

ОСНОВЫ Программирования

Лекция #4

28/09/2019

Темы:

1. Инкапсуляция
2. Константные методы
 - mutable
3. Классы
4. this
5. Конструкторы
 1. По умолчанию
 2. Копирования
 3. Преобразования
6. Деструктор

Группировка данных

```
double length(double x1, double y1,  
              double x2, double y2);
```

Группировка данных

```
double length(double x1, double y1,  
              double x2, double y2);
```

```
struct Point {  
    double x;  
    double y;
```

```
};
```

```
struct Segment {  
    Point p1;  
    Point p2;
```

```
};
```

```
//....
```

```
double length(Segment s);
```

Работа со структурами

```
double length(Segment s) {  
    double dx = s.p1.x - s.p2.x;  
    double dy = s.p1.y - s.p2.y;  
    return sqrt(dx * dx + dy * dy);  
}
```

```
double length(Segment* s) {  
    double dx = s->p1.x - s->p2.x;  
    double dy = s->p1.y - s->p2.y;  
    return sqrt(dx * dx + dy * dy);  
}
```

Инкапсуляция – базовый принцип ООП

- Инкапсуляция, или сокрытие реализации, является фундаментом объектного подхода к разработке ПО.
 - Следуя данному подходу, программист рассматривает задачу в терминах предметной области, а создаваемый им продукт видит как **совокупность абстрактных сущностей – классов** (в свою очередь формально являющихся пользовательскими типами).
 - Инкапсуляция предотвращает прямой доступ к внутреннему представлению класса из других классов и функций программы.
 - Без нее теряют смысл остальные основополагающие принципы объектно-ориентированного программирования (ООП): наследование и полиморфизм.
 - Сущность инкапсуляции можно отразить формулой:
Открытый интерфейс + скрытая реализация

Инкапсуляция

```
struct Segment {  
    Point p1;  
    Point p2;  
    double length() {  
        double dx = p1.x - p2.x;  
        double dy = p1.y - p2.y;  
        return sqrt(dx * dx + dy * dy);  
    }  
};
```

Инкапсуляция

```
struct Segment {  
    Point p1;  
    Point p2;  
    double length();  
};  
  
double Segment::length() {  
    double dx = p1.x - p2.x;  
    double dy = p1.y - p2.y;  
    return sqrt(dx * dx + dy * dy);  
}
```


Константные методы

- Константные методы не могут изменять поля объекта

```
struct Segment {  
    Point p1;  
    Point p2;  
    double length() const {  
        double dx = p1.x - p2.x;  
        double dy = p1.y - p2.y;  
        return sqrt(dx * dx + dy * dy);  
    }  
};
```

- У константных объектов можно вызывать только константные методы
- Внутри константных методов можно вызывать только константные методы

Ключевое слово mutable

- Позволяет определять поля, которые можно изменять внутри константных методов

```
struct IntArray {  
    Point p1;  
    Point p2;  
    double length() const {  
        counter++;  
        double dx = p1.x - p2.x;  
        double dy = p1.y - p2.y;  
        return sqrt(dx * dx + dy * dy);  
    }  
private:  
    mutable int counter;  
};
```

Класс: в узком или широком смысле?

- **Принцип инкапсуляции распространяется** не только на **классы** (class), но и на **структуры** (struct), а также **объединения** (union). Это связано с расширительным толкованием понятия "класс" в языке C++, трактуемом как в узком, так и широком смысле:
 - **Класс в узком смысле** – одноименный составной пользовательский тип данных являющийся контейнером для данных и алгоритмов их обработки. Вводится в текст программы определением типа со спецификатором class
 - **Класс в широком смысле** – любой составной пользовательский тип данных, агрегирующий данные и алгоритмы их обработки. Вводится в текст программы определением типа с одним из спецификаторов struct, union или class.
- Каждое определение класса вводит **новый тип данных**.

Указатель this

- Указатель `this` – неявно определяемый константный указатель на объект класса, через который происходит вызов соответствующего нестатического метода.
- Указатель `this` допускает разыменование (`*this`) и его применение внутри методов допустимо, но чаще всего излишне. Исключение составляют 2 ситуации:
 1. Сравнение адресов объектов:
`if (this != someObj) /*...*/`
 2. Оператор `return`
`return *this;`

Указатель this

```
struct Point {  
    double x;  
    double y;  
  
    void shift(double x, double y) {  
        this->x += x;  
        this->y += y;  
    }  
};
```

Класс как область видимости

- **Класс** – наряду с блоком, функцией и пространством имен – **является конструкцией C++**, которая **вводит** в состав программы одноименный **область видимости**. (Строго говоря, область видимости вводит определение класса, а именно его тело.)
 - Все члены класса видны в нем самом с момента своего объявления. Порядок объявления членов класса важен: нельзя ссылаться на члены, которые предстоит объявить позднее. Исключение составляют разрешение имен, используемых как аргументы по умолчанию.
- В области видимости класса находится не только его тело, но и внешние определения его членов: методов и статических атрибутов.

