Структуры и классы

Artem Isaev, Sergey Nikulin

2016 CMF

Структуры

- Структура – объединение разнородных типов данных. Допускает вложенность.

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;

struct vanilla_option {
    string type
    string sub_type;
    double price;
    string expiration_date;
    int size;
};
```

← объявление структуры

← поля структуры

Структуры

```
int main(){
    vanilla_option american_call;
    american_call.type = 'Call';
    american_call.sub_type = 'American';
    american_call.price = 1000;
    american_call.expiration_date = '15.03.2016';
    american_call.size = 11;
    cout<<"Тип" <<american_call.type;
    cout<<"Подтип" <<american_call.sub_type;
    cout<<"Цена" <<american_call.price;
    cout<<"Дата Экспирации" <<american_call.expiration_date;
    cout<<"Размер позиции" <<american_call.size;
}
```

← определение структурной переменной

← инициализация полей

Классы

- Класс – объединение данных класса и функций (методов класса).

Определение класса:

```
1  class dress {
2      private:
3          int size;
4          double length;
5          double sleeve_length;
6      public:
7          void setdata(int s, double l,
8              double sl_l) {
9                  size = s;
10                 length = l;
11                 sleeve_length = sl_l;
12             }
13         void showdata() {
14             cout <<  "Размер платья" << size;
15             cout << ", длина" << length;
16             cout << ", длина рукава" << sleeve_length;
17             }
18 };
```

Классы

- ***private*** – данные, защищенные от доступа функций вне класса
- ***public*** – данные доступные за пределами класса
- Определение объектов:

```
dress mine1, mine2;
```

- Вызов методов класса:

```
mine1.setdata( 42, 1.05, 0.75);
mine2.setdata( 42, 0.9, 1.0);
```

Конструкторы

- Конструктор – метод класса, выполняющийся автоматически в момент создания объекта. (например, автоматическая инициализация полей)

Счетчик:

```
4  class Counter {
5      private:
6          usigned int count;
7      public:
8          Counter():count(0)
9              {/*пустое тело*/}
10         void inc_count()
11             {count ++;}
12         int get_count()
13             {return count;}
14 };
```

конструктор

Конструкторы

- Конструктор должен быть ***public***
- Имя конструктора в точности совпадает с именем класса
- У конструкторов не существует возвращаемого значения
- Конструктор может принимать параметры:

```
Counter (unsigned int i): count(i) {}
```

Создание объекта: `Counter count1(1);`

- Инициализации нескольких параметров:

```
NewClass(): x1(0), x2(4), y1(33) {}
```

- Конструкторов может быть несколько, но они должны отличаться числом передаваемых параметров. Конструктор без параметров – конструктор по умолчанию.

Деструктор

- Деструктор – метод класса, вызываемы при уничтожении объекта.
- Имя деструктора совпадает с именем конструктора и предваряется символом ~

```
~Counter() {}
```

- Объявляется в разделе ***public***
- Деструктор не возвращает значений и не принимает параметров
- В классе может быть объявлен только один деструктор, так как уничтожение объекта возможно только одним способом
- Применение: освобождение памяти, выделенной конструктором при создании объекта

Пример

```
1  #include <iostream>
2  using namespace std;
3
4  class MyClass {
5      private:
6          double f;
7          double g;
8      public:
9          MyClass():f(0.0), g(0.0){
10             cout << "Default constructor!"<< endl;}
11         MyClass(double x, double y):f(x), g(y) {
12             cout << "Constructor with parameters!"<< endl;}
13         void ShowData() {
14             cout << "f = " << f << endl;
15             cout << "g = " << g << endl<< endl;}
16         ~MyClass() {
17             cout << "Destructor"<< endl;}
18 };
```

Пример

```
20  int main() {
21      MyClass obj1;
22      MyClass obj2(2.2, 3.0);
23
24      obj1.ShowData();
25      obj2.ShowData();
26
27      return 0;
28  }
```

Что будет результатом работы такой программы?

Определение методов класса вне класса

Внутри класса можно создавать только лишь прототип функции, в которой будут передаваться типы используемых переменных:

```
void Portfolio_Sum(vanilla_option, vanilla_option);
```

Сама же функция может быть вынесена за пределы класса:

```
void vanilla_option :: Portfolio_Sum(vanilla_option o1, vanilla_option o2){
summprice = o1.price + o2.price;
}
```

Из чего состоит функция?

```
void vanilla_option :: Portfolio_Sum(vanilla_option o1, vanilla_option o2){
```

Возвращаемый тип — Имя класса — Операция разрешения — Имя функции — Аргументы функции

Объекты в качестве аргументов

Пример:

```
using namespace std;

class Distance{
    int feet;
    float inches;
    public Distance: feet(0),inches(0.0) /Конструктор без аргумента
    {}
    void Distance :: add_dist(Distance d2, Distance d3){
        inches = d2.inches + d3.inches; /Сложение дюймов
        feet = 0                        /С возможным заемом
        if(inches >= 12.0)
        {
            inches-=12.0;               /Уменьшаем числ
            feet++;                     /Увеличиваем число футов
        }
        feet += d2.feet + d3.feet;
        }
    }
    }
    int main(){
        Distance dist1, dist3;
        Distance dist2(11, 6.25);
        dist3.add_dist(dist1, dist2); /Функция принимает, как аргументы - объекты( dist3 = dist1 + dist2)
    }
};
```

Объекты в качестве аргументов

В данном примере можно заметить, что функция принимает на вход два объекта, dist2 и dist1 и записывает полученные результат в другой объект класса Distance:

```
dist3.add_dist(dist1, dist2);
```

Аргумент dist3 в данном случае можно рассматривать, как псевдоаргумент функции add_dist(): формально он не является аргументом, но функция имеет доступ к его полям.

Типом исходной функции является void(), поэтому полученный результат в ходе выполнения работы функции автоматически присваивается переменной dist3.

Использование классов. Пример карточной игры.