**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 8**

Тема: Асинхронное программирование

Студент: Ефимов Александр

Группа: 80-201

Преподаватель: Чернышов Л.Н.

Дата: 24.12.2019

Оценка:

Москва, 2019

**Постановка задачи**

Программа должна:

1. Осуществлять ввод из стандартного ввода данных фигур, согласно варианту задания;
2. Программа должна создавать классы, соответствующие введенным данных фигур;
3. Программа должна содержать внутренний буфер, в который помещаются фигуры. Для создания буфера допускается использовать стандартные контейнеры STL. Размер буфера задается параметром командной строки. Например, для буфера размером 10 фигур: oop\_exercise\_08 10
4. При накоплении буфера они должны запускаться на асинхронную обработку, после чего буфер должен очищаться;
5. Обработка должна производиться в отдельном потоке;
6. Реализовать два обработчика, которые должны обрабатывать данные буфера:
   1. Вывод информации о фигурах в буфере на экран;
   2. Вывод информации о фигурах в буфере в файл. Для каждого буфера должен создаваться файл с уникальным именем.
7. Оба обработчика должны обрабатывать каждый введенный буфер. Т.е. после каждого заполнения буфера его содержимое должно выводиться как на экран, так и в файл.
8. В программе должно быть ровно два потока (thread). Один основной (main) и второй для обработчиков;
9. В программе должен явно прослеживаться шаблон Publish-Subscribe. Каждый обработчик должен быть реализован как отдельный подписчик.
10. Реализовать в основном потоке (main) ожидание обработки буфера в потоке-обработчике. Т.е. после отправки буфера на обработку основной поток должен ждать, пока поток обработчик выведет данные на экран и запишет в файл.

Вариант 1: *Треугольник, прямоугольник, квадрат*.

1. **Описание программы**

Шаблон квадрата написан в *Shape.h*, вместе шаблонными функциями *Area*, *Centre* и *Print*, которые ищут его площадь, центр фигуры и печатают его соответственно.

В *main.cpp* содержится меню, позволяющее работать с вектором, содержащим в себе общие указатели на абстрактный класс *Square*, тип точек которого **int**. Перед входом в меню создается отдельный поток обработчиков для печати буфера. После создания, поток блокируется и ожидает, пока не придет сигнал из **main** о заполненности буфера (т.е. вектора), после чего он печатает содержимое и записывает его в файл. Вектор также выполняет функцию очереди сообщений.

1. **Набор тестов**

* **Test\_01.txt**

Полное заполнение буфера размером в 2 фигуры

1. **Результаты выполнения тестов**

Поток обработчиков выводит буфер и чистит его, что соответствует ожиданиям.

**test\_01**

|  |
| --- |
| rookstar@Refrigerator:~/Git/oop\_exercise\_8$ ./oop\_exercise\_8 2  Max buffer size is set to 2  1. Add shape  2. Print all elements  3. Delete element  4. Print this menu  5. Clear shapes  0. Exit  -----------------------  Menu choice: 1  1. Triangle  2. Rectangle  3. Square  Any other number returns to menu  Pick the type of shape: 1  Input point coordinates  A: 0 0  B: 0 3  C: 3 3  -----------------------  Menu choice: 2  1. Triangle: (0 , 0); (0 , 3); (3 , 3); Centre = (1 , 2); Area = 4.5  -----------------------  Menu choice: 1  1. Triangle  2. Rectangle  3. Square  Any other number returns to menu  Pick the type of shape: 2  Input point coordinates clockwise or counter clockwise  A: 0 0  B: 0 4  C: 5 4  D: 5 0  //=========================\\  || Beginning buffer output ||  \\=========================//  1. Triangle: (0 , 0); (0 , 3); (3 , 3); Centre = (1 , 2); Area = 4.5  2. Rectangle: (0 , 0); (0 , 4); (5 , 4); (5 , 0); Centre = (2.5 , 2); Area = 20  -----------------------  Menu choice: 2  No shapes inserted  -----------------------  Menu choice: |

