**СЕМИНАР 2. Организация класса. Точки на декартовой плоскости**

**Теоретический материал**

Конструктор — это специальная функция-член класса (метод), которая создает экземпляр класса, инициализируя его значения. Имя конструктора совпадает с именем класса.

Конструктор не возвращает значения, результат его работы – создается объект класса.

Как правило конструкторы объявляются в открытой (публичной) секции класса, поскольку деятельность конструктором заключается в создании объекта класса и производится извне класса.

Вызов конструкторов как правило производится не явно, то есть указывается

CLASS\_NAME obj;

При котором создается объект obj, а скобки вызова конструктора отсутсвуют.

**Конструктор со списком инициализации** приведен в примере.

Имеются две формы инициализации полей класса при создании объектов. Во всех приведенных выше определениях конструкторов была продемонстрирована одна из этих форм, которую условно можно назвать первой. Простейший ее пример:

TString(int n) { N = n; }

Эта форма предполагает использование операции присваивания в теле конструктора. Вторая форма предусматривает использование списка инициализаторов, который размещается между списком аргументов и телом конструктора:

**класс(аргументы):инициализаторы { операторы конструктора; }**

Такая форма инициализации считается более эффективной.

**class** Point {

**private**:

**int** x;

**int** y;

**public**:

Point(**int** \_х, **int** \_y) : x(\_x), y(\_y) { }; *// конструктор точки*

};

Выделенное – список инициализации. Последовательность инициализации данные – строго как указана в списке объявления в спецификации класса. Если изменить последовательность – компилятор выдаст предупреждения (warning)

**Конструктор по умолчанию** – конструктор не имеющий параметров. Если конструктор по умолчанию не задан явно и нет конструкторов с параметрами, то он создастся автоматически, при этом поля объекта будут инициализированы мусором.

**class** Point {

**public**:

Point(); *// конструктор по умолчанию*

};

Еще один вариант конструктора по умолчанию – конструктор, всем параметрам которого присвоены значения.

**DATA(int a=0, int b=10);**

Такой конструктор может вызываться как с аргументами

**DATA Н(3, 4);**

Так и без аргументов (как обычный конструктор по умолчанию:

**DATA R;**

Существует промежуточный вариант – когда части параметром заданы значений по умолчанию, а второй части параметров требуется инициализация значений.

**DATA(int а, int b=10);**

Создание объекта таким конструктором будет выглядеть следующим образом

**DATA G(5);**

ВАЖНО: аргументу можно задать значение по умолчанию, если он находится правее всех в списке аргументов, или если все аргументы правее него имеют значения по умолчанию.

В рамках одного класса может быть объявлен только один конструктор по умолчанию так как в противном случае конструкторы будут неразличимы при вызове.

Если необходимо создать массив объектов, это реализуется вызовом конструктора по умолчанию. Альтернативный вариант – конструктор с инициализированными полями.

**Копирующий конструктор**. Конструктор, который в качестве аргумента принимает другой объект того же типа — называется **копирующим конструктором**. В этом случае новый объект будет создан как копия уже существующего объекта. Обычно аргумент копирующего конструктора объявляется как ссылка на константу, чтобы копия никак не влияла на оригинал. Копирующий конструктор создаётся по умолчанию, если не задан явно. В поля нового объекта он копирует значения соответствующих полей объекта-аргумента.

Общий синтаксис конструктора копирования

**My\_Class(const My\_Class&);**

// здесь My\_Class - это имя класса

const My\_Class& - ссылка на объект этого же класса;

Использование спецификатора *const* не случайно: как известно, выражение вызова функции с параметром типа X ничем не отличается от выражения вызова функции, у которой параметром является ссылка на объект типа X. При вызове такой функции не приходится копировать объекты как параметры. Передача адреса не требует копирования объекта, а значит, при этом не будет и рекурсии.

Использование стандартного конструктора копирования в случае, если в классе нет полей, используемых для динамического выделения памяти, не представляет никакой особенности. Ситуация меняется, если у класса есть поля, используемые для динамического выделения памяти, как, например поле **str** в TString. В этом случае стандартный конструктор копирования будет напрямую присваивать значение указателя на динамическую память для одного (существующего) объекта указателю для другого (нового) объекта. В итоге два разных указателя будут адресовать одну и ту же область памяти.

Если один из этих объектов изменяет содержимое динамической памяти, то изменяется содержимое такой памяти и для второго объекта, что является неправильным (в большинстве алгоритмов обработки объектов, кроме специально предусмотренных случаев). Кроме того, если один из этих объектов является локальным в блоке, то при выходе из блока он уничтожается и занимаемая им память считается свободной для дальнейшего

30использования. Но другой объект всѐ ещѐ существует, и его память не должна считаться свободной и подвергаться изменениям.

Для предотвращения такой ситуации, если она нежелательна, нужно явно создать конструктор копирования (нестандартный), который будет не просто напрямую копировать значения полей, но и, при необходимости выделять динамическую память и копировать в новую область памяти данные из такой же памяти исходного объекта.

