Московский авиационный институт

(национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программировании

Студент: Синдюков В.Р.

Преподаватель: Поповкин А. В.

Группа: 08-204Б

Вариант: 13

Дата:

Оценка:

Подпись:

Лабораторная работа №1

Задача: Необходимо спроектировать и запрограммировать на языке C++ классы фигур, согласно вариантов задании.

Классы должны удовлетворять следующим правилам:

• Должны иметь общий родительский класс Figure.

• Должны иметь общий виртуальный метод Print, печатающий параметры фи-

гуры и ее тип в стандартный поток вывода cout.

• Должный иметь общий виртуальный метод расчета площади фигуры – Square.

• Должны иметь конструктор, считывающий значениосновных параметров фи-

гуры из стандартного потока cin.

• Должны быть расположенны в раздельных файлах: отдельно заголовки (.h),

отдельно описание методов (.cpp)

Фигура: шестиугольник, ромб, пятиугольник

Описание

Абстракция данных — Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой. В ООП рассматривают лишь абстракцию данных

(нередко называя её просто «абстракцией»), подразумевая набор значимых характеристик объекта, доступный остальной программе.

Абстракция данных — Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой. В ООП рассматривают лишь абстракцию данных (нередко называя её просто «абстракцией»), подразумевая набор значимых характеристик объекта, доступный остальной программе.

Наследование — свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом.

Полиморфизм подтипов (в ООП называемый просто «полиморфизмом») — свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. Другой вид полиморфизма — параметрический — в ООП называют обобщённым программированием.

Класс — универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций длработы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей). В частности, в классах широко используются специальные блоки из одного или чаще двух спаренных методов, отвечающих за элементарные операции с определенным полем, которые имитируют непосредственный доступ к полю. Эти блоки называются «свойствами» и почти совпадают по конкретному имени со своим полем. Другим проявлением интерфейсной природы класса является то, что при копировании соответствующей переменной через присваивание, копируется только интерфейс, но не сами данные, то есть класс — ссылочный тип данных. Переменная-объект, относящаяся к заданному классом типу, называется экземпляром этого класса. При этом в некоторых исполняющих системах класс также может представляться некоторым объектом при выполнении программы посредством динамической идентификации типа данных. Обычно классы разрабатывают таким образом, чтобы обеспечить отвечающие природе объекта и решаемой задаче целостность данных объекта, а также удобный и простой интерфейс. В свою очередь, целостность предметной области объектов и их интерфейсов, а также удобство их проектирования, обеспечивается наследованием.

Объект — сущность в адресном пространстве вычислительной системы, позволяющая при создании экземпляра класса (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

Сходный код

```
class Rhomb : public Figure {
public:
	Rhomb();
	Rhomb(std::istream &is);
	Rhomb(size_t i, size_t j);
	Rhomb(const Rhomb& orig);

	double Square() override;
	void Print() override;
```

```
	virtual ~Rhomb();
private:
	size_t a;
	size_t h;

};
class Pentagon : public Figure {
public:
	Pentagon();
	Pentagon(std::istream &is);
	Pentagon(size_t i, size_t j, size_t k, size_t l, size_t m, size_t n);
	Pentagon(const Pentagon& orig);

	double Square() override;
	void Print() override;
	virtual ~Pentagon();

private:
	size_t side1;
	size_t side2;
	size_t side3;
	size_t side4;
	size_t side5;
	size_t rad;
};
class Hexagon : public Figure {
public:
	Hexagon();
	Hexagon(std::istream &is);
	Hexagon(size_t i, size_t j);
	Hexagon(const Hexagon& orig);

	Hexagon& operator++();
	friend bool operator==(const Hexagon& left, const Hexagon& right);
	friend Hexagon operator+(const Hexagon& left, const Hexagon& right);
	friend std::ostream& operator<<(std::ostream& os, const Hexagon& obj);
	friend std::istream& operator >> (std::istream& is, Hexagon& obj);

	double Square() override;
	void Print() override;
	virtual ~Hexagon();
	int32_t Side();
	Hexagon& operator=(const Hexagon& right);

private:
	size_t side;
	size_t rad;
};
```

