Курс лекций "Программирование"
Основы программирования на языках C и C++

Лекция 9. Введение в классы C++

Глухих Михаил Игоревич, к.т.н., доц.

mailto: glukhikh@mail.ru

Пример

- Во входном файле приведены координаты точек на плоскости. Необходимо составить из этих точек три треугольника максимальной площади и вывести их в выходной файл
- Начало решения см. лекцию 8

Как искать треугольники наибольшей площади?

- Надо перебрать все тройки точек (например, в тройном цикле)
- Следует завести массив, в котором будут храниться треугольники по убыванию площади. Изначально он заполнен треугольниками нулевой площади (реальные площади гарантированно больше)
- Когда определена площадь очередного треугольника, следует определить, нужно ли ее вставить в массив, после чего «раздвинуть» массив, убрав крайний элемент, и на освободившееся место вставить очередной треугольник (функция **findAndInsert**)

Переопределение оператора <=

- У данного оператора логический результат и два аргумента (в данном случае – треугольника). В данном случае при упорядочивании нас интересуют площади треугольников, поэтому и сравниваем мы площади

```
bool operator <=(const Triangle& tr1,
                 const Triangle& tr2)
{
   return tr1.area <= tr2.area;
}
```

Другие операторы сравнения

- Важно понимать, что другие операторы (например, >) при необходимости нужно определять отдельно. Компилятор сам не сделает из оператора <= оператор >. Компилятор не проверяет, что эти операторы дают противоположные результаты.

Поиск треугольников максимальной площади

```
void searchLargestTriangles(const Point* pointArray,
                            int pointNum,
                            Triangle* trArray,
                            int maxTrNum)
{
   if (maxTrNum <= 0)
      return;
   // Очистка треугольников
   for (int i=0; i<maxTrNum; i++)
      clearTriangle(trArray[i]);
   // ...
```

Поиск треугольников максимальной площади

```
    Triangle triangle;
    // Перебор троек точек
    for (int i=0; i<pointNum; i++)
    {
        triangle.vertexes[0] = pointArray[i];
        for (int j=i+1; j<pointNum; j++)
        {
            triangle.vertexes[1] = pointArray[j];
            for (int k=j+1; k<pointNum; k++)
            {
                triangle.vertexes[2] = pointArray[k];
                calcTriangleArea(triangle);
                findAndInsert(trArray, maxTrNum, triangle);
            }
        }
    }
}
```

Функции очистки

```
void clearPoint(Point& p)
{
   p.x=p.y=0.0;
}

void clearTriangle(Triangle& tr)
{
   for (int i=0; i<3; i++)
      clearPoint(tr.vertexes[i]);
   tr.area=0.0;
}
```

Вставка очередного треугольника в массив

```
void findAndInsert(Triangle* trArray,
                   int length, const Triangle& triangle)
{
   int pos;
   for (pos=length-1; pos>=0; pos--)
   {
      if (triangle <= trArray[pos])
         break;
   }
   pos++;
   if (pos==length)
      return;
   for (int m=length-2; m>=pos; m--)
      trArray[m+1]=trArray[m];
   trArray[pos]=triangle;
}
```

Обратите внимание…

- Что у нас выделились несколько функций, явно предназначенных для обработки точек и треугольников
- Для точек это функции:

```
clearPoint
calcDistance
operator >>
operator <<
```

- Для треугольников это функции:

```
clearTriangle
calcTriangleArea
operator <<
operator <=
```

Отсюда в свое время появилась идея…

- Четко закрепить функции программы (все или некоторую часть) за теми понятиями, с которыми они работают. Из этой идеи в свое время выросло объектно-ориентированное программирование и (в частности) язык C++

Отсюда в свое время появилась идея…

- На языке C++ подобную привязку можно сделать двумя способами (чаще всего они используются совместно):
  - выделить отдельный модуль, который опишет как само понятие (в форме структуры), так и функции, с ним работающие. В данном случае модуль **geometry.h** разделился бы на два модуля: **point.h** и **triangle.h**. Этот способ реализуем и на языке C.
  - определить функции, работающие с данным понятием, внутри определения структуры (или класса). Как результат – образуется так называемый **объект**, соединяющий в себе **данные (поля)**, описывающие определенное понятие, и **функции (методы)**, которые определяют, как работать с данным понятием.

Зачем нужна группировка функций по объектам?

