Московский Авиационный Институт
(Национальный Исследовательский Университет)

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

Лабораторная работа №9.
Тема: «Лямбда выражения»

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

Москва
2017

Цель работы

Целью лабораторной работы является:

- Знакомство с лямбда-выражениями

Задание

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер первого уровня и классы-фигуры) необходимо разработать:

- Контейнер второго уровня с использованием шаблонов.
- Реализовать с помощью лямбда-выражений набор команд, совершающих операции над контенйром 1-го уровня:
    - Генерация фигур со случайным значением параметров;
    - Печать контейнера на экран;
    - Удаление элементов со значением площади меньше определенного числа;
- В контенер второго уровня поместить цепочку команд.
- Реализовать цикл, который проходит по всем командам в контенере второго уровня и выполняет их, применяя к контейнеру первого уровня.

Для создания потоков использовать механизмы:

- future
- packaged_task/async

Для обеспечения потоко-безопасности структур данных использовать:

- mutex
- lock_guard

Нельзя использовать:

- Стандартные контейнеры std.

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;
}

trapezoid.h:
#ifndef TRAPEZOID_H
#define TRAPEZOID_H

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

class Trapezoid : public Figure
{
public:
        Trapezoid();
        Trapezoid(double sideA, double sideB, double height);
        Trapezoid(std::istream& is);

        void print() const override;
        double area() 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 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(double side);
	Square(std::istream& is);

	void print() const override;
	double area() 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(double sideA, double sideB, double height)
{
	m_sideA = sideA;
	m_sideB = sideB;
	m_height = height;
```

```
}

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;
}

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;
}

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(double sideA, double sideB);
	Rectangle(std::istream& is);

	void print() const override;
	double area() 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;
}

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>::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 "allocator.h"

class Figure
{
public:
	virtual ~Figure() {}
	virtual void print() const = 0;
	virtual double area() 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
```

square.cpp:

```
#include "square.h"

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

Square::Square(double side)
{
	m_side = side;
}

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

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

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

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;
}
```

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 -pthread
FILES = main.cpp square.cpp rectangle.cpp trapezoid.cpp figure.cpp allocator.cpp
PROG = lab9

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

clean:
	rm $(PROG)
```

