Московский авиационный институт
(Национальный исследовательский университет)

Институт: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»
Дисциплина: «Объектно-ориентированное программирование»

Лабораторная работа № 6

Тема: Основы работы с коллекциями: аллокаторы

Студент: Суханов Егор Алексеевич
Группа: 80-206
Преподаватель: Чернышов Л.Н.
Дата:
Оценка:

Москва, 2020

1. Постановка задачи

Разработать шаблоны классов согласно варианту задания. Параметром шаблона должен являться скалярный тип данных задающий тип данных для оси координат. Классы должны иметь публичные поля. Фигуры являются фигурами вращения, т.е. равносторонние (кроме трапеции и прямоугольника). Для хранения координат фигур необходимо использовать шаблон std::pair. Так как мой вариант задания 25, мне нужно сделать класс треугольника, который:

1. Имеет параметр шаблона: скалярный тип осей системы координат;
2. Имеет перегруженные операторы ввода-вывода;
3. Имеет метод вычисления площади.

Класс треугольника возьмем из прошлой ЛР.

Создать шаблон динамической коллекции, согласно варианту задания. Так как мой вариант задания 25, мне нужно создать динамический массив:

1. Динамический массив будет основан на коллекции из прошлой ЛР;
2. Коллекция должна использовать аллокатор для выделения и освобождения памяти для своих элементов.

Нужно создать аллокатор, согласно варианту задания:

1. Выделяет фиксированное кол-во блоков памяти (кол-во задается параметром шаблона);
2. Должен хранить указатель на используемый блок памяти;
3. Должен хранить динамическую коллекцию (динамический массив) свободных элементов;
4. Аллокатор должен быть совместим с стандартными коллекциями std::map и std::list.

Реализовать программу, которая:

- позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию использующую аллокатор;
- позволяет удалять элемент из коллекции по номеру элемента;
- выводит на экран введенные фигуры с помощью std::for_each.

Опишем порядок выполнения работы:

1. Изучение теоретической информации:
   - Аллокаторы в C++;
   - Стандартные коллекции.
2. Реализация аллокатора;
3. Добавление поддержки аллокаторов для моего динамического массива из прошлой ЛР;
4. Интерактивный ввод-вывод;

2. Описание программы

Ссылка на GITHUB: https://github.com/Reterer/oop_exercise_06

Программа состоит из 3-ех компонентов:

◦ Интерактивный ввод-вывод. Находится в файле main.cpp. В цикле происходит считывание команды и последующее ее выполнение. Для каждой команды реализована своя функция. Алгоритм работы данного компонента не отличается от работы аналогичных компонентов в прошлых лабораторных работах;

◦ Класс Triangle. Данный шаблонный класс имплементирует работу с треугольниками. Он позволяет узнать координаты вершин, площадь, а также обеспечивает ввод-вывод. Треугольник строится по двум точкам: вершине и центру описанной окружности. Этот класс использует вспомогательные шаблонные функции для работы с точками. Все функции и классы для работы с треугольниками находятся в файле figure.hpp;

◦ Динамический массив. Находится в файле vector.hpp. За основу был взят класс Vector из прошлой ЛР. Туда была добавлена поддержка аллокаторов. Кроме того
Координаты вводятся как два последовательных числа. Для ввода треугольника нужно ввести сначала центр описанной окружности, затем одну из его вершин. Для получения справки по командам нужно ввести help;

◦ Аллокатор. Мой аллокатор имеет выделенный массив блоков. Изначально имеется один свободный “кадр”. При аллоцировании памяти находится первый подходящий кадр (размер которого больше или равен запрашиваемому), если размер больше нужного, то он разрезается на два кадра. При освобождении памяти, освободившийся кадр , если есть возможность, соединяется со соседями. Достоинство такого аллокатора состоит в возможности выделения массивов. Однако каждый кадр требует заголовок, который занимает 8 байт.

