Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу
«Объектно-ориентированное программирование»

Студент: С. Д. Старчеус
Преподаватель: А. А. В. Поповкин
Группа: М8О-206Б

Вариант: 25
Дата:
Оценка:
Подпись:

Москва, 2019

Вариант 25:

Фигуры: треугольник, квадрат, прямоугольник.
Контейнер 1-ого уровня: очередь.
Контейнер 2-ого уровня: массив.

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: С. Д. Старчеус
Преподаватель: А. А. В. Поповкин
Группа: М8О-206Б

Вариант: 25
Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №1

Цель:

- Программирование классов на языке C++
- Управление памятью в языке C++
- Изучение базовых понятий ООП
- Знакомство с:
  - Классами в C++
  - Перегрузкой операторов
  - Дружественными функциями
  - Операциями ввода-вывода из стандартных библиотек

Задача: необходимо спроектировать и запрограммировать на языке C++ классы фигур, согласно вариантов задания. Классы должны удовлетворять следующим правилам:

- Иметь общий родительский класс Figure
- Иметь общий виртуальный метод Print, печатающи параметры фигуры и ее тип в стандартный поток вывода
- Иметь общий виртуальный метод расчета площади фигуры
- Иметь конструктор, считывающий значения основных параметров фигуры из стандартного потока ввода
- 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.

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, C++, 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 объ-2екта 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, появляюща-

яся при создании экземпляра класса (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

Исходный код

```
class Figure {
  public:
    virtual double Area() = 0;
    virtual void Print() = 0;
    virtual ~Figure() {};
};

class Rect : public Figure {
public:
  Rect();
  Rect(std::istream &is);
  Rect(uint a, uint b);
  double Area() override;
  void Print() override;
  virtual ~Rect();
private:
  uint side_a, side_b;
};

class Square : public Figure {
public:
 Square();
 Square(std::istream &is);
 Square(uint a);
 double Area();
 void Print() override;
 virtual ~Square();
private:
 uint side_a;
};

class Triangle : public Figure {
public:
  Triangle();
  Triangle(std::istream &is);
  Triangle(uint a, uint b, uint c);
  double Area() override;
  void Print() override;
  virtual ~Triangle();
private:
  uint side_a, side_b, side_c;
};
```

triangle.cpp	
Triangle(std::istream &is)	Ввод из потока std::stream
Triangle(int a, int b, int c)	Создание фигуры через указание сторон
double Area() override	Вычисление площади
void Print() override	Вывод информации о фигуре
~Triangle()	Деструктор класса

Остальные фигуры реализованы таким же образом.

Выводы

В этой работе я познакомился с базовыми понятиями ООП, такими как наследование, полиморфизм, абстрактный тип данных и инкапсуляция. Спроектировал классы фигур, заданные вариантом, в которых использовались перегруженные операторы, дружественные функции и операции ввода-вывода из стандартных библиотек.

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: С. Д. Старчеус
Преподаватель: А. А. В. Поповкин
Группа: М8О-206Б

Вариант: 25
Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №2

Цель:

- 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 (колонка фигура 1), 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).

Нельзя использовать:

- Стандартные контейнеры std.
- Шаблоны (template).
- Различные варианты умных указателей (shared_ptr, weak_ptr).

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера.
- Удалять фигуры из контейнера.

Описание

Динамические структуры данных используют в тех случаях, когда не известно, 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: 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 ядрес 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.

Исходный код

```
class TQueue {
public:
  TQueue();
  ~TQueue();
  void Push(Triangle &&triangle);
  bool IsEmpty();
  Triangle Pop();
  friend std::ostream& operator<<(std::ostream& os, const TQueue& queue);
private:
  TQueueItem *first, *last;
  uint size;
};

class TQueueItem {
public:
  TQueueItem(const Triangle &triangle);
  ~TQueueItem();
  void SetNext(TQueueItem *item);
  TQueueItem* GetNext();
  Triangle Get();
  friend std::ostream& operator<<(std::ostream& os, const TQueueItem& obj);
private:
  Triangle triangle;
  TQueueItem *next;
};

class Triangle : public Figure {
public:
  Triangle();
  Triangle(std::istream &is);
  Triangle(Triangle &triangle);
  Triangle(uint a, uint b, uint c);
  double Area() override;
  void Print() override;
  virtual ~Triangle();
  bool operator==(const Triangle& right);
  friend std::ostream& operator<<(std::ostream& os, const Triangle& obj);
  friend std::istream& operator>>(std::istream& is, Triangle& obj);
  Triangle\& operator=(const Triangle& right);
  uint side_a, side_b, side_c;
};
```