1. **Листинг программы**

**Shape.h**

|  |
| --- |
| #ifndef SHAPE\_H  #define SHAPE\_H  #include <iostream>  #include <utility>  #include <cmath>  #include <vector>  typedef std::pair<double, double> DoublePoint;  //---------------------------------------------------------------------  // Necessary for friend declaration  //---------------------------------------------------------------------  template <typename T>  class Shape;  template <typename T>  std::ostream& operator << (std::ostream& os, const Shape<T>& shape);  //---------------------------------------------------------------------  template <typename T>  class Shape {  public:  friend std::ostream& operator << <T> (std::ostream& os, const Shape& shape);  typedef std::pair<T, T> Point;    DoublePoint getCentre() const  { return centre; }  double getArea() const  { return area; }  bool getAbstract() const  { return abstract; }  int getType() const  { return type; }  std::vector<Point> getPoints() const  { return p; }  protected:  DoublePoint centre;  double area = 0;  bool abstract = false;  int type;  std::vector<Point> p;  private:  virtual void Centre() = 0;  virtual void Area() = 0;  virtual std::ostream& print(std::ostream&) const = 0;  };  template <typename T>  class Triangle : public Shape<T> {  public:  using typename Shape<T>::Point;  using Shape<T>::getCentre;  using Shape<T>::getArea;  using Shape<T>::centre;  using Shape<T>::area;  using Shape<T>::abstract;  using Shape<T>::type;  using Shape<T>::p;  Triangle(Point a, Point b, Point c);  Triangle(std::istream& is);  void Centre() override;  void Area() override;  std::ostream& print(std::ostream&) const override;  };  template <typename T>  class Rectangle : public Shape<T> {  public:  using typename Shape<T>::Point;  using Shape<T>::getCentre;  using Shape<T>::getArea;  using Shape<T>::centre;  using Shape<T>::area;  using Shape<T>::abstract;  using Shape<T>::type;  using Shape<T>::p;  Rectangle(Point a, Point b, Point c, Point d);  Rectangle(std::istream& is);  void Centre() override;  void Area() override;  std::ostream& print(std::ostream&) const override;  };  template <typename T>  class Square : public Shape<T> {  public:  using typename Shape<T>::Point;  using Shape<T>::getCentre;  using Shape<T>::getArea;  using Shape<T>::centre;  using Shape<T>::area;  using Shape<T>::abstract;  using Shape<T>::type;  using Shape<T>::p;  Square(Point a, Point b, Point c, Point d);  Square(std::istream& is);  void Centre() override;  void Area() override;  std::ostream& print(std::ostream&) const override;  };  //-----------------------------------------------------------------------------------------------------------  // Geometrical correctness  //-----------------------------------------------------------------------------------------------------------  bool DoubleEqual(double lhs, double rhs)  {  const double EPS = 0.00001;  if (lhs > rhs)  return (lhs - rhs) < EPS ? true : false;  else  return (rhs - lhs) < EPS ? true : false;  }  bool IsRight(std::pair<double, double> a, std::pair<double, double> b, std::pair<double, double> c)  {  std::pair<double, double> vec1 = { b.first - a.first, b.second - a.second };  std::pair<double, double> vec2 = { c.first - a.first, c.second - a.second };  double result = vec1.first \* vec2.first + vec1.second \* vec2.second;  if (DoubleEqual(result, 0)) return true;  return false;  }  bool IsRight(std::pair<int, int> a, std::pair<int, int> b, std::pair<int, int> c)  {  std::pair<int, int> vec1 = { b.first - a.first, b.second - a.second };  std::pair<int, int> vec2 = { c.first - a.first, c.second - a.second };  int result = vec1.first \* vec2.first + vec1.second \* vec2.second;  if (result == 0) return true;  return false;  }  //-----------------------------------------------------------------------------------------------------------  // Check if shape is a