3. Набор и результаты выполнения тестов

Все тесты находятся в папке tests.

test_01.txt -- проверка работы команд: вставки, удаления, вывода, вычисления кол-ва подходящих фигур.

```
insert 0 0 0 1 1
insert 1 0 0 2 2
insert 0 0 0 3 3
insert 1 0 0 4 4
print
erase 3
erase 1
erase 0
```

```
erase 0
print
```
Результаты выполнения:

```
Координаты треугольника: (3 , 3) (-4.09808 , 1.09808) (1.09808 , -4.09808)
площадь: 23.3827
Координаты треугольника: (3 , 3) (-4.09808 , 1.09808) (1.09808 , -4.09808)
площадь: 23.3827
Координаты треугольника: (1 , 1) (-1.36603 , 0.366025) (0.366025 , -1.36603)
площадь: 2.59808
Координаты треугольника: (2 , 2) (-2.73205 , 0.732051) (0.732051 , -2.73205)
площадь: 10.3923
```

test_02.txt -- проверка обработки ошибок.

```
sjehfkj kjefh w
insert 0 0 0 0 0
insert 0 kehfkewjfh 0 0 0 0
insert 0 kehfkewjfh 0 0 1 1
insert 1 0 0 1 1
erase 0
```
Результаты выполнения:

```
Введена неизвестная команда. Чтобы вывести справку, введите
"help"
Введены некорректные координаты.
Введены некорректные координаты.
Введены некорректные координаты.
Введено некорректное число.
Введено некорректное число.
```

4. Листинг программы

файл **main.cpp**:

```
/*
  Лабораторная работа: 6
  Вариант: 25
  Группа: М8О-206Б-19
  Автор: Суханов Егор Алексеевич

  Разработать шаблоны классов согласно варианту задания.
```

```
  Параметром шаблона должен являться скалярный тип данных задающий тип данных
для оси координат.
  Классы должны иметь публичные поля. Фигуры являются фигурами вращения, т.е.
равносторонними (кроме трапеции и прямоугольника).
  Для хранения координат фигур необходимо использовать шаблон  std::pair.

  Реализовать аллокатор, с помощью которго будет работать структура данных.
Свободные блоки
  в аллокатороре следует хранить с помощью предложенной структуры данных.

  Фигура:
      Треугольник
  Структура данных:
      Динамический массив
  Аллокатор:
    Динамический массив
*/
#include <iostream>
#include <algorithm>
#include "vector.hpp"
#include "figure.hpp"


void clear()
{
      std::cin.clear();
      std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}

void help()
{
      std::cout <<
            "команды:\n"
            "    help        -- выводит этот текст\n"
            "    exit        -- завершает работу программы\n"
            "    print       -- выводит вектор фигур (выполняет их
методы)\n"
            "    erase <id> -- удаляет элемент с индексом id\n"
            "    insert <id> <center> <vertex> -- вставляет треугольник на
позицию с номером id\n"
            << std::endl;
}

template<class T, class Alloc>
void print(Vector<T, Alloc>& figures) {
      std::for_each(figures.begin(), figures.end(), [](T& val) {
            std::cout << val;
      });
}

template<class T, class Alloc>
```

```
void erase(Vector<T, Alloc>& vec) {
	int del_element_idx;
	if (!(std::cin >> del_element_idx) || del_element_idx < 0 ||
del_element_idx >= vec.size()) {
		std::cout << "Введено некоректное число.\n";
		clear();
		return;
	}

	vec.erase(vec.iterator_by_index(del_element_idx));
}

template<class T, class Alloc>
void insert(Vector<T, Alloc>& vec) {
	int new_element_index;
	if (!(std::cin >> new_element_index) || new_element_index < 0 ||
new_element_index > vec.size()) {
		std::cout << "Введено некоректное число.\n";
		clear();
		return;
	}

	T new_el;
	if (!(std::cin >> new_el))
	{
		std::cout << "Введена некоректные координаты.\n";
		clear();
		return;
	}

	try {
		vec.insert(vec.iterator_by_index(new_element_index), new_el);
	}
	catch (std::bad_alloc&) {
		std::cout << "К сожалению аллокатор не может выделить больше
памяти\n";
	}
}

#include <vector>
#include "allocator.hpp"
int main() {
	setlocale(LC_ALL, "russian");
	Vector<Triangle<int>, Allocator<int, 10>> figures;
	std::string cmd;
	std::cout << '>';
	while (std::cin >> cmd)
	{
		if (cmd == "help")
			help();
		else if (cmd == "exit")
```

```
                break;
            else if (cmd == "print")
                print(figures);
            else if (cmd == "erase")
                erase(figures);
            else if (cmd == "insert")
                insert(figures);
            else
            {
                std::cout << "Введена неизвестная команда. Чтобы вывести
справку, введите \"help\"" << std::endl;
                clear();
            }

            std::cout << '>';
        }

        return 0;
}
```