Консоль

```
Menu:
1) Rhomb
2) Pentagon
3) Hexagon
4) Exit
1
4 5
side: 4
height: 5
Square = 20
```

```
Rhomb deleted
Menu:
1) Rhomb
2) Pentagon
3) Hexagon
4) Exit
2
7 8 9 6 5
4
side 1: 7
side 2: 8
side 3: 9
side 4: 6
side 5: 5
rad: 4
Square = 70
Pentagon deleted
Menu:
1) Rhomb
2) Pentagon
3) Hexagon
4) Exit
3
7 5 3 6 5 4
7
side 1: 7
side 2: 5
side 3: 3
side 4: 6
side 5: 5
side 6: 4
radius: 7
Square = 105
Hexagon deleted
Menu:
1) Rhomb
2) Pentagon
3) Hexagon
4) Exit
```

Выводы

В этой лабораторной были реализованы классы фигур. В первой лабораторной работе я познакомился с синтаксисом C++, хоть он похож на C, но всё же отличается наличием ООП. Также я познакомился с различными понятиями, такими, как класс, объект, инкапсуляция, абстрактный тип данных, наследование, полиморфизм.

Лабораторная работа №2

Задача: Необходимо спроектировать и запрограммировать на языке C++ класс- контейнер первого уровня, содержащий одну фигуру, согласно варианту задания 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.

• Классы фигур должны иметь переопределенный оператор ввода фигуры из потока std::istream(»). Оператор должен вводить параметры фигуры.

• Классы фигур должны иметь операторы копирования (=).

• Классы фигур должны иметь операторы сравнения с такими же фигурами (==).

• Класс-контейнер должен содержать 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.

• Класс-контейнер должен иметь метод по удалению фигуры из 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 расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Фигура: шестиугольник.

Контейнер: бинарное дерево.

Описание

Динамические структуры данных используются в тех случаях, когда мы заранее не знаем, сколько памяти необходимо выделить для нашей программы – это выясняется только в процессе работы. В общем случае эта структура представляет собой отдельные элементы, связанные между собой с помощью ссылок. Каждый элемент состоит из двух областей памяти: полданных и ссылок. Ссылки – это адреса других узлов того же типа, с которыми данный элемент логически связан. При добавлении ново- го элемента в такую структуру выделяется новый блок памяти и устанавливаются связи этого элемента с уже существующими.

Структура данных список является простейшим типом данных динамической структуры, состоящей из узлов. Каждый узел включает в себя в классическом варианте два поля: данные и указатель на следующий узел в списке. Элементы связного списка можно вставлять и удалять произвольным образом. Доступ к списку осуществляется через указатель, который содержит адрес первого элемента списка, называемого головой списка.

Параметры в функцию могут передаваться одним из следующих способов: по значению и по ссылке. При передаче аргументов по значению компилятор создает временную копию объекта, который должен быть передан, и размещает его 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 воз- действуют на объект. Использование передачи адреса объекта весьма эффективный способ работы с большим числом данных. Кроме того, так как передается адрес, а не сам объект, существенно экономится стековая память.

Исходный код

```
class Tree;
class TreeItem
{
public:
	TreeItem();
	TreeItem(Rhomb& rhomb);

	int32_t Side();
	Rhomb GetRhomb();
	~TreeItem();
	friend Tree;
private:
	Rhomb rhomb;
	TreeItem* left;
	TreeItem* right;
};
class Rhomb : public Figure
{
public:
	Rhomb();
	Rhomb(size_t i, size_t j);
	Rhomb(const Rhomb& orig);
	Rhomb& operator++();
	friend bool operator==(const Rhomb& left, const Rhomb& right);
	friend Rhomb operator+(const Rhomb& left, const Rhomb& right);
	friend std::ostream& operator<<(std::ostream& os, const Rhomb& obj);
	friend std::istream& operator >> (std::istream& is, Rhomb& obj);

	Rhomb(std::istream &is);
	double Square() override;
	int32_t Side();
	void Print() override;
	Rhomb& operator=(const Rhomb& right);
	virtual ~Rhomb();
private:
	size_t side;
	size_t hight; };
```

