Объектно-ориентированное программирование в C++

Перегрузка операторов. Продолжение

Статические элементы класса

Глобальные переменные и статические переменные

Правило ODR

Переопределение операций

```
Point p3 = p1 + p2;     // а что делать?
bool b = line1 > 0;
```

- C++ поддерживает возможность переопределения большинства операций (арифметических, логических, побитовых, сравнения и др.)
- Не поддерживается перегрузка операций:
  - «точка» (выборка элементов класса или структуры)
  - «тернарный оператор» (a ? b : c)
  - “::” (открытие области видимости)
- Специфические операции: ->, [ ], (), (int)
- Для переопределения операции используется «operator»
- Операция может быть элементом класса, *дружественной* функцией или внешней функцией (для некоторых операций разрешено переопределение только внутри пользовательского типа)

Операции приведения типа

- Конструктор с одним аргументом (*конструктор преобразования*)

```
// Если есть конструктор Line::Line(const char* s)
Line line  = "just string";
```

- Можно определить пользовательскую операцию преобразования

```
 // Line -> const char*
operator const char*() {  return c_str(); }
```

Функторы

- Операция «круглые скобки» позволяет использовать синтаксис вызова функций для объекта
- Такие объекты называют *функторами* и используются для настройки обработки в других функциях

```
F f;
f();
f(1);
f(4, 5);
```

```
class F
{
public:
    int operator()();
    int operator()(int);
    int operator()(int, int);
};
```

Статические элементы

- Статические элементы не связаны с конкретным экземпляром класса
- В смысловом плане статические элементы принадлежат *«классу как таковому»*, а не конкретным объектам.
- Статические поля – глобальные переменные, область видимости ограничена рамками класса
- Время жизни статических полей совпадает с временем жизни программы
- Доступ извне к статическому полю возможен с использованием *имени класса* с учетом модификаторов доступа

```
class A {
public:
    static int x;
    int y;
};
```

```
cout << A::x << endl;
```

Определение статических полей

- Описание статических полей в объявлении класса расценивается исключительно как *объявление*, а не определение
- Место под статическое поле должно быть выделено в одном из модулей через явное определение (*за рамками класса*)

```
// Определяем статическое поле с инициализацией
int A::x = 0;
```

- Определения статических полей должны быть вынесены в файл с реализацией (*.cpp)
- Статическое поле существует в единственном экземпляре на протяжении всей программы, вне зависимости от того, будут ли вводится экземпляры класса (объекты)

Доступ к статическим элементам

- Возможность доступа к статическим элементам определяется заданным правом доступа (помечены ли они как **public** или как **private**)
- Обращение к статическим элементам возможно *через объект* или *через класс*

```
A a;
// объекты могут обращаться к стат.элементам
a.x = 10;
// как глобальная переменная класса A
A::x = 10;
```

Применение статических полей

- Как правило, статические поля вводят для хранения констант и неизменяемых данных «внутри класса»

```
class MathConstants {
    public:
        static const double Pi = 3.14;
};
```

```
// определяем статические поля
R R::Unity = R(1, 1);
R R::Zero = R(0, 1);
int main()
{
    R r1 = R(1,3) + R::Unity;
    R r2 = r1 * R::Unity;
```

```
class R {
    int num, den;
public:
    R(int n, int d): num(n), den(d) { }
    static R Unity;
    static R Zero;
    ..
```

```
int A::counter = 0;
int main()
{
   A::showCounter();
   A a;
   {
      A a;
      a.showCounter();
   }
   a.showCounter();
}
```

```
class A {
private:
    static int counter;
public:
    static void showCounter();
    A() { ++counter; }
    ~A() { --counter; }
};
```

Используем статическое поле counter в качестве счётчика «активных» объектов, для которых уже вызван конструктор, но еще не вызван деструктор

Применение статических переменных

- Изменяемые статические поля могут создавать трудности для прогнозирования работы кода (как и любые глобальные переменные) и приводить к «ошибкам масштабирования» (в многопоточных приложениях)
- В императивных языках (C++, Java, C#, Python, ..) функция зависит не только аргументов, но и от внешних элементов (глобальные переменные, статические поля), которые **явно** не передаются при вызове.

