Реализация класса треугольников

Для некоторого множества заданных координатами своих вершин треугольников найти треугольник максимальной площади (если максимальную площадь имеют несколько треугольников, то найти первый из них). Предусмотреть возможность перемещения треугольников и проверки включения одного треугольника в другой.

Для реализации этой задачи составить описание класса треугольников на плоскости. Предусмотреть возможность объявления в клиентской программе (`main`) экземпляра треугольника с заданными координатами вершин. Предусмотреть наличие в классе методов, обеспечивающих: 1) перемещение треугольников на плоскости; 2) определение отношения > для пары заданных треугольников (мера сравнения — площадь треугольников); 3) определение отношения включения типа: «Треугольник 1 входит в (не входит в) Треугольник 2».

Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Отметим, что сейчас мы еще не готовы провести полномасштабную объектно-ориентированную декомпозицию программы. Для этого нам не хватает знаний, которые будут получены на втором семинаре. Поэтому применим гибридный подход: разработку главного клиента `main()` проведем по технологии функциональной декомпозиции, а функции-серверы, вызываемые из `main()`, будут использовать объекты.

Начнем с выявления основных понятий/классов для нашей задачи. Первый очевидный класс `Triangle` необходим для представления треугольников. Из нескольких способов определения треугольника на плоскости выберем самый простой — через три точки, задающие его вершины. Сделанный выбор опирается на новое понятие, отсутствующее в условии задачи, — понятие *точки*. Точку на плоскости можно представить различными способами; остановимся на наиболее популярном — с помощью пары вещественных чисел, задающих координаты точки по осям x и y.

Таким образом, с понятием точки связывается как минимум пара атрибутов. В принципе, этого уже достаточно, чтобы подумать о создании класса `Point`. Если же представить, что можно делать с объектом типа точки — например, перемещать ее на плоскости или определять ее вхождение в заданную фигуру, — то становится ясным, что такой класс `Point` будет полезен.

Итак, объектно-ориентированная декомпозиция дала нам два класса: `Triangle` и `Point`. Как эти два класса должны быть связаны друг с другом? На втором семинаре мы будем более подробно рассматривать взаимоотношения между классами. Пока же отметим, что наиболее часто для двух классов имеет место одно из двух:

- отношение наследования (отношение *is a*);
- отношение агрегации или включения (отношение *has a*).

Если класс `B` является «частным случаем» класса `A`, то говорят, что `B` *is a* `A` (например, класс треугольников есть частный вид класса многоугольников: `Triangle` *is a* `Polygon`).

Если класс `A` содержит в себе объект класса `B`, то говорят, что `A` *has a* `B` (например, класс треугольников может содержать в себе объекты класса точек: `Triangle` *has a* `Point`).

Теперь уточним один вопрос: *в каком порядке* мы будем перечислять точки, задавая треугольник? Порядок перечисления вершин важен для нас потому, что в дальнейшем, решая подзадачу определения отношения включения одного треугольника в другой, мы будем рассматривать стороны треугольника как *векторы*. Условимся, что вершины треугольника перечисляются в направлении *по часовой стрелке*.

Займемся теперь основным клиентом — `main()`. Здесь мы применяем функциональную декомпозицию, или технологию нисходящего проектирования. В соответствии с данной технологией основной алгоритм представляется как последовательность нескольких подзадач. Каждой подзадаче соответствует вызываемая серверная функция. На начальном этапе проектирования тела этих функций могут быть заполнены «заглушками» — отладочной печатью. Если при этом в какой-то серверной функции окажется слабое сцепление, то она в свою очередь разбивается на несколько подзадач.

То же самое происходит и с классами, используемыми в программе: по мере реализации подзадач они пополняются необходимыми для этого методами. Такая технология облегчает отладку и поиск ошибок, сокращая общее время разработки программы.

В соответствии с описанной технологией мы представим решение задачи 1.2 как последовательность нескольких этапов. Иногда как бы по забывчивости мы будем допускать некоторые «ляпы», поскольку в процессе поиска ошибок можно глубже понять нюансы программирования с классами.

На первом этапе мы напишем код для начального представления классов `Point` и `Triangle`, достаточный для того, чтобы создать несколько объектов типа `Triangle` и реализовать первый пункт меню — вывод всех объектов на экран.

