Лекция 5

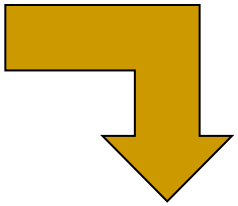
Методы классов

Объект как аргумент функции

Особенностью языка C++ является возможность использования объекта некоторого класса как **аргумента функции**.

```
class Rvector
{
   public:
      float x, y;
};
```

Класс радиус-вектора на плоскости



Функция вычисления модуля

```
float module(Rvector r)
{
    return sqrt(r.x*r.x + r.y*r.y);
}
```

Объект как возвращаемое значение

Объект некоторого класса может также быть **возвращаемым значением** функции. В этом случае внутри функции создается новый объект, над которым производятся необходимые операции. Этот объект далее передается как возвращаемое значение.

```
Rvector polar(float r, float a)
{
    Rvector tmp;
    tmp.x = r*cos(a);
    tmp.y = r*sin(a);
    return tmp;
}
```

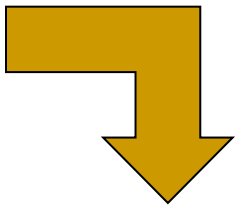
Радиус-вектор задан в полярных координатах модулем (r) и полярным углом (a)

Объект в методах класса

```
class Rvector
{
   public:
     float x, y;
     Rvector add(Rvector r1);
};

Rvector Rvector::add(Rvector r1)
{
   Rvector tmp;
   tmp.x = x + r1.x;
   tmp.y = y + r1.y;
   return tmp;
}
```

Реализует функцию сложения с другим вектором (r1)



Использование метода

```
Rvector r1, r2, r3;
...
r3 = r1.add(r2);
```

Способы инициализации объекта

Как известно, инициализация полей объекта является задачей конструктора объекта (конструктора по умолчанию, конструктора с параметрами и др.).

Рассмотрим несколько способов инициализации

```
Rvector r1;
Rvector r2(8, 12);
Rvector r3 = r2;
```

В последнем случае объект **r3** инициализируется значениями полей объекта **r2**. Фактически происходит копирование одного объекта в другой.

Копирование объектов

Всякий раз, когда объект некоторого класса передается в функцию или возвращается из нее, происходит копирование этого объекта в стек.

По умолчанию, компилятор реализует "поверхностное" копирование, при котором дублируются значения всех полей объекта.

Однако это не всегда приемлемо, например, в случаях, когда объект содержит указатели или ссылки на неразделяемые объекты (файлы, потоки ввода-вывода и др.). В этих случаях требуется определить специальный метод - **конструктор копирования**.

Конструктор копии

Конструктор копии (copy constructor) - конструктор, применяемый для создания нового объекта как копии уже существующего. Принимает как минимум один аргумент: ссылку на копируемый объект.

Примеры объявления конструктора копии

```
X(const X&);
X(X&);
X(const X&, int = 10);
X(const X&, double = 1.0, int = 40);
```

Первый конструктор должен применяться в случае отсутствия остальных.

Пример определения конструктора копирования:

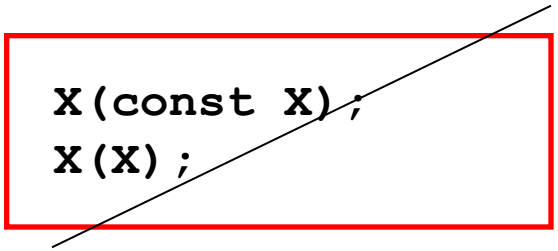
```
class Person
{
  public:
    char *name;
    int age;
    ...
    Person(const Person &p);
    ...
};
```

Сначала объявляем конструктор копии внутри класса Person

Далее определяем его, используя "глубокое" копирование. Для этого выделяем новый участок памяти под поле name.

```
Person::Person(const Person &p)
{
   if (name!=NULL)
      delete []name;
   name = new char[strlen(p.name)];
   strcpy(name, p.name);
   age = p.age;
}
```

Следующие объявления конструкторов копирования некорректны, так как приводят к бесконечному рекурсивному вызову самих себя:



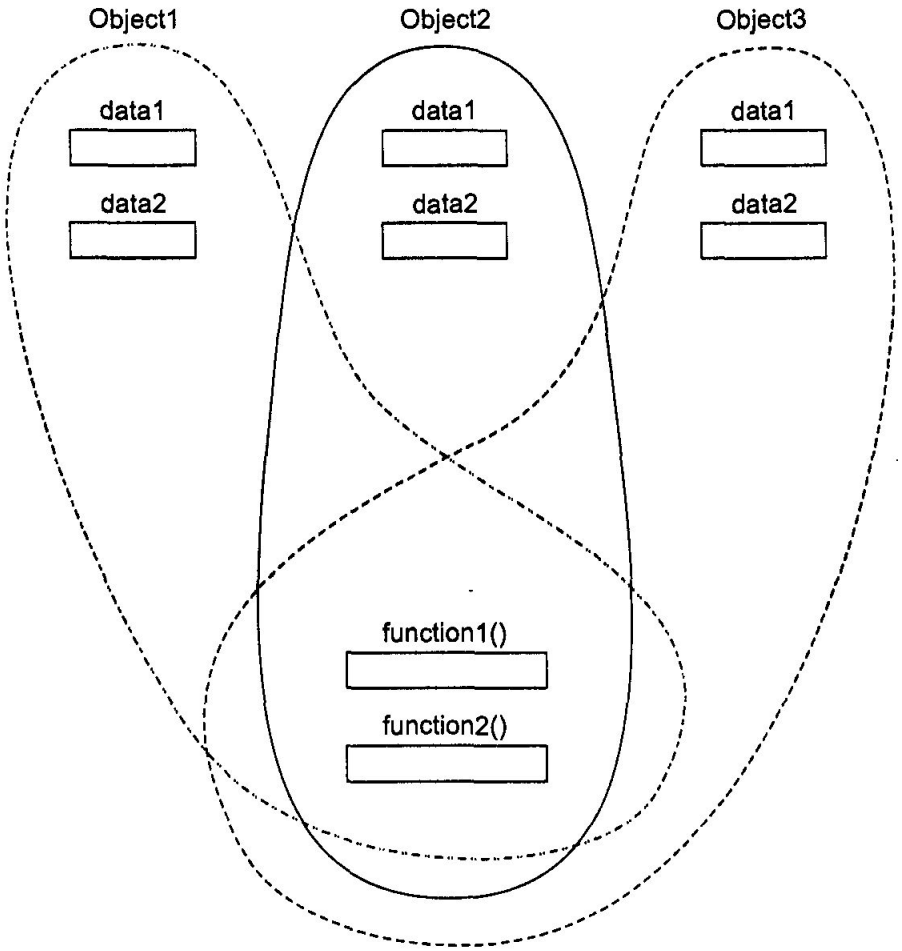
Конструктор копии вызывается в одном из 4 случаев:

1) объект является возвращаемым значением
2) объект передается функции по значению в качестве аргумента
3) объект конструируется на основе другого объекта (того же класса)
4) компилятор генерирует временный объект (при преобразованиях и т.д.)

Правило Трех

Явное определение конструктора копии необходимо в случаях, когда требуется "глубокое копирование" объектов. Эта ситуация возникает в случаях, когда объекты хранят указатели или другие неразделяемые ссылки. Как правило, в этих случаях кроме **конструктора копии**, требуется также явное определение **деструктора** и **оператора присваивания** (Правило Трех).

