Московский авиационный институт
(национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Отчет по лабораторным работам по курсу «Объектно-ориентированное программирование»

Студентка:	Слесарева Д.С.
Преподаватель:	Поповкин А.В.
Группа:	08-202
Вариант:	26
Дата:	
Оценка:	
Подпись:	

Москва, 2017

Оглавление

Лабораторная работа №1

Цель работы:

- Программирование классов на языке C++
- Управление памятью в языке C++
- Изучение базовых понятий ООП.
- Знакомство с классами в C++.
- Знакомство с перегрузкой операторов.
- Знакомство с дружественными функциями.
- Знакомство с операциями ввода-вывода из стандартных библиотек.

Задача:

Необходимо спроектировать и запрограммировать на языке C++ классы фигур, согласно варианту задания Классы должны удовлетворять следующим правилам:

- Должны иметь общий родительский класс Figure.
- Должны иметь общий виртуальный метод Print, печатающий параметры фигуры и ее тип в стандартный поток вывода cout.
- Должный иметь общий виртуальный метод расчета площади фигуры – Square.
- Должны иметь конструктор, считывающий значения основных параметров фигуры из стандартного
потока cin.
- Должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание
методов (.cpp).

Программа должна позволять вводить фигуру каждого типа с клавиатуры, выводить параметры фигур на экран и их площадь.

Вариант задания: 26.
Фигуры: квадрат, прямоугольник, трапеция.
Варианты структуры данных: контейнер первого уровня – очередь, контейнер второго уровня – связный список.

Описание

Объектно-ориентированное программирование или ООП — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного типа, использующая механизм пересылки сообщений и классы, организованные в иерархию наследования.
Центральный элемент ООП — абстракция. Данные с помощью абстракции преобразуются в объекты, а последовательность обработки этих данных превращается в набор сообщений, передаваемых между этими объектами. Каждый из объектов имеет свое собственное уникальное поведение. С объектами можно обращаться как с конкретными сущностями, которые реагируют на сообщения, приказывающие им выполнить какие-то действия.
ООП характеризуется следующими принципами:

- все является объектом;
- вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некоторое действие; объекты взаимодействуют, посылая и получая сообщения; сообщение — это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия;
- каждый объект имеет независимую **память**, которая состоит из других объектов;
- каждый объект является представителем класса, который выражает общие свойства объектов данного типа;
- в классе задается **функциональность** (поведение объекта); тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия;
- классы организованы в единую древовидную структуру с общим корнем, называемую **иерархией наследования**;
- память и поведение, связанное с экземплярами определенного класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

Абстрагирование — метод решения задачи, при котором объекты разного рода объединяются общим понятием (концепцией), а затем сгруппированные сущности рассматриваются как элементы единой категории.
Абстрагирование позволяет отделить логический смысл фрагмента программы от проблемы его реализации, разделив внешнее описание(интерфейс) объекта и его внутреннюю организацию (реализацию).

Инкапсуляция — техника, при которой несущественная с точки зрения интерфейса объекта информация прячется внутри него.

Наследование — свойство объектов, посредством которого экземпляры класса получают доступ к данным и методам классов-предков без их повторного определения. Наследование позволяет различным типам данных совместно использовать один и тот же код, приводя к уменьшению его размера и повышению функциональности.

Полиморфизм — свойство, позволяющее использовать один и тот же интерфейс для различных действий; полиморфной переменной, например, может соответствовать несколько различных методов.
Полиморфизм перекраивает общий код, реализующий некоторый интерфейс, так, чтобы удовлетворить конкретным особенностям отдельных типов данных.

Класс — множество объектов, связанных общностью структуры и поведения; абстрактное описание данных и поведения (методов) для совокупности похожих объектов, представители которой называются экземплярами класса.

Объект — конкретная реализация класса, обладающая характеристиками состояния, поведения и индивидуальности, синоним экземпляра.
Конструктор класса — специальный блок инструкций, вызываемый при создании объекта.
Деструктор — специальный метод класса, который служит для уничтожения элементов класса.

Виртуальная функция — это функция-член, которую предполагается переопределить в производных классах. При ссылке на объект производного класса с помощью указателя или ссылки на базовый класс можно вызвать виртуальную функцию для этого объекта и выполнить версию функции производного класса. Виртуальные функции обеспечивают вызов соответствующей функции для объекта независимо от выражения, используемого для вызова функции.

Исходный код

square.cpp	
Square();	Конструктор класса
Square(std::istream& is);	Конструктор класса из стандартного потока
double area() const override;	Площадь фигуры
void print() const override;	Печать фигуры
rectangle.cpp	
Rectangle();	Конструктор класса
Rectangle(std::istream& is);	Конструктор класса из стандартного потока
double area() const override;	Площадь фигуры
void print() const override;	Печать фигуры
trapezoid.cpp	
Trapezoid();	Конструктор класса
Trapezoid(std::istream& is);	Конструктор класса из стандартного потока
double area() const override;	Площадь фигуры
void print() const override;	Печать фигуры

- figure.h

```
#ifndef FIGURE_H
#define FIGURE_H

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

#endif
```

- 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;

private:
	double m_side;
};

#endif
```

- rectangle.cpp

```
#include "rectangle.h"
```

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

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

void Rectangle::print() const
{
   std::cout << "================" << std::endl;
   std::cout << "Figure type: rectangle" << std::endl;
   std::cout << "Side A size: " << m_sideA << std::endl;
   std::cout << "Side B size: " << m_sideB << std::endl;
}

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

- 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;

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

#endif
```

Консоль

```
dasha@belosnezzzka:~/OOP$ cd 1
dasha@belosnezzzka:~/OOP/1$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -o lab1 *.cpp
dasha@belosnezzzka:~/OOP/1$ ./lab1
================
Enter side: 2
================
Figure type: square
Side size: 2
Area: 4
================
Enter side A: 3
Enter side B: 4
================
Figure type: rectangle
Side A size: 3
Side B size: 4
Area: 12
================
Enter side A: 5
Enter side B: 6
Enter height: 7
================
Figure type: trapezoid
Side A size: 5
Side B size: 6
Height: 7
Area: 38.5
```

Выводы

Сделав данную лабораторную работу, я познакомилась с перегрузкой операторов, операциями ввода и вывода из стандартных библиотек и классами в C++. Была создана программа, позволяющая вводить фигуру каждого типа с клавиатуры и выводить параметры фигур на экран и их площадь. Опыт, полученный при создании этой лабораторной работы пригодится мне при создании других лабораторных работ и дальнейшем обучении Объектно-ориентированному программированию.