Конструкторы и деструкторы

- **Конструктор** – метод класса, автоматически применяемый к каждому экземпляру (объекту) класса перед первым использованием (в случае динамического выделения памяти – после успешного выполнения операции new).
- Освобождение ресурсов, захваченных в конструкторе класса либо на протяжении времени жизни соответствующего экземпляра, осуществляет **деструктор**.
- В связи с принятым по умолчанию почленным порядком инициализации и копирования объектов класса в большинстве случаев возникает необходимость в реализации, - наряду с конструктором по умолчанию, - **конструктор копирования** и перегруженной **операции-функции присваивания** operator=

Конструкторы и деструкторы

- **Выполнение** любого конструктора состоит из двух фаз:
 1. Фаза явной (неявной) инициализации (обработка списка инициализации)
 2. Фаза вычислений (исполнение тела конструктора)
- Конструктор **не может определяться** со спецификатором `const`. Константность и объекта устанавливается по завершении работы конструктора и снимается перед вызовом деструктора.

Инициализация без конструктора

- Класс, все члены которого открыты, может задействовать механизм **явной позиционной инициализации**, ассоциирующий значения в списке инициализации с членами данных в соответствии с их порядком.

```
struct Sample {  
    int          int_prm;  
    double      dbl_prm;  
    std::string  str_prm;  
};  
//...  
Sample sample = { 1, -3.14, "hello world" };
```

Инициализация без конструктора

- Преимущество:
 - Скорость и эффективность, особо значимые при выполнении во время запуска программы (для глобальных объектов).
- Недостатки:
 - Только для классов, члены которых открыты
 - Отсутствие поддержки инкапсуляции и абстрактных типов
 - Требование предельной точности и аккуратности в применении

Конструкторы по умолчанию

- Явный конструктор по умолчанию **не требует задания значений** его параметров, хотя таковые могут присутствовать в сигнатуре (но в таком случае должны иметь значения по умолчанию)

```
struct Sample {  
    Sample(int ipr = 0, double dpr = 0.0);  
    //...  
};
```

Конструкторы по умолчанию

- Если в классе определен хотя бы один конструктор с параметрами, то при использовании класса со стандартными контейнерами и данническими массивами экземпляров конструктор по умолчанию **обязателен**.

```
Sample *samples = new Sample[Num_Of_Samples];
```

Если конструктор по умолчанию **не определен**, но существует хотя бы один конструктор с параметрами, в определении объектов должны присутствовать аргументы. Если ни одного конструктора не определено, объект класса не инициализируется.

Конструкторы с параметрами

```
struct Sample {  
    Sample(int prm) : _prm(prm) { }  
private:  
    int _prm;  
};
```

```
// все вызовы конструктора допустимы и эквивалентны  
Sample sample1(10),  
    sample2 = Sample(10),  
    sample3 = 10; // для одного аргумента
```

Массивы объектов

```
// массивы объектов класса определяются аналогично массивам  
// объектов базовых типов
```

```
// для конструктора с одним аргументом  
Sample array1[] = { 10, -5, 0, 127 };
```

```
// для конструктора с несколькими аргументами  
Sample array2[5] = {  
    Sample(10, 0.1),  
    Sample(-5, -3.6),  
    Sample(0, 0.0),  
    Sample() // если есть конструктор по умолчанию  
};
```

Конструкторы копирования

- Конструктор копирования принимает в качестве первого формального параметра **ссылку** на существующий объект класса. Другими словами, этот параметр имеет тип T&, const T&
 - *Второй и последующие параметры конструктора копирования, если есть, должны иметь значение по умолчанию.*
- В случае отсутствия явного конструктора копирования в определении класса производится почленная инициализация объекта по умолчанию

```
struct Sample {  
    Sample(const Sample &rhs);  
    //...  
};
```

Конструкторы преобразования

- **Конструкторы преобразования** служат для построения объектов класса по одному или нескольким значениям иных типов
- **Операции преобразования** позволяют преобразовать содержимое объектов класса к требуемым типам данных

```
struct Sample {  
    // конструкторы преобразования  
    Sample(const char* c);  
    Sample(const std::string& s);  
    // операции преобразования  
    operator int() { return int_prm; }  
    operator double()    { return dbl_prm; }  
};
```


Конструкторы

- Итого, при простом объявлении класса

```
struct A { }
```

Неявным образом создаются следующие методы:

1. `A();` // конструктор по умолчанию
2. `A(const A& a);` // конструктор копирования
3. `operator= (const A& a);` // копирующий оператор
4. `A(const A&& a);` // конструктор перемещения
5. `operator= (const A&& a);` // оператор перемещения
6. `~A();` // деструктор

Если в реализации класса используется доступ к динамически выделенной памяти (или указатели), то хорошим тоном является переопределение конструктора копирования

Деструкторы.

- Деструктор – не принимающий параметров и не возвращающий результат метод класса, автоматически вызываемый при выходе объекта из области видимости и применении к указателю на объект класса операции delete

```
struct Sample {  
    // ....  
    ~Sample();  
};
```

Примечание: деструктор не вызывается при выходе из области видимости ссылки или указателя на объект.

Деструкторы.

- Задачи деструктора:
 1. Освобождение (возврат) системных ресурсов, главным образом – оперативной памяти
 2. Заккрытие файлов или устройств
 3. Снятие блокировок, таймеров и т.д.

ЯВНЫЙ ВЫЗОВ ДЕСТРУКТОРОВ

- Потребность в явном вызове деструктора обычно связана с необходимостью **уничтожить** динамически размещенный объект **без освобождения памяти**.

```
char* buf = new char[sizeof(Sample)];  
// “размещающий” вариант new  
Sample* sample1 = new (buf) Sample(100);  
// ...  
sample1->~Sample(); // вызов 1  
Sample* sample2 = new (buf) Sample(200);  
// ...  
sample2->~Sample(); // вызов 2  
delete[] buf;
```