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

Лекция 8. Структурное и модульное программирование средствами языков С и С++

Глухих Михаил Игоревич, к.т.н., доц.

mailto: glukhikh@mail.ru

Разбиение программы на модули (файлы)

- Самые простые программы могут состоять из одной функции main
- Чуть более сложные включают в себя другие функции
- По мере возрастания сложности программы функций становится слишком много, в них становится тяжело ориентироваться
- Выход – разбиение функций на отдельные модули по смысловому значению

Пример

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

Какие модули можно выделить в данной задаче?

- Есть следующие группы задач:
  - ввод-вывод (модуль **file**)
  - геометрия (модуль **geometry**)
  - поиск треугольников наибольшей площади (модуль **search**)

Как организуется модуль на языке C/C++?

- Модуль логически состоит из двух файлов - файла с исходным кодом (**source file**) и заголовочного файла (**header file**)
  - файл с исходным кодом (**module.cpp**) включает в себя определения функций, а также определения глобальных переменных и констант (если они есть); в первой строчке обычно подключается заголовочный файл того же модуля:

    ```
    #include "module.h"
    ```
  - заголовочный файл (**module.h**) включает в себя прототипы функций, определения констант, объявления глобальных переменных - но только для тех элементов модуля, о которых должны знать другие модули

Как организуется модуль на языке C/C++?

- Если какой-либо модуль использует данный, необходимо подключить его заголовочный файл (в двойных кавычках – это указывает на то, что заголовочный файл не системный, а собственный):

```
#include "module.h"
```

- Главный модуль программы обычно содержит только функцию **main** и не имеет заголовочного файла (поскольку функция **main** не используется в других модулях)
- Некоторые модули включают только заголовочный файл – например, содержащий определения глобальных констант

Что такое «объявление переменной»?

- Запись типа

  ```
  extern int Var;
  ```

- Которая указывает, что в одном из файлов с исходным кодом на верхнем уровне определена глобальная переменная:

  ```
  int Var;
  ```

- Объявленную переменную можно использовать, даже если в данном файле с исходным кодом нет ее определения
- Если несколько модулей используют одни и те же константы, они обычно определяются в заголовочном файле; объявления констант не используются

Как происходит сборка программы?

- Сборка (build) состоит из 3 основных этапов:
  - на этапе **препроцессинга (preprocessing)** директивы препроцессора (например, **#include**) заменяются содержимым указанного в них заголовочного файла, в результате файл с исходным кодом дополняется прототипами указанных там функций и объявлениями глобальных переменных – за счет этого в файле можно вызывать указанные функции (прототип впереди) и использовать глобальные переменные (объявление впереди). Препроцессор может создавать на диске временные файлы, которые, однако, удаляются после окончания сборки

Как происходит сборка программы?

- Сборка (build) состоит из 3 основных этапов:
  - на этапе **компиляции (compiling)** для каждого исходного файла составляются таблицы определенных в нем функций и глобальных переменных, все определенные функции переводятся на машинный язык (при переводе на машинный язык могут выявляться ошибки компиляции). Результатом работы компилятора являются файлы **module.obj** (объектные файлы) для каждого использованного в программе модуля

Как происходит сборка программы?

- Сборка (build) состоит из 3 основных этапов:
  - на этапе **связывания (linking)** происходит привязка всех используемых (вызванных) функций и глобальных переменных к той таблице, в которой они определены; если определения не обнаружены ни в одной таблице, происходит ошибка связывания (unresolved external symbol …). Результатом связывания является файл **program.exe** (для MVS имя совпадает с именем проекта) – исполняемый файл

Что при сборке происходит с библиотечными функциями?

- Прототипы функций находятся в заголовочных файлах (iostream, math.h, stdio.h, string.h и т.п.)
- Объектные файлы с уже переведенными на машинный язык определениями функций заранее собраны в библиотеки (файлы с расширением lib или dll)
  - определения из статических библиотек (расширение lib) на этапе связывания добавляются в исполняемый файл программы
  - определения из динамических библиотек (расширение dll) в исполняемый файл не добавляются; вместо этого в ссылке на соответствующую функцию указывается, что она находится в динамической библиотеке, и при ее вызове происходит обращение к библиотеке
  - поэтому, статические библиотеки нужны только на этапе связывания, а динамические – и на этапе исполнения программы