Лабораторная работа №2

Цель работы:

- Закрепление навыков работы с классами.
- Создание простых динамических структур данных.
- Работа с объектами, передаваемыми «по значению».

Задача:

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий **одну фигуру,** согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

- Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
- Классы фигур должны иметь переопределенный оператор вывода в поток std::ostream (<<).
- Оператор должен распечатывать параметры фигуры (тип фигуры, длины сторон, радиус и т.д).
- Классы фигур должны иметь переопределенный оператор ввода фигуры из потока std::istream (>>).
- Оператор должен вводить основные параметры фигуры (длины сторон, радиус и т.д).
- Классы фигур должны иметь операторы копирования (=).
- Классы фигур должны иметь операторы сравнения с такими же фигурами (==).
- Класс-контейнер должен соджержать объекты фигур “по значению” (не по ссылке).
- Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.
- Класс-контейнер должен иметь методы по получению фигуры из контейнера (опеределяется
- структурой контейнера).
- Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опеределяется
- структурой контейнера).
- Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток
- std::ostream (<<).
- Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
- Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно
- описание методов (.cpp).

Вариант задания: 26.

Фигуры: квадрат, прямоугольник, трапеция.

Варианты структуры данных: контейнер первого уровня – очередь, контейнер второго уровня – связный список.

Описание

Очередь — абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» (FIFO, First In — First Out). Добавление элемента возможно лишь в конец очереди, выборка — только из начала очереди, при этом выбранный элемент из очереди удаляется.

Параметры в функцию могут передаваться одним из следующих способов:

- по значению;
- по ссылке.

При передаче аргументов **по значению** компилятор создает временную копию объекта, который должен быть передан, и размещает ее в области стековой памяти, предназначенной для хранения локальных объектов. Вызываемая функция оперирует именно с этой копией, не оказывая влияния на оригинал объекта. Прототипы функций, принимающих аргументы по значению, предусматривают в качестве параметров указание типа объекта, а не его адреса.

Если же необходимо, чтобы функция модифицировала оригинал объекта, используется передача параметров **по ссылке**. При этом в функцию передается не сам объект, а только его адрес. Таким образом, все модификации в теле функции переданных ей по ссылке аргументов воздействуют на объект. Принимая во внимание тот факт, что функция может возвращать лишь единственное значение, использование передачи адреса объекта оказывается весьма эффективным способом работы с большим числом данных. Кроме того, так как передается адрес, а не сам объект, существенно экономится стековая память.

Исходный код

square.cpp	
Square();	Конструктор класса
Square(std::istream& is);	Конструктор класса из стандартного потока
double area() const override;	Площадь фигуры
void print() const override;	Печать фигуры
Square& operator = (const Square& other);	Переопределённый оператор копирования
bool operator == (const Square& other) const;	Переопределённый оператор сравнения
friend std::ostream& operator << (std::ostream& os, const Square& square);	Переопределённый оператор вывода в поток std::ostream
friend std::istream& operator >> (std::istream& is, Square& square);	Переопределённый оператор вывода в поток std::istream
queue.cpp	
Queue();	Конструктор класса
~Queue();	Деструктор класса
void push(const Square& square);	Добавить в очередь
void pop();	Удалить из очереди
friend std::ostream& operator << (std::ostream& os, const Queue& queue);	Переопределённый оператор вывода в поток std::ostream
queue_item.cpp	
QueueItem(const Square& square);	Конструктор класса
void setNext(QueueItem* next);	Установка ссылки на следующий элемент
QueueItem* getNext();	Получение ссылки на следующий элемент
Square getSquare() const;	Получение площади

- figure.h

```
#ifndef FIGURE_H
#define FIGURE_H

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

```
#endif
```

- 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;

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

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

private:
   double m_side;
};

#endif
```

- queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

#include "queue_item.h"

class Queue
{
public:
   Queue();
   ~Queue();

   void push(const Square& square);
   void pop();
   unsigned int size() const;
   Square front() const;

   friend std::ostream& operator << (std::ostream& os, const Queue& queue);

private:
   QueueItem* m_front;
   QueueItem* m_end;
   unsigned int m_size;
};

#endif
```

- queue_item.h

```
#ifndef QUEUE_ITEM_H
#define QUEUE_ITEM_H

#include "square.h"

class QueueItem
{
public:
   QueueItem(const Square& square);

   void setNext(QueueItem* next);
   QueueItem* getNext();
   Square getSquare() const;

private:
   Square m_square;
   QueueItem* m_next;
};

#endif
```

Консоль

```
dasha@belosnezzzka:~/OOP$ cd 2
dasha@belosnezzzka:~/OOP/2$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -o lab2 *.cpp
dasha@belosnezzzka:~/OOP/2$ ./lab2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
Enter side: 2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
Enter side: 3
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: square
Side size: 2
================
Figure type: square
Side size: 3
================
```