Консоль

```
in
77 8
Rhomb created: 77 8
Rhomb created: 0 0
Rhomb copied
in
4 5
Rhomb created: 4 5
Rhomb created: 0 0
Rhomb copied
```

```
in
11 23
Rhomb created: 11 23
Rhomb created: 0 0
Rhomb copied
p
77
  4
    null
    11
  null

destroy
Destroied
p
Empty
in
7 4
Rhomb created: 7 4
Rhomb created: 0 0
Rhomb copied
in
5 6
Rhomb created: 5 6
Rhomb created: 0 0
Rhomb copied
in
9 3
Rhomb created: 9 3
Rhomb created: 0 0
Rhomb copied
f
5
p
7
  5
  9
in
4 5
Rhomb created: 4 5
Rhomb created: 0 0
Rhomb copied
in
9 6
Rhomb created: 9 6
Rhomb created: 0 0
Rhomb copied
in
3 7
Rhomb created: 3 7
```

```
Rhomb created: 0 0
Rhomb copied
in
55 6
Rhomb created: 55 6
Rhomb created: 0 0
Rhomb copied
rem
9
6
p
4
  3
  55
```

Вывод

В этой лабораторной было предложено написать свою структуру данных. В моем случае это бинарное дерево. Я сделал его бинарным деревом поиска. Конечно, все эти контейнеры есть в стандартной библиотеке шаблонов, но чтобы стать хорошим программистом, нужно уметь реализовывать свой список, стек, очередь или любую другую широко используемую структуру данных. Фигуры передаются в бинарное дерево по значению, чтобы в них хранился сам объект, а не его копи.

Лабораторная работа №3

Умный указатель – класс (обычно шаблонный), имитирующий интерфейс обычного указателя и добавляющий некую новую функциональность, например, проверку границ при доступе или очистку памяти.

Существует 3 вида умных указателей стандартной библиотеки C++:

• unique ptr – обеспечивает, чтобы у базового указателя был только один владелец. Может быть передан новому владельцу, но не может быть скопирован или
сделан общим. Заменяет auto ptr, использовать который не рекомендуется.

• shared ptr – умный указатель с подсчитанными ссылками. Используется, когда

необходимо присвоить один необработанный указатель нескольким владельцам, например, когда копии указателя возвращается из контейнера, но требуется сохранить оригинал. Необработанный указатель не будет удален до тех пор, пока все владельцы shared ptr не выйдут из области или не откажутся от владения.

• weak ptr – умный указатель дл особых случаев использования с shared ptr. weak ptr предоставляет доступ к объекту, который принадлежит одному или нескольким экземплярам shared ptr, но не участвует в подсчете ссылок. Используется, когда требуется отслеживать объект, но не требуется, чтобы он оставался в активном состоянии.

Исходный код

```
template <class T> class Tree;

template <class T> class TreeItem
{
public:
	TreeItem();
	TreeItem(const std::shared_ptr<T> &obj);

	std::shared_ptr<T> GetFigure();
	~TreeItem();
	friend class Tree<T>;
private:
	std::shared_ptr<T> item;
	std::shared_ptr<TreeItem<T>> left;
	std::shared_ptr<TreeItem<T>> right;

};
template <class T> class Tree
{
public:
	Tree();
	std::shared_ptr<TreeItem<T>> find(std::shared_ptr<T> &obj);
	void remove(int32_t side);
	void insert(std::shared_ptr<T> &obj);
	void print();
	void print(std::ostream& os);
	template <class A> friend std::ostream& operator<<(std::ostream& os, Tree<A>
&tree);
	bool empty();
	virtual ~Tree();
private:
	std::shared_ptr<TreeItem<T>> head;
	std::shared_ptr<TreeItem<T>> minValueNode(std::shared_ptr<TreeItem<T>> root);
	std::shared_ptr<TreeItem<T>> deleteNode(std::shared_ptr<TreeItem<T>> root,
int32_t side);
	void print_tree(std::shared_ptr<TreeItem<T>> item, int32_t a, std::ostream& os);
};
```

Консоль

```
List of operations:
1) Insert rhomb
2) Insert hexagon
3) Insert pentagon
4) Remove figure
5) Print
```

```
0) Exit
1
2 7
2
6
3
9


5
2
7
14
  null
  6
3

1
88 4
5
2
7
14
  null
  6
3
    null
    88
4
352

4
88
5
2
7
14
  null
  6
3

4
6
Pentagon deleted
5
2
7
14
```

Выводы

В этой лабораторной работе необходимо было реализовать умные указатели. В данном случае sharedptr, который ведет подсчет ссылок на объект и если счет равен 0, то указатель на объект удаляется. Так же в данной лабораторной работе был изучен makeshared, который возвращает sharedptr на объект с числом 1.

