве основы для класса множество было решено взять класс вектор (my\_vector). Все функции, которого реализованы без использования сторонних библиотек.

Методы, такие как ***add*** и ***operator[]*** производного класса Bunch перегружают одноименные методы базового класса my\_vector, а другая часть элементов класса my\_vector наследуются классом Bunch.

Методы и функции, которые необходимо реализовать:

Перегруженная операция потокового вывода;

Операторы **+** (объединение), **-** (разность),\*(пересечение), **==** (сравнение: истина, если элементы двух множеств совпадают) и также !=. Функции erase() и find(), которые можно использовать в комбинации, чтобы удалить найденный элемент erase(find()).

Примеры операций над множествами (приведены для целых чисел):

{1, 4, 5, 6} + {1, 2, 3, 4} => {1, 2, 3, 4, 5, 6}

{1, 4, 5, 6} \* {1, 2, 3, 4} => {1, 4}

{1, 4, 5, 6} - {1, 2, 3, 4} => {5, 6}

**Текст программы**

* class\_bunch.h

#pragma once

#include "my\_vector.h"

#include <string>

template <class T>

class Bunch

{

my\_vector<T> elements;

public:

Bunch<T>() {}

~Bunch<T>() {}

Bunch<T> operator+(const Bunch<T>&); //объединение множеств

Bunch<T> operator-(const Bunch<T>&); //вычитает одно множество из другого

Bunch<T> operator\*(const Bunch<T>&); //пересечение множеств

T& operator[](int) const;

bool operator==(const Bunch<T>&); //сравнивает 2 множества, если они равны возвращает true

bool operator!=(const Bunch<T>&); //сравнивает 2 множества, если они не равны возвращает false

void add(T); //добавляет элемент в множество

int find(T) const; //возвращает индекс элемента

void erase(int); //удаляет элемент под индексом

friend std::ostream& operator<<(std::ostream& stream, const Bunch<T>& b)

{

for (int i = 0; i != b.elements.size(); ++i)

stream << b[i] << ';';

return stream;

}

friend std::istream& operator>>(std::istream& stream, Bunch<T>& b)

{

std::string str;

stream >> str;

b.elements.erase();

while (str.find(';') != std::string::npos)

{

b.add(std::atoi((str.substr(0, str.find(';')).c\_str())));

str.erase(0, str.find(';') + 1);

}

return stream;

}

};

template<class T>

inline void Bunch<T>::add(T b)

{

if (find(b) == -1)

elements.push\_back(b);

}

template<class T>

inline int Bunch<T>::find(T b) const

{

for (int i = 0; i != elements.size(); ++i)

if (b == elements[i])

return i;

return -1;

}

template<class T>

inline void Bunch<T>::erase(int b)

{

elements.erase(b);

}

template<class T>

inline Bunch<T> Bunch<T>::operator+(const Bunch<T> &b)

{

Bunch new\_bunch = \*this;

for (int i = 0; i != b.elements.size(); ++i)

new\_bunch.add(b[i]);

return new\_bunch;

}

template<class T>

inline Bunch<T> Bunch<T>::operator-(const Bunch<T> &b)

{

Bunch new\_bunch;

for (int i = 0; i != elements.size(); ++i)

if (b.find((\*this)[i]) == -1)

new\_bunch.add((\*this)[i]);

return new\_bunch;

}

template<class T>

inline Bunch<T> Bunch<T>::operator\*(const Bunch<T>& b)

{

Bunch new\_bunch;

if (elements.size() > b.elements.size())

{

for (int i = 0; i != elements.size(); ++i)

if (b.find((\*this)[i]) != -1)

new\_bunch.add((\*this)[i]);

}

else

{

for (int i = 0; i != b.elements.size(); ++i)

if (find(b[i]) != -1)

new\_bunch.add(b[i]);

}

return new\_bunch;

}

template<class T>

inline T & Bunch<T>::operator[](int b) const

{

return elements[b];