```
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: square
Side size: 3
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
```

Выводы

Сделав данную лабораторную работу, я закрепила навыки работы с классами, создала простую динамическую структуру данных под названием «очередь», также поработала с передачей объектов по значению. Была создана программа, позволяющая вводить произвольное количество фигур и добавлять их в контейнер, распечатывать содержимое контейнера и удалять фигуры из контейнера. Опыт, полученный при создании этой лабораторной работы пригодится мне при создании других лабораторных работ и дальнейшем обучении объектно-ориентированному программированию.

Лабораторная работа №3

Цель работы:

- Закрепление навыков работы с классами.
- Знакомство с умными указателями.

Задача:

Необходимо спроектировать и запрограммировать на языке С++ класс-контейнер первого уровня, содержащий **все три фигуры,** согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

- Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
- Класс-контейнер должен соджержать объекты используя std:shared_ptr<…>.
- Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.
- Класс-контейнер должен иметь методы по получению фигуры из контейнера (опередеяется структурой контейнера).
- Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опередеяется структурой контейнера).
- Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).
- Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
- Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Вариант задания: 26.

Фигуры: квадрат, прямоугольник, трапеция.

Варианты структуры данных: контейнер первого уровня – очередь, контейнер второго уровня – связный список.

Описание

Умный указатель — это тот же обычный указатель, обеспечивающий безопасность благодаря автоматическому управлению памятью. Такой указатель помогает избежать множества проблем: «висячие» указатели, «утечки» памяти и отказы в выделении памяти. Интеллектуальный указатель должен подсчитывать количество ссылок на указанный объект.

- std::auto_ptr<>

  auto_ptr является умным указателем, реализующим семантику владения. Стоит отметить, что auto_ptr – это единственный умный указатель, включенный в нынешний стандарт C++

- std::shared_ptr<>

  shared_ptr является более гибкой и универсальной версией умного указателя. shared_ptr обладает во многом схожей с auto_ptr семантикой, основные отличия заключаются в методике копирования. Класс shared_ptr реализует разделяемое (shared) владение с подсчётом ссылок, что позволяет использовать более привычные конструкторы копирования и операторы присваивания. Благодаря такой семантике shared_ptr можно использовать в стандартных контейнерах STL. Кроме того, shared_ptr позволяет задавать функтор удаления (deleter), что может быть полезно для классов, имеющих необычную семантику удаления.

- std::weak_ptr<>

  weak_ptr – это «слабый», не владеющий экземпляром объекта умный указатель. weak_ptr ссылается на экземпляр, которым владеет shared_ptr, однако не разделяет прав владения этим экземпляром. Это гарантирует, что если экземпляр уже уничтожен, мы сможем это надёжно и безопасно проверить.

Исходный код

queue.cpp	
Queue();	Конструктор класса
~Queue();	Деструктор класса
void push(const std::shared_ptr<Figure>& figure);	Добавить в очередь
void pop();	Удалить из очереди
std::shared_ptr<Figure> front() const;	Получение фигуры из узла
std::shared_ptr<QueueItem> m_front;	Получение ссылки на первый узел
std::shared_ptr<QueueItem> m_end;	Получение ссылки на последний элемент
friend std::ostream& operator << (std::ostream& os, const Queue& queue);	Переопределённый оператор вывода в поток std::ostream
queue_item.cpp	
QueueItem(const std::shared_ptr<Figure>& figure);	Конструктор класса
void setNext(std::shared_ptr<QueueItem> next);	Установка ссылки на следующий элемент
std::shared_ptr<QueueItem> getNext();	Получение ссылки на следующий элемент
std::shared_ptr<Figure> getFigure() const;	Получение фигуры

- queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

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

class Queue
{
public:
	Queue();
	~Queue();

	void push(const std::shared_ptr<Figure>& figure);
	void pop();
```

```
        unsigned int size() const;
        std::shared_ptr<Figure> front() const;

        friend std::ostream& operator << (std::ostream& os, const Queue& queue);

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

#endif
```

- queue_item.h

```
#ifndef QUEUE_ITEM_H
#define QUEUE_ITEM_H

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

class QueueItem
{
public:
        QueueItem(const std::shared_ptr<Figure>& figure);

        void setNext(std::shared_ptr<QueueItem> next);
        std::shared_ptr<QueueItem> getNext();
        std::shared_ptr<Figure> getFigure() const;

private:
        std::shared_ptr<Figure> m_figure;
        std::shared_ptr<QueueItem> m_next;
};

#endif
```

Консоль

```
dasha@belosnezzzka:~/OOP$ cd 3
dasha@belosnezzzka:~/OOP/3$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -o lab3 *.cpp
dasha@belosnezzzka:~/OOP/3$ ./lab3
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
1
================
Enter side: 2
================
```

```
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
2
================
Enter side A: 3
Enter side B:
4
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
3
================
Enter side A: 5
Enter side B: 6
Enter height:
7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: square
Side size: 2
================
Figure type: rectangle
Side A size: 3
Side B size: 4
================
Figure type: trapezoid
Side A size: 5
Side B size: 6
Height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
2
================
```

```
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: rectangle
Side A size: 3
Side B size: 4
================
Figure type: trapezoid
Side A size: 5
Side B size: 6
Height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
0
```

Выводы

Сделав данную лабораторную работу, я закрепила навыки работы с классами и познакомилась с умными указателями. Была создана программа, позволяющая вводить произвольное количество фигур и добавлять их в контейнер, распечатывать содержимое контейнера и удалять фигуры из контейнера. Опыт, полученный при создании этой лабораторной работы пригодится мне при создании других лабораторных работ и дальнейшем обучении объектно-ориентированному программированию.

Лабораторная работа №3

Цель работы:

- Закрепление навыков работы с классами.
- Знакомство с умными указателями.

Задача:

Необходимо спроектировать и запрограммировать на языке С++ класс-контейнер первого уровня, содержащий **все три фигуры,** согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

- Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
- Класс-контейнер должен соджержать объекты используя std:shared_ptr<…>.
- Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.
- Класс-контейнер должен иметь методы по получению фигуры из контейнера (опередеяется структурой контейнера).
- Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опередеяется структурой контейнера).
- Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).
- Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
- Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Вариант задания: 26.