- Приведенные вызовы Calculate потенциально могут возвращать разные результаты, не смотря на одинаковые аргументы

```
A b = a;
int res1 = a.Calculate(3);
int res2 = a.Calculate(3);
int res3 = b.Calculate(3);
```

```
class A
{
private:
    int x, y;
    static int sharedData;
public:
    int Calculate(int z);
};
```

Статические методы

- Функции класса, имеющие полный доступ к элементам класса (как и все другие элементы) и доступные без конкретных экземпляров
- Обращение к статическим методам, как и к статическим полям, возможно *через объект* или с использованием имени класса (*через класс*)
- Внутри статического метода не доступен «текущий объект» через указатель **this**

```
class Matrix {
    public :
        static Matrix getUnityMatrix(int size);
        static Matrix getZeroMatrix(int size);
    ..
};
```

```
// вызываем стат.методы без объекта
Matrix zero = Matrix::getZeroMatrix(4);
Matrix unity = Matrix::getZeroMatrix(4);
```

Глобальные переменные и **static**

- Переменные, которые объявляются за рамками структурного блока - функции, являются глобальными (*внешними*)
- Глобальные переменные доступны во всей программе, т.е. во всех модулях (файлах с кодом *.cpp), в которых введены их имена (*объявлены*)
- Переменные на разных уровнях иерархии могут иметь одинаковые имена (перекрываться)
- Для обращения к глобальной переменной (в случае неоднозначности) можно использовать символ **::**

```
int v = 10;
void f() { v = 10; }
int main()
{
    float v = 0.0f;
    f();
    int res = ::v + 1;
}
```

- Для работы с глобальной переменной в разных

модулях необходимо ввести *объявление*

в каждом модуле с использованием extern:

extern int v ;

Статическая глобальная переменная

- Статическая глобальная переменная – доступна только в пределах модуля (*внутренняя переменная*)
- Модуль компиляции (= *единица трансляции*) – файл с программным кодом (*.cpp) с учётом подключаемых файлов (*.h)

```
// source1.cpp
static int value;
void f1()  {  value = -10;  }
```

```
// source2.cpp
static int value;
void f2() {value = 20;  }
```

- В каждом модуле компиляции могут быть одноименные статические переменные, но совершенно разные переменные.
- Можно вводить статические функции, которые доступны в рамках одного модуля

```
// source1.cpp
static int f() {return 10;}
```

```
// source2.cpp
static int f() {return -20;}
```

Статические «локальные» переменные

- Если переменная объявляется внутри функции с ключевым словом **static**, то в этом случае она является глобальной в плане времени существования, но доступна только внутри функции.
- Время жизни такой переменной – от первого вызова функции, когда происходит инициализация, до конца работы приложения.

```
int f()
{
    static int counter = 0;
    return ++counter;
}
int main()
{
    cout << f() << endl;
    cout << f() << endl;
```

Правило одного определения

Правило **one-definition-rule** (ODR):

- В пределах любой единицы трансляции сущности (функция, тип данных, объект) не могут иметь более одного определения
- В пределах программы глобальные переменные (*внешние*) и неподставляемые функции (*не inline*) не могут иметь больше одного определения
- Классы и подставляемые функции (т.е. сущности, у которых определение полностью или частично совмещается с объявлением) могут определяться в более чем одной единице трансляции, но определения *обязаны* совпадать
- Определения сущностей, не являющихся внешними, в разных единицах трансляции определяют различные сущности, даже если их имена и типы совпадают. Эти определения беспрепятственно могут различаться.

Объявление и определение класса: *.h vs *.cpp

- Объявление класса *частично или полностью* определяет его
- Классы, которые используются в нескольких файлах (модулях) или независимых проектах рекомендуется объявлять в заголовочном файле *.h.
- Нетривиальные методы класса рекомендуется переносить в *.cpp файл.
- Заголовочные файлы представляют собой открытый интерфейс для внешних пользователей.
- Некоторые фрагменты C++ кода должны компилироваться **совместно** с кодом «пользователей», поэтому определяются в h-файлах (шаблонные функции и шаблонные классы)
- Изменение h-файла требует перекомпиляции всех «пользователей»