rectangle  //-----------------------------------------------------------------------------------------------------------  template <class T>  bool IsTriangle(std::vector< std::pair<T, T> > &p)  {  return !DoubleEqual  ( ((p[1].first - p[0].first) \* (p[2].second - p[0].second) - (p[1].second - p[0].second) \* (p[2].first - p[0].first)), 0 );  }  //-----------------------------------------------------------------------------------------------------------  // Check if shape is a rectangle  //-----------------------------------------------------------------------------------------------------------  template <class T>  bool IsRectangle(const std::vector< std::pair<T, T> > &p)  {  std::pair<T, T> null(0, 0);  if (p[0] == null && p[1] == null && p[2] == null && p[3] == null) return false;  if (  IsRight(p[2], p[1], p[3]) &&  IsRight(p[3], p[2], p[0]) &&  IsRight(p[1], p[0], p[2]) &&  IsRight(p[0], p[3], p[1])  ) return true;  return false;  }  //-----------------------------------------------------------------------------------------------------------  // Check if shape is a square  //-----------------------------------------------------------------------------------------------------------  template <class T>  double distance(const std::pair<T, T> a, std::pair<T, T> b)  {  return (b.first - a.first) \* (b.first - a.first) + (b.second - a.second) \* (b.second - a.second);  }  template <class T>  bool IsSquare(const std::vector< std::pair<T, T> > &p)  {  if (  IsRectangle(p) &&  DoubleEqual(distance(p[0], p[1]), distance(p[1], p[2])) &&  DoubleEqual(distance(p[1], p[2]), distance(p[2], p[3])) &&  DoubleEqual(distance(p[2], p[3]), distance(p[3], p[0])) &&  DoubleEqual(distance(p[3], p[0]), distance(p[0], p[1]))  ) return true;  return false;  }  //-----------------------------------------------------------------------------------------------------------  // Polymorphism safe out operator overload  //-----------------------------------------------------------------------------------------------------------  template <class T>  std::ostream& operator << (std::ostream& os, const Shape<T>& shape)  {  shape.print(os);  return os;  }  template <class T>  std::ostream& operator << (std::ostream& os, const std::pair<T,T>& p)  {  os << '(' << p.first << " , " << p.second << ')';  return os;  }  //-----------------------------------------------------------------------------------------------------------  // Triangle functions  //-----------------------------------------------------------------------------------------------------------  template <class T>  Triangle<T>::Triangle(Point a, Point b, Point c)  {  p.push\_back(a);  p.push\_back(b);  p.push\_back(c);  if (!IsTriangle(p)) { abstract = true; }  Centre();  Area();  type = 0;  }  template <class T>  Triangle<T>::Triangle(std::istream& is)  {  Point points[3];  std::cout << "Input point coordinates\nA: ";  is >> points[0].first >> points[0].second;  std::cout << "B: ";  is >> points[1].first >> points[1].second;  std::cout << "C: ";  is >> points[2].first >> points[2].second;  p.push\_back(points[0]);  p.push\_back(points[1]);  p.push\_back(points[2]);  if (!IsTriangle(p)) { abstract = true; }  Centre();  Area();  }  template <class T>  void Triangle<T>::Centre()  {  for (int i = 0; i < 3; ++i)  {  centre.first += p[i].first;  centre.second += p[i].second;  }  centre.first /= 3;  centre.second /= 3;  }  template <class T>  void Triangle<T>::Area()  {  area = (double) (  p[0].first \* (p[1].second - p[2].second)  + p[1].first \* (p[2].second - p[0].second)  + p[2].first \* (p[0].second - p[1].second)  ) / 2;  if (area < 0) area = -area;  }  template <class T>  std::ostream& Triangle<T>::print(std::ostream& os) const  {  if (abstract)  {  os << "Abstract: ";  }  else  {  os << "Triangle: ";  }  os << p[0] << "; " << p[1] << "; " << p[2]  << "; Centre = " << getCentre()  << "; Area = " << getArea();  return os;  }  //-----------------------------------------------------------------------------------------------------------  // Rectangle functions  //-----------------------------------------------------------------------------------------------------------  template <class T>  Rectangle<T>::Rectangle(Point a, Point b, Point c, Point d)  {  p.push\_back(a);  p.push\_back(b);  p.push\_back(c);  p.push\_back(d);  if (!IsRectangle(p)) { abstract = true; }  Centre();  Area();  type = 1;  }  template <class T>  Rectangle<T>::Rectangle(std::istream& is)  {  Point points[4];  std::cout << "Input point coordinates clockwise or counter clockwise\nA: ";  is >> points[0].first >> points[0].second;  std::cout << "B: ";  is >> points[1].first >> points[1].second;  std::cout << "C: ";  is >> points[2].first >> points[2].second;  std::cout << "D: ";  is >> points[3].first >> points[3].second;  p.push\_back(points[0]);  p.push\_back(points[1]);  p.push\_back(points[2]);  p.push\_back(points[3]);  if (!IsRectangle(p)) { abstract = true; }    Centre();  Area();  type = 1;  }  template <class T>  void Rectangle<T>::Centre()  {  for (int i = 0; i < 4; ++i)  {  centre.first += p[i].first;  centre.second += p[i].second;  }  centre.first /= 4;  centre.second /= 4;  }  template <class T>  void Rectangle<T>::Area()  {  area = sqrt( distance(p[0], p[1]) \* distance(p[1], p[2]) );  }  template <class T>  std::ostream& Rectangle<T>::print(std::ostream& os) const  {  if (abstract)  {  os << "Abstract: ";  }  else  {  os << "Rectangle: ";  }    os << p[0] << "; " << p[1] << "; " << p[2] << "; " << p[3]  << "; Centre = " << getCentre()  << "; Area = " << getArea();  return os;  }  //-----------------------------------------------------------------------------------------------------------  // Square functions  //-----------------------------------------------------------------------------------------------------------  template <class T>  Square<T>::Square(Point a, Point b, Point c, Point d)  {  p.push\_back(a);  p.push\_back(b);  p.push\_back(c);  p.push\_back(d);  if (!IsSquare(p)) { abstract = true; }  Centre();  Area();  type = 2;  }  template <class T>  Square<T>::Square(std::istream& is)  {  Point points[4];  std::cout << "Input point coordinates clockwise or counter clockwise\nA: ";  is >> points[0].first >> points[0].second;  std::cout << "B: ";  is >> points[1].first >> points[1].second;  std::cout << "C: ";  is >> points[2].first >> points[2].second;  std::cout << "D: ";  is >> points[3].first >> points[3].second;  p.push\_back(points[0]);  p.push\_back(points[1]);  p.push\_back(points[2]);  p.push\_back(points[3]);  if (!IsSquare(p)) { abstract = true; }  Centre();  Area();  type = 2;  }  template <class T>  void Square<T>::Centre()  {  for (int i = 0; i < 4; ++i)  {  centre.first += p[i].first;  centre.second += p[i].second;  }  centre.first /= 4;  centre.second /= 4;  }  template <class T>  void Square<T>::Area()  {  area = sqrt( distance(p[0], p[1]) \* distance(p[1], p[2]) );  }  template <class T>  std::ostream& Square<T>::print(std::ostream& os) const  {  if (abstract)  {  os << "Abstract: ";  }  else  {  os << "Square: ";  }  os << p[0] << "; " << p[1] << "; " << p[2] << "; " << p[3]  << "; Centre = " << getCentre()  << "; Area = " << getArea();  return os;  }  //-----------------------------------------------------------------------------------------------------------  // Compare points  //-----------------------------------------------------------------------------------------------------------  bool operator == (std::pair<int, int> lhs, std::pair<int, int> rhs)  {  return lhs.first == rhs.first && lhs.second == rhs.second;  }  bool operator == (std::pair<double, double> lhs, std::pair<double, double> rhs)  {  return DoubleEqual(lhs.first, rhs.first) && DoubleEqual(lhs.second, rhs.second);  }  #endif |