Этап 1

```
//////////////////////////////////////////////////////////
// Проект Task1_2
//////////////////////////////////////////////////////////
// Point.h
#ifndef POINT_H
#define POINT_H
```

```
class Point {
public:
    // Конструктор
    Point(double _x = 0, double _y = 0) : x(_x), y(_y) {}
    // Другие методы
    void Show() const;
public:
    double x, y;
};
#endif /* POINT_H */
////////////////////////////////////////////////////////////
// Point.cpp
#include <iostream>
#include "Point.h"
using namespace std;

void Point::Show() const {
    cout << "  (" << x << ", " << y << ")";
}
////////////////////////////////////////////////////////////
// Triangle.h
#ifndef TRIANGLE_H
#define TRIANGLE_H

#include "Point.h"

class Triangle {
public:
    Triangle(Point, Point, Point, const char*);  // конструктор
    Triangle(const char*);      // конструктор пустого (нулевого) треугольника
    ~Triangle();                            // деструктор
    Point Get_v1() const { return v1; }     // Получить значение v1
    Point Get_v2() const { return v2; }     // Получить значение v2
    Point Get_v3() const { return v3; }     // Получить значение v3
    char* GetName() const { return name; } // Получить имя объекта
    void  Show() const;                     // Показать объект
    void  ShowSideAndArea() const;          // Показать стороны и площадь объекта
public:
    static int count;       // количество созданных объектов
private:
    char* objID;            // идентификатор объекта
    char* name;             // наименование треугольника
    Point v1, v2, v3;       // вершины
    double a;               // сторона, соединяющая v1 и v2
    double b;               // сторона, соединяющая v2 и v3
    double c;               // сторона, соединяющая v1 и v3
};
#endif /* TRIANGLE_H */
```

```
////////////////////////////////////////////////////////////
//Triangle.cpp
// Реализация класса Triangle
#include <math.h>
#include <iostream>
#include <iomanip>
#include <cstring>
//#include "CyrIOS.h"                           // for Visual C++ 6.0
#include "Triangle.h"
using namespace std;

// Конструктор
Triangle::Triangle(Point _v1, Point _v2, Point _v3, const char* ident)
     : v1(_v1), v2(_v2), v3(_v3) {
    char buf[16];

    objID = new char[strlen(ident) + 1];
    strcpy(objID, ident);

    count++;
    sprintf(buf, "Треугольник %d", count);
    name = new char[strlen(buf) + 1];
    strcpy(name, buf);
    a = sqrt((v1.x - v2.x) * (v1.x - v2.x) + (v1.y - v2.y) * (v1.y - v2.y));
    b = sqrt((v2.x - v3.x) * (v2.x - v3.x) + (v2.y - v3.y) * (v2.y - v3.y));
    c = sqrt((v1.x - v3.x) * (v1.x - v3.x) + (v1.y - v3.y) * (v1.y - v3.y));
    cout << "Constructor_1 for: " << objID
            << " (" << name << ")" << endl;   // отладочный вывод
}

// Конструктор пустого (нулевого) треугольника
Triangle::Triangle(const char* ident) {
    char buf[16];
    objID = new char[strlen(ident) + 1];
    strcpy(objID, ident);

    count++;
    sprintf(buf, "Треугольник %d", count);
    name = new char[strlen(buf) + 1];
    strcpy(name, buf);
    a = b = c = 0;
    cout << "Constructor_2 for: " << objID
            << " (" << name << ")" << endl;   // отладочный вывод
}

// Деструктор
Triangle::~Triangle() {
    cout << "Destructor for: " << objID << endl;
```

```
    delete [] objID;
    delete [] name;
}

// Показать объект
void Triangle::Show() const {
    cout << name << ":";
    v1.Show();   v2.Show();   v3.Show();
    cout << endl;
}

// Показать стороны и площадь объекта
void Triangle::ShowSideAndArea() const {
    double p = (a + b + c) / 2;
    double s = sqrt(p * (p - a) * (p - b) * (p - c));
    cout << "-----------------------" << endl;
    cout << name << ":";
    cout.precision(4);
    cout << " a= " << setw(5) << a;
    cout << ", b= " << setw(5) << b;
    cout << ", c= " << setw(5) << c;
    cout << ";\ts= " << s << endl;
}
//////////////////////////////////////////////////////////
// Main.cpp
#include <iostream>
#include "Triangle.h"
//#include "CyrIOS.h"           // for Visual C++ 6.0
using namespace std;

int  Menu();
int  GetNumber(int, int);
void ExitBack();
void Show(Triangle* [], int);
void Move(Triangle* [], int);
void FindMax(Triangle* [], int);
void IsIncluded(Triangle* [], int);

// Инициализация глобальных переменных
int Triangle::count = 0;

// ------------------- главная функция
int main() {

// Определения точек
    Point p1(0, 0);    Point p2(0.5, 1);
    Point p3(1, 0);    Point p4(0, 4.5);
    Point p5(2, 1);    Point p6(2, 0);
    Point p7(2, 2);    Point p8(3, 0);

// Определения треугольников
    Triangle triaA(p1, p2, p3, "triaA");
    Triangle triaB(p1, p4, p8, "triaB");
    Triangle triaC(p1, p5, p6, "triaC");
    Triangle triaD(p1, p7, p8, "triaD");

// Определение массива указателей на треугольники
    Triangle* pTria[] = { &triaA, &triaB, &triaC, &triaD };
    int n = sizeof (pTria) / sizeof (pTria[0]);

// Главный цикл
    bool done = false;
    while (!done) {
        switch (Menu()) {
        case 1: Show(pTria, n);         break;
        case 2: Move(pTria, n);         break;
        case 3: FindMax(pTria, n);      break;
        case 4: IsIncluded(pTria, n);   break;
        case 5: cout << "Конец работы." << endl;
                done = true;    break;
        }
    }
    return 0;
}
// ------------------- вывод меню
int Menu() {

    cout << "\n============== Главное меню ==================" <<
endl;
    cout << "1 - вывести все объекты\t 3 - найти максимальный" << endl;
    cout << "2 - переместить\t\t 4 - определить отношение включения" << endl;
    cout << "\t\t    5 - выход" << endl;

    return GetNumber(1, 5);
}
// ------- ввод целого числа в заданном диапазоне
int GetNumber(int min, int max) {
    int number = min - 1;
    while (true) {
        cin >> number;
        if ((number >= min) && (number <= max) && (cin.peek() == '\n'))
            break;
        else {
            cout << "Повторите ввод (ожидается число от " << min
                    << " до " << max << "):" << endl;
            cin.clear();
            while (cin.get() != '\n') {};
        }
```

```
    }
    return number;
}
// ————————— возврат в функцию с основным меню
void ExitBack() {
    cout << "Нажмите Enter." << endl;
    cin.get(); cin.get();
}
// ————————————— вывод всех треугольников
void Show(Triangle* p_tria[], int k) {
    cout << "======= Перечень треугольников ========" << endl;
    for (int i = 0; i < k; ++i) p_tria[i]->Show();
    for (i = 0; i < k; ++i) p_tria[i]->ShowSideAndArea();
    ExitBack();
}
// ———————————————— перемещение
void Move(Triangle* p_tria[], int k) {
    cout << "============ Перемещение ============" << endl;
    // здесь будет код функции...
    ExitBack();
}
// ————————— поиск максимального треугольника
void FindMax(Triangle* p_tria[], int k) {
    cout << "=== Поиск максимального треугольника ==" << endl;
    // здесь будет код функции...
    ExitBack();
}
// ————————— определение отношения включения
void IsIncluded(Triangle* p_tria[], int k) {
    cout << "======== Отношение включения ==========" << endl;
    // здесь будет код функции...
    ExitBack();
}
//———— конец проекта Task1_2 ————
////////////////////////////////////////////////////////////
```