triangle.cpp	
Triangle& operator=(const Triangle& right)	Переопределённый оператор копирования
friend std::istream& operator»(std::istream& is, Triangle& obj)	Переопределённый оператор вывода
bool operator==(const Triangle& right)	Переопределённый оператор сравнения

queue.cpp	
Queue()	Конструктор класса
Triangle Pop() override	Взятие элемента из очереди
void Push() override	Ввод элемента в очередь
friend std::ostream& operator«(std::ostream& os, const TQueue& queue)	Переопределённый оператор вывода

tqueueitem.cpp	
TQueueItem(const Triangle &triangle)	Конструктор класса
void SetNext(TQueueItem *item)	Установка следующего элемента
TQueueItem* GetNext()	Получение следующего элемента
Triangle Get()	Получение фигуры из узла
friend std::ostream& operator«(std::ostream& os, const TQueueItem& obj)	Переопределённый оператор вывода

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 реализованы в стандартной библиотеке, для пониманиях их работы и уверенности в своих действиях, самостоятельная разработка этих структур весьма полезна.

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: С. Д. Старчеус
Преподаватель: А. А. В. Поповкин
Группа: М8О-206Б
Вариант: 25
Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №3

Цель:

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

Задача: необходимо спроектировать и запрограммировать на языке C++ класс- контейнер первого уровня, содержащий все три фигуры класса фигуры, согласно вариантов задания (реализованную в ЛР1). Классы должны удовлетворять следую- щим 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 (опередеряется структурой контейнера).
- Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опередеряется 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).

Нельзя использовать:

- Стандартные контейнеры std.
- Шаблоны (template).
- Объекты «по-значению»

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер
- Распечатывать содержимое контейнера
- Удалять фигуры из контейнера

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 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 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.

Исходный код

```
class TQueue {
public:
  TQueue();
  ~TQueue();
  void Push(std::shared_ptr<Figure> &&figure);
  bool IsEmpty();
  std::shared_ptr<Figure> Pop();
  friend std::ostream& operator<<(std::ostream& os, const TQueue& queue);
private:
  std::shared_ptr<TQueueItem> first, last;
  uint size;
};

class TQueueItem {
public:
  TQueueItem(const std::shared_ptr<Figure>& figure);
  ~TQueueItem();
  void SetNext(std::shared_ptr<TQueueItem> item);
  std::shared_ptr<TQueueItem> GetNext();
  std::shared_ptr<Figure> Get();
  friend std::ostream& operator<<(std::ostream& os, const TQueueItem& obj);
private:
  std::shared_ptr<Figure> figure;
  std::shared_ptr<TQueueItem> next;
};
```

Изменилась реализация контейнера: класс Figure заменил класс Triangle. Добавление фигуры происходит при помощи полиморфизма времени выполнения: сначала создаётся желаемый объект, после чего при помощи dynamic_cast он рассматривается как объект класса Figure, после чего добавляется в контейнер, например:

```
  TQueue q;
  Triangle *fig = new Triangle(cin);
  Figure *f = dynamic_cast<Figure*>(fig);
  q.Push(std::shared_ptr<Figure>(f));
```

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, и количество ошибок, вызванных неумелой работе с памяьтю, сокращается.

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: С. Д. Старчеус
Преподаватель: А. А. В. Поповкин
Группа: М8О-206Б

Вариант: 25
Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №4

Цель:

- Знакомство с шаблонами классов
- Построение шаблонов динамических структур данных

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 классу 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 (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 расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер
- Распечатывать содержимое контейнера
- Удалять фигуры из контейнера

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 особенно полезны при работе с коллекциями и умными указателями.

Исходный код

```
1  template <class T>
2  class TQueue {
3  public:
4    TQueue();
5    ~TQueue();
6    void Push(std::shared_ptr<T> &&figure);
7    bool IsEmpty();
8    std::shared_ptr<T> Pop();
9    template <class A> friend std::ostream& operator<<(std::ostream& os, const TQueue<A
         >& queue);
10 private:
11   std::shared_ptr<TQueueItem<T>> first, last;
12   uint size;
13 };
14
15
16 template <class T>
17 class TQueueItem {
18 public:
19   TQueueItem(const std::shared_ptr<T>& figure);
20   ~TQueueItem();
21   void SetNext(std::shared_ptr<TQueueItem<T>> item);
22   std::shared_ptr<TQueueItem<T>> GetNext();
23   std::shared_ptr<T> Get();
24   template <class A> friend std::ostream& operator<<(std::ostream& os, const
         TQueueItem<A>& obj);
25 private:
26   std::shared_ptr<T> figure;
27   std::shared_ptr<TQueueItem<T>> next;
28 };
```

Разница между текущей и предыдущей работой: контейнер реализован через шаблон.