Лабораторная работа №4

Задача:

Необходимо спроектировать и запрограммировать на языке C++ шаблон класса- контейнера первого уровня, содержащий все три фигуры класса фигуры, 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 содержать объекты используя std:sharedptr<. . . >.
- Шаблон класса-контейнера должен иметь метод по добавлению фигуры в контейнер.
- Шаблон класса-контейнера должен иметь методы по получению фигуры из контейнера (опередеряется структурой контейнера).
- Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (опередеряется 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 расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Фигуры: треугольник, шестиугольник, восьмиугольник.

Контейнер: бинарное дерево.

Описание

Шаблоны (template) предназначены для кодирования обобщенных 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 типа или значение одного из допустимых типов (целое число, перечисляемый тип, указатель на любой объект с глобально доступным именем, ссылка). Шаблоны используются в случаях дублирования одного и того же кода для нескольких типов. Например, можно использовать шаблоны функций для создания набора функций, которые применяют один и тот же алгоритм к различным типам данных. Кроме того, шаблоны классов можно использовать для разработки набора типобезопасных классов. Иногда рекомендуется использовать шаблоны вместо макросов C и пустых указателей. Шаблоны особенно полезны при работе с коллекциями и умными указателями.

Исходный код

```
template <class T> class Tree
{
public:
	Tree();
	std::shared_ptr<TreeItem<T>> find(std::shared_ptr<T> &obj);
	void remove(int32_t side);
	void insert(std::shared_ptr<T> &obj);
	void print();
	void print(std::ostream& os);
	template <class A> friend std::ostream& operator<<(std::ostream& os, Tree<A>
&tree);
	bool empty();
	virtual ~Tree();



	TIterator<TreeItem<T>, T> end();
private:
	std::shared_ptr<TreeItem<T>> head;
	std::shared_ptr<TreeItem<T>> minValueNode(std::shared_ptr<TreeItem<T>> root);
	std::shared_ptr<TreeItem<T>> deleteNode(std::shared_ptr<TreeItem<T>> root,
int32_t side);
	void print_tree(std::shared_ptr<TreeItem<T>> item, int32_t a, std::ostream& os);
};

template <class T> class Tree;

template <class T> class TreeItem
{
public:
	TreeItem();
	TreeItem(const std::shared_ptr<T> &obj);

	std::shared_ptr<T> GetFigure();
	~TreeItem();
	friend class Tree<T>;
private:
	std::shared_ptr<T> item;
	std::shared_ptr<TreeItem<T>> left;
```

```
        std::shared_ptr<TreeItem<T>> right;
};
```

Консоль

```
List of operations:
1) Insert rhomb
2) Insert hexagon
3) Insert pentagon
4) Remove figure
5) Print
0) Exit
1
4 8
2
6
3
7

5

4
8
32
  null
  6
3
18

2
99
3
56
1
456 87

5

4
8
32
  null
  6
3
18
    null
    99
3
297
      null
      456
87
39672

4
4
4
456

5
```

```
6
3
18
  null
  99
3
297
```

Выводы

В этой лабораторной работе был спроектирован и запрограммирован на языке C++ шаблон класса-контейнера первого уровня, что позволило держать в вершинах бинарного дерева три фигуры на выбор. Были подробно изучены шаблоны и их правильное использование.

Лабораторная работа №5

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.
Итератор должен позволять использовать структуру данных в операторах типа for. Например: for(auto i : stack) std::cout « *i « std::endl;

Фигуры: ромб, шестиугольник, пятиугольник
Контейнер: бинарное дерево.

Описание

Для доступа к элементам некоторого множества элементов используют специальные объекты, называемые итераторами. В контейнерных типах stl они доступны через методы класса . Функциональные возможности указателей и итераторов близки, так что обычный указатель тоже может использоваться как итератор.

Категории итераторов:

- Итератор ввода (input iterator) – используется потоками ввода.
- Итератор вывода (output iterator) – используется потоками вывода.
- Однонаправленный итератор (forward iterator) – для прохода по элементам в одном направлении.

• Двунаправленный итератор (bidirectional iterator) – способен пройти по элементам в любом направлении. Такие итераторы реализованы в некоторых контейнерных типах stl (list, set, multiset, map, multimap).

• Итераторы произвольного доступа (random access) – через них можно иметь доступ к любому элементу. Такие итераторы реализованы в некоторых контейнерных типах stl (vector, deque, string, array).