Пример. Заголовочный файл geometry.h

```
#ifndef _GEOMETRY_H
#define _GEOMETRY_H

// Расстояние между двумя точками по их координатам
double calcDistance(double xa, double ya, double xb,
    double yb);

// Площадь треугольника по координатам точек
double calcAreaByPoints(double xa, double ya, double
    xb, double yb, double xc, double yc);

#endif
```

Геометрические функции

- Расстояние между точками:
  - $((x_2-x_1)^2+(y_2-y_1)^2)^{0.5}$
- Площадь треугольника – используем формулу Герона:
  - полупериметр: p=(a+b+c)/2
  - площадь: $s=(p*(p-a)*(p-b)*(p-c))^{0.5}$

Что такое #ifndef-#define-#endif?

- Это, как и **#include**, директивы препроцессора
- Директива **#define** позволяет определить переменную препроцессора (например, _GEOMETRY_H)
- Директива **#ifndef** позволяет выяснить, определена ли переменная препроцессора. Если она **не определена**, участок кода до директивы **#endif** вставляется в программу, если же **определена** – выбрасывается из нее
- Нужны эти скобки для того, чтобы текст заголовочного файла не мог вставиться в исходный файл дважды

Как лучше хранить в памяти набор точек на плоскости

- Каждая точка имеет x-координату и y-координату. Поэтому нам необходимо сохранить два массива: px[] и py[].
- Лучше было бы создать собственный **структурный** тип Point (точка) и один массив point[]
- В C/C++ для этого используются т.н. **структуры** и **классы**

Реализация в C++, определение структуры

- Для описания структурных типов используются **структуры** и **классы**. Рассмотрим вначале структуры.
- Например, опишем собственный тип «точка»:

```
struct Point // структура
{
   double x, y; // поля (данные) структуры
};
```

- Определения собственных типов, как правило, располагаются в заголовочных файлах (например, geometry.h)

Как лучше хранить в памяти набор треугольников

- Каждый из треугольников имеет три точки, у каждой из которых две координаты. Поэтому можно создать шесть массивов: ax[], ay[], bx[], by[], cx[], cy[]
  - такой подход возможен, но неудобен – слишком много переменных
- Можно создать двумерный массив с x-координатами трех точек: tx[][3] и y-координатами трех точек: ty[][3].
  - число переменных сокращается до двух
- А идеалом было бы создать собственный тип Triangle и хранить массив triangles[]

Реализация в C++, использование структуры

- Описание структурного типа «треугольник»:

```
// В языке C++
struct Triangle
{
    Point vertexes[3]; // три вершины треугольника
    double area; // площадь треугольника (если нужно)
};
// В языке C
struct Triangle
{
    struct Point vertexes[3]; // три вершины треугольника
    double area; // площадь треугольника (если нужно)
};
```

Реализация в C++, использование структуры

- Также мы можем использовать тип непосредственно в программе:

```
// В языке C++
int main(void)
{
   Point p1, p2;
   cin>>p1.x>>p1.y>>p2.x>>p2.y;
}
// В языке C
int main(void)
{
   struct Point p1, p2;
   scanf("%lf %lf %lf %lf", &p1.x, &p1.y, &p2.x, &p2.y);
}
```

Имена типов

- Формируются по обычным правилам формирования имен в языках C/C++, аналогично именам переменных и функций
- Рекомендуется, однако, придерживаться определенных правил, позволяющих быстро отличать переменные, функции и типы друг от друга

Имена типов

- Следующие правила применяются среди разработчиков на языке Java:
  - имена переменных и функций начинаются со строчной буквы, например varName
  - имя переменной обозначает предмет (существительное), например redBall
  - имя функции обозначает действие (глагол), например strike
  - имена типов начинаются с прописной буквы, например, LargeTriangle

Как создать переменную нового типа?

- Важно понимать, что, написав определение нового типа, мы **НЕ СОЗДАЕМ** никаких переменных этого типа. Для этой цели, необходимо написать определение переменной! Например:

```
Point p1, p2; // Две переменных типа «точка»
struct Point p1, p2; // То же в C
```

- Чтобы обратиться к полям структуры, используется оператор .

```
cout<<"Введите координаты точек"<<endl;
cin>>p1.x>>p1.y>>p2.x>>p2.y;
```

Как создать переменную нового типа?

