Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу «Объектно-ориентированное программирование»

Студент: Литвина А. А.

Преподаватель: Поповкин А. В.

Группа: 8О-206Б-17

Вариант: 3

Дата:

Оценка:

Подпись:

**Москва, 2018**

Лабораторная работа №1

1 Цель работы

• Программирование классов на языке С++

• Управление памятью в языке С++

• 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.

2 Задача

Необходимо спроектировать и запрограммировать на языке C++ классы фигур, согласно вариантов 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)

Фигура: прямоугольник, ромб, трапеция

3 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.

Инкапсуляция — свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе. Одни языки (например, С++, Java или Ruby) отождествляют инкапсуляцию с сокрытием, но другие (Smalltalk, Eiffel, OCaml) различают эти понятия.

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 для работы 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 (интерфейс присваивания и считывания значения), которые имитируют непосредственный доступ к полю. Эти блоки называются «свойствами» и почти совпадают по конкретному имени со своим полем (например, имя поля может начинаться со строчной, а имя свойства — с заглавной буквы). 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 — cущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

4 Исходный код

|  |
| --- |
| class Figure { |
|  | public: |
|  | virtual double Square() = 0; |
|  | virtual void Print() = 0; |
|  | virtual ~Figure() {}; |
|  | }; |

|  |
| --- |
| class Rectangle : public Figure { |
|  | public: |
|  | Rectangle(); |
|  | Rectangle(std::istream &is); |
|  | Rectangle(size\_t length, size\_t height); |
|  | Rectangle(const Rectangle& orig); |
|  | double Square() override; |
|  | void Print() override; |
|  | virtual ~Rectangle(); |
|  | private: |
|  | size\_t length; |
|  | size\_t height; |
|  | }; |

|  |
| --- |
| class Rhombus : public Figure { |
|  | public: |
|  | Rhombus(); |
|  | Rhombus(std::istream &is); |
|  | Rhombus(size\_t side, size\_t smaller\_angle); |
|  | Rhombus(const Rhombus& orig); |
|  | double Square() override; |
|  | void Print() override; |
|  | virtual ~Rhombus(); |
|  | private: |
|  | size\_t side; |
|  | size\_t smaller\_angle; |
|  | }; |

|  |
| --- |
| class Trapeze : public Figure { |
|  | public: |
|  | Trapeze(); |
|  | Trapeze(std::istream &is); |
|  | Trapeze(size\_t smaller\_base, size\_t bigger\_base, size\_t left\_side, size\_t right\_side); |
|  | Trapeze(const Trapeze& orig); |
|  | double Square() override; |
|  | void Print() override; |
|  | virtual ~Trapeze(); |
|  | private: |
|  | size\_t smaller\_base; |
|  | size\_t bigger\_base; |
|  | size\_t left\_side; |
|  | size\_t right\_side; |
|  | }; |

5 Выводы

В ходе реализации классов фигур, я ознакомилась с базовыми понятиями ООП, такими, как класс, объект, инкапсуляция, абстрактный тип данных, наследование, полиморфизм, а так же познакомился с синтаксисом C++, который все же отличается от языков, в которых мы работали ранее.

Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Объектно-ориентированное программирование»

Студент: Литвина А. А.

Преподаватель: Поповкин А. В.

Группа: 8О-206Б-17

Вариант: 3

Дата:

Оценка:

Подпись:

**Москва, 2018**

Лабораторная работа №2

1 Цель работы

• Закрепление навыков работы с классами.

• Создание простых динамических структур данных.

• Работа с объектами, передаваемыми «по значению».

2 Задача

Необходимо спроектировать и запрограммировать на языке С++ класс-контейнер первого уровная, 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: Прямоугольник. Контейнер: Массив.

3 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 собо отдельные элементы, связанные между собой с помощью ссылок. Каждый элемент состоит из двух областей памяти: поля данных и ссылок. Ссылки – это адреса других узлов того же типа, с которыми данный элемент логически связан. При добавлении нового 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 воздействуют 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.

4 Исходный код

|  |
| --- |
| class TArray |
|  | { |
|  | public: |
|  | TArray() = default; |
|  | TArray(size\_t size); |
|  | TArray(size\_t size, const Rectangle \*initObj); |
|  | size\_t ElementInsert(Rectangle \*obj, size\_t index); |
|  | void ElementDelete(size\_t index); |
|  | Rectangle& ElementGet(size\_t index); |
|  | friend std::ostream& operator <<(std::ostream& os, const TArray& arr); |
|  | ~TArray(); |
|  |  |
|  | private: |
|  | Rectangle \*array; |
|  | size\_t size; |
|  | }; |

5 Выводы

В этой работе было необходимо реализовать собственную структуру данных. В моем случае ей являлся массив. В него можно добавлять произвольное количество элементов в конец, печатать содержимое, брать элемент по индексу, удалять элемент по индексу. Несмотря на то, что все основные структуры уже созданы до нас, каждый программист должен уметь реализовать их.

Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Объектно-ориентированное программирование»

Студент: Литвина А. А.

Преподаватель: Поповкин А. В.

Группа: 8О-206Б-17

Вариант: 3

Дата:

Оценка:

Подпись:

**Москва, 2018**

Лабораторная работа №3

1 Цель работы

• Закрепление навыков работы с классами.

• Знакомство с умными указателями.

2 Задача

Необходимо спроектировать и запрограммировать на языке С++ класс-контейнер первого уровня, содержащий все три фигуры, согласно варианту задания.

Классы должны удовлетворять следующим правилам:

• Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.

• Класс-контейнер должен содержать объекты, используя std::shared ptr.

• Класс-контейнер должен иметь метод по добалению 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 ostream.

• Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.

• Классы должны быть расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Фигуры: Прямоугольник, трапеция, ромб. Контейнер: Массив.

3 Описание

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 С++:

• 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 копия 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 для 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.

4 Исходный код

|  |
| --- |
| class TArrayItem |
|  | { |
|  | public: |
|  | TArrayItem(); |
|  | TArrayItem(const std::shared\_ptr<Figure>& figure); |
|  | void SetPtr(const std::shared\_ptr<Figure>& figure); |
|  | friend std::ostream& operator<<(std::ostream& os, const TArrayItem& obj); |
|  | double Square(); |
|  | ~TArrayItem(); |
|  | private: |
|  | std::shared\_ptr<Figure> figure; |
|  | }; |

5 Выводы

В данной работе были получены навыки работы с умными указателями. В нашем случае были использованы shared\_ptr. Их особенность в том, что они хранят количество ссылок на объект, и в случае если таковых не имеется, удаляются. Это весьма удобно, так как не нужно заботиться об удалении вручную.

Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Объектно-ориентированное программирование»

Студент: Литвина А. А.

Преподаватель: Поповкин А. В.

Группа: 8О-206Б-17

Вариант: 3

Дата:

Оценка:

Подпись:

**Москва, 2018**

Лабораторная работа №4

1 Цель работы

• Знакомство с шаблонами классов.

• Построение шаблонов динамических структур данных.

2 Задача

Необходимо спроектировать и запрограммировать на языке С++ класс-контейнер первого уровня, содержащий все три фигуры, согласно варианту задания.

Классы должны удовлетворять следующим правилам:

• Требования к классам фигуры аналогичны требованиям из лабораторной работы 1.

• Шаблон класса-контейнера должен соджержать 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 (опеределяется структурой контейнера).

• Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (опеределяется 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).

Фигуры: Прямоугольник, трапеция, ромб. Контейнер: Массив.

3 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 собо отдельные элементы, связанные между собой с помощью ссылок. Каждый элемент состоит из двух областей памяти: поля данных и ссылок. Ссылки – это адреса других узлов того же типа, с которыми данный элемент логически связан. При добавлении нового 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 воздействуют 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.

4 Исходный код

|  |
| --- |
| template <class T> |
|  | class TArray |
|  | { |
|  | public: |
|  | TArray() = default; |
|  | TArray(size\_t size); |
|  | TArray(size\_t size, const TArrayItem<T> \* initObj); |
|  | size\_t ElementInsert(TArrayItem<T> \*obj, size\_t index); |
|  | void ElementDelete(size\_t index); |
|  | TArrayItem<T>& ElementGet(size\_t index); |
|  | int Size(); |
|  | template <class A> friend std::ostream& operator <<(std::ostream& os, const TArray<A>& arr); |
|  | ~TArray(); |
|  |  |
|  | private: |
|  | std::shared\_ptr<TArrayItem<T>> array; |
|  | size\_t size; |
|  | }; |

5 Выводы

В данной работе был спроектирован шаблон класс массива, который позволяет пользователю хранить произвольные типы данных. Было изучено использование шаблонов и принципы их работы.

Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу «Объектно-ориентированное программирование»

Студент: Литвина А. А.

Преподаватель: Поповкин А. В.

Группа: 8О-206Б-17

Вариант: 3

Дата:

Оценка:

Подпись:

**Москва, 2018**

Лабораторная работа №5

1 Цель работы

• Закрепление навыков работы с шаблонами классов.

