## Задачи по геометрии, программирование.

Задачи по геометрии очень хорошо подходят для того, чтобы начать изучение C++. Основная концепция C++ — понятие объекта, который имеет некоторое состояние (хранит в себе некоторый набор данных) и определенным образом взаимодействует с другими объектами (допускает выолнение некоторых операций). В геометрии естественными объектами являются, например, точки, векторы, линии, отрезки и т.д., а с точки зрения взаимодействия и операций мы имеем, например, операции с векторами, отношения принадлежности (точка лежит на отрезке) или взаимного расположения (отрезки пересеакаются, линии параллельны, точка внутри или снаружи фигуры и т.п.).

Простейший способ определения объекта в C++ — это понятие структуры. Структура может содержать объявления разнообразных (внутренних переменных) и также обяъвления (и иногда определения) разнообразных приписанных ей функций (методов).

Рассмотрим простейший пример — объект двумерный вектор. Он имеет внутреннее состояние — две координаты $x, y$, ему присущи некоторые свойства (например, длина), и он может вступать в некоторые операции с другими объектами, например, умножаться на число, составлять скалярное произведение, и т.д.

Мы можем все это записать формально, создав новый тип с именем, скажем, Vector, и наделив ее необходимыми внутренними состояниями (данными), свойствами и операциями.

```
struct Vector {
   double x, y;       // переменные состояния --- координаты
   double Length() {  // внутренняя функция, вычисляющая свойство "длина"
      return sqrt(x*x + y*y);
   }
};
// и так с ними можно работать:
int main()
{
   Vector a, b;              // создали два вектора
   a.x = 1;  a.y = 3;        // задали им координаты
   b.x = 5;  b.y = -6.28;    // операция . --- это доступ к конкретному полю
                             //    (элементу) структуры
   cout << "a: "  << a.Length() << endl;   // а тут доступ (вызов) функции Length()
   cout << "b: "  << b.Length() << endl;   // от имени конкретного объекта - a или b
   return 0;
}
```

В данном варианте векторы изначально создаются с неопределенным состоянием, как и любая другая объявленная, но не инициализированная переменная. Поэтому мы потом и присваивали координатам конкретные значения. В C++ можно записать как создавать объект с требуемым состоянием. Для этого служат специальные функции — конструкторы, которые по имени совпадают с именем типа объекта. Конструктор не возвращает значения по определению. Конструкторов может быть несколько, если они различаются по набору и типам своих параметров. Сделаем конструкторы для вектора.

```
struct Vector {
   double x, y;       // переменные состояния - координаты

   // конструктор без параметров
   Vector() { x = y = 0; }
   // конструктор с параметрами-координатами
   Vector(double xx, double yy) { x = xx; y = yy; }
   // конструктор копирования
```

```
   Vector(const Vector &b) { x = b.x; y = b.y; }

   double Length() { return sqrt(x*x + y*y); }
};
// теперь предыдущий пример можно записать так:
int main()
{
   Vector a(1, 3), b(5, -6.28);
   cout << "a: "  << a.Length() << endl;
   cout << "b: "  << b.Length() << endl;

   Vector c;      // это будет нулевой вектор
   Vector d(a);   // это будет копия a
   Vector pt[10]; // массив векторов, каждый элемент инициализируется
                  // конструктором без параметров (и только им!)
   return 0;
}
```

Конструктор копирования очень важен (в общем случае). Он используется при передаче объекта в функцию параметром по значению и при возвращении объекта оператором return. В тривиальных случаях (как здесь) его может по умолчанию реализовать компилятор, но лучше знать о его необходимости и записывать самому.