Исходный код

```
#ifndef TITERATOR_H
#define TITERATOR_H
#include <memory>
#include <iostream>
template <class node, class T>
class TIterator
{
public:
  TIterator(std::shared_ptr<node> n) {

    node_ptr = n;
  }
  std::shared_ptr<T> operator * () {
    return node_ptr->GetFigure();
  }
  std::shared_ptr<T> operator -> () {
    return node_ptr->GetFigure();
  }
  void operator ++ () {
    node_ptr = Next(node_ptr);
  }

        std::shared_ptr<node> Next(std::shared_ptr<node> _cur)
        {
                if (_cur->right != NULL)
                {
                        _cur = _cur->right;

                        while (_cur->left != NULL)
                                _cur = _cur->left;
                }
                else
                {
                        std::shared_ptr<TreeItem<T>> succ = NULL;
                        std::shared_ptr<TreeItem<T>> root = node_ptr;

                        while (root != NULL)
                        {
                                if (_cur->item < root->item)
                                {
                                        succ = root;
                                        root = root->left;
                                }
                                else if (root->item < _cur->item)
                                        root = root->right;
                                else
                                        break;
```

```
                    }

                    _cur = succ;
                }

                return _cur;
            }

    TIterator operator ++ (int) {
        TIterator iter(*this);
        ++(*this);
        return iter;
    }
    bool operator == (TIterator const& i) {
        return node_ptr == i.node_ptr;
    }
    bool operator != (TIterator const& i) {
        return !(*this == i);
    }
private:
    std::shared_ptr<node> node_ptr;
};
#endif
```

Консоль

```
List of operations:
1) Insert rhomb
2) Insert hexagon
3) Insert pentagon
4) Remove figure
5) Print
6) Print tree with iterator
0) Exit
1
4 8
2
66

6
4
66
```

Выводы

В этой лабораторной работе был спроектирован и разработан Итератор для бинарного дерева. Так же были изучены и другие типы итераторов. Итератор был разработан в виде шаблона и работает фигурами: ромб, шестиугольник, пятиугольник

Лабораторная работа №6

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 дл хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти.
Аллокатор должен хранить списки использованных/свободных блоков. Дл хранения списка свободных блоков нужно применять динамическую структуру данных
(контейнер 2-ого уровня, согласно варианту задания).
Для вызова аллокатора должны быть переопределены операторы new и delete у классов-фигур.

Описание

Аллокатор памяти – часть программы, обрабатывающая запросы на выделение и освобождение оперативной памяти или запросы на включение заданной области памяти в адресное пространство процессора.
Основное назначение аллокатора памяти в первом смысле – реализация динамической памяти. В языке С динамическое выделение памяти производится через функцию malloc.
Программисты должны учитывать последствия динамического выделения памяти и дважды обдумать использование функции malloc или оператора new. Фрагментация и потери в производительности, связанные с использование динамической памяти, могут иметь

катастрофические последствия в вашем дальнейшем цикле разработки. Проекты, где управление и распределение памяти не продумано надлежащим образом, часто страдают от случайных сбоев после длительной сессии из-за нехватки памяти.

Исходный код

Описание классов фигур и классов-контейнеров остается неизменным

```
class TAllocationBlock {
public:
	TAllocationBlock(size_t size, size_t count);
	void *allocate();
	void deallocate(void *pointer);
	bool has_free_blocks();

	virtual ~TAllocationBlock();
private:
	size_t _size;
	size_t _count;

	char *_used_blocks;
	void **_free_blocks;

	size_t _free_count;};
```

Консоль

```
TAllocationBlock: Memory init
List of operations:
1) Insert rhomb
2) Insert hexagon
3) Insert pentagon
4) Remove figure
5) Print
6) Print tree with iterator
0) Exit
1
22 8
2
44
3
55 9
5
22
8
176
  null
  44
3
132

3
6 9
5
22
8
176
  6
9
54
  44
3
132
```

```
4
6
4
22

5

44
3
132
```

Выводы

В этой лабораторной работе был спроектирован и разработан аллокатор памяти для бинарного дерева. Был минимизирован вызов операции malloc. Для вызова аллокатора были переопределены операторы new и delete у классов-фигур. Для хранения свободных блоков было реализовано бинарное дерево.