Рекомендуем вам обратить внимание на следующие моменты в проекте Task1_2.

- Класс Point (файлы Point.h, Point.cpp).
  - Реализация класса Point пока что содержит единственный метод Show(), назначение которого очевидно: показать объект типа Point на экране. Здесь следует заметить, что при решении реальных задач в какой-либо графической оболочке метод Show() действительно нарисовал бы нашу точку, да еще в цвете. Но мы-то изучаем «чистый» C++, так что придется удовольствоваться текстовым выводом на экран основных атрибутов точки — ее координат.
- Класс Triangle (файлы Triangle.h, Triangle.cpp).
  - Назначение большинства полей и методов очевидно из их имен и комментариев.
  - Поле static int count играет роль глобального[1] счетчика создаваемых объектов; мы сочли удобным в конструкторах генерировать имена треугольников автоматически: «Треугольник 1», «Треугольник 2» и т. д., используя текущее значение count (возможны и другие способы именования треугольников).
  - Поле char* objID избыточно для решения нашей задачи — оно введено исключительно для целей отладки и обучения; вскоре вы увидите, что благодаря отладочным операторам печати в конструкторах и деструкторе удобно наблюдать за созданием и уничтожением объектов.
  - Метод ShowSideAndArea() введен также только для целей отладки, — убедившись, что стороны треугольника и его площадь вычисляются правильно (с помощью калькулятора), в дальнейшем этот метод можно удалить.
  - Конструктор пустого (нулевого) треугольника предусмотрен для создания временных объектов, которые могут модифицироваться с помощью присваивания.
  - Метод Show() — см. комментарий выше по поводу метода Show() в классе Point. К сожалению, здесь нам тоже не удастся нарисовать треугольник на экране; вместо этого печатаются координаты его вершин.
- Основной модуль (файл Main.cpp).
  - Инициализация глобальных переменных: обратите внимание на оператор int Triangle::count = 0; — если вы забудете это написать, компилятор очень сильно обидится.
  - Функция main():

    определения восьми точек p1, ..., p8 выбраны произвольно, но так, чтобы из них можно было составить треугольники;

    определения четырех треугольников сделаны тоже произвольно, впоследствии на них будут демонстрироваться основные методы класса; однако не забывайте, что вершины в каждом треугольнике должны перечисляться *по часовой стрелке*;

    далее определяются массив указателей Triangle* pTria[] с адресами объявленных выше треугольников и его размер n; в таком виде удобно передавать адрес pTria и величину n в вызываемые серверные функции;

    главный цикл функции довольно прозрачен и дополнительных пояснений не требует.
  - Функция Menu() — после вывода на экран списка пунктов меню вызывается функция GetNumber(), возвращающая номер пункта, введенный пользователем с клавиатуры. Для чего написана такая сложная функция — GetNumber()? Ведь можно было просто написать: cin >> number;? Но тогда мы не обеспечили бы защиту программы от непреднамеренных ошибок при вводе. Вообще-то

[1] Свойство глобальности обеспечивается благодаря модификатору static.