}

template<class T>

inline bool Bunch<T>::operator==(const Bunch<T> &b)

{

if (elements.size() != b.elements.size())

return false;

for (int i = 0; i != elements.size(); ++i)

if (find(b[i]) == -1)

return false;

return true;

}

template<class T>

inline bool Bunch<T>::operator!=(const Bunch<T> &b)

{

return !(\*this == b);

}

* my\_vector.h

#pragma once

#include <cstdlib>

const int size\_l = 50;

using namespace std;

template<class T>

class my\_vector

{

int length;

int length\_of\_used;

T \*ptr\_;

int add\_memory(T);

public:

my\_vector<T>();

my\_vector<T>(const my\_vector<T> &);

~my\_vector<T>();

void push\_back(T);

void pop\_back();

void erase();

my\_vector<T> operator=(const my\_vector<T>&);

void erase(int);

void erase(int, int);

//void set\_size(int);

int size() const;

T& operator[](int) const;

};

template<class T>

inline int my\_vector<T>::add\_memory(T b)

{

length += size\_l / 2;

T \*p = new T[length]; //

//T \*p = (T\*)malloc(sizeof(T) \* length);

if (p == NULL)

{

length -= size\_l / 2;

throw std::exception("Memory is full!");

}

for (int i = 0; i != length\_of\_used; ++i)

\*(p + i) = \*(ptr\_ + i);

delete[] ptr\_; //

//free(ptr\_);

ptr\_ = p;

push\_back(b);

return 0;

}

template<class T>

inline void my\_vector<T>::push\_back(T b)

{

length > length\_of\_used ? \*(ptr\_ + length\_of\_used++) = b : add\_memory(b);

}

template<class T>

inline void my\_vector<T>::pop\_back()

{

if (length\_of\_used)

\*(ptr\_ + --length\_of\_used) = 0;

}

template<class T>

inline void my\_vector<T>::erase()

{

while (length\_of\_used)

\*(ptr\_ + --length\_of\_used) = 0;

}

template<class T>

inline my\_vector<T> my\_vector<T>::operator=(const my\_vector<T>& b)

{

length = b.length;

length\_of\_used = b.length\_of\_used;

ptr\_ = new T[length]; //

//ptr\_ = (T\*)malloc(sizeof(T) \* length);

//if (ptr\_ == NULL) throw std::exception("Memory is full!");

for (int i = 0; i != length\_of\_used; ++i)

\*(ptr\_ + i) = \*(b.ptr\_ + i);

return \*this;

}

template<class T>

inline void my\_vector<T>::erase(int b)

{

if (b < 0 || b >= length\_of\_used)

throw std::exception("Error! This area of memory is not available!");

for (int i = b; i != length\_of\_used - 1; ++i)

\*(ptr\_ + i) = \*(ptr\_ + i + 1);

pop\_back();

}

template<class T>

inline void my\_vector<T>::erase(int b, int c)

{

if (b < 0 || b >= length\_of\_used || c >= length\_of\_used)

throw std::exception("Error! This area of memory is not available!");

if (b > c)

throw std::exception("Error! Range is not exist!");

int delta = c - b + 1;

for (int i = b; i != length\_of\_used - delta; ++i)

\*(ptr\_ + i) = \*(ptr\_ + i + delta);

delta = length\_of\_used - delta;

for (int i = length\_of\_used; i != delta; --i)

pop\_back();

}

template<class T>

inline int my\_vector<T>::size() const

{

return length\_of\_used;

}

template<class T>

inline T & my\_vector<T>::operator[](int b) const

{

return length\_of\_used > b ? \*(ptr\_ + b) : throw std::exception("Error! This area of memory is not available!");

}

template<class T>

inline my\_vector<T>::my\_vector()

{

length = size\_l;

length\_of\_used = 0;

ptr\_ = new T[length]; //

//ptr\_ = (T\*)malloc(sizeof(T) \* length);

if (ptr\_ == NULL)

{

length = 0;

throw std::exception("Memory is full!");