main.cpp:

```
#include <functional>
#include <random>
#include <chrono>
#include <string>
#include <mutex>
#include <thread>
#include "queue.h"
#include "square.h"
#include "rectangle.h"
#include "trapezoid.h"

typedef std::function<void(void)> Command;

int main()
{
	Queue<Figure> q;
	List<Command> cmds;
	List<std::string> cmdsNames;
	std::mutex mtx;

	Command cmdInsert = [&]()
	{
		std::lock_guard<std::mutex> guard(mtx);

		unsigned int seed = std::chrono::system_clock::now().time_since_epoch().count();

		std::default_random_engine generator(seed);
		std::uniform_int_distribution<int> distrFigureType(1, 3);
		std::uniform_int_distribution<int> distrFigureParam(1, 10);

		std::cout << "================" << std::endl;
		std::cout << "Command: insert" << std::endl;

		switch (distrFigureType(generator))
		{
			case 1:
			{
				std::cout << "================" << std::endl;
				std::cout << "Inserted: square" << std::endl;

				double side = distrFigureParam(generator);

				q.push(std::shared_ptr<Square>(new Square(side)));

				break;
			}

			case 2:
			{
				std::cout << "================" << std::endl;
				std::cout << "Inserted: rectangle" << std::endl;
```

```
                double sideA = distrFigureParam(generator);
                double sideB = distrFigureParam(generator);

                q.push(std::shared_ptr<Rectangle>(new Rectangle(sideA, sideB)));

                break;
            }

            case 3:
            {
                std::cout << "================" << std::endl;
                std::cout << "Inserted: trapezoid" << std::endl;

                double sideA = distrFigureParam(generator);
                double sideB = distrFigureParam(generator);
                double height = distrFigureParam(generator);

                q.push(std::shared_ptr<Trapezoid>(new Trapezoid(sideA, sideB, height)));

                break;
            }
        }
};

Command cmdErase = [&]()
{
        std::lock_guard<std::mutex> guard(mtx);

        const double AREA = 24.0;

        std::cout << "================" << std::endl;
        std::cout << "Command: erase" << std::endl;

        if (q.size() == 0)
        {
            std::cout << "================" << std::endl;
            std::cout << "Queue is empty" << std::endl;
        }
        else
        {
            std::shared_ptr<Figure> first = q.front();

            while (true)
            {
                bool isRemoved = false;

                for (auto figure : q)
                {
                    if (figure->area() < AREA)
                    {
                        std::cout << "================" << std::endl;
                        std::cout << "Removed" << std::endl;

                        figure->print();
                        std::cout << "Area: " << figure->area() << std::endl;

                        q.pop();
                        isRemoved = true;

                        break;
                    }
                }
```

```
                        if (!isRemoved)
                                break;
                }
        }
};

Command cmdPrint = [&]()
{
        std::lock_guard<std::mutex> guard(mtx);

        std::cout << "================" << std::endl;
        std::cout << "Command: print" << std::endl;

        for (auto figure : q)
        {
                figure->print();

                std::cout << "Area: " << figure->area() << std::endl;
        }
};

while (true)
{
        unsigned int action;

        std::cout << "================" << std::endl;
        std::cout << "Menu:" << std::endl;
        std::cout << "1) Add command" << std::endl;
        std::cout << "2) Erase command" << std::endl;
        std::cout << "3) Execute commands" << std::endl;
        std::cout << "4) Print commands" << std::endl;
        std::cout << "0) Quit" << std::endl;
        std::cin >> action;

        if (action == 0)
                break;

        if (action > 4)
        {
                std::cout << "Error: invalid action" << std::endl;

                continue;
        }

        switch (action)
        {
                case 1:
                {
                        unsigned int commandType;

                        std::cout << "================" << std::endl;
                        std::cout << "1) Insert" << std::endl;
                        std::cout << "2) Erase" << std::endl;
                        std::cout << "3) Print" << std::endl;
                        std::cout << "0) Quit" << std::endl;
                        std::cin >> commandType;

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

```
                                        continue;
                                }

                                switch (commandType)
                                {
                                        case 1:
                                        {
                                                cmds.add(std::shared_ptr<Command>(&cmdInsert, []
(Command*){}));
                                                cmdsNames.add(std::shared_ptr<std::string>(new
std::string("Insert")));

                                                break;
                                        }

                                        case 2:
                                        {
                                                cmds.add(std::shared_ptr<Command>(&cmdErase, []
(Command*){}));
                                                cmdsNames.add(std::shared_ptr<std::string>(new
std::string("Erase")));

                                                break;
                                        }

                                        case 3:
                                        {
                                                cmds.add(std::shared_ptr<Command>(&cmdPrint, []
(Command*){}));
                                                cmdsNames.add(std::shared_ptr<std::string>(new
std::string("Print")));

                                                break;
                                        }
                                }
                        }

                        break;
                }

                case 2:
                {
                        unsigned int commandIndex;

                        std::cout << "================" << std::endl;
                        std::cout << "Command index: ";
                        std::cin >> commandIndex;

                        if (commandIndex >= cmds.size())
                        {
                                std::cout << "Error: invalid command index" << std::endl;

                                continue;
                        }

                        cmds.erase(cmds.get(commandIndex));
                        cmdsNames.erase(cmdsNames.get(commandIndex));

                        break;
                }

                case 3:
```

```
                {
                        Queue<std::thread> ths;

                        for (auto cmd : cmds)
                                ths.push(std::shared_ptr<std::thread>(new std::thread(*cmd)));

                        for (auto th : ths)
                                th->join();

                        break;
                }

                case 4:
                {
                        std::cout << "================" << std::endl;

                        if (cmds.size() == 0)
                                std::cout << "Commands list is empty" << std::endl;
                        else
                                for (auto cmdName : cmdsNames)
                                        std::cout << *cmdName << std::endl;

                        break;
                }
        }
    }

    return 0;
}
```

rectangle.cpp:

```
#include "rectangle.h"

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

Rectangle::Rectangle(double sideA, double sideB)
{
    m_sideA = sideA;
    m_sideB = sideB;
}

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

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

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

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(32, 100);
```

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
```