К определению объекта можно добавить операции с этим объектом. Можно это сделать напрямую в стиле C с помощью обычной функции. Например, как вычислить сумму векторов

```
   Vector Summa(Vector a, Vector b) {
       Vector c(a.x + b.x, a.y + b.y);
       return c;
   }
и далее писать что-то вроде
   c = Summa(a, b);
   c = Summa(c, d);
```

Но C++ предоставляет возможность "переопределить"существующие операции для ваших новых типов объектов, в нашем случае — для векторов. Для этого используется функция со специальным именем operator. Напишем так (и заодно чуть короче):

```
   Vector operator+(Vector a, Vector b) {
       return Vector(a.x + b.x, a.y + b.y);
   }
Теперь мы можем точно так же писать
   c = operator+(a, b);
   c = operator+(c, d);
но зато нам также доступна и вот такая форма записи
   c = a + b + d;
```

Теперь мы можем (пере)определить любые существующие операции, придав им новый смысл для наших векторов. Нужно только иметь в виду:

— можно переопределять только уже существующие операции (т.е. старые значки);

— для операции сохраняется исходная ассоциативность и приоритетность;

— нужно строго отслеживать типы первого и второго аргументов операции и готовить функции для каждого требуемого сочетания таких типов.

В резельтате набор переопределнных операций может оказаться очень большим, но это уже издержки удобства и универсальности их использования, и с этим надо пока смириться.

В качестве примера можно привести несколько таких определений

```
   Vector operator+(Vector a, Vector b) {  // сумма векторов
       return Vector(a.x + b.x, a.y + b.y);
   }
   Vector operator-(Vector a, Vector b) {  // разность векторов
       return Vector(a.x - b.x, a.y + b.y);
   }
   double operator*(Vector a, Vector b) {  // скалярное произведение
       return a.x*b.x + a.y*b.y;
   }
   Vector operator*(double n, Vector b) {  // умножение на число справа
       return Vector(a.x*n, a.y*n);
   }
   Vector operator*(Vector a, double n) {  // умножение на число слева
       return n*a;  // по сути нам все равно с какой стороны умножать,
                    // но правила контроля типа аргументов требуют
                    // отдельной функции
   }
   double operator*(Vector a, Vector b) {  // "векторное" произведение
       return a.x*b.y - a.y*b.x;
   }
```

Будем считать, что идея понятна, хотя тут есть еще много о чем поговорить.

Теперь операции с векторами можено записывать выражениями, аналогичными операциям с обычными числами.

Но векторами геометрия не исчерпывается. Придется вводить другие объекты и определять операции и отношения между всеми ними. Например, прямая линия. Ее можно представить разными способами, например, как вектор базисной точки и вектор направления, или как коэффициенты уравнения $ax+by+d=0$. Какое из представлений удобнее — зависит от задачи. В большнстве случаев удобна форма с уравнением, но иногда может потребоваться и первая форма. Значит, нужно уметь их преобразовывать друг в друга. По каким исходным данным должна создаваться линия? Какие операции допустимы с линиями? Эти вопросы надо решить, исходя из сути своих задач.

Мы не будем здесь писать честный код, а только наметим пути для его построения.

```
struct Line {
   double a, b, d; // коэффициенты уравнения, a^2+b^2=1
   Line() { s = b = d = 0; }  // неопределенная линия!
   Line(Vector p1, Vector p2);  // линия, проходящая через две данные точки
              // другой альтернативный вариант - базовая точка и направление
   Line (const Line &s);   // конструктор копирования

   Vector Direction(); // направляющий вектор
   // и т.п. по необходмости
};
// теперь некоторые операции

bool operator||(Line s1, Line s2);  // s1||s2  прямые параллельны?  взяли знак ||  :)
bool operator<<(Vector p, Line s2); // p<<s2   точка лежит на прямой?  взяли знак <<

Vector operator&(Line s1, Line s2); // p = s1&s2; точка как пересечение прямых
                                    // но как быть, если прямые не пересекаются ?
                                    // чему будет тогда равно p ???
double operator%(Line s1, Vector p);  // s1%p  - расстояние от точки до прямой
```

Здесь функции внутри структуры только объявлены. Их определение может оказаться нетривиальным, и тогда его следует выность из описания объекта. В таком случае перед именем функции ставится контекст объекта (класса), т.е. имя типа и оператор ::.

```
Line::Line (const Line &s) {
   a = s.a; b = s.b; c = s.c;
}
Vector Line::Direction() {
   return Vector(-b, a);
}
```