файл **figure.hpp**:

```
#pragma once
#include <stdexcept>
#include <utility>
#include <iostream>

const double PI = 3.141592653589793;
// Векторное сложение
template <class T>
std::pair<T, T> operator + (const std::pair<T, T> a, const std::pair<T, T> b)
{
        return { a.first + b.first, a.second + b.second };
}
// Векторное вычитание
template <class T>
std::pair<T, T> operator - (const std::pair<T, T> a, const std::pair<T, T> b)
{
        return { a.first - b.first, a.second - b.second };
}
// Скалярное умножение векторов
template <class T>
T operator * (const std::pair<T, T> a, const std::pair<T, T> b) {
        return a.first * b.first + a.second * b.second;
}
// Умножение на скаляр
template <class T>
std::pair<T, T> operator * (const T a, const std::pair<T, T> b) {
        return { a * b.first, a * b.second };
}
```

```cpp
// Вывод координат точки
template <class T>
std::ostream& operator<< (std::ostream& out, const std::pair<T, T>& p) {
	out << '(' << p.first << " , " << p.second << ')';
	return out;
};

// Ввод координат
template <class T>
std::istream& operator>> (std::istream& in, std::pair<T, T>& p) {
	in >> p.first >> p.second;
	return in;
}

// Поворот вектора pair на угол angle
template <class T>
std::pair<T, T> rotate(std::pair<T, T> pair, const double angle) {
	std::pair<T, T> rotated;
	rotated.first = (T)(pair.first * std::cos(angle) - pair.second *
std::sin(angle));
	rotated.second = (T)(pair.first * std::sin(angle) + pair.second *
std::cos(angle));
	return rotated;
}


template<typename T>
class Triangle {
public:
	using Vertex = std::pair<T, T>;

	Triangle();
	Triangle(Vertex center, Vertex vertex);

	// Возвращает площадь треугольника
	double Square() const;

	friend std::ostream& operator<< (std::ostream& out, const Triangle<T>&
t) {
		const double ANGLE = 2 * PI / Triangle<T>::vertex_count; // угол
между двумя соседними вершинами и центром

		out << "Координаты треугольника: ";
		out << t.vertex;		// Выводим первую вершину
		auto vec = t.vertex - t.center;
		for (int i = 2; i <= Triangle<T>::vertex_count; ++i)
		{
			vec = rotate(vec, ANGLE); // Получаем координаты следующий
вершины
			out << ' ' << t.center + vec;
		}
```

```
		out << "\t площадь: " << t.Square() << '\n';
		return out;
	}

	friend std::istream& operator>> (std::istream& in, Triangle<T>& t) {
		in >> t.center >> t.vertex;
		if (t.center == t.vertex)
			in.setstate(std::ios_base::failbit);
		return in;
	}

private:
	static const int vertex_count = 3; // Кол-во вершин треугольника

	Vertex center; // Центр описанной окружности
	Vertex vertex; // Одна из вершин треугольника
};

template<typename T>
Triangle<T>::Triangle()
{}

template<typename T>
Triangle<T>::Triangle(Vertex center, Vertex vertex)
	: center{ center }, vertex{ vertex }
{
	if (center == vertex)
		throw std::invalid_argument("center cannot be eq to the
vertex");
}

template<typename T>
double Triangle<T>::Square() const {
	auto vecRadius = vertex - center; // Вектор-радиус описанной
окружности
	double sqRadius = vecRadius * vecRadius;	// Квадрат радуиса
описанной окружности
	return vertex_count / 2. * sqRadius * std::sin(2 * PI / vertex_count);
}
```

