Московский Авиационный Институт
(Национальный Исследовательский Университет)

Кафедра: 806 «Вычислительная математика и программирование»
Факультет: «Прикладная математика и физика»
Дисциплина: «Объектно-ориентированное программирование»

Лабораторная работа №7.
Тема: «Хранилище объектов»

Группа: 8О-408Б
Студент: Забарин Никита Игоревич
Преподаватель: Поповкин Александр Викторович
Вариант: №26

Москва
2017

Цель работы

Целью лабораторной работы является:

- Создание сложных динамических структур данных.
- Закрепление принципа OCP.

Задание

Необходимо реализовать динамическую структуру данных – «Хранилище объектов» и алгоритм работы с ней. «Хранилище объектов» представляет собой контейнер 1-го уровня. Каждым элементом контейнера 1-го уровня, в свою, является динамической структурой данных — контейнером 2-го уровня. Таким образом у нас получается контейнер в контейнере. Элементом второго контейнера является объект-фигура, определенная вариантом задания.

При этом должно выполняться правило, что количество объектов в контейнере второго уровня не больше **5**. Т.е. если нужно хранить больше 5 объектов, то создается еще один контейнер второго уровня. Например, для варианта (1,2) добавление объектов будет выглядеть следующим образом:
1. Вначале массив пустой.
2. Добавляем Объект1: В массиве по индексу 0 создается элемент с типом список, в список добавляется Объект 1.
3. Добавляем Объект2: Объект добавляется в список, находящийся в массиве по индекс 0.
4. Добавляем Объект3: Объект добавляется в список, находящийся в массиве по индекс 0.
5. Добавляем Объект4: Объект добавляется в список, находящийся в массиве по индекс 0.
6. Добавляем Объект5: Объект добавляется в список, находящийся в массиве по индекс 0.
7. Добавляем Объект6: В массиве по индексу 1 создается элемент с типом список, в список добавляется Объект 6.

Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию Имя объекта (в том числе и для деревьев).
При удалении объектов должно выполняться правило, что контейнер второго уровня не должен быть пустым. Т.е. если он становится пустым, то он должен удалится.

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера (1-го и 2-го уровня).
- Удалять фигуры из контейнера по критериям:
    - По типу (например, все квадраты).
    - По площади (например, все объекты с площадью меньше чем заданная).

Выводы

Листинг

queue_item_impl.cpp:

```
template <class T>
QueueItem<T>::QueueItem(const std::shared_ptr<T>& item)
{
	m_item = item;
}

template <class T>
```

```
void QueueItem<T>::setNext(std::shared_ptr<QueueItem<T>> next)
{
	m_next = next;
}

template <class T>
std::shared_ptr<QueueItem<T>> QueueItem<T>::getNext()
{
	return m_next;
}

template <class T>
std::shared_ptr<T> QueueItem<T>::getItem() const
{
	return m_item;
}
```

criteria_area.h:

```
#ifndef CRITERIA_AREA_H
#define CRITERIA_AREA_H

#include "criteria.h"

template <class T>
class CriteriaArea : public Criteria<T>
{
public:
	CriteriaArea(double area);

	bool check(const std::shared_ptr<T>& item) const override;

private:
	double m_area;
};

#include "criteria_area_impl.cpp"

#endif
```

trapezoid.h:

```
#ifndef TRAPEZOID_H
#define TRAPEZOID_H

#include <iostream>
#include "figure.h"

class Trapezoid : public Figure
{
public:
	Trapezoid();
	Trapezoid(std::istream& is);

	void print() const override;
	double area() const override;
	const char* getName() const override;

	Trapezoid& operator = (const Trapezoid& other);
	bool operator == (const Trapezoid& other) const;

	void* operator new (size_t size);
	void operator delete (void* p);

	friend std::ostream& operator << (std::ostream& os, const Trapezoid& trapezoid);
```

```
        friend std::istream& operator >> (std::istream& is, Trapezoid& trapezoid);

private:
        double m_sideA;
        double m_sideB;
        double m_height;
};

#endif
```