Выводы

В данной работе был спроектирован шаблон класса очередь, который позволяет пользователю хранить произвольные типы данных. Было изучено использование шаблонов и принципы их работы. Так же были реализованы методы, необходимые для работы с очередью.

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: С. Д. Старчеус
Преподаватель: А. А. В. Поповкин
Группа: М8О-206Б

Вариант: 25
Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №5

Цель:

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

Задача: используя структуры данных, разработанные для предыдущей лабора- торной 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.

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер
- Распечатывать содержимое контейнера
- Удалять фигуры из контейнера

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).

Исходный код

```
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->Get();
  }
  std::shared_ptr<T> operator -> () {
    return node_ptr->Get();
  }
  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:
  std::shared_ptr<node> node_ptr;
};
```

1 Выводы

Был разработан итератор для очереди. Итераторы - удобный инструмент для перебора всех элементов структуры. Так, например, в случае работы с очередью итератор позволяет выводить на экран все элементы стека без их извлечения. С помощью итераторов можно значительно упростить работу с какой-либо структурой данных, так как достаточно реализовать итератор один раз; далее, любой проход с помощью них, будет значительно проще, так как отбрасывается повторение 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: С. Д. Старчеус
Преподаватель: А. А. В. Поповкин
Группа: М8О-206Б

Вариант: 25
Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №6

Цель:

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

Задача: используя структуры данных, разработанные для предыдущей лабораторной работы (ЛРNo5) 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.

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер
- Распечатывать содержимое контейнера
- Удалять фигуры из контейнера

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 – уменьшение количества операций по выделению памяти, в следствие чего, программа работает эффективнее.

Исходный код

```
1  class TAllocationBlock {
2  public:
3    TAllocationBlock(size_t size,size_t count);
4    void *allocate();
5    void deallocate(void *pointer);
6    bool has_free_blocks();
7    virtual ~TAllocationBlock();
8  private:
9    size_t _size;
10   size_t _count;
11   char *_used_blocks;
12   void **_free_blocks;
13   size_t _free_count;
14 };
15
16 template <class T>
17 class TQueueItem {
18 public:
19   TQueueItem(const std::shared_ptr<T>& figure);
20   ~TQueueItem();
21   void SetNext(std::shared_ptr<TQueueItem<T>> item);
22   std::shared_ptr<TQueueItem<T>> GetNext();
23   std::shared_ptr<T> Get();
24   void* operator new (size_t size);
25   void operator delete (void *p);
26   template <class A> friend std::ostream& operator<<(std::ostream& os, const
         TQueueItem<A>& obj);
27 private:
28   std::shared_ptr<T> figure;
29   std::shared_ptr<TQueueItem<T>> next;
30   static TAllocationBlock queueitem_allocator;
31 };
```

Появился сам линейный аллокатор, и он встраивается как статическая переменная в обёртку над элементами контейнера - TQueueItem.

Выводы

В этой работе был создан аллокатор памяти, помогающий оптимизировать выделение и освобождение памяти. Свободные блоки хранятся в аллокаторе в контейнере второго уровня – в массиве.

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: С. Д. Старчеус
Преподаватель: А. А. В. Поповкин
Группа: М8О-206Б

Вариант: 25
Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №7

Цель:

- Создание сложных динамических структур данных
- Закрепление принципа OCP

Задача: необходимо реализовать динамическую структуру данных – «Хранили- ще объектов» и алгоритм работы с неи. «Хранили- ще объектов» представляет собои контеинер, одного из следующих видов (Контейнер 1-го уровня).
Каждым элементом контеинера, в свою, является динамическои структурои данных одного из следующих видов (Контейнер 2-го уровня).
При этом должно выполняться правило, что количество объектов в контеинере второго уровня не больше 5. Т.е. если нужно хранить больше 5 объектов, то создается еще один контеинер второго уровня.

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

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

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". 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 C++, std::list (шаблонный класс) является контейнером, а объект его класса-конкретизации, как например, std::list<int> mylist является коллекцией.