Лабораторная работа №7

Задача: Необходимо реализовать динамическую структуру данных – "Хранилище объектов"и алгоритм работы с ней. "Хранилище объектов"представляет собой контейнер бинарное дерево. Каждым элементом контейнера является динамическая структура список. Таким образом, у нас получается контейнер в контейнере. Элементов второго контейнера является объект-фигура, определенная вариантом задания. При этом должно выполняться правило, что количество объектов в контейнере второго уровня не больше 5. Т.е. если нужно хранить больше 5 объектов, то создается еще один контейнер второго уровня.
Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию площади объекта. При удалении объектов должно выполняются правило, что контейнер второго уровня не должен быть пустым. Т.е. если он становится пустым, то он должен удалиться.

Описание

Принцип открытости/закрытости – принцип ООП, устанавливающий следующее положение: "программные сущности должны быть открыты для расширения, но закрыты для изменения". Контейнер в программировании – структура (АТД), позволяющая инкапсулировать в себе объекты любого типа. Объектами контейнеров являются коллекции, которые уже могут содержать в себе объекты определенного типа. Например, в зыке C++, std::list (шаблонный класс) является контейнером, а объект его класса-конкретизации, как например, std::list<int> mylist является коллекцией. Среди программистов наиболее известны контейнеры, построенные на основе шаблонов, однако, существуют и реализации в виде библиотек. Примерами контейнеров в C++ являются контейнеры из стандарт- ной библиотеки (STL) – map, vector и т.д. В контейнерах часто встречаются реализации алгоритмов для них. В ряде языков программирования (особенно в скриптовых типа Perl или PHP) контейнеры и работа с ними встроена в язык.

Исходный код

```
template <class T> class TMassivItem {
public:
    TMassivItem();
    TMassivItem(const shared_ptr<Figure> &figure);
    TMassivItem(const shared_ptr<TMassivItem<T>>& orig);
    template <class A> friend ostream& operator<< (ostream& os, const
shared_ptr<TMassivItem<A>> &obj);
    double Square();
    virtual ~TMassivItem();
    shared_ptr<Figure> obj;
};

template <class T> class TMassiv {
public:
    TMassiv();
    TMassiv(const TMassiv& orig);
    void push(shared_ptr<Figure> obj);
    void clear();
    void pop(size_t n);
    template <class A> friend ostream& operator<<(ostream& os,const TMassiv<A>&
massiv);
    virtual ~TMassiv();
private:
    shared_ptr<TMassivItem<T>> *token;
    size_t size = 0;
};
```

Консоль

```
Oparations:
1) Insert rhomb
2) Insert pentagon
3) Insert hexagon
4) Delete of type 1)rhomb 2)pentagon 3)hexagon
5) Delete side
6) Print
0) Exit
1
66
2
48
3
57
6
Side = 48
Side = 57
Side = 66

4
3
6
Side = 48
Side = 66

4
1
6
Side = 48

4
2
```

```
6
Empty
```

Выводы

В этой лабораторной работе был реализован контейнер второго уровня. В вершинах которого хранятся массивы . Было реализовано два вида удаления из контейнера второго уровня. Первый вид удаляет все фигуры одного типа. В добавок к этому была реализована обычная вставка в бинарное дерево и его печать.

Лабораторная работа №8

Задача: Используя структуры данных, разработанные для лабораторной работы №6 (контейнер 1-ого уровня и классы-фигуры) разработать алгоритм быстрой сортировки для класс-контейнера.

Необходимо разработать два вида алгоритма:
1. Обычный, без параллельных вызовов.
2. 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 потокобезопасности структур использовать механизмы:
- mutex
- lock quard

Программа должна позволять:
- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера.
- Удалять фигуры из контейнера.
- Проводить сортировку контейнера.

Фигуры: треугольник, шестиугольник, восьмиугольник.
Контейнер: бинарное дерево.

Описание

Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцесорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading). Оно включает в себя все черты более традиционного, последовательного программирования, но в параллельном программировании имеются три дополнительных, четко определенных этапа:

- Определение параллелизма: анализ задачи с целью выделить подзадачи, которые могут выполняться одновременно.
- Выявление параллелизма: изменение структуры задачи таким образом, чтобы

можно было эффективно выполнять подзадачи. Длэтого часто требуется найти зависимости между подзадачами и организовать исходный код так, чтобы
ими можно было эффективно управлять.

• Выражение параллелизма: реализация параллельного алгоритма в исходном коде с помощью системы обозначений параллельного программирования.