list_item.h:

```
#ifndef LIST_ITEM_H
#define LIST_ITEM_H

#include <memory>

template <class T>
class ListItem
{
public:
        ListItem(const std::shared_ptr<T>& item);

        void swap(ListItem<T>& other);
        void setPrev(std::shared_ptr<ListItem<T>> prev);
        void setNext(std::shared_ptr<ListItem<T>> next);
        std::shared_ptr<ListItem<T>> getPrev();
        std::shared_ptr<ListItem<T>> getNext();
        std::shared_ptr<T> getItem() const;

private:
        std::shared_ptr<T> m_item;
        std::shared_ptr<ListItem<T>> m_prev;
        std::shared_ptr<ListItem<T>> m_next;
};

#include "list_item_impl.cpp"

#endif
```

queue_item.h:

```
#ifndef QUEUE_ITEM_H
#define QUEUE_ITEM_H

#include <memory>

template <class T>
class QueueItem
{
public:
        QueueItem(const std::shared_ptr<T>& item);

        void setNext(std::shared_ptr<QueueItem<T>> next);
        std::shared_ptr<QueueItem<T>> getNext();
        std::shared_ptr<T> getItem() const;

private:
        std::shared_ptr<T> m_item;
        std::shared_ptr<QueueItem<T>> m_next;
};

#include "queue_item_impl.cpp"

#endif
```

list_impl.cpp:

```
template <class T>
List<T>::List()
{
	m_size = 0;
}

template <class T>
List<T>::~List()
{
	while (size() > 0)
		erase(begin());
}

template <class T>
void List<T>::add(const std::shared_ptr<T>& item)
{
	std::shared_ptr<ListItem<T>> itemPtr = std::make_shared<ListItem<T>>(item);

	if (m_size == 0)
	{
		m_begin = itemPtr;
		m_end = m_begin;
	}
	else
	{
		itemPtr->setPrev(m_end);
		m_end->setNext(itemPtr);
		m_end = itemPtr;
	}

	++m_size;
}

template <class T>
void List<T>::erase(const Iterator<ListItem<T>, T>& it)
{
	if (m_size == 1)
	{
		m_begin = nullptr;
		m_end = nullptr;
	}
	else
	{
		std::shared_ptr<ListItem<T>> left = it.getItem()->getPrev();
		std::shared_ptr<ListItem<T>> right = it.getItem()->getNext();
		std::shared_ptr<ListItem<T>> mid = it.getItem();

		mid->setPrev(nullptr);
		mid->setNext(nullptr);

		if (left != nullptr)
			left->setNext(right);
		else
			m_begin = right;

		if (right != nullptr)
			right->setPrev(left);
		else
			m_end = left;
	}
```

```
	--m_size;
}

template <class T>
unsigned int List<T>::size() const
{
	return m_size;
}

template <class T>
Iterator<ListItem<T>, T> List<T>::get(unsigned int index) const
{
	if (index >= size())
		return end();

	Iterator<ListItem<T>, T> it = begin();

	while (index > 0)
	{
		++it;
		--index;
	}

	return it;
}

template <class T>
Iterator<ListItem<T>, T> List<T>::begin() const
{
	return Iterator<ListItem<T>, T>(m_begin);
}

template <class T>
Iterator<ListItem<T>, T> List<T>::end() const
{
	return Iterator<ListItem<T>, T>(nullptr);
}

template <class K>
std::ostream& operator << (std::ostream& os, const List<K>& list)
{
	if (list.size() == 0)
	{
		os << "================" << std::endl;
		os << "List is empty" << std::endl;
	}
	else
		for (std::shared_ptr<K> item : list)
			item->print();

	return os;
}

square.h:
#ifndef SQUARE_H
#define SQUARE_H

#include <iostream>
#include "figure.h"

class Square : public Figure
{
public:
```

```
	Square();
	Square(std::istream& is);

	void print() const override;
	double area() const override;
	const char* getName() const override;

	Square& operator = (const Square& other);
	bool operator == (const Square& other) const;

	void* operator new (size_t size);
	void operator delete (void* p);

	friend std::ostream& operator << (std::ostream& os, const Square& square);
	friend std::istream& operator >> (std::istream& is, Square& square);

private:
	double m_side;
};

#endif
```