**Деструкторы**

     Деструкторы выполняют работу, обратную той, что проделывают конструкторы. Хотя класс может иметь несколько конструкторов, но деструктор может быть только один. Синтаксис деструктора очень похож на синтаксис конструктора по умолчанию.

**~CLASS\_NAME();**

Деструктор почти всегда вызывается неявно. Вызов деструктора происходит:

1. при выходе объекта за пределы своей области видимости: выходе программы из блока, где создавались автоматические объекты классов, при завершении программы (вызове функции exit) для глобальных и статических объектов
2. при уничтожении динамического созданного (операцией new) объекта операцией delete.

Деструкторы всегда вызываются в порядке, обратном последовательности определения объектов

Если в классе нет явно созданного деструктора, то, также как и конструктор, компилятор создаѐт его автоматически, с открытым режимом доступа.

Типы данных **Class** и **struct**:

Они очень похожи, отличительная особенность – в структуре все поля по умолчанию открыты (public), а в классе – закрыты, то есть не доступны для внешнего обращения.

**New/delete и malloc(calloc)/free**

* malloc выделяет неинициализированную память. Её нужно освобождать, используя free.
* calloc делает то же, что и malloc, только выделенная память инициализируется константой (0). Освобождается с помощью free.
* new выделяет и инициализирует память, вызывая конструктор (если память выделяется под объект). Освобождается вызовом оператора delete (который, по сути, вызывает деструктор объекта).

**Оператор malloc** выделяет блок памяти. Размер блока задается в байтах. Возвращает адрес выделенного блока или ноль, если не удалось выделить память.

new и delete являются улучшенными вариантом операторов malloc и free.

new позволяет выделять память, необходимую для размещения переменной, структурной переменной или массива. Для переменной необходимо указывать тип и в случае массива - размер. new может записывать значение в созданную переменную.

В отличие от malloc, при использовании оператора new ненужно указывать количество байт памяти, которое необходимо выделить.

Следует отметить, что допустимо только совместное использование однотипных средств выделения/освобождения памяти: New/delete и malloc(calloc)/free

Варианты New/free и malloc(calloc)/delete не допустимы!

**Компоненты пустого класса:**

Таким образом, в пустом классе вида class A{}; имеются:

– стандартный конструктор;

– стандартный конструктор копирования;

– деструктор;

– стандартная перегруженная операция присваивания.

**Дружественная функция** — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным в полях private или protected. Дружественная функция может располагаться в любом поле класса – private, public или protected. Она при любых обстоятельствах будет иметь доступ к private-элементам класса и, даже если она сама находится в поле private (как в нашем примере), к ней можно будет обратиться вне класса, не используя специальных методов.

Их можно объявлять и определять в любом  поле класса, будь то private, public или protected,

Объявление дружественных функций отличается от объявления методов класса тем, что перед типом возвращаемого функцией значения используется зарезервированное слово friend.

Обратите внимание, что в  виде параметра мы передаем этим функциям ссылку на объект нашего класса.

Дружественная функция  не принадлежит классу, не является его компонентом, поэтому при ее описании не используется оператор ::

Дружественные функции широко используются когда одну функцию необходимо использовать для нескольких классов.

***Статические компоненты класса***

Члены класса, как поля, так и функции, могут быть определены (для полей точнее сказать, объявлены) в классе как статические с помощью спецификатора **static**.

Статические поля не тиражируются при создании объектов класса, в отличие от обычных полей, а существуют каждое (например, stN1 и stN2) в единственном экземпляре и являются общими для всех объектов класса.

К закрытым статическим полям можно обращаться только через вызовы функций класса. Однако для вызовов обычных функций класса требуется наличие объектов. Объектов может быть несколько и тогда требуется выбрать какой-либо один из них для вызова функции, работающей со статическим полем, либо объектов может вообще не быть. В обоих случаях для обращения к закрытым статическим полям должны использоваться статические функции.

Если вы создаете *статический* метод, ваша программа может вызывать такой метод, даже если объекты не были созданы.

inline-функция — это такая функция, чье тело подставляется в каждую точ­ку вызова, вместо того, чтобы генерировать код вызова. Причина использования inline-функции заключается в их эффективности. Всякий раз, когда вы­зывается функция, необходимо выполнить серию инструкций для формирования вызова функции, вставки аргументов в стек и возврата значения из функции. В некоторых случаях для этого прихо­дится использовать много тактов центрального процессора. При использовании inline-функции нет необходимости в таких дополнительных действиях и скорость выполнения программы возрастает.

Inline-функция будет растиражирована в коде компилятором столько раз, сколько она вызывается - это делается для того, чтобы минимизировать время работы со стеком, регистрами. Эти функции приводят к разбуханию исполняемого кода.