Исходный код

```
template <class T>
void Tree<T>::Sort(std::shared_ptr<T> *&arr, int l, int r)
{
	int x = l + (r - l) / 2;
	int i = l;
	int j = r;

	while (i <= j) {
		while (arr[i]->Side() < arr[x]->Side()) {
			i++;
		}
		while (arr[j]->Side() > arr[x]->Side()) {
			j--;
		}
		if (i <= j) {
			std::shared_ptr<T> tmp = arr[i];
			arr[i] = arr[j];
			arr[j] = tmp;
			i++;
			j--;
		}
	}
	if (i < r) {
		Sort(arr, i, r);
	}

	if (l < j) {
		Sort(arr, l, j);
	}
}

template <class T>
int ParalSort(std::shared_ptr<T> *&arr, int l, int r)
{
	int x = l + (r - l) / 2;
	int i = l;
	int j = r;

	while (i <= j) {
		while (arr[i]->Side() < arr[x]->Side()) {
			i++;
		}
		while (arr[j]->Side() > arr[x]->Side()) {
			j--;
		}
		if (i <= j) {
			std::shared_ptr<T> tmp = arr[i];
			arr[i] = arr[j];
			arr[j] = tmp;
			i++;
			j--;
		}
```

```
	}
	if (i < r) {
		std::packaged_task<int(std::shared_ptr<T> *&, int, int)> task(ParalSort);
		auto result = task.get_future();

		std::thread task_td(std::move(task), std::ref(arr), i, r);
		task_td.join();
		result.get();
	}
	if (l < j) {
		std::packaged_task<int(std::shared_ptr<T> *&, int, int)> task(ParalSort);
		auto result = task.get_future();

		std::thread task_td(std::move(task), std::ref(arr), l, j);
		task_td.join();
		result.get();
	}
	return 0;
}
```

Консоль

```
Operations:
1) Add rhomb(2 sides)
2) Add pentagon
3) Add hexagon
4) Delete max side
5) Print
6) Print wirh iterator
7) ParSort of tree
8) Parallel sort of tree
0) Exit
7
Enter number of sides
5
Left 5 sides
56
Left 4 sides
4
Left 3 sides
78
Left 2 sides
2
Left 1 sides
3
5
4
  2
    null
    3
  56
    null
    78

Operations:
1) Add rhomb(2 sides)
2) Add pentagon
3) Add hexagon
4) Delete max side
5) Print
6) Print wirh iterator
7) ParSort of tree
```

```
8) Parallel sort of tree
0) Exit
8
Enter number of sides
7
Left 7 sides
55
Left 6 sides
67
Left 5 sides
41
Left 4 sides
2
Left 3 sides
3
Left 2 sides
9
Left 1 sides
21
5
55
  41
    2
      null
      3
        null
        21
          9
          null
    null
  67
```

Выводы

В этой лабораторной работе была реализована быстрая сортировки и быстрая сортировка с использованием потоков. Как эта сортировка проходит для бинарного дерева? Легко. Вводится размер массива, потом вводится n-элементов для массива, где n - размер массива. Затем массив сортируется с/без помощи потоков. Далее просто из отсортированного массива строится сбалансированное бинарное дерево. Лабораторная помогла лучше разобраться в том, что такое потоки, как их представить, как проверить с помощью стороннего ПО.

Лабораторная работа №9

Задача: Используя структуры данных, разработанные для лабораторной работы №6 (контейнер 1-ого уровня и классы-фигуры) необходимо разработать:

- Контейнер второго уровня с использованием шаблонов.
- Реализовать с помощью лямбда-выражений набор команд, совершающих операции над контейнером 1-ого уровня: генерация фигур со случайными значениями 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 конейнеру первого уровня.

Для создания потоков ипользовать механизмы:

- future
- packaged task/async

Для обеспечения потокобезопасности структур использовать механизмы:
• mutex
• lock quard
Фигуры: ромб, шестиугольник, пятиугольник.