}

}

template<class T>

inline my\_vector<T>::my\_vector(const my\_vector<T>& b)

{

length = b.length;

length\_of\_used = b.length\_of\_used;

ptr\_ = new T[length]; //

//ptr\_ = (T\*)malloc(sizeof(T) \* length);

//if (ptr\_ == NULL) throw std::exception("Memory is full!");

for (int i = 0; i != length\_of\_used; ++i)

\*(ptr\_ + i) = \*(b.ptr\_ + i);

}

template<class T>

inline my\_vector<T>::~my\_vector()

{

delete[] ptr\_; //

//free(ptr\_);

}

* main.cpp

#include <iostream>

#include "class\_bunch.h"

int main()

{

setlocale(0, "ru");

Bunch<int> b, a;

char sign;

while (true)

{

try

{

std::cout << "Enter first bunch: \n";

std::cin >> a;

std::cout << "Enter sign: \n";

std::cin >> sign;

std::cout << "Enter second bunch: \n";

std::cin >> b;

switch (sign)

{

case '+':

std::cout << a + b;

break;

case '-':

std::cout << a - b;

break;

case '=':

std::cout << (a == b);

break;

case '\*':

std::cout << a \* b;

break;

case (int)'1':

a.erase(a.find(55));

a.erase(2);

std::cout << a;

break;

}

std::cout << "\n";

}

catch (std::exception &ex)

{

std::cout << "\n" << ex.what() << "\n";

}

}

system("pause");

return 0;

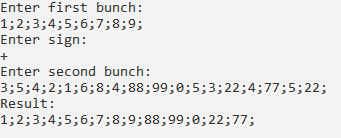
}

**Анализ результатов**

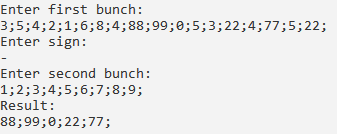
В данной программе в качестве «точки входа» выступает основная функция main(). В ней написан тестируемый код. Ниже представлены результаты выполнения. Для реализации методов данного класса был «с нуля» написан вспомогательный класс вектор, который можно использовать и отдельно от класса «Bunch». Большинство методов и функций класса множество реализованы простым вызовом методов класса my\_vector.

Далее приведены результаты тестов программы:

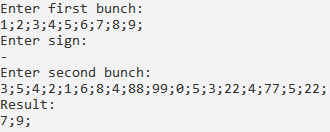
Тест №1:



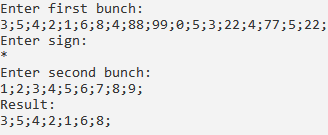
Тест №2:



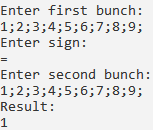
Тест №3:



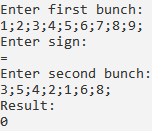
Тест №4:



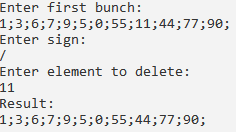
Тест №5:



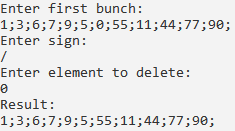
Тест №6:



Тест №7:



Тест №8:



Тесты были проверены вручную. Программа выдала верные результаты. Также были проведены многочисленные тесты всех операторов, многие из которых не были прикреплены. Но основываясь на правильности приведенных тестов, можно сделать вывод, что остальные то же отработают правильно.

Примечание:

Знак «/» используется про тестирования операции удаления элемента.

**Вывод**

В результате написания программы был получен опыт работы с шаблонными классами (ООП). Были реализованы перегрузки основных операторов простых арифметических действий для множеств. В качестве доработки можно было бы переопределить остальные вспомогательные операторы присваивания, а также создать сортировку элементов множества. Еще из минусов – это возможность переполнения int значение элемента, но и это решаемо, т.к. можно подключить библиотеку для работы с длинной арифметикой на c++. В итоге был написан класс множества, основанный на классе my\_vector, который удобно использовать работы с множествами.