Файл **vector.hpp**:

```
#pragma once
#include <stdexcept>
#include <memory>

template<typename T, typename Allocator = std::allocator<T>>
class Vector {
public:
	using allocator_type = typename Allocator::template rebind<T>::other;
```

```
	explicit Vector()
		: buf_{ nullptr, { alloc_, 0 } }, size_{ 0 }, cap_{ 0 },
alloc_{}
	{}
	explicit Vector(const int size)
		: Vector(size, size)
	{}
	explicit Vector(const int size, const int cap) : Vector() {
		this->size_ = size;
		this->cap_ = cap;

		if (size_ > cap_)
			throw std::invalid_argument("size_ of vector can't be more
then capacity.");
		auto new_buf = std::unique_ptr<T[], ptr_deleter>{
alloc_.allocate(cap_), { alloc_, (std::size_t)cap_ } };
		this->buf_.swap(new_buf);
	}

	Vector(const std::initializer_list<T>& list)
		: Vector(list.size())
	{
		int count = 0;
		for (const T& el : list) {
			this->buf_.get()[count] = el;
			++count;
		}
	}

	Vector(Vector<T, Allocator>& vec)
		: Vector(vec.size_, vec.cap_)
	{
		for (int i = 0; i < vec.size_; i++) {
			this->buf_[i] = vec[i];
		}
	}

	int size() const {
		return this->size_;
	}

	T& operator[] (const int i) {
		if (i < 0 || i >= this->size_)
			throw std::out_of_range("Out of range");
		return this->buf_.get()[i];
	}

	// Итераторы
	class iterator
	{
		friend class Vector;
```

```
public:
	using iterator_category = std::forward_iterator_tag;
	using value_type = T;
	using difference_type = int;
	using pointer = T*;
	using reference = T&;

	iterator(Vector* vec, T* ptr) : vec{vec}, ptr{ptr}
	{}

	bool operator== (const iterator& it) const {
		return it.vec == vec && it.ptr == ptr;
	}
	bool operator!= (const iterator& it) const {
		return it.vec != vec || it.ptr != ptr;
	}
	T& operator* () {
		is_valid();
		return *ptr;
	}
	T* operator-> () {
		is_valid();
		return ptr;
	}

	iterator& operator++() {
		is_valid();
		ptr++;
		return *this;
	}
	iterator operator++(int) const {
		is_valid();
		return { vec, ptr + 1 };
	}

private:
	void is_valid() const {
		if (vec == nullptr || ptr == nullptr)
			throw std::runtime_error("Nullptr iterator");
		if (vec->buf_.get() + vec->size_ <= ptr)
			throw std::out_of_range("Iterator gt or eq end");
	}
	Vector<T, Allocator>* vec;
	T* ptr;
};

iterator begin() {
	return { this, buf_.get() };
}
iterator end() {
	return {this, buf_.get() + size_};
```

```
}
// Возвращает итератор, который указывает на элемент под номером index
iterator iterator_by_index(int index) {
	if (index < 0 || index > size_)
		throw std::out_of_range("bad index");

	return {this, buf_.get() + index};
}

void push_back(const T& val) {
	if (size_ >= cap_)
		grow();
	buf_.get()[size_] = val;
	++size_;
}
void push_back(T&& val) {
	if (size_ >= cap_)
		grow();
	buf_.get()[size_] = val;
	++size_;
}
iterator insert(const iterator& position, const T& value) {
	if (position.vec != this || buf_.get() + size_ < position.ptr)
		throw std::invalid_argument("Invalid iterator");

	if (position == end()) {
		push_back(value);
		return end();
	}

	int new_element_idx = (int)(position.ptr - buf_.get());
	push_back(buf_[size_ - 1]);
	T temp = buf_.get()[new_element_idx];
	buf_.get()[new_element_idx] = value;
	for(int i = new_element_idx + 1; i < size_; i++) {
		std::swap(temp, buf_.get()[i]);
	}

	return iterator_by_index(new_element_idx);
}
iterator erase(const iterator& position) {
	if (position.vec != this || buf_.get() + size_ <= position.ptr)
		throw std::invalid_argument("Invalid iterator");

	int del_element_idx = (int)(position.ptr - buf_.get());
	T temp = buf_.get()[size_ - 1];
	for (int i = size_ - 2; i >= del_element_idx; i--) {
		std::swap(temp, buf_.get()[i]);
	}

	--size_;
```

```
            return iterator_by_index(del_element_idx);
      }

private:
      void grow() {
            int new_cap = (cap_ == 0) ? 1 : cap_ * 2;
            T* new_buf = alloc_.allocate(new_cap);
            for (int i = 0; i < size_; i++) {
                  new_buf[i] = buf_.get()[i];
            }
            auto new_buf_ptr = std::unique_ptr<T[], ptr_deleter>(new_buf, {
alloc_, (std::size_t)new_cap });
            buf_.swap(new_buf_ptr);
            cap_ = new_cap;
      }

      struct ptr_deleter {

            ptr_deleter(allocator_type& alloc, size_t size) : alloc{&alloc},
size{size}
            {}

            void operator() (T* ptr) {
                  for(std::size_t i = 0; i < size; ++i)

std::allocator_traits<allocator_type>::destroy(*alloc, &ptr[i]);
                  alloc->deallocate(ptr, size);
            }

            allocator_type* alloc;
            std::size_t size;
      };

private:
      allocator_type alloc_;
      std::unique_ptr<T[], ptr_deleter> buf_;
      int size_;
      int cap_;
};
```