• Построение итераторов для динамических структур данных.

2 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;

• Требования к классам фигуры аналогичны требованиям из лабораторной работы 1.

• Шаблон класса-контейнера должен соджержать 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 (опеределяется структурой контейнера).

• Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (опеределяется 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).

Фигуры: Прямоугольник, трапеция, ромб. Контейнер: Массив.

3 Описание

Для доступа к элементам некоторого множества элементов используют специальные объекты, называемые итераторами. В контейнерных типах stl они доступны через методы класса (например, begin() в шаблоне класса vector). 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).

4 Исходный код

|  |
| --- |
| template <class node, class T> |
|  | class TIterator |
|  | { |
|  | public: |
|  | TIterator(node \* n) { |
|  | node\_ptr = n; |
|  | } |
|  | node \* operator \* () { |
|  | return node\_ptr; |
|  | } |
|  | std::shared\_ptr<node> operator -> () { |
|  | return node\_ptr; |
|  | } |
|  | void operator ++ () { |
|  | node\_ptr = (node\_ptr->GetNext()); |
|  | } |
|  | 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: |
|  | node \* node\_ptr; |
|  | }; |

5 Выводы

Был разработан итератор для массива. Объяснение принципа работы займет много времени, но он становится очевиден при прочтении функции GetNext(). Итераторы - удобный инструмент для перебора всех элементов структуры. Благодаря им можно значительно упростить нагромождения кода.

Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №6 по курсу «Объектно-ориентированное программирование»

Студент: Литвина А. А.

Преподаватель: Поповкин А. В.

Группа: 8О-206Б-17

Вариант: 3

Дата:

Оценка:

Подпись:

**Москва, 2018**

Лабораторная работа №6

1 Цель работы

• Закрепление навыков по работе с памятью в C++.

• Создание аллокаторов памяти для динамических структур данных.

2 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.

Аллокатор должен хранить списки использованных/свободных блоков. Для 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.

Фигуры: Прямоугольник, трапеция, ромб. Контейнер: Массив.

3 Описание

Аллокатор памяти – часть программы (как прикладной, так и операционной системы), 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. Легко убедить себя, что вы не делаете так уж много аллокаций, а значит большого значения это не имеет, но такой тип мышления распространяется лавиной по всей команде и приводит к медленной смерти. Фрагментация и потери в производительности, связанные с использование динамической памяти, не будучи пресеченными в зародыше, могут иметь катастрофические трудноразрешаемые 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 и стоят сотни часов работы программистов, пытающихся освободить память и реорганизовать ее выделение.

4 Исходный код

|  |
| --- |
| 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; |
|  | }; |

5 Выводы

В результате выполнения был разработан аллокатор памяти для массива, оптимизирован вызов операции malloc, переопределены операторы new-delete. Для хранения свободных блоков использовался массив памяти.

Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №7 по курсу «Объектно-ориентированное программирование»

Студент: Литвина А. А.

Преподаватель: Поповкин А. В.

Группа: 8О-206Б-17

Вариант: 3

Дата:

Оценка:

Подпись:

**Москва, 2018**

Лабораторная работа №7

1 Цель работы

• Создание сложных динамических структур данных.

• Закрепление принципа OCP.

2 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 выполянться правило, что контейнер второго уровня не должен быть пустым. Т.е. если он становится пустым, то он должен удалиться.

Фигуры: Прямоугольник, трапеция, ромб.

Контейнер 1: массив. Контейнер 2: В-дерево

3 Описание

Принцип открытости/закрытости (ОСР) – принцип ООП, устанавливающий следующее положение: "программные сущности (классы, модули, функции и т.п.) должны быть открыты для расширения, но закрыты для изменения".

Контейнер в программировании – структура (АТД), позволяющая инкапсулировать в себе объекты любого типа. Объектами (переменными) контейнеров являются коллекции, которые уже могут содержать в себе объекты определенного типа.

Например, в языке С++, std::list (шаблонный класс) является контейнером, а объект его класса-конкретизации, как например, std::list mylist является коллекцией. Среди "широких масс"программистов наиболее известны контейнеры, построенные на основе шаблонов, однако, существуют и реализации в виде библиотек (наиболее широко известна библиотека GLib). Кроме того, применяются и узкоспециализированные решения. Примерами контейнеров в С++ являются контейнеры из стандартной 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.

Стек – тип или структура данных в виде набора элементов, которые расположены по принципу LIFO, т.е. "последний пришел, первый вышел". Доступ к элементам осуществляет через обращение к головному элементу (тот, который был добавлен последним).