- Во-первых, это улучшает структурированность программы
- Во-вторых, при решении достаточно сложных задач человеку часто бывает проще оперировать вначале понятиями, а уже потом – конкретными алгоритмами. Дополнение понятий теми действиями, которые можно с ними выполнять, приближает их к реальности

Определения наших структур будут выглядеть так

- Определение точки (файл point.h):

```
struct Point
{
    double x, y;  // поля
    void clear(); // функция-член (метод)
    double calcDistance(const Point& p) const;
};
// глобальные функции
istream& operator >>(istream& in, Point& p);
ostream& operator <<(ostream& out, const Point& p);
```

Определения наших структур будут выглядеть так

- Определение треугольника (файл triangle.h):

```
struct Triangle
{
    // Поля
    Point vertexes[3];
    double area;
    // Методы
    void clear();
    double calcArea() const;
    bool operator <=(const Triangle& tr) const;
};
// Глобальная функция
ostream& operator <<(ostream& out,
                     const Triangle& tr);
```

Следует иметь в виду…

- что функции, определенные внутри структуры (функции-члены или методы), всегда вызываются для определенной переменной данного типа, например, так:

```
Point p;
p.clear(); // эквивалентно старому clearPoint(p)
```

- или:

```
Point p1, p2;
cin>>p1>>p2;
// эквивалентно старому calcDistance(p1, p2)
double dist=p1.calcDistance(p2);
```

Определения функций-членов (один экземпляр)

```
// Немножко иначе пишется заголовок
// Используется оператор ::
void Point::clear()
// К полям структуры можно обращаться
// непосредственно, без оператора .
{
   x=y=0.0;
}
```

Определения функций-членов (два экземпляра)

```
// Ключевое слово const в конце указывает,
// что переменная, для которой вызывается
// calcDistance, не изменится
double Point::calcDistance(const Point& p) const
// Здесь мы имеем дело с двумя точками
// Для одной метод вызвали
// (к ее полям обращаемся непосредственно)
// Другая – аргумент
// (к ее полям обращаемся как p.x)
{
   return sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y));
}
```

Аналогично – для треугольника

- Очистка

```
void Triangle::clear()
{
   for (int i=0; i<3; i++)
      vertexes[i].clear();
   area=0.0;
}
```

- Расчет площади

```
double Triangle::calcArea()
{
   double a=vertexes[0].calcDistance(vertexes[1]);
   double b=vertexes[1].calcDistance(vertexes[2]);
   double c=vertexes[2].calcDistance(vertexes[0]);
   return area=calcAreaBySides(a, b, c);
}
```

Оператор-член имеет два аргумента...

- Работаем аналогично calcDistance - для одного треугольника функция будет вызвана, второй будет аргументом

```
bool Triangle::operator <=(const Triangle& tr)
    const
{
    return area <= tr.area;
}
```

- Пользуемся оператором так же, как и раньше:

```
Triangle tr1, tr2;
// …
if (tr1 <= tr2) { … }
// полная форма: if (tr1.operator <=(tr2)) { … }
```

Почему не определить внутри операторы ввода и вывода?

- Все дело в том, что вводимый (или выводимый) объект для них является вторым (правым) аргументом.
- Бинарный оператор такого рода

```
SomeStream stream;
SomeObject obj;
stream << obj;
```

- может быть переопределен двумя способами:

```
SomeStream& operator <<(SomeStream& s, SomeObject& o);
```

- или

```
struct SomeStream
{
   SomeStream& operator <<(SomeObject& obj);
}
```

- То есть, операторы ввода и вывода могут быть определены внутри структуры-потока, но не внутри структуры, соответствующей выводимому объекту

Заметим также, что

- Функции clear для точки (треугольника) в нашей программе, по сути, задают начальное состояние точки (треугольника).
- Для задания начального состояния обычно используют особую функцию, называемую «**конструктор объекта**», которая вызывается автоматически при его создании (то есть, явно его вызывать не нужно). Это гарантирует нам, что конструктор будет обязательно вызван (мы про это не забудем).

Как записывается конструктор?