Исходный код

```
1  template <class T>
2  class TVector {
3  public:
4    TVector();
5    ~TVector();
6    bool Insert(std::shared_ptr<T> &&item);
7    template<class A> void Remove_type();
8    void Remove_lesser(double area);
9    std::shared_ptr<T> Remove(int i);
10   std::shared_ptr<T> Get(int i) const;
11   int Size();
12   template <class A> friend std::ostream& operator<<(std::ostream& os, const TVector<A
         >& queue);
13 private:
14   std::shared_ptr<T> arr[5];
15   int size;
16 };
17
18 template <class T>
19 class TQueue {
20 public:
21   TQueue();
22   ~TQueue();
23   void Insert(std::shared_ptr<Figure> &&figure);
24   std::shared_ptr<Figure> Remove(int i, int j);
25   void Remove_lesser(double area);
26   template <class A> void Remove_type();
27   void Push(std::shared_ptr<T> &&figure);
28   bool IsEmpty();
29   std::shared_ptr<T> Pop();
30   TIterator<TQueueItem<T>,T> begin();
31   TIterator<TQueueItem<T>,T> end();
32   template <class A> friend std::ostream& operator<<(std::ostream& os, const TQueue<A
         >& queue);
33 private:
34   std::shared_ptr<TQueueItem<T>> first, last;
35   uint size;
36 };
```

Появился TVector - массив с фиксированным количеством элементов. Так же добавились новые методы в TQueue - Remove_lesser, Remove_type<A>, вызывающие такие же методы массива в цикле (по сути являются обёрткой для массивов).

Выводы

В этой работе был реализован контейнер второго уровня – вектор. Векторы хранятся в очереди, и контролируют количество элементов, содержащихся в них. Так же для вектора реализованы два метода - удаление по типу и по площади, для очереди реализованы соответствующие обёрточные методы вызывающие методы вектора в цикле.

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: С. Д. Старчеус
Преподаватель: А. А. В. Поповкин
Группа: М8О-206Б

Вариант: 25
Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №8

Цель:

- Знакомство с параллельным программированием в C++

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

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер
- Распечатывать содержимое контейнера
- Удалять фигуры из контейнера по критериям
- Проводить сортировку контейнера

Описание

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

- 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.

Исходный код

```
template <class T>
void TQueue<T>::Quicksort() {
  if (this->size < 2) {
    return;
  }

  TQueue<T> lesser, greater;

  std::shared_ptr<T> pivot_figure = this->Pop();
  double pivot = pivot_figure->Area();

  std::shared_ptr<T> figure;
  while (!this->IsEmpty()) {
    figure = this->Pop();
    if (figure->Area() < pivot) {
      lesser.Push(std::move(figure));
    }
    else {
      greater.Push(std::move(figure));
    }
  }

  lesser.Quicksort();
  greater.Quicksort();

  while (!lesser.IsEmpty()) {
    this->Push(lesser.Pop());
  }

  this->Push(std::move(pivot_figure));

  while (!greater.IsEmpty()) {
    this->Push(greater.Pop());
  }
}


template <class T>
void TQueue<T>::Quicksort_parallel() {
  if (this->size < 2) {
    return;
  }

  TQueue<T> lesser;
  TQueue<T> greater;

  std::shared_ptr<T> pivot_figure = this->Pop();
```

```
48    double pivot = pivot_figure->Area();
49
50    std::shared_ptr<T> figure;
51    while (!this->IsEmpty()) {
52      figure = this->Pop();
53      if (figure->Area() < pivot) {
54        lesser.Push(std::move(figure));
55      }
56      else {
57        greater.Push(std::move(figure));
58      }
59    }
60
61    std::packaged_task<void()>
62      task1(std::bind(&TQueue<T>::Quicksort_parallel, &lesser));
63    std::packaged_task<void()>
64      task2(std::bind(&TQueue<T>::Quicksort_parallel, &greater));
65
66    auto futureGreater = task1.get_future();
67    auto futureLesser = task2.get_future();
68
69    std::thread(std::move(task1)).join();
70    std::thread(std::move(task2)).join();
71    futureLesser.get();
72    futureGreater.get();
73
74    while (!lesser.IsEmpty()) {
75      this->Push(lesser.Pop());
76    }
77
78    this->Push(std::move(pivot_figure));
79
80    while (!greater.IsEmpty()) {
81      this->Push(greater.Pop());
82    }
83  }
```

Сортировка реализована прямо в лоб - берём первый элемент в качестве опорного, далее раскидываем все элементы в две очереди - меньшие элемента и большие либо равные, далее рекурсивно их сортируем, затем объединяем. Изменений в параллельной версии сортировки немного - всё что изменилось - это запуск сортировок в новом потоке. Так как срок жизни локальных очередей больше, чем рекурсивных вызовов, то их можно передавать без опасения получить segmentation_fault.