Фигуры: квадрат, прямоугольник, трапеция.

Варианты структуры данных: контейнер первого уровня – очередь, контейнер второго уровня – связный список.

Описание

Умный указатель — это тот же обычный указатель, обеспечивающий безопасность благодаря автоматическому управлению памятью. Такой указатель помогает избежать множества проблем: «висячие» указатели, «утечки» памяти и отказы в выделении памяти. Интеллектуальный указатель должен подсчитывать количество ссылок на указанный объект.

- std::auto_ptr<>

  auto_ptr является умным указателем, реализующим семантику владения. Стоит отметить, что auto_ptr – это единственный умный указатель, включенный в нынешний стандарт C++

- std::shared_ptr<>

  shared_ptr является более гибкой и универсальной версией умного указателя. shared_ptr обладает во многом схожей с auto_ptr семантикой, основные отличия заключаются в методике копирования. Класс shared_ptr реализует разделяемое (shared) владение с подсчётом ссылок, что позволяет использовать более привычные конструкторы копирования и операторы присваивания. Благодаря такой семантике shared_ptr можно использовать в стандартных контейнерах STL. Кроме того, shared_ptr позволяет задавать функтор удаления (deleter), что может быть полезно для классов, имеющих необычную семантику удаления.

- std::weak_ptr<>

  weak_ptr – это «слабый», не владеющий экземпляром объекта умный указатель. weak_ptr ссылается на экземпляр, которым владеет shared_ptr, однако не разделяет прав владения этим экземпляром. Это гарантирует, что если экземпляр уже уничтожен, мы сможем это надёжно и безопасно проверить.

Исходный код

queue.cpp	
Queue();	Конструктор класса
~Queue();	Деструктор класса
void push(const std::shared_ptr<Figure>& figure);	Добавить в очередь
void pop();	Удалить из очереди
std::shared_ptr<Figure> front() const;	Получение фигуры из узла
std::shared_ptr<QueueItem> m_front;	Получение ссылки на первый узел
std::shared_ptr<QueueItem> m_end;	Получение ссылки на последний элемент
friend std::ostream& operator << (std::ostream& os, const Queue& queue);	Переопределённый оператор вывода в поток std::ostream
queue_item.cpp	
QueueItem(const std::shared_ptr<Figure>& figure);	Конструктор класса
void setNext(std::shared_ptr<QueueItem> next);	Установка ссылки на следующий элемент
std::shared_ptr<QueueItem> getNext();	Получение ссылки на следующий элемент
std::shared_ptr<Figure> getFigure() const;	Получение фигуры

- queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

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

class Queue
{
public:
	Queue();
	~Queue();
```

```
        void push(const std::shared_ptr<Figure>& figure);
        void pop();
        unsigned int size() const;
        std::shared_ptr<Figure> front() const;

        friend std::ostream& operator << (std::ostream& os, const Queue& queue);

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

#endif
```

- queue_item.h

```
#ifndef QUEUE_ITEM_H
#define QUEUE_ITEM_H

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

class QueueItem
{
public:
        QueueItem(const std::shared_ptr<Figure>& figure);

        void setNext(std::shared_ptr<QueueItem> next);
        std::shared_ptr<QueueItem> getNext();
        std::shared_ptr<Figure> getFigure() const;

private:
        std::shared_ptr<Figure> m_figure;
        std::shared_ptr<QueueItem> m_next;
};

#endif
```

Консоль

```
dasha@belosnezzzka:~/OOP$ cd 3
dasha@belosnezzzka:~/OOP/3$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -o lab3 *.cpp
dasha@belosnezzzka:~/OOP/3$ ./lab3
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
1
```

```
================
Enter side: 2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
2
================
Enter side A: 3
Enter side B:
4
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
3
================
Enter side A: 5
Enter side B: 6
Enter height:
7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: square
Side size: 2
================
Figure type: rectangle
Side A size: 3
Side B size: 4
================
Figure type: trapezoid
Side A size: 5
Side B size: 6
Height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
```

```
0) Quit
2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: rectangle
Side A size: 3
Side B size: 4
================
Figure type: trapezoid
Side A size: 5
Side B size: 6
Height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
0
```

Выводы

Сделав данную лабораторную работу, я закрепила навыки работы с классами и познакомилась с умными указателями. Была создана программа, позволяющая вводить произвольное количество фигур и добавлять их в контейнер, распечатывать содержимое контейнера и удалять фигуры из контейнера. Опыт, полученный при создании этой лабораторной работы пригодится мне при создании других лабораторных работ и дальнейшем обучении объектно-ориентированному программированию.

Лабораторная работа №4

Цель работы:

- Знакомство с шаблонами классов.
- Построение шаблонов динамических структур данных.

Задача:

Необходимо спроектировать и запрограммировать на языке С++ класс-контейнер первого уровня, содержащий **одну фигуру,** согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

Требования к классам фигуры аналогичны требованиям из лабораторной работы 1.

- Шаблон класса-контейнера должен соджержать объекты используя std:shared_ptr<…>.
- Шаблон класса-контейнера должен иметь метод по добавлению фигуры в контейнер.
- Шаблон класса-контейнера должен иметь методы по получению фигуры из контейнера (опередеяется структурой контейнера).
- Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (опередеяется структурой контейнера).
- Шаблон класса-контейнера должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).
- Шаблон класса-контейнера должен иметь деструктор, удаляющий все элементы контейнера.
- Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Вариант задания: 26.

Фигуры: квадрат, прямоугольник, трапеция.

Варианты структуры данных: контейнер первого уровня – очередь, контейнер второго уровня – связный список.

Описание

Шаблоны (template) — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).
В C++ возможно создание шаблонов функций и классов.
Шаблоны позволяют создавать параметризованные классы и функции. Параметром может быть любой тип или значение одного из допустимых типов (целое число, enum, указатель на любой объект с глобально доступным именем, ссылка).
Шаблоны объявляются и определяются подобно другим функциям и классам с некоторыми основными отличиями. Объявление шаблона не полностью определяет функцию или класс; определяется только синтаксическая структура класса или функции. Фактический класс или функция создаются с помощью шаблона в ходе процесса, называемого созданием экземпляра. Отдельные создаваемые классы или функции называются экземплярами.

Исходный код

queue.cpp	
Queue();	Конструктор класса
~Queue();	Деструктор класса
void push(const Square& square);	Добавить в очередь
void pop();	Удалить из очереди
friend std::ostream& operator << (std::ostream& os, const Queue& queue);	Переопределённый оператор вывода в поток std::ostream
queue_item.cpp	
QueueItem(const Square& square);	Конструктор класса
void setNext(QueueItem* next);	Установка ссылки на следующий элемент
QueueItem* getNext();	Получение ссылки на следующий элемент
Square getSquare() const;	Получение площади

- queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

#include <iostream>
#include "queue_item.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;

	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;
};


#endif
```