trapezoid.cpp:

```
#include "trapezoid.h"

Trapezoid::Trapezoid()
{
	m_sideA = 0.0;
	m_sideB = 0.0;
	m_height = 0.0;
}

Trapezoid::Trapezoid(std::istream& is)
{
	is >> *this;
}

void Trapezoid::print() const
{
	std::cout << *this;
}

double Trapezoid::area() const
{
	return m_height * (m_sideA + m_sideB) / 2.0;
}

const char* Trapezoid::getName() const
{
	return "Trapezoid";
}

Trapezoid& Trapezoid::operator = (const Trapezoid& other)
{
	if (&other == this)
		return *this;

	m_sideA = other.m_sideA;
	m_sideB = other.m_sideB;
	m_height = other.m_height;

	return *this;
}
```

```
bool Trapezoid::operator == (const Trapezoid& other) const
{
	return m_sideA == other.m_sideA && m_sideB == other.m_sideB && m_height == other.m_height;
}

void* Trapezoid::operator new (size_t size)
{
	return Figure::allocator.allocate();
}

void Trapezoid::operator delete (void* p)
{
	Figure::allocator.deallocate(p);
}

std::ostream& operator << (std::ostream& os, const Trapezoid& trapezoid)
{
	os << "================" << std::endl;
	os << "Figure type: trapezoid" << std::endl;
	os << "Side A size: " << trapezoid.m_sideA << std::endl;
	os << "Side B size: " << trapezoid.m_sideB << std::endl;
	os << "Height: " << trapezoid.m_height << std::endl;

	return os;
}

std::istream& operator >> (std::istream& is, Trapezoid& trapezoid)
{
	std::cout << "================" << std::endl;
	std::cout << "Enter side A: ";
	is >> trapezoid.m_sideA;
	std::cout << "Enter side B: ";
	is >> trapezoid.m_sideB;
	std::cout << "Enter height: ";
	is >> trapezoid.m_height;

	return is;
}

container_impl.cpp:
template <class T>
void Container<T>::add(const std::shared_ptr<T>& item)
{
	auto lastContIt = m_container.begin();

	if (lastContIt == m_container.end())
		m_container.push(std::make_shared<List<T>>());

	lastContIt = m_container.begin();

	while (lastContIt.getItem()->getNext() != nullptr)
		++lastContIt;

	if ((*lastContIt)->size() == 5)
	{
		m_container.push(std::make_shared<List<T>>());
		++lastContIt;
	}

	(*lastContIt)->add(item);

	for (unsigned int i = (*lastContIt)->size() - 1; i > 0; --i)
```

```
		{
			auto lastElemIt = (*lastContIt)->get(i);
			auto preLastElemIt = (*lastContIt)->get(i - 1);

			if (strcmp(preLastElemIt->getName(), lastElemIt->getName()) <= 0)
				break;

			preLastElemIt.getItem()->swap(*lastElemIt.getItem());
		}
	}
}

template <class T>
void Container<T>::erase(const Criteria<T>& criteria)
{
	for (auto subCont : m_container)
	{
		while (true)
		{
			bool isRemoved = false;

			for (unsigned int i = 0; i < subCont->size(); ++i)
			{
				auto elemIt = subCont->get(i);

				if (criteria.check(*elemIt))
				{
					subCont->erase(elemIt);
					isRemoved = true;

					break;
				}
			}

			if (!isRemoved)
				break;
		}
	}

	while (m_container.size() > 0 && m_container.front()->size() == 0)
		m_container.pop();

	if (m_container.size() > 0)
	{
		auto firstSubCont = m_container.front();

		m_container.push(firstSubCont);
		m_container.pop();

		while (m_container.front() != firstSubCont)
		{
			if (m_container.front()->size() > 0)
				m_container.push(m_container.front());

			m_container.pop();
		}
	}
}

template <class K>
std::ostream& operator << (std::ostream& os, const Container<K>& container)
{
	if (container.m_container.size() == 0)
	{
```

```
			os << "================" << std::endl;
			os << "Container is empty" << std::endl;
		}
		else
		{
			unsigned int containerCnt1 = 1;

			for (auto subCont : container.m_container)
			{
				unsigned int containerCnt2 = 1;

				os << "================" << std::endl;
				os << "Container #" << (containerCnt1++) << ":" << std::endl;

				for (auto subItem : *subCont)
				{
					os << "================" << std::endl;
					os << "Item #" << (containerCnt2++) << ":" << std::endl;

					subItem->print();

					os << "Area: " << subItem->area() << std::endl;
				}
			}
		}

		return os;
}
```

allocator.h:

```
#ifndef ALLOCATOR_H
#define ALLOCATOR_H

#include <cstdlib>
#include "list.h"

#define R_CAST(__ptr, __type) reinterpret_cast<__type>(__ptr)

class Allocator
{
public:
		Allocator(unsigned int blockSize, unsigned int count);
		~Allocator();

		void* allocate();
		void deallocate(void* p);
		bool hasFreeBlocks() const;

private:
		void* m_memory;
		List<unsigned int> m_freeBlocks;
};

#endif
```

rectangle.h:

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

#include <iostream>
#include "figure.h"

class Rectangle : public Figure
```

```
{
public:
	Rectangle();
	Rectangle(std::istream& is);

	void print() const override;
	double area() const override;
	const char* getName() const override;

	Rectangle& operator = (const Rectangle& other);
	bool operator == (const Rectangle& other) const;

	void* operator new (size_t size);
	void operator delete (void* p);

	friend std::ostream& operator << (std::ostream& os, const Rectangle& rectangle);
	friend std::istream& operator >> (std::istream& is, Rectangle& rectangle);

private:
	double m_sideA;
	double m_sideB;
};

#endif
```

allocator.cpp:

```
#include "allocator.h"

Allocator::Allocator(unsigned int blockSize, unsigned int count)
{
	m_memory = malloc(blockSize * count);

	for (unsigned int i = 0; i < count; ++i)
		m_freeBlocks.add(std::make_shared<unsigned int>(i * blockSize));
}

Allocator::~Allocator()
{
	free(m_memory);
}

void* Allocator::allocate()
{
	void* res = R_CAST(R_CAST(m_memory, char*) + **m_freeBlocks.get(0), void*);

	m_freeBlocks.erase(m_freeBlocks.begin());

	return res;
}

void Allocator::deallocate(void* p)
{
	unsigned int offset = R_CAST(p, char*) - R_CAST(m_memory, char*);

	m_freeBlocks.add(std::make_shared<unsigned int>(offset));
}

bool Allocator::hasFreeBlocks() const
{
	return m_freeBlocks.size() > 0;
}
```

queue_impl.cpp:

```
template <class T>
Queue<T>::Queue()
{
	m_size = 0;
}

template <class T>
Queue<T>::~Queue()
{
	while (size() > 0)
		pop();
}

template <class T>
void Queue<T>::push(const std::shared_ptr<T>& item)
{
	std::shared_ptr<QueueItem<T>> itemPtr = std::make_shared<QueueItem<T>>(item);

	if (m_size == 0)
	{
		m_front = itemPtr;
		m_end = m_front;
	}
	else
	{
		m_end->setNext(itemPtr);
		m_end = itemPtr;
	}

	++m_size;
}

template <class T>
void Queue<T>::pop()
{
	if (m_size == 1)
	{
		m_front = nullptr;
		m_end = nullptr;
	}
	else
		m_front = m_front->getNext();

	--m_size;
}

template <class T>
unsigned int Queue<T>::size() const
{
	return m_size;
}

template <class T>
std::shared_ptr<T> Queue<T>::front() const
{
	return m_front->getItem();
}

template <class T>
Iterator<QueueItem<T>, T> Queue<T>::begin() const
{
	return Iterator<QueueItem<T>, T>(m_front);
}
```

```
template <class T>
Iterator<QueueItem<T>, T> Queue<T>::end() const
{
	return Iterator<QueueItem<T>, T>(nullptr);
}

template <class K>
std::ostream& operator << (std::ostream& os, const Queue<K>& queue)
{
	if (queue.size() == 0)
	{
		os << "================" << std::endl;
		os << "Queue is empty" << std::endl;
	}
	else
		for (std::shared_ptr<K> item : queue)
			item->print();

	return os;
}
```

criteria_type.h:

```
#ifndef CRITERIA_TYPE_H
#define CRITERIA_TYPE_H

#include <cstring>
#include "criteria.h"

template <class T>
class CriteriaType : public Criteria<T>
{
public:
	CriteriaType(const char* type);

	bool check(const std::shared_ptr<T>& item) const override;

private:
	char m_type[16];
};

#include "criteria_type_impl.cpp"

#endif
```

list_item_impl.cpp:

```
template <class T>
ListItem<T>::ListItem(const std::shared_ptr<T>& item)
{
	m_item = item;
}

template <class T>
void ListItem<T>::swap(ListItem<T>& other)
{
	m_item.swap(other.m_item);
}

template <class T>
void ListItem<T>::setPrev(std::shared_ptr<ListItem<T>> prev)
{
	m_prev = prev;
}
```

```
template <class T>
void ListItem<T>::setNext(std::shared_ptr<ListItem<T>> next)
{
	m_next = next;
}

template <class T>
std::shared_ptr<ListItem<T>> ListItem<T>::getPrev()
{
	return m_prev;
}

template <class T>
std::shared_ptr<ListItem<T>> ListItem<T>::getNext()
{
	return m_next;
}

template <class T>
std::shared_ptr<T> ListItem<T>::getItem() const
{
	return m_item;
}
```

figure.h:

```
#ifndef FIGURE_H
#define FIGURE_H

#include <cstring>
#include "allocator.h"

class Figure
{
public:
	virtual ~Figure();
	virtual void print() const = 0;
	virtual double area() const = 0;
	virtual const char* getName() const = 0;

	static Allocator allocator;
};

#endif
```

iterator_impl.cpp:

```
template <class N, class T>
Iterator<N, T>::Iterator(const std::shared_ptr<N>& item)
{
	m_item = item;
}

template <class N, class T>
std::shared_ptr<N> Iterator<N, T>::getItem() const
{
	return m_item;
}

template <class N, class T>
std::shared_ptr<T> Iterator<N, T>::operator * ()
{
	return m_item->getItem();
}
```

```
template <class N, class T>
std::shared_ptr<T> Iterator<N, T>::operator -> ()
{
	return m_item->getItem();
}

template <class N, class T>
Iterator<N, T> Iterator<N, T>::operator ++ ()
{
	m_item = m_item->getNext();

	return *this;
}

template <class N, class T>
Iterator<N, T> Iterator<N, T>::operator ++ (int index)
{
	Iterator tmp(m_item);

	m_item = m_item->getNext();

	return tmp;
}

template <class N, class T>
bool Iterator<N, T>::operator == (const Iterator& other) const
{
	return m_item == other.m_item;
}

template <class N, class T>
bool Iterator<N, T>::operator != (const Iterator& other) const
{
	return !(*this == other);
}
```

queue.h:

```
#ifndef QUEUE_H
#define QUEUE_H

#include <iostream>
#include "queue_item.h"
#include "iterator.h"

template <class T>
class Queue
{
public:
	Queue();
	~Queue();

	void push(const std::shared_ptr<T>& item);
	void pop();
	unsigned int size() const;
	std::shared_ptr<T> front() const;

	Iterator<QueueItem<T>, T> begin() const;
	Iterator<QueueItem<T>, T> end() const;

	template <class K>
	friend std::ostream& operator << (std::ostream& os, const Queue<K>& queue);
```

```
private:
	std::shared_ptr<QueueItem<T>> m_front;
	std::shared_ptr<QueueItem<T>> m_end;
	unsigned int m_size;
};

#include "queue_impl.cpp"

#endif
```

criteria_area_impl.cpp:

```
template <class T>
CriteriaArea<T>::CriteriaArea(double area)
{
	m_area = area;
}

template <class T>
bool CriteriaArea<T>::check(const std::shared_ptr<T>& item) const
{
	return item->area() < m_area;
}
```

container.h:

```
#ifndef CONTAINTER_H
#define CONTAINTER_H

#include <memory>
#include <cstring>
#include "queue.h"
#include "list.h"
#include "criteria.h"

template <class T>
class Container
{
public:
	void add(const std::shared_ptr<T>& item);
	void erase(const Criteria<T>& criteria);
	//void print() const;

	template <class K>
	friend std::ostream& operator << (std::ostream& os, const Container<K>& container);

private:
	Queue<List<T>> m_container;
};

#include "container_impl.cpp"

#endif
```

square.cpp:

```
#include "square.h"