вопрос надежного ввода чисел с клавиатуры подробно разбирается на семинаре 4 при решении задачи 4.2. Там же вы можете найти описание работы аналогичной функции GetInt()[1].

- Функция Show()просто выводит на экран перечень всех треугольников. В завершение вызывается функция ExitBack(), которая обеспечивает заключительный диалог с пользователем после обработки очередного пункта меню.
- Остальные функции по обработке оставшихся пунктов меню выполнены в виде заглушек, выводящих только наименование соответствующего пункта.

Тестирование и отладка первой версии программы

После компиляции и запуска программы вы должны увидеть на экране следующий текст:

```
Constructor_1 for: triaA (Треугольник 1)
Constructor_1 for: triaB (Треугольник 2)
Constructor_1 for: triaC (Треугольник 3)
Constructor_1 for: triaD (Треугольник 4)

============== Главное  меню ==================
1 - вывести все объекты  3 - найти максимальный
2 - переместить      4 - определить отношение включения
          5 - выход
```

Введите с клавиатуры цифру 1[2]. Программа выведет:

```
1
======= Перечень треугольников ========
Треугольник 1:  (0, 0)  (0.5, 1)  (1, 0)
Треугольник 2:  (0, 0)  (0, 4.5)  (3, 0)
Треугольник 3:  (0, 0)  (2, 1)  (2, 0)
Треугольник 4:  (0, 0)  (2, 2)  (3, 0)
-------------------------------------------
Треугольник 1: a= 1.118, b= 1.118, c=    , 1:    s= 0.5
-------------------------------------------
Треугольник 2: a=   4.5, b= 5.408, c=     3:    s= 6.75
-------------------------------------------
Треугольник 3: a= 2.236, b=     1, c=     2:    s= 1
-------------------------------------------
Треугольник 4: a= 2.828, b= 2.236, c=     3:    s= 3
Нажмите Enter.
```

[1] В задаче 9.2 из первой книги практикума было дано другое решение проблемы ввода номера пункта меню (с защитой от ошибок), но оно рассчитано на использование функций библиотеки stdio, унаследованных из языка C.

[2] После ввода числовой информации всегда подразумевается нажатие клавиши Enter.

Выбор первого пункта меню проверен. Нажмите Enter. Программа выведет:

```
============== Главное  меню ==================
```

Теперь проверим выбор второго пункта меню. Введите с клавиатуры цифру 2. На экране должно появиться:

```
2
====================== Перемещение =====================
Нажмите Enter.
```

Выбор второго пункта проверен. Нажмите Enter. Программа выведет:

```
============== Главное  меню ==================
. . .
```

Теперь проверим ввод ошибочного символа. Введите с клавиатуры любой буквенный символ, например w, и нажмите Enter. Программа должна выругаться:

```
Повторите ввод (ожидается число от 1 до 5):
```

Проверяем завершение работы. Введите цифру 5. Программа выведет:

```
5
Конец работы.
Destructor for: triaD
Destructor for: triaC
Destructor for: triaB
Destructor for: triaA
```

Тестирование закончено. Обратите внимание на то, что деструкторы объектов вызываются в порядке, обратном вызову конструкторов.

Продолжим разработку программы. На втором этапе мы добавим в классы Point и Triangle методы, обеспечивающие перемещение треугольников, а в основной модуль — реализацию функции Move(). Кроме этого, в классе Triangle мы удалим метод ShowSideAndArea(), поскольку он был введен только для целей отладки и свою роль уже выполнил.

Этап 2

Внесите следующие изменения в тексты модулей проекта.

1. Модуль Point.h: добавьте сразу после объявления метода Show() объявление операции-функции «+=», которая позволит нам впоследствии реализовать метод перемещения Move() в классе Triangle:

```
void operator +=(Point&);
```

2. Модуль Point.cpp. Добавьте код реализации данной функции:

```
void Point::operator +=(Point& p) {
    x += p.x;    y += p.y;
}
```

3. Модуль Triangle.h.
   - Удалите объявление метода ShowSideAndArea().
   - Добавьте объявление метода:

```
void Move(Point);
```

4. Модуль Triangle.cpp.
   - Удалите метод ShowSideAndArea().
   - Добавьте код метода Move():

```
// Переместить объект на величину (dp.x, dp.y)
void Triangle::Move(Point dp) {
    v1 += dp;    v2 += dp;    v3 += dp;
}
```

5. Модуль Main.cpp.
   - В список прототипов функций в начале файла добавьте сигнатуру:

```
double GetDouble();
```

   - Добавьте в файл текст новой функции GetDouble() либо сразу после функции Show(), либо в конец файла. Эта функция предназначена для ввода вещественного числа и вызывается из функции Move(). В ней предусмотрена защита от ввода недопустимых (например, буквенных) символов аналогично тому, как это решено в функции GetNumber():

```
// ---------- ввод вещественного числа
double GetDouble() {
    double value;
    while (true) {
        cin >> value;
        if (cin.peek() == '\n') break;
        else {
            cout << "Повторите ввод (ожидается вещественное число):" << endl;
            cin.clear();
            while (cin.get() != '\n') {};
        }
    }
    return value;
}
```