- Если поле структуры само по себе является структурой, используется цепочка обращений. Например:

```
Triangle tr;
tr.vertexes[0].x=p1.x;
tr.vertexes[0].y=p1.y;
```

- Структуры (без дополнительных усилий) нельзя складывать, умножать, выводить в поток, вводить из потока и т.п. Однако, их можно присваивать, например:

```
tr.vertexes[1]=p2;
```

- Для того, чтобы использовать структуру, необходимо иметь ее определение – то есть, подключить заголовочный файл

Что мы выигрываем, используя структуры?

- В программе уменьшается число отдельных переменных. Например, для создания массивов координат мы вместо:

```
double* px = new double[pointNum];
double* py = new double[pointNum];
```

- напишем так:

```
Point* points = new Point[pointNum];
```

- При описании массива треугольников мы вместо:

```
double trX[maxTrNum][3], trY[maxTrNum][3];
```

- напишем так:

```
Triangle triangles[maxTrNum];
```

Что мы выигрываем, используя структуры?

- Уменьшается количество аргументов функций. Структуры (и классы), как правило, передаются в функции по ссылке (чтобы не тратить время на копирование значения), например:

```
double calcDistance(const Point& pa, const Point& pb)
// const указывает, что содержимое не будет меняться
{
   return sqrt((pb.x-pa.x)*(pb.x-pa.x)+
               (pb.y-pa.y)*(pb.y-pa.y));
}
```

- Для сравнения, вариант без структур:

```
double calcDistance(double xa, double ya,
                    double xb, double yb)
{
   return sqrt((xb-xa)*(xb-xa)+(yb-ya)*(yb-ya));
}
```

Вроде бы не стало короче. Но…

- Если раньше мы могли написать так:

```
double xa, xb, ya, yb;
cin>>xa>>ya>>xb>>yb;
// Найдите здесь ошибку
double dist=calcDistance(xa, xb, ya, yb);
```

- То теперь мы можем написать только так:

```
Point pa, pb;
cin>>pa.x>>pa.y>>pb.x>>pb.y;
// Максимум, что можно,
// это поменять точки местами
// Впрочем, такая замена ни на что не повлияет
double dist=calcDistance(pa, pb);
```

То же верно для других функций

- Вместо функции

```
double calcAreaByPoints(double xa, double ya,
                        double xb, double yb,
                        double xc, double yc)
{
   double ab=calcDistance(xa, ya, xb, yb);
   double bc=calcDistance(xb, yb, xc, yc);
   double ca=calcDistance(xc, yc, xa, ya);
   return calcAreaBySides(ab, bc, ca);
}
```

То же верно для других функций

- Мы имеем другую

```
double calcTriangleArea(Triangle& tr)
{
    Point* v = tr.vertexes;
    double a=calcDistance(v[0], v[1]);
    double b=calcDistance(v[1], v[2]);
    double c=calcDistance(v[2], v[0]);
    return tr.area=calcAreaBySides(a, b, c);
}
```

Функция calcAreaBySides

```
// Используется формула Герона
double calcAreaBySides(double a, double b, double c)
{
   double p2=(a+b+c)/2.0;
   // Не забываем проверить, что корень извлечется
   if (p2<=0.0 || p2<=a || p2<=b || p2<=c)
      return 0.0;
   return sqrt(p2*(p2-a)*(p2-b)*(p2-c));
}
```

Пример. Заголовочный файл file.h

```
#ifndef _FILE_H
#define _FILE_H

// Подсчет числа точек во входном файле
int countPoints(const char* fileName);
// Чтение точек из входного файла
bool readPoints(const char* fileName,
    Point* pointArray, int maxPointNum);
// Вывод треугольников в выходной файл
bool writeTriangles(const char* fileName,
    const Triangle* trArray, int trNum);

#endif
```

Пример. Заголовочный файл search.h

```
#ifndef _SEARCH_H
#define _SEARCH_H

// Поиск треугольников максимальной площади
void searchLargestTriangles(const Point* pointArray,
    int pointNum, Triangle* trArray, int maxTrNum);

#endif
```

Теперь попробуем написать функцию main

- Имена входного и выходного файлов будем читать из командной строки
- Нам необходимо
  - прочитать имена файлов и открыть их
  - посчитать точки во входном файле
  - записать их в массив
  - найти наибольшие треугольники
  - вывести результат в выходной файл