Square::Square()
{
	m_side = 0.0;
}

Square::Square(std::istream& is)
{
	is >> *this;
```

```
}

void Square::print() const
{
	std::cout << *this;
}

double Square::area() const
{
	return m_side * m_side;
}

const char* Square::getName() const
{
	return "Square";
}

Square& Square::operator = (const Square& other)
{
	if (&other == this)
		return *this;

	m_side = other.m_side;

	return *this;
}

bool Square::operator == (const Square& other) const
{
	return m_side == other.m_side;
}

void* Square::operator new (size_t size)
{
	return Figure::allocator.allocate();
}

void Square::operator delete (void* p)
{
	Figure::allocator.deallocate(p);
}

std::ostream& operator << (std::ostream& os, const Square& square)
{
	os << "================" << std::endl;
	os << "Figure type: square" << std::endl;
	os << "Side size: " << square.m_side << std::endl;

	return os;
}

std::istream& operator >> (std::istream& is, Square& square)
{
	std::cout << "================" << std::endl;
	std::cout << "Enter side: ";
	is >> square.m_side;

	return is;
}

criteria.h:
#ifndef CRITERIA_H
#define CRITERIA_H
```

```
template <class T>
class Criteria
{
public:
	virtual bool check(const std::shared_ptr<T>& item) const = 0;
};

#endif
```

criteria_type_impl.cpp:

```
template <class T>
CriteriaType<T>::CriteriaType(const char* type)
{
	strcpy(m_type, type);
}

template <class T>
bool CriteriaType<T>::check(const std::shared_ptr<T>& item) const
{
	return strcmp(m_type, item->getName()) == 0;
}
```

list.h:

```
#ifndef LIST_H
#define LIST_H

#include <iostream>
#include "list_item.h"
#include "iterator.h"

template <class T>
class List
{
public:
	List();
	~List();

	void add(const std::shared_ptr<T>& item);
	void erase(const Iterator<ListItem<T>, T>& it);
	unsigned int size() const;
	Iterator<ListItem<T>, T> get(unsigned int index) const;

	Iterator<ListItem<T>, T> begin() const;
	Iterator<ListItem<T>, T> end() const;

	template <class K>
	friend std::ostream& operator << (std::ostream& os, const List<K>& list);

private:
	std::shared_ptr<ListItem<T>> m_begin;
	std::shared_ptr<ListItem<T>> m_end;
	unsigned int m_size;
};

#include "list_impl.cpp"

#endif
```

makefile:

```
CC = g++
CFLAGS = -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result
FILES = main.cpp square.cpp rectangle.cpp trapezoid.cpp figure.cpp allocator.cpp
```

```
PROG = lab7

all:
	$(CC) $(CFLAGS) -o $(PROG) $(FILES)

clean:
	rm $(PROG)
```

main.cpp:

```
#include "container.h"
#include "square.h"
#include "rectangle.h"
#include "trapezoid.h"
#include "criteria_type.h"
#include "criteria_area.h"

int main()
{
	unsigned int action;
	Container<Figure> cont;

	while (true)
	{
		std::cout << "================" << std::endl;
		std::cout << "Menu:" << std::endl;
		std::cout << "1) Add figure" << std::endl;
		std::cout << "2) Delete figure" << std::endl;
		std::cout << "3) Print" << std::endl;
		std::cout << "0) Quit" << std::endl;
		std::cin >> action;

		if (action == 0)
			break;

		if (action > 3)
		{
			std::cout << "Error: invalid action" << std::endl;

			continue;
		}

		switch (action)
		{
			case 1:
			{
				if (!Figure::allocator.hasFreeBlocks())
					std::cout << "Error. No free blocks" << std::endl;
				else
				{
					unsigned int figureType;

					std::cout << "================" << std::endl;
					std::cout << "1) Square" << std::endl;
					std::cout << "2) Rectangle" << std::endl;
					std::cout << "3) Trapezoid" << std::endl;
					std::cout << "0) Quit" << std::endl;
					std::cin >> figureType;

					if (figureType > 0)
					{
						if (figureType > 3)
						{
							std::cout << "Error: invalid figure type" << std::endl;
```

```
                                        continue;
                                }

                                switch (figureType)
                                {
                                        case 1:
                                        {
                                                cont.add(std::shared_ptr<Square>(new
Square(std::cin)));

                                                break;
                                        }

                                        case 2:
                                        {
                                                cont.add(std::shared_ptr<Rectangle>(new
Rectangle(std::cin)));

                                                break;
                                        }

                                        case 3:
                                        {
                                                cont.add(std::shared_ptr<Trapezoid>(new
Trapezoid(std::cin)));