   - Замените заглушку функции Move() следующим кодом:

```
// ---------- перемещение
void Move(Triangle* p_tria[], int k) {
    cout << "========== Перемещение ==========" << endl;
    cout << "Введите номер треугольника (от 1 до " << k << "): ";
    int i = GetNumber(1, k) - 1;
    p_tria[i]->Show();

    Point dp;
    cout << "Введите смещение по x: ";
    dp.x = GetDouble();
    cout << "Введите смещение по y: ";
    dp.y = GetDouble();

    p_tria[i]->Move(dp);
    cout << "Новое положение треугольника:" << endl;
    p_tria[i]->Show();
    ExitBack();
}
```

Выполнив компиляцию проекта, проведите его тестирование аналогично тестированию на первом этапе. После выбора второго пункта меню и ввода данных, задающих номер треугольника, величину сдвига по x и величину сдвига по y, вы должны увидеть на экране примерно следующее:[1]

```
2
========== Перемещение ==========
Введите номер треугольника (от 1 до 4): 1
Треугольник 1:  (0, 0)  (0.5, 1)  (1, 0)
Введите смещение по x: 2.5
Введите смещение по y: -7
Новое положение треугольника:
Треугольник 1:  (2.5, -7)  (3, -6)  (3.5, -7)
Нажмите Enter.
```

Продолжим разработку программы.

Этап 3

На этом этапе мы добавим в класс Triangle метод, обеспечивающий сравнение треугольников по их площади, а в основной модуль — реализацию функции FindMax(). Внесите следующие изменения в тексты модулей проекта:

1. Модуль Triangle.h: добавьте объявление функции-операции:

```
bool operator >(const Triangle&) const;
```

2. Модуль Triangle.cpp: добавьте код реализации функции-операции:

```
// Сравнить объект (по площади) с объектом tria
bool Triangle::operator >(const Triangle& tria) const {
    double p = (a + b + c) / 2;
    double s = sqrt(p * (p - a) * (p - b) * (p - c));
    double p1 = (tria.a + tria.b + tria.c) / 2;
    double s1 = sqrt(p1 * (p1 - tria.a) * (p1 - tria.b) * (p1 - tria.c));
    if (s > s1) return true;
    else        return false;
}
```

[1] Жирным шрифтом выделено то, что вводилось с клавиатуры.

3. Модуль Main.cpp: замените заглушку функции FindMax() следующим кодом:

```
// ———————— поиск максимального треугольника
void FindMax(Triangle* p_tria[], int k) {
    cout << "=== Поиск максимального треугольника ==" << endl;
    // Создаем объект triaMax, который по завершении поиска будет
    // идентичен максимальному объекту.
    // Инициализируем его значением 1-го объекта из массива объектов.
    Triangle triaMax("triaMax");
    triaMax = *p_tria[0];
    // Поиск
    for (int i = 1; i < 4; ++i)
        if (*p_tria[i] > triaMax)
            triaMax = *p_tria[i];
    cout << "Максимальный треугольник: " << triaMax.GetName() << endl;
    ExitBack();
}
```

Откомпилируйте программу и запустите. Выберите третий пункт меню. На экране должно появиться:

```
3
=== Поиск максимального треугольника ==
Constructor_2 for: triaMax (Треугольник 5)
Максимальный треугольник: Треугольник 2
Нажмите Enter.
```

Как видите, максимальный треугольник найден правильно. Повинуясь указанию, нажмите Enter. Появится текст:

```
Destructor for: triaB
============== Г л а в н о е   м е н ю ==================
1 - вывести все объекты   3 - найти максимальный
2 - переместить       4 - определить отношение включения
            5 - выход
```

Чтобы завершить работу, введите цифру 5 и нажмите Enter. Что за ... — получили??? Ха-ха-ха!!! Программа вылетела! Если это происходит в Visual Studio, то вас порадует нагло выскочившая на передний план диалоговая панель с жирным белым крестом на красном кружочке и с сообщением об ошибке:

```
Debug Assertion Failed!
Program: C:\. . . MAIN.EXE
File: dbgdel.cpp
Line 47
```

А далее добрый совет, смысл которого в переводе на русский язык следующий: «Если вас интересует, по каким причинам могут выскакивать такие диалоговые панельки, обратитесь к разделу документации по Visual C++, посвященному макросу assert». Попробуйте обратиться. Скорее всего, вы будете разочарованы...

Давайте лучше посмотрим на нашу отладочную печать. Перед тем как так некрасиво умереть, программа успела вывести на экран следующее:

```
5
Конец работы.
Destructor for: triaD
Destructor for: triaC
Destructor for: HHHHHHHHHHэээээЗЗЗЗЗЗЗЗЗЗЗQ
```

Пришло время для аналитической работы нашим серым клеточкам. Вспомните, как выглядела отладочная печать при завершении нормально работающей первой версии программы? Деструкторы вызывались в порядке, обратном вызову конструкторов. Значит, какая-то беда случилась после вызова деструктора для объекта triaB! Почему?