Пример. Главный файл main.cpp

```
#include "file.h"
#include "search.h"
#include <iostream>
#include <locale.h>
using namespace std;
int main(int argc, char** argv)
{
   setlocale(LC_ALL, "Russian");
   if (argc < 3)
   {
      cout<<"Запуск: Triangles.exe inf.txt outf.txt"<<endl;
      return -1;
   }
   const char* inFileName = argv[1];  // Имя входного файла
   const char* outFileName = argv[2]; // Имя выходного файла
   int pointNum = countPoints(inFileName);
```

Пример. Главный файл main.cpp

```
if (pointNum < 0)
{
   cout<<"Входной файл не существует"<<endl;
   return -2;
} else if (pointNum < 4)
{
   cout<<"Входной файл слишком мал"<<endl;
   return -3;
}
Point* pointArray = new Point[pointNum];
if (!readPoints(inFileName, pointArray, pointNum))
{
   cout<<"Неизвестная ошибка при вводе точек "<<endl;
   return -3;
}
```

Пример. Главный файл main.cpp

```
    const int maxTrNum = 3;
    Triangle trArray[maxTrNum];
    searchLargestTriangles(pointArray, pointNum,
                           trArray, maxTrNum);
    if (!writeTriangles(outFileName, trArray, maxTrNum))
    {
       cout<<"Не удалось записать результат"<<endl;
       return -4;
    }
    cout<<"Программа успешно завершена"<<endl;
    delete[] pointArray;
    return 0;
}
```

Файл file.cpp, функция countPoints

```
// Подключить file.h, fstream, std
int countPoints(const char* fileName)
{
   ifstream in(fileName);
   if (!in.is_open()) return -1;
   double x,y;
   int i;
   for (i=0; ; i++)
   {
      in>>x>>y;
      if (in.fail()) break;
   }
   return i;
}
```

Файл file.cpp, функция readPoints

```
bool readPoints(const char* fileName,
                Point* pointArray, int maxPointNum)
{
   ifstream in(fileName);
   // Файл не открыт
   if (!in.is_open())
      return 0;
   for (int i=0; i<maxPointNum; i++)
   {
      in>>pointArray[i];
      if (in.fail())
         return false;
   }
   return true;
}
```

Разрешается определить свои операции над структурами

- Например, собственные операции ввода:

```
istream& operator >>(istream& in, Point& p)
{
   in>>p.x>>p.y;
   // Результатом должен быть сам поток
   // Это позволяет команды типа cin>>a>>b>>c
   return in;
}
```

- Подобный прием называется **перегрузкой операторов**. По правилам, для бинарных операторов первым идет левый от оператора аргумент, вторым - правый. Например, выполнение операции ввода:

```
Point p;
cin>>p; // эквивалентно
operator >>(cin, p);
```

Аналогично определяется операция вывода точки

- Программист сам определяет удобный для него формат вывода. Например:

```
ostream& operator <<(ostream& out,
                     const Point& p)
{
    out<<"("<<p.x<<" "<<p.y<<")";
    return out;
}
```

- При этом, точка выводится в формате
  - (1 2)

Аналогично определяется операция вывода треугольника

- Например, так:

```
ostream& operator <<(ostream& out,
                     const Triangle& tr)
{
   out<<"A="<<tr.vertexes[0]<<
        " B="<<tr.vertexes[1]<<
        " C="<<tr.vertexes[2]<<
        " area="<<tr.area;
   return out;
}
```

Файл file.cpp, функция writeTriangles

```
bool writeTriangles(const char* fileName,
                    Triangle* trArray, int trNum)
{
   ofstream out(fileName);
   if (!out.is_open())
      return false;
   for (int i=0; i<trNum; i++)
      out<<"#"<<i+1<<": "<<trArray[i]<<endl;
   return true;
}
```

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**)

Файл search,cpp, вставка очередного треугольника

```
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;
}
```

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);
            }
        }
    }
}
```

Вспомогательная функция 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;
}
```

Вот и все!

- Демонстрация

Итоги

- Рассмотрено
  - Структура модуля в C/C++
  - Пример разбиения на модули в C/C++
  - Структурные типы
- Далее
  - Применение инкапсуляции