Базовые возможности C++

ЛЕКЦИЯ №3

Прежде чем начать

- Это механизм, при неумелом использовании которого можно полностью запутать код.
- Непонятный код – причина сложных ошибок!
- Перегруженные операции помогают определить «свойства» созданного вами класса, но не алгоритма работы с классами!

Перегрузка операций

Можно описать функции, для описания следующих операций:

+ - * / % ^ & | ~ !

= < > += -= *= /= %= ^= &=

|= << >> >>= <<= == != <= >= &&

|| ++ -- ->* , -> [] () new delete

Нельзя изменить приоритеты этих операций, равно как и синтаксические правила для выражений. Так, нельзя определить унарную операцию % , также как и бинарную операцию !.

Синтаксис

```
type operator operator-symbol ( parameter-list )
```

Ключевое слово operator позволяет перегружать операции. Например:

- Перегрузка унарных операторов:
  - **ret-type operator** op **(** arg **)**
  - где **ret-type** и op соответствуют описанию для функций-членов операторов, а arg — аргумент типа класса, с которым необходимо выполнить операцию.
- Перегрузка бинарных операторов
  - **ret-type operator** op**(arg1, arg2)**
  - где *ret-type* и op — элементы, описанные для функций операторов членов, а arg1 и arg2 — аргументы. Хотя бы один из аргументов **должен принадлежать типу класса**.

Пример

Example16_Operator

```
1.class Rectangle {
2.public:
3.     Rectangle(int,int);
4.     int          operator[](int i);
5.     operator int();
6.     void         print();
7.     virtual ~Rectangle();
8.private:
9.    int _width,_height;
10.};
```

Префиксные и постфиксные операторы ++ и --

Операторы инкремента и декремента относятся к особой категории, поскольку имеется два варианта каждого из них:

- преинкрементный и постинкрементный операторы;
- предекрементный и постдекрементный операторы.

При написании функций перегруженных операторов полезно реализовать отдельные версии для префиксной и постфиксной форм этих операторов. Для различения двух вариантов используется следующее правило: **префиксная** форма оператора объявляется точно так же, как и любой другой унарный оператор; в **постфиксной** форме принимается дополнительный аргумент типа **int**.

Пример:

```
friend Point& operator++( Point& ) // Prefix increment

friend Point& operator++( Point&, int ) // Postfix increment

friend Point& operator--( Point& ) // Prefix decrement

friend Point& operator--( Point&, int ) // Postfix decrement
```

Пример

Example17_OperatorPlus

```
1.class Rectangle {
2.public:
3.    Rectangle(int,int);
4.    Rectangle& operator++();
5.    Rectangle& operator++(int);
6.    int        operator[](int i);
7.    operator int();
8.    void       print();
9.    virtual ~Rectangle();
10.private:
11.    int _width,_height;
12.};
```

Бинарные операторы

Example18_BinaryOperator

```
1.class Rectangle {
2.public:
3.    Rectangle(int,int);
4.    Rectangle& operator++();
5.    Rectangle& operator++(int);
6.    int        operator[](int i);
7.    operator int();

8.    friend std::ostream& operator <<(std::ostream &os,Rectangle &rec);
9.    friend Rectangle     operator +(Rectangle &left, Rectangle &right);

1.    virtual ~Rectangle();
2.private:
3.    int _width,_height;
4.};
```

Переопределение оператора присваивания

Example19_OperatorAssign

Он должен быть нестатической функцией-членом. Никакой оператор **operator=** не может быть объявлен как функция, не являющаяся членом.

Он не наследуется производными классами.

Компилятор может создавать для типов классов функции **operator=** по умолчанию, если они не существуют. В этом случае оператор равно применяется к каждому члену класса.

Спецификатор delete и default

Example27_Delete

Иногда очень важно сделать так, что бы у объекта был только один экземпляр. Т.е., что бы его нельзя было скопировать.

Сейчас стандартная идиома «запрещения копирования» может быть явно выражена следующим образом:

```
class X {
// ...

X& operator=(const X&) = delete;     // Запрет копирования
X(const X&) = delete; // запрет копирование в момент конструирования
};
```

Такая конструкция запрещает компилятору «создавать» конструкторы и оператор копирования «по умолчанию».

Ключевое слово **=default**, наоборот, указывает что мы хотим что бы компилятор использовал операцию «по-умолчанию». Вообще говоря, она является избыточной.

Делаем функтор

CPP_Examples21_OperatorFunctor

Функторы в C++ являются сокращением от "**функциональные объекты**". Функциональный объект является экземпляром класса C++, в котором определён **operator()**. Если вы определите **operator()** для C++ класса, то вы получите объект, который действует как функция, но может также хранить состояние.

```
class A{
private:
    int _value;
public:   A(int a) : _value(a) {};
   A operator()(int a) {
std::cout << _value+a << std::endl;
return A(_value+a);
}
};
```

Литералы

Литерал — это некоторое выражение, создающее объект. В языке C++ существуют литералы для разных встроенных типов (2.14 Literals):

123 // int

1.2 // double

1.2F // float

'a' // char

1ULL // unsigned long long

0xD0 // unsigned int в шестнадцатеричном формате

"as" // string

Пользовательские литералы

Example22_Literal

Должны начинаться с подчеркивания:

```
OutputType operator "" _suffix(unsigned long long);
```

Конструктор типа должен так же иметь спецификатор **constexpr**

Могут иметь следующие параметры:

```
const char*
unsigned long long int
long double
char
wchar_t
char16_t
char32_t
const char*, std::size_t
const wchar_t*, std::size_t
const char16_t*, std::size_t
const char32_t*, std::size_t
```

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