Всмотримся внимательней в предыдущий вывод нашей программы. После того как функция FindMax() выполнила основную работу и вывела на экран сообщение

```
Максимальный треугольник: Треугольник 2
```

программа пригласила пользователя нажать клавишу Enter. Это приглашение выводится функцией ExitBack(). А вот после нажатия клавиши Enter на экране появился текст:

```
Destructor for: triaB
```

после которого опять было выведено главное меню.

Значит, деструктор для объекта triaB был вызван в момент возврата из функции FindMax(). Но почему? Ведь объект triaB создается в основной функции main(), а значит, там же должен и уничтожаться! Что-то нехорошее происходит, однако, в теле функции FindMax(). Хотя внешне вроде все прилично... Стоп! Ведь внутри функции объявлен объект triaMax, и мы даже видели работу его конструктора:

```
Constructor_2 for: triaMax (Треугольник 5)
```

А где же вызов деструктора, который по идее должен произойти в момент возврата из функции FindMax()? Кажется, мы нашли причину ненормативного поведения нашей программы. Объект triaMax после своего объявления неоднократно модифицируется с помощью операции присваивания. Последняя такая модификация происходит в цикле, причем объекту triaMax присваивается значение объекта triaB. А теперь давайте вспомним, что если мы не перегрузили операцию присваивания для некоторого класса, то компилятор сделает это за нас, но в такой «операции присваивания по умолчанию» будут поэлементно копироваться все поля объекта. При наличии же полей типа указателей возможны проблемы, что мы и получили.

В поля objID и name объекта triaMax были скопированы значения одноименных полей объекта triaB. В момент выхода из функции FindMax() деструктор объекта освободил память, на которую указывали эти поля. А при выходе из основной

функции main() деструктор объекта triaB попытался еще раз освободить эту же память. Это делать нельзя, потому что этого делать нельзя никогда.

Займемся починкой нашей программы. Нужно добавить в класс Triangle перегрузку операции присваивания, а заодно и конструктор копирования.

Внесите следующие изменения в тексты модулей проекта:

1. Модуль Triangle.h.
   - Добавьте объявление конструктора копирования:

```
Triangle(const Triangle&);  // конструктор копирования
```

   - Добавьте объявление операции присваивания:

```
Triangle& operator =(const Triangle&);
```

2. Модуль Triangle.cpp.
   - Добавьте реализацию конструктора копирования:

```
// Конструктор копирования
Triangle::Triangle(const Triangle& tria) : v1(tria.v1), v2(tria.v2),
v3(tria.v3) {
    cout << "Copy constructor for: " << tria.objID << endl; // отладочный вывод

    objID = new char[strlen(tria.objID) + strlen("(копия)") + 1];
    strcpy(objID, tria.objID);
    strcat(objID, "(копия)");

    name = new char[strlen(tria.name) + 1];
    strcpy(name, tria.name);
    a = tria.a;
    b = tria.b;
    c = tria.c;
}
```

   - Добавьте реализацию операции присваивания:

```
// Присвоить значение объекта tria
Triangle& Triangle::operator =(const Triangle& tria) {
    cout << "Assign operator: " << objID << " = " << tria.objID << endl;
    // отладочный вывод

    if (&tria == this) return *this;
    delete [] name;
    name = new char[strlen(tria.name) + 1];
    strcpy(name, tria.name);
    a = tria.a;  b = tria.b;  c = tria.c;
    return *this;
}
```

И в конструкторе копирования, и в операторе присваивания перед копированием содержимого полей, на которые указывают поля типа char*, для них выделяется новая память. Обратите внимание, что в конструкторе копирования после переписи поля objID мы добавляем в конец этой строки текст «(копия)». А в операции присваивания это поле, идентифицирующее объект, вообще не затрагивается и остается в своем первоначальном значении. Все это полезно для целей отладки.

Откомпилируйте и запустите программу. Выберите третий пункт меню. На экране должен появиться текст:

```
3
=== Поиск максимального треугольника ==
Constructor_2 for: triaMax (Треугольник 5)
Assign operator: triaMax = triaA
Assign operator: triaMax = triaB
Максимальный треугольник: Треугольник 2
Нажмите Enter.
```

Обратите внимание на отладочный вывод операции присваивания. Продолжим тестирование. Нажмите Enter. Программа выведет:

```
Destructor for: triaMax
============== Главное меню ==================
1 - вывести все объекты   3 - найти максимальный
2 - переместить       4 - определить отношение включения
                5 - выход
```

Обратите внимание на то, что был вызван деструктор для объекта triaMax, а не triaB. Продолжим тестирование. Введите цифру 5. Программа выведет:

```
5
Конец работы.
Destructor for: triaD
Destructor for: triaC
Destructor for: triaB
Destructor for: triaA
```

Все. Программа работает, как часы.

Но, однако, мы еще не все сделали. Нам осталось решить самую сложную подзадачу — определение *отношения включения* одного треугольника в другой.