4 Исходный код

|  |
| --- |
| template <class T> |
|  | class TBinaryTree |
|  | { |
|  | public: |
|  | TBinaryTree(); |
|  | void rebuild(); |
|  | void push(T \* value); |
|  | int Size(); |
|  | void Sort(); |
|  | bool empty(); |
|  | T\* pop(); |
|  | template <class A> friend std::ostream& operator<<(std::ostream& os, const TBinaryTree<A> & obj); |
|  | template <class A> friend std::istream& operator >> (std::istream& is, const TBinaryTree<A>& obj); |
|  | ~TBinaryTree(); |
|  | //private: |
|  | //TBinaryTree \*left; |
|  | //TBinaryTree \*right; |
|  | //T data; |
|  | T\* bt[5]; |
|  | int size; |
|  | }; |

5 Выводы

В этой работе был реализован контейнер второго уровня – В-дерево. В его элементах хранятся массивы, с количеством элементов не более 5. Так же было реализовано два вида удаления из массива - удаление всех фигур одного типа и удаление всех фигур, совпадающих с заданной. В результате работы продемонстрирована возможность хранения и использования структур в структурах.

Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №8 по курсу «Объектно-ориентированное программирование»

Студент: Литвина А. А.

Преподаватель: Поповкин А. В.

Группа: 8О-206Б-17

Вариант: 3

Дата:

Оценка:

Подпись:

**Москва, 2018**

Лабораторная работа №8

1 Цель работы

• Знакомство с параллельным программированием в C++.

2 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 ипользовать 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: Прямоугольник, трапеция, ромб.

Контейнер 1: массив. Контейнер 2: В-дерево

3 Описание

Параллельное программирование – это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности (multithreading).

Параллельное программирование может быть сложным, но его легче понять, если считать его не “трудным“ , а просто “немного иным“. 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. Для этого 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.

4 Исходный код

|  |
| --- |
| template<class T> |
|  | void TArray<T>::QSort(int l, int r) |
|  | { |
|  | if (array[(l + r) / 2].GetItemValue() == nullptr) { |
|  | return; |
|  | } |
|  | double x = array[(l + r) / 2].GetItemValue()->Square(); |
|  | std::cout << "X is " << x << "\n"; |
|  | //çàïèñü ýêâèâàëåíòíà (l+r)/2, |
|  | //íî íå âûçââàåò ïåðåïîëíåíèÿ íà áîëüøèõ äàííûõ |
|  | int i = l; |
|  | int j = r; |
|  | //êîä â while îáû÷íî âûíîñÿò â ïðîöåäóðó particle |
|  | while (i <= j) |
|  | { |
|  | while (array[i].GetItemValue() != nullptr && array[i].GetItemValue()->Square() < x && i < Size()) i++; |
|  | while (array[j].GetItemValue() != nullptr && array[j].GetItemValue()->Square() > x && j >= 0) j--; |
|  | if (i <= j) |
|  | { |
|  | TArrayItem<T> tmp; |
|  | tmp = array[i]; |
|  | array[i] = array[j]; |
|  | array[j] = tmp; |
|  | i++; |
|  | j--; |
|  | } |
|  | } |
|  | if (i<r) |
|  | QSort(i, r); |
|  |  |
|  | if (l<j) |
|  | QSort(l, j); |
|  | } |

|  |
| --- |
|  |
| template<class T> |
|  | void TArray<T>::MQSort(int l, int r) |
|  | { |
|  | if (array[(l + r) / 2].GetItemValue() == nullptr) { |
|  | return; |
|  | } |
|  | double x = array[(l + r) / 2].GetItemValue()->Square(); |
|  | std::cout << "X is " << x << "\n"; |
|  | //çàïèñü ýêâèâàëåíòíà (l+r)/2, |
|  | //íî íå âûçââàåò ïåðåïîëíåíèÿ íà áîëüøèõ äàííûõ |
|  | int i = l; |
|  | int j = r; |
|  | //êîä â while îáû÷íî âûíîñÿò â ïðîöåäóðó particle |
|  | while (i <= j) |
|  | { |
|  | while (array[i].GetItemValue() != nullptr && array[i].GetItemValue()->Square() < x && i < Size()) i++; |
|  | while (array[j].GetItemValue() != nullptr && array[j].GetItemValue()->Square() > x && j >= 0) j--; |
|  | if (i <= j) |
|  | { |
|  | TArrayItem<T> tmp; |
|  | tmp = array[i]; |
|  | array[i] = array[j]; |
|  | array[j] = tmp; |
|  | i++; |
|  | j--; |
|  | } |
|  | } |
|  | if (i < r) { |
|  | MQSort(i, r); |
|  | } |
|  | if (l < j) { |
|  | auto left = std::async(std::launch::async, [&]() { |
|  | return MQSort(l, j); |
|  | }); |
|  | //MQSort(l, j); |
|  | } |
|  |  |
|  | } |
|  |  |