И как заключительный пример приведем фрагмент кода, который мог бы проверять пересакаются ли два отрезка с концами $p1, p2$ и $q1, q2$.

```
bool HaveIntersection (Vector p1, Vector p2, Vector q1, Vector q2) {
   Line line1(p1, p2);
   Line line1(q1, q2);
   double d1 = line1 % q1;
   double d2 = line1 % q2;
   if (d1*d2 > 0) return false; // q1, q2 по одну сторону от line1
   d1 = line2 % p1;
   d2 = line2 % p2;
   if (d1*d2 > 0) return false; // p1, p2 по одну сторону от line2
   // здесь обе пары точек лежат по разные стороны от соответствующих линий
   // т.е. отрезки должны пересекаться
   return true;   // но это неверно, см. ниже
}
```

Приведенный выше код на самом деле не совсем корректен. Мы не рассмотрели случаи, когда точки лежат на соответствующих линиях (например, $d1 = 0$) или "почти лежат" с указанной точностью EPS. Более правильно было бы писать

```
   if (d1*d2 > 0 && fabs(d1) >= EPS && fabs(d2) >= EPS) return false;
```

И дальше разбираться со случаями, когда расстояние точек до прямых меньше EPS. То есть надо исключить случаи, когда точки близки к прямым, но отрезки все же не пересекаются т.е. точка не попадает в EPS окрестность отрезка. Запишем это в виде функции расстояния от точки до отрезка

```
double DistToSegment(Vector pt, Vector segpt1, Vector segpt2) {
   Line line(segp1, segp2);
   Vector v0(segpt2 - segpt1); // вектор отрезка
   Vector v1(segpt1 - pt);     // векторы из точки в концы отрещка
   Vector v2(segpt2 - pt);
   double sc1 = v1*v0;          // скалярные произведения
   double sc2 = v2*v0;
   double dist;
   if (sc1 > 0) { // pt "левее" точки segpt1
       dist = v1.Length();
   } else if (sc2 > 0) {  // pt "правее" segpt2
       dist = v2.Length();
   } else {  // точка проектируется на отрезок
       dist = fabs(line % pt);
   }
   return dist;
}
```

Теперь можно исключить случаи попадания точек на отрезки при проверке пересечения отрезков. Выпишем скорректированную функцию этой проверки.

```
#define EPS  1e-12

bool DistToSegment(Vector pt, Vector segpt1, Vector segpt2) { ..... }

bool HaveIntersection (Vector p1, Vector p2, Vector q1, Vector q2) {
   // проверим сначала попадание точек на соответствующие отрезки
   // т.е. заведомо есть пересечение
   if (DistToSegment(p1, q1, q2) < EPS) return true;
   if (DistToSegment(p2, q1, q2) < EPS) return true;
   if (DistToSegment(q1, p1, p2) < EPS) return true;
   if (DistToSegment(q2, p1, p2) < EPS) return true;

   // теперь точки заведомо отстоят от отрезков более EPS
   // и можно применить логику проверки положения с разных сторон
   Line line1(p1, p2);
   Line line1(q1, q2);
   double d1 = line1 % q1;
   double d2 = line1 % q2;
   if (d1*d2 > 0) return false; // q1, q2 по одну сторону от line1
   d1 = line2 % p1;
   d2 = line2 % p2;
   if (d1*d2 > 0) return false; // p1, p2 по одну сторону от line2
   // здесь обе пары точек лежат по разные стороны от соответствующих линий
   // т.е. отрезки должны пересекаться
   // (попадание точек на отрезки  мы уже обработали)
   return true;
}
```

Мораль. Написание корректного кода здесь достаточно кропотливое занятие. Возможно, код придется перекраивать под различные найденные частные случаи. Но при подготовленной базе объектов и операций этот код просто отражает естественные геометрические отношения.