Этап 4

Из многочисленных подходов к решению этой подзадачи наш выбор остановился на алгоритме, в основе которого лежит определение относительного положения точки и вектора на плоскости[1]. Вектор — это направленный отрезок прямой линии, начинающийся в точке beg_p и заканчивающийся в точке end_p. При графическом изображении конец вектора украшают стрелкой. Теперь призовите

[1] Известен другой алгоритм решения этой задачи, основанный на использовании формулы Герона. Наш выбор обоснован тем, что это решение гармонично вписывается в технологию «нисходящего проектирования».

ваше пространственное воображение или вооружитесь карандашом и бумагой, чтобы проверить следующее утверждение. Вектор (beg_p, end_p) делит плоскость на пять непересекающихся областей:

1) все точки *слева* от воображаемой бесконечной прямой[1], на которой лежит наш вектор (область LEFT),
2) все точки *справа* от воображаемой бесконечной прямой, на которой лежит наш вектор (область RIGHT),
3) все точки на воображаемом продолжении прямой *назад* от точки beg_p в бесконечность (область BEHIND),
4) все точки на воображаемом продолжении прямой *вперед* от точки end_p в бесконечность (область AHEAD),
5) все точки, *принадлежащие* самому вектору (область BETWEEN).

Для выяснения относительного положения точки, заданной некоторым объектом класса Point, добавим в класс Point перечисляемый тип:

```
enum ORIENT { LEFT, RIGHT, AHEAD, BEHIND, BETWEEN };
```

а также метод Classify(beg_p, end_p), возвращающий значение типа ORIENT для данной точки относительно вектора (beg_p, end_p).

Обладая этим мощным методом, совсем нетрудно определить, находится ли точка внутри некоторого треугольника. Мы договорились перед началом решения задачи, что треугольники будут задаваться перечислением их вершин в порядке изображения их на плоскости *по часовой стрелке*. То есть каждая пара вершин образует вектор, и эти векторы следуют один за другим по часовой стрелке. При этом условии некоторая точка находится внутри треугольника тогда и только тогда, когда ее ориентация относительно каждой вектора-стороны треугольника имеет одно из двух значений: либо RIGHT, либо BETWEEN. Эту подзадачу будет решать метод InTriangle() в классе Point.

Изложим по порядку, какие изменения нужно внести в тексты модулей.

1. Модуль Point.h.
   - Добавьте перед объявлением класса Point объявление нового типа ORIENT, а также упреждающее объявление типа Triangle:

     ```
     enum ORIENT { LEFT, RIGHT, AHEAD, BEHIND, BETWEEN };
     class Triangle;
     ```

     Последнее необходимо, чтобы имя типа Triangle было известно компилятору в данной единице трансляции, так как оно используется в сигнатуре метода InTriangle().
   - Добавьте внутри класса Point объявления функций:

     ```
     Point operator +(Point&);
     Point operator -(Point&);
     double Length() const;                    // определяет длину вектора точки
                                               // в полярной системе координат
     ORIENT Classify(Point&, Point&) const;    // определяет положение точки
                                               // относительно вектора,
                                               // заданного двумя точками
     bool InTriangle(Triangle&) const;         // определяет,
                                               // находится ли точка внутри
                                               // треугольника
     ```

Функция-операция «–» и метод Length() будут использованы при реализации метода Classify(), а функция-операция «+» добавлена для симметрии. Метод Classify(), в свою очередь, вызывается из метода InTriangle().

1. Модуль Point.cpp.
   - Добавьте после директивы #include <iostream> директиву #include <math.h>.

     Она необходима для использования функции sqrt(x) из математической библиотеки C++ в алгоритме метода Length().
   - Добавьте после директивы #include «Point.h» директиву #include «Triangle.h».
   - Последняя необходима в связи с использованием имени класса Triangle в данной единице трансляции.
   - Добавьте реализацию функций-операций:

     ```
     Point Point::operator +(Point& p) {
         return Point(x + p.x, y + p.y);
     }
     Point Point::operator -(Point& p) {
         return Point(x - p.x, y - p.y);
     }
     ```

   - Добавьте реализацию метода Length():

     ```
     double Point::Length() const {
         return sqrt(x*x + y*y);
     }
     ```

   - Добавьте реализацию метода Classify():

     ```
     ORIENT Point::Classify(Point& beg_p, Point& end_p) const {
         Point p0 = *this;
         Point a = end_p - beg_p;
         Point b = p0 - beg_p;
         double sa = a.x * b.y - b.x * a.y;

         if (sa > 0.0) return LEFT;
         if (sa < 0.0) return RIGHT;

     if ((a.x * b.x < 0.0) || (a.y * b.y < 0.0)) return BEHIND;
         if (a.Length() < b.Length()) return AHEAD;

         return BETWEEN;
     }
     ```

[1] Точнее говоря, от воображаемого *бесконечного вектора*, поскольку направление здесь очень важно.

Алгоритм заимствован из [6], поэтому мы не будем здесь подробно его разбирать. Обратите внимание, что аргументы передаются в функцию по ссылке — это позволяет избежать вызова конструктора копирования.