Описание

Лямбда-выражение – это удобный способ определения анонимного объекта-функции непосредственно в месте его вызова или передачи в функцию в качестве аргумента. Обычно лямбда-выражения используются для инкапсуляции нескольких строк кода, передаваемых алгоритмам или асинхронным методам. В итоге, мы получаем крайне удобную конструкцию, которая позволяет сделать код более лаконичным и устойчивым к изменениям.
Непосредственное объявление лямбда-функции состоит из трех частей. Первая часть (квадратные скобки) позволяет привязывать переменные, доступные в текущей области видимости. Вторачасть (круглые скобки) указывает список принимаемых параметров лямбда-функции. Третьчасть (фигурные скобки) содержит тело лямбда- функции.
В настоящее время, учитывая, что достигли практически потолка по тактовой частоте и дальше идет рост количества ядер, появился запрос на параллелизм. В результате снова в моде стал функциональный подход, так как он очень хорошо работает в условиях параллелизма и не требует явных синхронизаций. Поэтому сейчас усиленно думают, как задействовать растущее число ядер процессора и как обеспечить автоматическое распараллеливание. А в функциональном программировании практически основа всего – лямбда. Учитывая, что функциональные зыки переживают второе рождение, было бы странным, если бы функциональный подход не добавляли во все популярные языки. C++ – язык, поддерживающий много парадигм, поэтому нет ничего странного в использовании лямбда-функций и лямбда-выражений в нем.

Исходный код

```
int main(void) {
	Tree<Figure> ttree;
	typedef std::function<void(void)> command;
	TTree<command> tree(4);
	command cmdInsert = [&]() {
		std::cout << "Command: Insert" << std::endl;
		std::default_random_engine generator;
		std::uniform_int_distribution<int> distribution(1, 10);
		for (int i = 0; i < 10; i++) {
			int side = distribution(generator);
			if ((side % 2) == 0) {
				std::shared_ptr<Figure> ptr =
std::make_shared<Rhomb>(Rhomb(side, side));
				if (ttree.find(ptr) == nullptr) {
					ttree.insert(ptr);
				}
			}
			else if ((side % 3) == 0) {
				std::shared_ptr<Figure> ptr =
std::make_shared<Pentagon>(Pentagon(side));
				if (ttree.find(ptr) == nullptr) {
					ttree.insert(ptr);
				}
			}
			else {
```

```
                        std::shared_ptr<Figure> ptr =
std::make_shared<Hexagon>(Hexagon(side));
                        if (ttree.find(ptr) == nullptr) {
                                ttree.insert(ptr);
                        }
                }
        }
    };
    command cmdPrint = [&]() {
            std::cout << "Command: Print" << std::endl;
            for (auto i : ttree) {
                    i->Print();
            }
    };
    command cmdRemove = [&]() {
            std::cout << "Command: Remove" << std::endl;
            std::default_random_engine generator;
            std::uniform_int_distribution<int> distribution(1, 10);
            int side = distribution(generator);
            std::cout << "Lesser than " << side << std::endl;
            for (int i = 0; i < 10; i++) {

                    for (auto iter : ttree) {
                            if (iter->Side() < side) {
                                    ttree.remove(iter->Side());
                                    break;
                            }
                    }
            }
    };

    tree.insert(std::shared_ptr<command>(&cmdInsert, [](command*) {}));
    tree.insert(std::shared_ptr<command>(&cmdPrint, [](command*) {}));
    tree.insert(std::shared_ptr<command>(&cmdRemove, [](command*) {}));
    tree.insert(std::shared_ptr<command>(&cmdPrint, [](command*) {}));
    tree.inorder();

    return 0;
}
```

Консоль

```
Command: Insert
Pentagon created: 3
Pentagon copy created
Pentagon created: 3
Pentagon copy created
Hexagon created: 5
Rhomb created: 6 6
Hexagon created: 5
Rhomb created: 2 2
Rhomb created: 10 10
Rhomb created: 6 6
Pentagon created: 9
Pentagon copy created
Rhomb created: 4 4
Command: Print
2
Side = 3
4
Side = 5
6
Side = 9
```

```
10
Command: Remove
Lesser than 3
Command: Print
Side = 3
4
Side = 5
6
Side = 9
10
```

Вывод

В этой лабораторной работе были реализованы лямбда-выражения, которые были помещены в вершины бинарного дерева и выполнялись поочередно. Так было реализовано удаление сторон меньшего, чем заданное число, которое генерировалось в каком-то диапазоне. Было реализовано добавление фигур с различными сторонами. Печать дерева.

Курс Объектно-Ориентированного Программирования подошел. Этот курс оказался довольно полезным для меня . Он немало помог мне в понимании других предметов этого семестра, а именно: операционными системами и дискретным анализом.