- queue_item.h

```
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;
};


#endif
```

Консоль

```
dasha@belosnezzzka:~/OOP$ cd 4
dasha@belosnezzzka:~/OOP/4$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -o lab4 main.cpp square.cpp rectangle.cpp
trapezoid.cpp
dasha@belosnezzzka:~/OOP/4$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -o lab4 main.cpp square.cpp rectangle.cpp
trapezoid.cpp
dasha@belosnezzzka:~/OOP/4$ ./lab4
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
1
================
Enter side: 2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
2
================
Enter side A: 3
Enter side B: 4
```

```
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
3
================
Enter side A: 5
Enter side B: 6
Enter height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: rectangle
Side A size: 3
Side B size: 4
================
Figure type: trapezoid
Side A size: 5
Side B size: 6
Height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
0
```

Выводы

Сделав данную лабораторную работу, я познакомилась с шаблонами классов и построением динамических структур. Была создана программа, позволяющая вводить произвольное количество фигур и добавлять их в контейнер, распечатывать содержимое контейнера и удалять фигуры из контейнера. Опыт, полученный при создании этой лабораторной работы пригодится мне при создании других лабораторных работ и дальнейшем обучении объектно-ориентированному программированию.

Лабораторная работа №5

Цель работы:

- Закрепление навыков работы с шаблонами классов.
- Построение итераторов для динамических структур данных.

Задача:
Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛР№4) спроектировать и разработать Итератор для динамической структуры данных.
Итератор должен быть разработан в виде шаблона и должен уметь работать со всеми типами фигур, согласно варианту задания.

Вариант задания: 26.
Фигуры: квадрат, прямоугольник, трапеция.
Варианты структуры данных: контейнер первого уровня – очередь, контейнер второго уровня – связный список.

Описание

Итератор (от англ. iterator — перечислитель) — интерфейс, предоставляющий доступ к элементам коллекции (массива или контейнера) и навигацию по ним. В различных системах итераторы могут иметь разные общепринятые названия. В терминах систем управления базами данных итераторы называются курсорами. В простейшем случае итератором в низкоуровневых языках является указатель.
Использование итераторов в обобщённом программировании позволяет реализовать универсальные алгоритмы работы с контейнерами.
В процедурных языках программирования широко используется индексация, основанная на счётчике цикла, для перебора всех элементов последовательности (например, массива). Хотя индексация может использоваться совместно с некоторыми объектно-ориентированными контейнерами, использование итераторов даёт свои преимущества:

- Индексация не подходит для некоторых структур данных, в частности, для структур данных с медленным произвольным доступом или вообще без поддержки такового (например, список или дерево).
- Итераторы предоставляют возможность последовательного перебора любых структур данных, поэтому делают код более читаемым, удобным для повторного использования и менее чувствительным к изменениям структур данных.
- Итераторы могут предоставлять дополнительные возможности при навигации по элементам. Например, проверку отсутствия пропусков элементов или защиту от повторного перебора одного и того же элемента.
- Некоторые контейнеры могут предоставлять возможность модифицировать свои объекты без влияния на сам итератор. Например, после того, как итератор уже «прошёл» первый элемент, можно вставить дополнительные элементы в начало контейнера без каких-либо нежелательных последствий. При использовании индексации это проблематично из-за смены номеров индексов.

Возможность модификации контейнера во время итерации его элементов стала необходимой в современном объектно-ориентированном программировании, где взаимосвязи между объектами и последствия выполнения операций могут быть не слишком очевидными. Использование итератора избавляет от этих видов проблем.

Исходный код

queue.cpp	
Queue();	Конструктор класса
~Queue();	Деструктор класса
void push(const std::shared_ptr<Figure>& figure);	Добавить в очередь
void pop();	Удалить из очереди
std::shared_ptr<Figure> front() const;	Получение фигуры из узла
std::shared_ptr<QueueItem> m_front;	Получение ссылки на первый узел
std::shared_ptr<QueueItem> m_end;	Получение ссылки на последний элемент
friend std::ostream& operator << (std::ostream& os, const Queue& queue);	Переопределённый оператор вывода в поток std::ostream
queue_item.cpp	
QueueItem(const std::shared_ptr<Figure>& figure);	Конструктор класса
void setNext(std::shared_ptr<QueueItem> next);	Установка ссылки на следующий элемент
std::shared_ptr<QueueItem> getNext();	Получение ссылки на следующий элемент
std::shared_ptr<Figure> getFigure() const;	Получение фигуры

- 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<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;
};


#endif
```

Консоль

```
dasha@belosnezzzka:~/OOP$ cd 5
dasha@belosnezzzka:~/OOP/5$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -o lab5 main.cpp square.cpp rectangle.cpp
trapezoid.cpp
dasha@belosnezzzka:~/OOP/5$ ./lab5
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
1
================
Enter side: 2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
2
================
Enter side A: 3
Enter side B: 4
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
3
```