**Factory.h**

|  |
| --- |
| #ifndef FACTORY\_H  #define FACTORY\_H  #include <iostream>  #include "Shape.h"  template <typename T>  class Factory {  private:  enum Type { triangle, rectangle, square };  public:  static std::shared\_ptr<Shape<T>> create(int num, std::istream &is)  {  switch (num)  {  case triangle:  return std::make\_shared< Triangle<T> >(is);  break;    case rectangle:  return std::make\_shared< Rectangle<T> >(is);  break;    case square:  return std::make\_shared< Square<T> >(is);  break;  default:  throw std::logic\_error("No such number");  break;  }  }  static std::shared\_ptr<Shape<T>> create(int num, std::pair<T, T> (&p)[4])  {  switch (num)  {  case triangle:  return std::make\_shared< Triangle<T> >(p[0], p[1], p[2]);  break;    case rectangle:  return std::make\_shared< Rectangle<T> >(p[0], p[1], p[2], p[3]);  break;    case square:  return std::make\_shared< Square<T> >(p[0], p[1], p[2], p[3]);  break;  default:  throw std::logic\_error("No such number");  break;  }  }  };  #endif |

**main.cpp**

|  |
| --- |
| //================================================//  // Ефимов А.В. //  // M8O-201Б //  // Вариант 1: Треугольник, Прямоугольник, Квадрат //  //================================================//  #include <iostream>  #include <fstream>  #include <string>  #include <exception>  #include <vector>  #include <memory>  #include <thread>  #include <mutex>  #include <condition\_variable>  #include "Shape.h"  #include "Factory.h"  using namespace std;  using WorkingType = int;  std::vector< std::shared\_ptr<Shape<WorkingType>> > shapes;  std::mutex mtx;  std::condition\_variable sync;  void getMenu()  {  cout  << "\n1. Add shape\n"  << "2. Print all elements\n"  << "3. Delete element\n"  << "4. Print this menu\n"  << "5. Clear shapes\n"  << "0. Exit" << endl;  }  int GetNum(void)  {  int k;  bool got = false;  while (!got)  {  cin >> k;  if (!cin.good())  {  cout << "Bad input ignored: " << flush;  cin.clear();  cin.ignore(256, '\n');  }  else got = true;  }  return k;  }  void save(const std::string& filePath)  {  std::ofstream outfile;  outfile.open(filePath, std::ios::out);  if (outfile.fail())  {  throw std::runtime\_error("File open operation failed");  }  std::vector< std::pair<WorkingType,WorkingType> > points;  outfile << shapes.size() << '\t';  for(std::shared\_ptr<Shape<WorkingType>> e : shapes)  {  points = e->getPoints();  outfile << e->getType() << '\t';  switch(e->getType())  {  case 0:  for (int i = 0; i < 3; ++i)  {  outfile << points[i].first << '\t' << points[i].second << '\t';  }  break;    case 1:  case 2:  for (int i = 0; i < 4; ++i)  {  outfile << points[i].first << '\t' << points[i].second << '\t';  }  break;  }  }  outfile.close();  }    void printer (int& work, const int size)  {  std::string path;  int saveNum = 1, figNum = 1;    while(true)  {  std::unique\_lock<std::mutex> lock(mtx);  sync.wait(lock, [&] { return (shapes.size() == size || !work); });  if (!work) break;  std::cout << "//=========================\\\\\n|| Beginning buffer output ||\n\\\\=========================//\n" << std::endl;  path = "Saves/Buffer" + std::to\_string(saveNum);  try {  save(path);  }  catch (std::runtime\_error &e) {  std::cout << "DURING SAVING: File failed to open" << std::endl;  }  ++saveNum;  for (std::shared\_ptr< Shape<WorkingType> > e : shapes)  {  std::cout << figNum++ << ". " << \*e << "\n";  }  figNum = 1;  shapes.clear();  sync.notify\_one();  }  }  int main(int argc, char\* argv[])  {  int size = 5;    if (argc > 1)  {  try  {  size = std::stoi(argv[1]);  if (size <= 0) throw std::invalid\_argument("stoi");  }  catch (std::invalid\_argument& e)  {  std::cerr << "Invalid parameter value given" << std::endl;  return 0;  }  }  std::pair<WorkingType, WorkingType> points[4];  int figNum = 1, k = 1, work = 1;  std::thread worker (printer, std::ref(work), size);  std::cout << "\nMax buffer size is set to " << size;  getMenu();  do  {  std::unique\_lock<std::mutex> lock(mtx);  cout << "-----------------------\nMenu choice: ";  try  {  k = GetNum();  if (!cin) throw runtime\_error("Failed to get input");  switch (k)  {  case 1:  cout  << "1. Triangle\n"  << "2. Rectangle\n"  << "3. Square\n"  << "Any other number returns to menu\n"  << "Pick the type of shape: ";  k = GetNum();  if (k >= 1 && k <= 3)  {  --k;  switch (k)  {  case 0:  std::cout << "Input point coordinates\nA: ";  cin >> points[0].first >> points[0].second;  std::cout << "B: ";  cin >> points[1].first >> points[1].second;  std::cout << "C: ";  cin >> points[2].first >> points[2].second;  break;    case 1:  // Intentionally empty  case 2:  std::cout << "Input point coordinates clockwise or counter clockwise\nA: ";  std::cin >> points[0].first >> points[0].second;  std::cout << "B: ";  std::cin >> points[1].first >> points[1].second;  std::cout << "C: ";  std::cin >> points[2].first >> points[2].second;  std::cout << "D: ";  std::cin >> points[3].first >> points[3].second;  break;    default:  break;  }  shapes.push\_back(Factory<WorkingType>::create(k, points));    if (shapes.size() == size) {  std::cout << "\n";  sync.notify\_one();  sync.wait(lock, [&] { return shapes.empty(); });  }  }  k = 1;  break;  case 2:  if (shapes.empty())  {  cout << "No shapes inserted" << endl;  }  else  {  for (std::shared\_ptr< Shape<WorkingType> > e : shapes)  {  std::cout << figNum++ << ". " << \*e << "\n";  }  figNum = 1;  }  break;  case 3:  if (shapes.empty()) throw runtime\_error("No shapes to delete");  cout << "Number of the shape: ";  k = GetNum();  if (k < 1 || k > shapes.size()) throw runtime\_error("Incorrect index");  shapes.erase(shapes.begin() + k - 1);  break;  case 4:  getMenu();  break;  case 5:  shapes.clear();  std::cout << "All shapes deleted" << std::endl;  break;  case 0:  work = 0;  break;  default:  cout << "No such number" << endl;  }  }  catch (runtime\_error& err)  {  cerr << err.what() << endl;  }  catch (logic\_error& err)  {  cerr << err.what() << endl;  }  catch (...)  {  cerr << "Exception thrown, but undefined." << endl;  }  } while (work);  sync.notify\_one();  worker.join();  return 0;  } |

1. **Выводы**

Создание отдельных поток разделяет работу и помогает ускорить программу. В частности, если сократить работу потока до вывода в файл, то можно не ожидать завершения его вывода и продолжать программу, что создает параллельность.

1. **Список литературы**

* Описание методов conditional\_variable (Справочник по языку C++) [Электронный ресурс].

URL:<https://en.cppreference.com/w/cpp/thread/condition_variable>

(Дата обращения: 22/12/2019)

* Примеры использования conditional\_variable [Электронный ресурс].

URL:<https://thispointer.com/c11-multithreading-part-7-condition-variables-explained/>

(Дата обращения: 22/12/2019)