файл **allocator.hpp**:

```
#pragma once
#include <vector>

template <typename T, std::size_t BLOCK_SIZE>
class Allocator {
public:
      using value_type = T;
      using pointer = T*;
      using const_pointer = const T*;
      using size_type = std::size_t;
```

```
	Allocator() noexcept : freeChunks(0), buffer{ nullptr } {
		static_assert(BLOCK_SIZE > 0, "Размер блока должен быть больше нуля");
		static_assert(sizeof(T) >= sizeof(Header), "Размер типа должен быть не меньше размера заголовка");
	}

	~Allocator() noexcept {
		free(buffer);
	}

	template <typename U>
	Allocator(const Allocator<U, BLOCK_SIZE>& Alloc) noexcept : Allocator() {}

	template <typename U>
	struct rebind
	{
		using other = Allocator<U, BLOCK_SIZE>;
	};

	T* allocate(std::size_t n) {
		if (n == 0)
			return nullptr;

		// Если память для аллокатора не выделена
		if (!buffer) {
			buffer = static_cast<T*>(malloc(sizeof(T) * (BLOCK_SIZE + 1)));

			freeChunks.resize(1);
			reinterpret_cast<Header*>(&buffer[0])->size = BLOCK_SIZE;
			freeChunks[0] = &buffer[0];
		}

		if (freeChunks.empty())
			throw std::bad_alloc();

		// Находим кадр достаточого размера
		Header* ptr = nullptr;
		for (int i = 0; i < freeChunks.size(); ++i) {
			if (reinterpret_cast<Header*>(freeChunks[i])->size >= n) {
				ptr = reinterpret_cast<Header*>(freeChunks[i]);
				freeChunks.erase(freeChunks.begin() + i);
				break;
			}
		}
		// Если не нашли, то кидаем ошибку
		if (!ptr)
			throw std::bad_alloc();
```

```
            // Если кадр больше чем нужно, то делим его
            if (ptr->size > n) {
                  if (ptr->size > n + 1) {
                        size_t size = ptr->size;
                        ptr->size = n;
                        // Создаем новый кадр
                        Header*                    next                    =
reinterpret_cast<Header*>(reinterpret_cast<T*>(ptr) + n + 1);
                        next->size = size - n - 1;
                        freeChunks.push_back(reinterpret_cast<T*>(next));
                  }
            }

            return reinterpret_cast<T*>(ptr)+1;
      }

      void deallocate(T* p, std::size_t n) {
            if (!p || n == 0)
                  return;

            // Получаем хедр кадра
            Header* head = reinterpret_cast<Header*>(p - 1);
            // Нужно проверить, являются ли соседние кадры свободными
            auto it = freeChunks.begin();
            while (it != freeChunks.end()) {
                  Header* neighbour = reinterpret_cast<Header*>(*it);
                  // Если сосед слева
                  if (neighbour + neighbour->size + 1 == head) {
                        neighbour->size += head->size + 1;
                        head = neighbour;
                        it = freeChunks.erase(it);
                  }
                  // Сосед справа
                  else if (head + head->size + 1 == neighbour) {
                        head->size += neighbour->size + 1;
                        it = freeChunks.erase(it);
                  }
                  else
                        ++it;
            }

            // Нужно пометить кадр как свободный
            freeChunks.push_back(reinterpret_cast<T*>(head));
      }


private:
      struct Header {
            size_t size;
      };
```

```
        pointer buffer;
        std::vector<pointer> freeChunks;
};
```

5. Выводы

В данной лабораторной работе я узнал о простых алгоритмах аллокации памяти: линейный аллокатор, стековый аллокатор, блочный аллокатор. Я узнал, что умные указатели могут работать с массивами. Я научился писать совместимый с stl аллокатор. Свои аллокаторы полезно использовать если требуется ускорить выделение памяти или, например, если нужно выделять память рядом с друг другом для большего попадания в кэш. Собственные аллокаторы могут уменьшить фрагментацию кучи.

6. Список литературы

1. Страуструп, Бьёрн. Язык программирования C++. Краткий курс, 2-е изд. : Пер. с англ. - СПб.: ООО "Диалектика", 2019. - 320 с.: ил. - Парал. тит. англ.;

2. Альтернативные аллокаторы памяти[Электронный ресурс]. URL: https://habr.com/ru/post/274827/ (дата обращения 01.12.20);

3. Аллокаторы памяти[Электронный ресурс]. URL: https://habr.com/ru/post/505632/ (дата обращения 01.12.20);

4. STL Allocators[Электронный ресурс]. URL: https://www.codeguru.com/cpp/cpp/cpp_mfc/stl/article.php/c4079/Allocators-STL.htm (дата обращения 01.12.20).