Однако в тех случаях, когда размер inline-функции достаточно большой, общий объем программы также возрастает. Поэтому в качестве inline-функции обычно используются очень маленькие функ­ции. Большие функции реализуются обычным способом.

Важно ясно понимать, что технически inline представляет запрос к компилятору сгенерировать подставляемый код, inline не является командой. В некоторых ситуациях компилятор не будет выполнять этот запрос.

**Разбор программ**

#include <stdio.h> *//библиотека функций стандартного ввода/вывода работы с файлами + printf/scanf*

#include <math.h> *//библиотека для выполнения простых математических операций - cos/acos, возведение в*

*квадрат и т.д.*

*// класс точки плоскости*

**class** Point

{

**private**:

**int** x;

**int** y;

**public**:

Point(**int** X, **int** Y) : x(X), y(Y) {}; *// конструктор точки со списком инициализации*

**int** getx() { **return** x; }; *// функция доступа к x*

**int** gety() { **return** y; }; *// функция доступа к y*

**friend** **double** distance(Point\*, Point\*); *// дружественная функция определения расстояния между 2-*

*мя точками*

};

**double** distance(Point\* r, Point\* q)

{

**double** d2=(r->x-q->x)\*(r->x-q->x)+(r->y-q->y)\*(r->y-q->y);

**return** (sqrt(d2));

}

*//Основная функция*

**int** main(**int** argc, **char**\* argv[ ])

{

**double** sum; *// суммарное расстояние от первой до остальных точек*

Point\*\* p; *// адрес массива адресов точек набора*

Point\* t; *// адрес текущей точки*

**int** x, y; *// координаты точки*

**int** i=0;

**int** j=0;

**if**(argc < 2) *// анализ числа аргументов командной строки*

**return**(-1);

p = **new** Point\*[argc]; *// распределить память под массив адресов точек*

**while**(++i < argc)

{ *//цикл ввода точек из командной строки*

sscanf(argv[i], **"(%d%\*c%d)"**, &x, &y); *// получение координат точки*

p[j]=**new** Point(x,y); *// динамическое конструирование точки*

j ++;

}

p[j]=NULL; *// маркировка конца массива адресов точек набора*

**for** (i=0; p[i] != NULL; i++)

{ *// цикл вычисления средних расстояний*

sum = 0.0; *// инициализировать сумму расстояний до текущей точки*

t = p[i]; *// фиксировать адрес текущей точки*

*// цикл суммирования расстояний от текущей точки до*

*остальных*

**for** (j=0; p[j] != NULL; j++)

sum += distance(t, p[j]); *//вычисление среднего расстояния до текущей точки*

printf(**"Dcp(%d;%d)=%f\n"**, t->getx(), t->gety(), sum/(argc-2));

}

*// освобождить дин. память точек набора по их адресам*

**for**(j=0; p[j] != NULL; j++)

**delete** p[j];

**delete** [ ]p; *// освободить дин. память массива адресов точек*

**return** 0; *//корректное завершение программы*

}

#include <stdlib.h>

#include <math.h>

#include <iostream.h>

*// Класс Point2 точки плоскости*

**class** Point

{

**private**:

**float** x;

**float** y;

Point(**float** X, **float** Y) : x(X), y(Y) {}; *// приватный конструктор*

**public**:

**float** getx() { **return** x; }; *// доступ к x*

**float** gety() { **return** y; }; *// доступ к y*

**static** Point Polar(**float**, **float**); *// именованный конструктор точки - статический метод*

Point() { x=y=0.0;}; *// конструктор по умолчанию (для массива точек)*

};

*// конструирование точки по полярным координатам*

Point Point::Polar(**float** R, **float** F)

{

Point p(R\*cos(F), R\*sin(F)); *// перевод полярных координат в декартовы*

**return** p; *// возврат объекта класса точки*

}

*// основная функция*

**int** main(**int** argc, **char**\* argv[ ])

{

**float** edge = 1.0; *// сторона треугольника (1.0 – длина по умолчанию)*

**float** angle = 0.0; *// полярный угол вершины*

**float** pi = acos (-1.0 ); *// вычислить pi (можно M\_PI из math.h)*

**float** radius; *// радиус-вектор вершины*

**int** i = 0; *// счетчик*

Point vertex [3]; *// массив вершин треугольника vertex[0], vertex[1], vertex[2]*

**if**(argc > 1)

edge = atof (argv[1]); *// получить длину стороны из командной строки*

radius = edge / (2\*sin(pi/3.0)); *// радиус описанной окружности*

**while** (i<3)

{ *// цикл вычисления декартовых координат вершин*

vertex[i] = Point::Polar(radius, angle);

angle += ((2.0\*pi)/3.0); *// +120 градусов =2\* pi/3*

i++;

}

**do**

{ *// Печать декартовых координат вершин в обратном порядке*

--i;

cout << vertex[i].getx()<<**';'**<<vertex[i].gety()<<endl;

} **while**(i>0);

**return** (0); *// Корректное завершение программы*

}