                                                break;
                                        }
                                }
                        }
                }

                break;
        }

        case 2:
        {
                unsigned int byCriteria;

                std::cout << "================" << std::endl;
                std::cout << "1) By type" << std::endl;
                std::cout << "2) By area" << std::endl;
                std::cout << "0) Quit" << std::endl;
                std::cin >> byCriteria;

                if (byCriteria > 0)
                {
                        if (byCriteria > 2)
                        {
                                std::cout << "Error: invalid criteria" << std::endl;

                                continue;
                        }

                        switch (byCriteria)
                        {
                                case 1:
                                {
                                        unsigned int figureType;

                                        std::cout << "================" << std::endl;
                                        std::cout << "1) Square" << std::endl;
```

```
                                                std::cout << "2) Rectangle" << std::endl;
                                                std::cout << "3) Trapezoid" << std::endl;
                                                std::cout << "0) Quit" << std::endl;
                                                std::cin >> figureType;

                                                if (figureType > 0)
                                                {
                                                        if (figureType > 3)
                                                        {
                                                                std::cout << "Error: invalid figure type"
<< std::endl;


                                                                continue;
                                                        }

                                                        switch (figureType)
                                                        {
                                                                case 1:
                                                                {

        cont.erase(CriteriaType<Figure>("Square"));

                                                                        break;
                                                                }

                                                                case 2:
                                                                {

        cont.erase(CriteriaType<Figure>("Rectangle"));

                                                                        break;
                                                                }

                                                                case 3:
                                                                {

        cont.erase(CriteriaType<Figure>("Trapezoid"));

                                                                        break;
                                                                }
                                                        }
                                                }

                                                break;
                                        }

                                        case 2:
                                        {
                                                double area;

                                                std::cout << "Enter area: ";
                                                std::cin >> area;

                                                cont.erase(CriteriaArea<Figure>(area));

                                                break;
                                        }
                                }
                        }

                        break;
                }
```

```
                        case 3:
                        {
                                std::cout << cont;

                                break;
                        }
                }
        }

        return 0;
}
```

rectangle.cpp:

```
#include "rectangle.h"

Rectangle::Rectangle()
{
        m_sideA = 0.0;
        m_sideB = 0.0;
}

Rectangle::Rectangle(std::istream& is)
{
        is >> *this;
}

void Rectangle::print() const
{
        std::cout << *this;
}

double Rectangle::area() const
{
        return m_sideA * m_sideB;
}

const char* Rectangle::getName() const
{
        return "Rectangle";
}

Rectangle& Rectangle::operator = (const Rectangle& other)
{
        if (&other == this)
                return *this;

        m_sideA = other.m_sideA;
        m_sideB = other.m_sideB;

        return *this;
}

bool Rectangle::operator == (const Rectangle& other) const
{
        return m_sideA == other.m_sideA && m_sideB == other.m_sideB;
}

void* Rectangle::operator new (size_t size)
{
        return Figure::allocator.allocate();
}

void Rectangle::operator delete (void* p)
```

```
{
	Figure::allocator.deallocate(p);
}

std::ostream& operator << (std::ostream& os, const Rectangle& rectangle)
{
	os << "================" << std::endl;
	os << "Figure type: rectangle" << std::endl;
	os << "Side A size: " << rectangle.m_sideA << std::endl;
	os << "Side B size: " << rectangle.m_sideB << std::endl;

	return os;
}

std::istream& operator >> (std::istream& is, Rectangle& rectangle)
{
	std::cout << "================" << std::endl;
	std::cout << "Enter side A: ";
	is >> rectangle.m_sideA;
	std::cout << "Enter side B: ";
	is >> rectangle.m_sideB;

	return is;
}
```

figure.cpp:

```
#include "figure.h"

Allocator Figure::allocator(48, 100);

Figure::~Figure() {}
```

iterator.h:

```
#ifndef ITERATOR_H
#define ITERATOR_H

template <class N, class T>
class Iterator
{
public:
	Iterator(const std::shared_ptr<N>& item);

	std::shared_ptr<N> getItem() const;

	std::shared_ptr<T> operator * ();
	std::shared_ptr<T> operator -> ();
	Iterator operator ++ ();
	Iterator operator ++ (int index);
	bool operator == (const Iterator& other) const;
	bool operator != (const Iterator& other) const;

private:
	std::shared_ptr<N> m_item;
};

#include "iterator_impl.cpp"

#endif
```