```
================
Enter side A: 5
Enter side B: 6
Enter height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: square
Side size: 2
================
Figure type: rectangle
Side A size: 3
Side B size: 4
================
Figure type: trapezoid
Side A size: 5
Side B size: 6
Height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: trapezoid
Side A size: 5
Side B size: 6
Height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
0
```

Выводы

Сделав данную лабораторную работу, я закрепила навыки работы с шаблонами классов и построила итератор для динамической структуры данных. Была создана программа, позволяющая вводить произвольное количество фигур и добавлять их в контейнер, распечатывать содержимое контейнера и удалять фигуры из контейнера. Опыт, полученный при создании этой лабораторной работы пригодится мне при создании других лабораторных работ и дальнейшем обучении объектно-ориентированному программированию.

Лабораторная работа №6

Цель работы:

- Закрепление навыков по работе с памятью в C++.
- Создание аллокаторов памяти для динамических структур данных.

Задача:

Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛР№5) спроектировать и разработать аллокатор памяти для динамической структуры данных.
Цель построения аллокатора – минимизация вызова операции malloc. Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти.
Алокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контейнер 2-го уровня, согласно варианта задания).
Для вызова аллокатора должны быть переопределены оператор **new** и **delete** у классов-фигур.
Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛР№4) спроектировать и разработать Итератор для динамической структуры данных.
Итератор должен быть разработан в виде шаблона и должен уметь работать со всеми типами фигур, согласно варианту задания.

Вариант задания: 26.
Фигуры: квадрат, прямоугольник, трапеция.
Варианты структуры данных: контейнер первого уровня – очередь, контейнер второго уровня – связный список.

Описание

Аллокатор умеет выделять и освобождать память в требуемых количествах определённым образом. std::allocator -- пример реализации аллокатора из стандартной библиотеки, просто использует new и delete, которые обычно обращаются к системным вызовам malloc и free. Более сложный пример -- pool allocator. Реальное выделение памяти только одно, системных вызовов почти нет, программа ускорилась.
Собственно, для этого и нужны аллокаторы, чтобы вставлять свое нестандартное, выделение памяти в любое место. Большинство стандартных контейнеров их принимают.

Исходный код

queue.cpp	
Allocator(unsigned int blockSize, unsigned int count);	Конструктор класса
~Allocator();	Деструктор класса
void* allocate();	Выделение памяти
void deallocate(void* p);	Освобождение памяти
bool hasFreeBlocks() const;	Проверка аллокатора на пустоту

- 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
```

Консоль

```
dasha@belosnezzzka:~/OOP$ cd 6
dasha@belosnezzzka:~/OOP/6$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -o lab6 main.cpp square.cpp rectangle.cpp
trapezoid.cpp figure.cpp allocator.cpp
dasha@belosnezzzka:~/OOP/6$ ./lab6
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
```

```
3) Trapezoid
0) Quit
1
================
Enter side: 2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
2
================
Enter side A: 3
Enter side B: 4
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
3
================
Enter side A: 5
Enter side B: 6
Enter height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: trapezoid
```

```
Side A size: 5
Side B size: 6
Height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
0
```

Выводы

Сделав данную лабораторную работу, я закрепила навыки по работе с памятью в C++ и создала аллокатор памяти динамической структуры данных. Была создана программа, позволяющая вводить произвольное количество фигур и добавлять их в контейнер, распечатывать содержимое контейнера и удалять фигуры из контейнера. Опыт, полученный при создании этой лабораторной работы пригодится мне при создании других лабораторных работ и дальнейшем обучении объектно-ориентированному программированию.

Лабораторная работа №7

Цель работы:

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

Задача:

Необходимо реализовать динамическую структуру данных – «Хранилище объектов» и алгоритм работы с ней. «Хранилище объектов» представляет собой контейнер, одного из видов (Контейнер 1-го уровня). Каждый элемент контейнера, в свою очередь, является динамической структурой данных одного из видов (Контейнер 2-го уровня).

Таким образом у нас получается контейнер в контейнере. Элементом второго контейнера является объект-фигура, определенная вариантом задания.

Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию площади 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.

Вариант задания: 26.

Фигуры: квадрат, прямоугольник, трапеция.

Варианты структуры данных: контейнер первого уровня – очередь, контейнер второго уровня – связный список.

Описание

Принцип открытости/закрытости (OCP) — принцип ООП, устанавливающий следующее положение: «программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения».
Принцип открытости/закрытости означает, что программные сущности должны быть:
открыты для расширения: означает, что поведение сущности может быть расширено путём создания новых типов сущностей.
закрыты для изменения: в результате расширения поведения сущности, не должны вноситься изменения в код, который эти сущности использует.
Это особенно значимо в производственной среде, когда изменения в исходном коде потребуют проведение пересмотра кода, модульного тестирования и других подобных процедур, чтобы получить право на использования его в программном продукте. Код, подчиняющийся данному принципу, не изменяется при расширении и поэтому не требует таких трудозатрат.

Исходный код

container.cpp	
void erase(const Criteria<T>& criteria);	Удаление по критерию площади
void add(const std::shared_ptr<T>& item);	Добавление по критерию площади
criteria_area.cpp	
bool check(const std::shared_ptr<T>& item) const override;	Критерий проверки площади

- 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;
};

#endif
```

- 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;
};
#endif
```

Консоль

```
dasha@belosnezzzka:~/OOP$ cd 7
dasha@belosnezzzka:~/OOP/7$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -o lab7 main.cpp square.cpp rectangle.cpp
trapezoid.cpp figure.cpp allocator.cpp
dasha@belosnezzzka:~/OOP/7$ ./lab7
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
1
================
Enter side: 2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
2
================
Enter side A: 3
Enter side B: 4
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Container #1:
================
Item #1:
================
Figure type: rectangle
Side A size: 3
Side B size: 4
Area: 12
```

```
================
Item #2:
================
Figure type: square
Side size: 2
Area: 4
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
0
```

Выводы

Сделав данную лабораторную работу, я создала очень сложную динамическую структуру и закрепила принцип OCP. Была создана программа, позволяющая вводить произвольное количество фигур и добавлять их в контейнер, распечатывать содержимое контейнеров 1 и 2 уровней и удалять фигуры из контейнера по критериям (все квадраты/площадь меньше, чем задана). Опыт, полученный при создании этой лабораторной работы пригодится мне при создании других лабораторных работ и дальнейшем обучении объектно-ориентированному программированию.