Выводы

Благодаря этой лабораторной работе я получил 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. В моём случае все рекурсивные вызовы сортировки выполнялись в отдельном потоке.

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: С. Д. Старчеус
Преподаватель: А. А. В. Поповкин
Группа: М8О-206Б

Вариант: 25
Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №9

Цель:

- 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 контенйром 1-го уровня:
  - Генерация фигур со случайным значением параметров
  - Печать контейнера на экран
  - Удаление элементов со значением площади меньше определенного числа
- В контенер второго уровня поместить цепочку команд
- Реализовать цикл, который проходит по всем командам в контенере второго уровня и выполняет их, применяя к контейнеру первого уровня

Для создания потоков использовать механизмы:

- future
- packaged_task/async

Для обеспечения потоко-безопасности структур данных использовать:

- mutex
- lock_guard

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер
- Распечатывать содержимое контейнера
- Удалять фигуры из контейнера по критериям
- Проводить сортировку контейнера

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.

Исходный код

```
int main() {
  Print_usage();
  TVector<void(*)(TQueue<Figure>&)> commands;
  TQueue<Figure> q;

  int flag;
  while(scanf("%d", &flag) == 1) {
    switch(flag) {
      case 1:
        commands.Push(
          [](TQueue<Figure> &q) -> void {
            int figure = rand() % 3;

            int size_a = rand() % 128 + 1;
            int size_b = rand() % 128 + 1;
            int size_c = rand() % (size_a + size_b - 1) + 1;

            if (size_a > size_b) {
              if (size_c < size_a - size_b + 1) {
                size_c = size_a - size_b + 1;
              }
            }
            else {
              if (size_c < size_b - size_a + 1) {
                size_c = size_b - size_a + 1;
              }
            }

            std::shared_ptr<Figure> fig;
            switch (figure) {
              case 0:
                fig = std::shared_ptr<Figure>(new Triangle(size_a, size_b, size_c));
                break;
              case 1:
                fig = std::shared_ptr<Figure>(new Rect(size_a, size_b));
                break;
              case 2:
                fig = std::shared_ptr<Figure>(new Square(size_a));
                break;
            }
            q.Push(std::move(fig));
            cout << endl;
          }
        );
        break;
      case 2:
        commands.Push(
```

```
48          [](TQueue<Figure> &q) -> void {
49            cout << endl;
50            cout << "Queue:\n";
51            for (auto i : q) {
52              cout << "[" <<*i << "]" << endl;
53            }
54            cout << endl;
55          }
56        );
57        break;
58      case 3:
59        commands.Push(
60          [](TQueue<Figure> &q) -> void {
61            double area;
62            cout << "Enter area:\n";
63            scanf("%lf", &area);
64
65            std::shared_ptr<TQueueItem<Figure>> prev = q.first;
66            std::shared_ptr<TQueueItem<Figure>> iter = q.first;
67
68            while (iter != nullptr) {
69              if (iter->Get()->Area() < area) {
70                if (iter == q.first) {
71                  q.first = iter->GetNext();
72                  if (iter == q.last) {
73                    q.last = nullptr;
74                  }
75                }
76                else if (iter == q.last) {
77                  prev->SetNext(nullptr);
78                  q.last = prev;
79                }
80                else {
81                  prev->SetNext(iter->GetNext());
82                }
83              }
84
85              prev = iter;
86              iter = iter->GetNext();
87            }
88          }
89        );
90        break;
91    }
92  }
93
94  for (int i = 0; i < commands.Size(); i++) {
95    commands[i](q);
96  }
```

```
97 }
```

Всё изменение - в коде main(): создаём вектор, добавляем в него комманды, запрошенные пользователем, выполняем.

Выводы

В этой работе я узнал о лямбда-выражениях. Теперь действия над контейнером первого уровня генерируются в виде команд, которые помещаются в контейнер второго уровня. Анонимные выражения могут быть полезными в ряде случаев, так как они не загрязняют пространство имён, лишая пользователя лишней возможности напрячь свой мозг.