- Имя конструктора всегда совпадает с именем типа.
- У конструктора не бывает результата (аргументы могут быть, такой случай мы рассмотрим позднее).
- Например:

```
// Определение типа
struct Point
{
   double x, y;
   Point(); // конструктор точки
   double calcDistance(const Point& p) const;
};
```

- Определение конструктора будет выглядеть при этом так:

```
Point::Point()
{
   x=y=0.0;
}
```

Аналогично будет определен конструктор треугольника

- Определение типа

```
struct Triangle
{
    Point vertexes[3];
    double area;
    Triangle(); // конструктор
    double calcArea();
    bool operator <=(const Triangle& tr) const;
};
```

- Определение конструктора

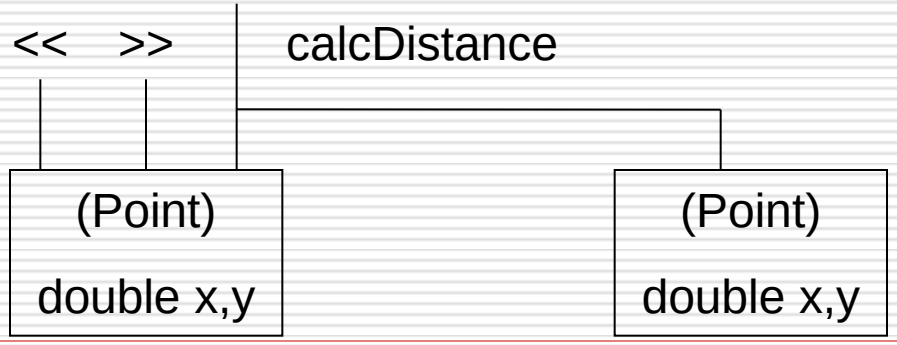
```
Triangle::Triangle()
{
    // конструктор вершин-точек вызовется автоматически
    area=0.0;
}
```

Идея инкапсуляции

- Предположим, что мы участвуем в большом проекте, и нам поручили реализовать определенные действия над точками и треугольниками (не рассказывая подробно, как это будет использоваться)
- В этом случае, по возможности, следует сделать так, чтобы, работая с нашими данными и функциями, нельзя было ничего случайно сломать
- Один из способов обеспечения этого – закрыть для внешнего доступа часть наших данных и/или функций

Идея инкапсуляции

- Представьте себе, что наш объект заключен в капсуле. До того, что находится внутри, никому не добраться. Однако на капсуле есть клавиши, на которые можно нажимать, и индикаторы, на которые можно смотреть - органы управления объектом. Для языка C++ органы управления - это функции и данные, открытые для внешнего доступа
- В нашем случае, к примеру, точка - это нечто, что можно прочитать, что можно записать, и между чем можно считать расстояние. Все остальное с точки зрения внешнего мира неинтересно.



Разграничение доступа на языке C++

```
struct Point
{
private: // После этого ключевого слова все закрыто
   // Закрытые поля (и методы)
   double x, y;
public: // После этого ключевого слова все открыто
   // Открытые методы (и поля)
   Point();
   double calcDistance(const Point& p) const;
   // Глобальные функции-друзья, имеющие доступ
   // к закрытым полям
   friend istream& operator >>(istream& in, Point& p);
   friend ostream& operator <<(ostream& out,
                               const Point& p);
};
```

Классы доступа

- Данные и функции структур (и классов тоже) могут иметь разные классы доступа. Основные из них – **private** (закрытый) и **public** (открытый).
- Данные и функции, объявленные как **private**, могут использоваться только функциями-членами данной структуры (класса), а также глобальными функциями-друзьями данной структуры (класса)
- Данные и функции, объявленные как **public**, могут использоваться всеми функциями в программе (если у них есть доступ к объекту данного типа)
- Традиционно закрываются все данные, функции, наоборот, открываются. Хотя многое зависит от конкретной решаемой задачи.

Чем отличаются классы и структуры?

```
class Point
{
private:
    // Закрытые поля
    double x, y;
public:
    // Открытые методы
    Point();
    double calcDistance(const Point& p) const;
    // Глобальные функции-друзья, имеющие доступ
    // к закрытым полям
    friend istream& operator >>(istream& in, Point& p);
    friend ostream& operator <<(ostream& out,
                                const Point& p);
};
```

Классы и структуры в C++ - примерно одно и то же…

- Единственное различие:
  - Данные и функции структур по умолчанию открытые (**public**)
  - Данные и функции классов по умолчанию закрытые (**private**)
- Как используются:
  - Как правило, структуры используются там, где открыто все
  - В других случаях обычно используются классы

Итоги

- Рассмотрено
  - Поиск наибольших объектов
  - Объединение методов и данных
  - Конструкторы
  - Разграничение доступа, инкапсуляция
- Далее
  - Инкапсуляция -- продолжение