Лабораторная работа №8

Цель работы:

- Знакомство с параллельным программированием.

Задача:

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер первого уровня и классы-фигуры) разработать алгоритм быстрой сортировки для класса-контейнера.

Необходимо разработать два вида алгоритма:

- Обычный, без параллельных вызовов.
- С использованием параллельных вызовов. В этом случае, каждый рекурсивный вызов сортировки должен создаваться в отдельном потоке.

Для создания потоков использовать механизмы:

- future
- packaged_task/async

Для обеспечения потоко-безопасности структур данных использовать:

- mutex
- lock_guard

Вариант задания: 26.

Фигуры: квадрат, прямоугольник, трапеция.

Варианты структуры данных: контейнер первого уровня – очередь, контейнер второго уровня – связный список.

Описание

Первоначально **основы параллельного программирования** закладывались в архитектуре вычислительных устройств. Была предложена классификация на основе понятия потока. Последовательность команд, данных, функционально полных последовательных алгоритмов рассматривалась как объект, который можно исполнить параллельно с другим подобным объектом. Разработчики больше ориентировались на аппаратные механизмы, что лишало параллельное многопоточное программирование возможности иметь семантику и не давало возможности программисту управлять процессами адекватно решаемой задаче.

В самом начале системы параллельного программирования также не придавали значения алгоритму, который должен будет исполняться. Процессор сам разделял код и данные на участки, которые исполнял параллельно. Это давало заметный прирост производительности, но волновала, в частности:

- проблема разделения памяти между процессами;
- логика ожидания одним потоком результатов работы другого потока;
- механизм защиты памяти одного процесса от другого процесса;
- логика взаимодействия независимых процессоров, ядер;
- логика переключения между процессами;
- обмен данными «на лету» между процессами

Исходный код

queue.cpp	
void sort();	Сортировка без параллельных вызовов
void sortParallel();	Сортировка с параллельными вызовами

- queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

#include <iostream>
#include <thread>
#include <future>
#include <functional>
#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;

	void sort();
	void sortParallel();

	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;

        void sortHelper(Queue<T>& q, bool isParallel);
        std::future<void> sortParallelHelper(Queue<T>& q);
};

#include "queue_impl.cpp"

#endif
```

Консоль

```
dasha@belosnezzzka:~/OOP$ cd 8
dasha@belosnezzzka:~/OOP/8$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -pthread -o lab8 main.cpp square.cpp rectangle.cpp
trapezoid.cpp figure.cpp allocator.cpp
dasha@belosnezzzka:~/OOP/8$ ./lab8
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
2
================
Enter side A: 3
Enter side B: 4
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
1
================
Enter side: 2
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
1
================
```

```
1) Square
2) Rectangle
3) Trapezoid
0) Quit
5
Error: invalid figure type
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
3
================
Enter side A: 5
Enter side B: 6
Enter height: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
3
================
Figure type: rectangle
Side A size: 3
Side B size: 4
Area: 12
================
Figure type: square
Side size: 2
Area: 4
================
Figure type: trapezoid
Side A size: 5
Side B size: 6
Height: 7
Area: 38.5
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
4
================
1) Single thread
2) Multithread
0) Quit
1
================
```

```
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
3
================
Figure type: square
Side size: 2
Area: 4
================
Figure type: rectangle
Side A size: 3
Side B size: 4
Area: 12
================
Figure type: trapezoid
Side A size: 5
Side B size: 6
Height: 7
Area: 38.5
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
1
================
1) Square
2) Rectangle
3) Trapezoid
0) Quit
1
================
Enter side: 7
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
4
================
1) Single thread
2) Multithread
0) Quit
2
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
3
================
Figure type: square
```

```
Side size: 2
Area: 4
================
Figure type: rectangle
Side A size: 3
Side B size: 4
Area: 12
================
Figure type: trapezoid
Side A size: 5
Side B size: 6
Height: 7
Area: 38.5
================
Figure type: square
Side size: 7
Area: 49
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
2
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
0
```

Выводы

Сделав данную лабораторную работу, я познакомилась с параллельным программированием. Была создана программа, позволяющая вводить произвольное количество фигур и добавлять их в контейнер, распечатывать содержимое контейнера, проводить сортировку контейнера и удалять фигуры из контейнера. Опыт, полученный при создании этой лабораторной работы пригодится мне при создании других лабораторных работ и дальнейшем обучении объектно-ориентированному программированию.

Лабораторная работа №9

Цель работы:

- Знакомство с лямбда-выражениями.

Задача:

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
- контейнером 1-го уровня:
    - Генерация фигур со случайным значением параметров;
    - Печать контейнера на экран;
    - Удаление элементов со значением площади меньше определенного числа;
- В контейнер второго уровня поместить цепочку команд.
- Реализовать цикл, который проходит по всем командам в контейнере 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.

Вариант задания: 26.

Фигуры: квадрат, прямоугольник, трапеция.

Варианты структуры данных: контейнер первого уровня – очередь, контейнер второго уровня – связный список.

Описание

Лямбда-выражение в программировании — специальный синтаксис для определения функциональных объектов, заимствованный из λ-исчисления. Применяется как правило для объявления анонимных функций по месту их использования, и обычно допускает замыкание на лексический контекст, в котором это выражение использовано.
Используя лямбда-выражения, можно объявлять функции в любом месте кода.
Возвращаемый тип лямбда-выражения выводится автоматически. Использовать ключевое слово auto не нужно, если не указывается завершающий возвращающий тип. trailing-return-type похож на часть стандартного метода или функции, содержащую возвращаемый тип. Однако тип возвращаемого значения следует списку параметров, и необходимо включить ключевое слово -> элемента trailing-return-type перед типом возвращаемого значения.
Можно опустить часть возвращаемого типа лямбда-выражения, если тело лямбда-выражения содержит только один оператор return или лямбда-выражение не возвращает значение. Если тело лямбда-выражения содержит один оператор return, компилятор выводит тип возвращаемого значения из типа возвращаемого выражения. В противном случае компилятор выводит следующий тип возвращаемого значения: void.

Исходный код

- main.cpp