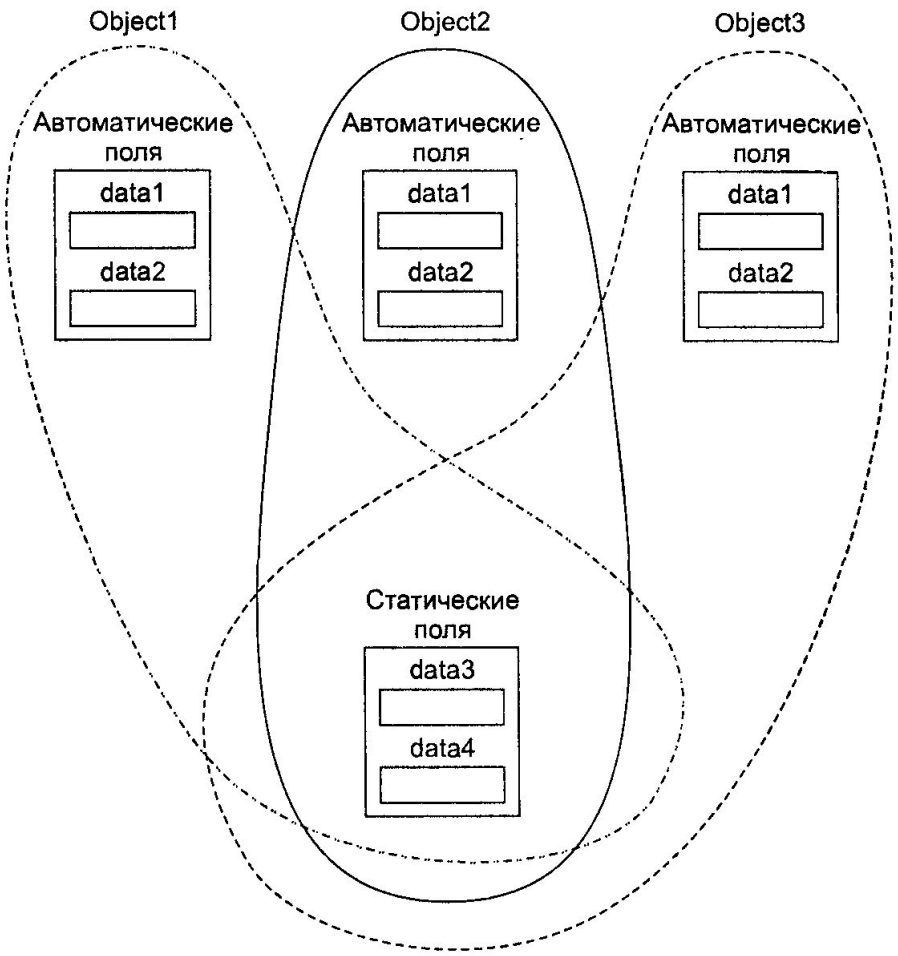
Поля и методы объекта в памяти



Каждый объект класса обладает своими собственными значениями полей данных, которые не зависят друг от друга (см. рис.). Таким образом, **состояние** каждого объекта индивидуально. С другой стороны, методы (функции) являются общими для всех объектов. То есть, **поведение** всех объектов идентично.

Статические поля класса

Однако существуют поля данных, которые являются общими для всех экземпляров класса. Такие поля называются **статическими**. Эти поля объявляются с ключевым словом static. Существует только один экземпляр статического поля, видимый всеми объектами (рис., поля data3 и data4).



Пример: счетчик объектов

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

class counter
{
   private:
      static int count;  // объявляем стат. поле
   pu
      
};

int c
```

```
int main()
{
   counter f1, f2, f3;      //создаем 3 объекта

   cout << f1.getcount() << endl; //все объекты
   cout << f2.getcount() << endl; //выведут на экран
   cout << f3.getcount() << endl; //одно значение 3
   return 0;
}
```

Константные методы

Константные методы - это методы, которые не изменяют значения полей своего объекта (то есть, не изменяют его состояние). Такие методы объявляются с ключевым словом const.

Пример – функция вывода на экран всех закрытых полей объекта. Если функция только выводит их на экран, не изменяя при этом самих значений, то она может быть объявлена как константная.

Попытка изменить значение поля объекта внутри константного метода будет приводить к ошибке.

```
class aClass
{
private:
   int alpha;
public:
   void func1()          // не-const метод
   {
      alpha = 99;        // OK
   }

   void func2() const    // const метод
   {
      alpha = 99;        // ОШИБКА!
   }
};
```

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

class Distance
{
   private:
     int feet;
     float inches;
   public:
     Distance() : feet(0), inches(0.0) {}
     Distance(int ft, float in):feet(ft),inches(in) {}

     void getdist()
     {
       cout << "\nEnter feet: ";  cin >> feet;
       cout << "Enter inches: ";  cin >> inches;
     }
     void showdist() const
     { cout << feet << "\'-" << inches << '\"'; }

     Distance add_dist(const Distance&) const;
};
...
```

```
...
Distance Distance::add_dist(const Distance& d2) const
{
  Distance temp;
  feet = 0;        // ОШИБКА: нельзя изменять поле
  d2.feet = 0;     // ОШИБКА: нельзя изменять d2

  temp.inches = inches + d2.inches;
  if(temp.inches >= 12.0)
  {
     temp.inches -= 12.0;
     temp.feet = 1;
  }
  temp.feet += feet + d2.feet;
  return temp;
}
...
```

```
...
int main()
{
  Distance dist1, dist3;
  Distance dist2(11, 6.25);

  dist1.getdist();
  dist3 = dist1.add_dist(dist2);

  cout << "\ndist1 = ";  dist1.showdist();
  cout << "\ndist2 = ";  dist2.showdist();
  cout << "\ndist3 = ";  dist3.showdist();
  cout << endl;
  return 0;
}
```

Константные объекты

Известно, что ключевое слово const применяется для защиты от изменений переменных стандартных типов. Например

```
const int A = 15;
```

Аналогичным образом от изменений могут быть защищены программные объекты. Если объект некоторого класса объявлен с ключевым словом const, то он становится недоступным для изменения. Такой объект называется **константным**.

```
const имя_класса имя_объекта(параметры);
```

Попытка изменить значение поля объекта внутри константного метода будет приводить к ошибке.

```
void main()
{
   const Distance football(300, 0);

   // football.getdist();    // ОШИБКА!
   cout << "football = ";
   football.showdist();      // OK!
   cout << endl;
}
```