- Добавьте реализацию метода InTriangle():

```
bool Point::InTriangle(Triangle& tria) const {
    ORIENT or1 = Classify(tria.Get_v1(), tria.Get_v2());
    ORIENT or2 = Classify(tria.Get_v2(), tria.Get_v3());
    ORIENT or3 = Classify(tria.Get_v3(), tria.Get_v1());

    if ((or1 == RIGHT || or1 == BETWEEN)
     && (or2 == RIGHT || or2 == BETWEEN)
     && (or3 == RIGHT || or3 == BETWEEN)) return true;
    else    return false;
}
```

2. Модуль Triangle.h: добавьте в классе Triangle объявление дружественной функции (все равно, в каком месте):

```
friend bool TriaInTria(Triangle, Triangle); // Определить,
                    // входит ли один треугольник во второй
```

3. Модуль Triangle.cpp: добавьте в конец файла реализацию внешней дружественной функции:

```
// Определить, входит ли треугольник tria1 в треугольник tria2
bool TriaInTria(Triangle tria1, Triangle tria2) {
    Point v1 = tria1.Get_v1();
    Point v2 = tria1.Get_v2();
    Point v3 = tria1.Get_v3();
    return (v1.InTriangle(tria2) &&
            v2.InTriangle(tria2) &&
            v3.InTriangle(tria2));
    return true;
}
```

Результат, возвращаемый функцией, основан на проверке вхождения каждой вершины первого треугольника (tria1) во второй треугольник (tria2).

4. Модуль Main.cpp: замените заглушку функции IsIncluded() следующим кодом:

```
// ----------- определение отношения включения
void IsIncluded(Triangle* p_tria[], int k) {
    cout << "======== Отношение включения ==========" << endl;
    cout << "Введите номер 1-го треугольника (от 1 до " << k << "): ";
    int i1 = GetNumber(1, k) - 1;

    cout << "Введите номер 2-го треугольника (от 1 до " << k << "): ";
    int i2 = GetNumber(1, k) - 1;

    if (TriaInTria(*p_tria[i1], *p_tria[i2]))
        cout << p_tria[i1]->GetName() << " - входит в - "
        << p_tria[i2]->GetName() << endl;
    else
        cout << p_tria[i1]->GetName() << " - не входит в - "
        << p_tria[i2]->GetName() << endl;
    ExitBack();
}
```

Модификация проекта завершена. Откомпилируйте и запустите программу. Выберите четвертый пункт меню. Следуя указаниям программы, введите номера сравниваемых треугольников, например 1 и 2. Вы должны получить следующий результат:

```
4
======== Отношение включения ==========
Введите номер 1-го треугольника (от 1 до 4): 1
Введите номер 2-го треугольника (от 1 до 4): 2
Copy constructor for: triaB
Copy constructor for: triaA
Destructor for: triaA(копия)
Destructor for: triaB(копия)
Треугольник 1 - входит в - Треугольник 2
Нажмите Enter
```

Обратите внимание на отладочную печать: конструкторы копирования вызываются при передаче аргументов в функцию TriaInTria(), а перед возвратом из этой функции вызываются соответствующие деструкторы.

Проведите следующий эксперимент: удалите с помощью скобок комментария конструктор копирования в файлах Triangle.h и Triangle.cpp, откомпилируйте проект и повторите тестирование четвертого пункта меню. Полюбовавшись результатом, верните проект в нормальное состояние.

Протестируйте остальные пункты меню, вызывая их в произвольном порядке.

Когда вы сочтете, что тестирование завершено, уберите с помощью символов комментария // всю отладочную печать из конструкторов, деструктора и операции присваивания. Повторите тестирование без отладочной печати.

Решение задачи 1.2 завершено.

Давайте повторим наиболее важные моменты этого семинара.

1. Использование классов лежит в основе объектно-ориентированной декомпозиции программных систем, которая является более эффективным средством борьбы со сложностью систем, чем функциональная декомпозиция.
2. В результате декомпозиции система разделяется на компоненты (модули, функции, классы). Чтобы обеспечить высокое качество проекта, его способность к модификациям, удобство сопровождения, необходимо учитывать сцепление внутри компонента (оно должно быть сильным) и связанность между компонентами (она должна быть слабой), а также правильно распределять обязанности между компонентом-клиентом и компонентом-сервером.
3. Класс — это определяемый пользователем тип, лежащий в основе ООП. Класс содержит ряд полей (переменных), а также методов (функций), имеющих доступ к этим полям.

4. Доступ к отдельным частям класса регулируется с помощью ключевых слов: `public` (открытая часть), `private` (закрытая часть) и `protected` (защищенная часть). Последний вид доступа имеет значение только при наследовании классов. Методы, расположенные в открытой части, формируют *интерфейс* класса и могут вызываться через соответствующий объект.
5. Если в классе имеются поля типа указателей и осуществляется динамическое выделение памяти, то необходимо позаботиться о создании конструктора копирования и перегрузке операции присваивания.
6. Для создания полного, минимального и интуитивно понятного интерфейса класса широко применяется перегрузка методов и операций.