5 Выводы

Благодаря этой лабораторной работе я получила опыт применения потоков в C++, что, несомненно, пригодится в дальнейшем. Распараллеливание вычислений - незаменимая вещь для уменьшения времени работы какого-то алгоритмом, например, сортировки работают в разы быстрее, чем в однопоточной реализации.

Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №9 по курсу «Объектно-ориентированное программирование»

Студент: Литвина А. А.

Преподаватель: Поповкин А. В.

Группа: 8О-206Б-17

Вариант: 3

Дата:

Оценка:

Подпись:

**Москва, 2018**

Лабораторная работа №9

1 Цель работы

• Знакомство с лямбда-выражениями

2 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 контейнеру первого уровня.

Для создания потоков ипользовать механизмы:

• future

• packaged task/async

Для обеспечения потокобезопасности структур использовать механизмы:

• mutex

• lock quard

Фигуры: Прямоугольник, трапеция, ромб.

Контейнер 1: массив. Контейнер 2: В-дерево

3 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. Вторая часть (круглые скобки) указывает список принимаемых параметров лямбда-функции. Третья часть (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 языки переживают второе рождение, было бы странным, если бы функциональный подход не добавляли во все популярные языки. С++ – язык, поддерживающий много парадигм, поэтому нет ничего странного в использовании лямбда-функций и лямбда-выражений в нем.

4 Исходный код

|  |  |  |
| --- | --- | --- |
|  | |  |
| int main() | |
|  | | { | |
|  | | Tree<Rectangle> treeRect; | |
|  | | typedef std::function<void(void) > command; | |
|  | | Vector<command> vecCmd; | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | | command cmdInsert = [&]() { | |
|  | | std::cout << "Creating random tree" << std::endl; | |
|  | | std::default\_random\_engine generator(time(NULL)); | |
|  | | std::uniform\_int\_distribution<int> distribution(1, 20); | |
|  | | for (int i = 0; i < 10; i++) { | |
|  | | int side\_a = distribution(generator); | |
|  | | int side\_b = distribution(generator); | |
|  | | std::shared\_ptr<Rectangle> ptr = std::make\_shared<Rectangle>(side\_a, side\_b); | |
|  | | int key = distribution(generator); | |
|  | | treeRect.insert(ptr, 0, key); | |
|  | | } | |
|  | | }; | |
|  | |  | |
|  | | command cmdPrint = [&]() { | |
|  | | std::cout << "Printing tree" << std::endl; | |
|  | | treeRect.print(); | |
|  | | }; | |
|  | |  | |
|  | | command removeIfEqualN = [&]() { | |
|  | | if (treeRect.empty()) { | |
|  | | std::cout << "Tree is empty" << std::endl; | |
|  | |  | |
|  | | } | |
|  | | else { | |
|  | | size\_t n; | |
|  | | std::cout << "Enter N: "; | |
|  | | std::cin >> n; | |
|  | | treeRect.remove(n); | |
|  | | } | |
|  | | }; | |
|  | |  | |
|  | | vecCmd.pushBack(std::shared\_ptr<command>(&cmdInsert, [](command\*){})); | |
|  | | vecCmd.pushBack(std::shared\_ptr<command>(&cmdPrint, [](command\*) {})); | |
|  | | vecCmd.pushBack(std::shared\_ptr<command>(&removeIfEqualN, [](command\*) {})); | |
|  | | vecCmd.pushBack(std::shared\_ptr<command>(&cmdPrint, [](command\*) {})); | |
|  | |  | |
|  | | while (!vecCmd.empty()) { | |
|  | | std::shared\_ptr<command> cmd = vecCmd.remove(0); | |
|  | | std::future<void> ft = std::async(\*cmd); | |
|  | | ft.get(); | |
|  | |  | |
|  | | } | |
|  | | system("pause"); | |
|  | | return 0; | |
|  | | } | |

5 Выводы

В этой лабораторной работе были получены базовые понятия о реализации лямбда-выражений в C++. Они были помещены в ранее реализованный массив, и выполнялись по очереди. Правильно реализованные лямбда-выражения упрощают код и дают прирост производительности.