```
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;
}
```

Консоль

```
dasha@belosnezzzka:~/OOP$ cd 9
dasha@belosnezzzka:~/OOP/9$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -pthread -o lab9 main.cpp square.cpp rectangle.cpp
trapezoid.cpp figure.cpp allocator.cpp
dasha@belosnezzzka:~/OOP/9$ ./lab9
================
Menu:
1) Add command
2) Erase command
3) Execute commands
4) Print commands
0) Quit
1
================
1) Insert
2) Erase
3) Print
0) Quit
1
================
Menu:
1) Add command
2) Erase command
3) Execute commands
4) Print commands
0) Quit
1
================
1) Insert
2) Erase
```

```
3) Print
0) Quit
2
================
Menu:
1) Add command
2) Erase command
3) Execute commands
4) Print commands
0) Quit
1
================
1) Insert
2) Erase
3) Print
0) Quit
3
================
Menu:
1) Add command
2) Erase command
3) Execute commands
4) Print commands
0) Quit
1
================
1) Insert
2) Erase
3) Print
0) Quit
1
================
Menu:
1) Add command
2) Erase command
3) Execute commands
4) Print commands
0) Quit
4
================
Insert
Erase
Print
Insert
================
Menu:
1) Add command
2) Erase command
3) Execute commands
4) Print commands
0) Quit
3
================
Command: insert
================
Inserted: trapezoid
================
Command: erase
================
Command: print
================
```

```
Figure type: trapezoid
Side A size: 9
Side B size: 5
Height: 6
Area: 42
================
Command: insert
================
Inserted: rectangle
================
Menu:
1) Add command
2) Erase command
3) Execute commands
4) Print commands
0) Quit
2
================
Command index: 3
================
Menu:
1) Add command
2) Erase command
3) Execute commands
4) Print commands
0) Quit
3
================
Command: insert
================
Inserted: square
================
Command: print
================
Figure type: trapezoid
Side A size: 9
Side B size: 5
Height: 6
Area: 42
================
Figure type: rectangle
Side A size: 8
Side B size: 4
Area: 32
================
Figure type: square
Side size: 10
Area: 100
================
Command: erase
================
Menu:
1) Add command
2) Erase command
3) Execute commands
4) Print commands
0) Quit
0
```

Выводы

Сделав данную лабораторную работу, я познакомилась с лямбда-выражениями. Была создана программа, позволяющая вводить произвольное количество фигур и добавлять их в контейнер, генерировать фигуры со случайным значением параметров, печатать контейнер на экран и удалять фигуры с площадью, меньше заданной, из контейнера. Опыт, полученный при создании этой лабораторной работы пригодится мне при создании других лабораторных работ и дальнейшем обучении объектно-ориентированному программированию.

Общие выводы

Закончился курс объектно-ориентированного программирования на C++. Ещё раз повторю определение ООП: «это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования». Должна сказать, пожалуй, самым сложным было научиться использовать объектное мышление. Но, когда получается мыслить объектно, легко решаются сложные задачи, потому что используются объекты и взаимодействие между ними.
Так и в нашем курсе, каждая лабораторная представляла собой определённое задание, каждый раз нового и связанного с предыдущим. Всего в курсе мной было сделано девять лабораторных работ.

1. Первая лабораторная работа
   Я познакомилась с ООП, узнала, что такое полиморфизм, наследование и инкапсуляция. По заданию лабораторной работы были спроектированы классы фигур (заданные вариантом), использовались перегруженные операторы, дружественные функции и операции ввода и вывода из стандартных библиотек.

2. Вторая лабораторная работа
   Темой второй лабораторной работы была передача объектов в методы контейнера по значению, была спроектирована структура данных – очередь.

3. Третья лабораторная работа
   В этой лабораторной работе было необходимо переписать контейнер с использованием умных указателей, что и было сделано.

4. Четвертая лабораторная работа
   Первое знакомство с шаблонами классов, соответственно был построен шаблон динамической структуры данных.

5. Пятая лабораторная работа
   Применение итераторов для облегчения перемещения по очереди.

6. Шестая лабораторная работа
   Цель данной лабораторной работы – применение аллокатора для оптимизации выделения и освобождения памяти. Свободные блоки хранятся в аллокаторе в контейнере второго уровня – связном списке.

7. Седьмая лабораторная работа
   Мне она показалась самой сложной из всех. Необходимо было создать контейнер в контейнере, причём во втором контейнере хранились отсортированные фигуры, а при отсутствии фигур во втором контейнере, он удалялся из первого.

8. Восьмая лабораторная работа
   Применение параллельного программирования для реализации алгоритма сортировки.

9. Девятая лабораторная работа
   Использование лямбда-выражений для генерации действий над первым контейнером. Очень интересная тема!

Подведём итог, на данном курсе были получены практические навыки программирования, использовано много разных возможностей языка C++, кроме того был изучен крупнейший веб-сервис для хостинга IT-проектов и их совместной разработки, Github. Все приобретённые навыки помогут мне заниматься программированием на других курсах и использовать полученные знания в жизни.

Ссылка на репозиторий на Github

https://github.com/FutureTopProger/OOP