

# СЕМИНАР 1.6 «Технические детали классов»

## Вопросы проектирования классов

- Специальные функции - глены класса:
    - К-р по умолчанию
    - Пользовательские к-ры
    - Деструктор
    - Операторы = (8, &&)
    - copy/move к-ры
- } ПОВТОРЕНИЕ (см. прошлый семинар)
- } РАЗБИРАЕМ СЕГОДНЯ ...
- Данные и функции - глены класса - ПОВТОРЕНИЕ\*  
\*(дополнительно остальные элементы повторить)
  - Операторы - разбираем сегодня...

## Операторы (перегрузка операторов)

$x + y * z$  - если для экземпляров класса?

```
class Complex
```

Complex operator+ (const Complex & other) {...} - не лучший  
вариант,  
см. далее

- допускается инфиксная и функциональная формы вызова

$\rightarrow []()$  - можно перегружать  
~~&~~, - не рекомендуется  
 $:: . .^*$  - нельзя перегружать

} нельзя унарный %, или тернарный +  
 необходимо соблюдать тип -  
 УНАРНЫЙ, БИНАРНЫЙ, ...

class X - комплексное число, дробь и т.п. (математический)

Хотим писать так  $X + 1$  и так  $1 + X$  - надо не ф-ю-глен

нужна только эта одна

|   |  |
|---|--|
| <u><math>X operator+(X, X)</math></u>   | много перегрузок - тоже плохо<br>+ вызывает оператор-глен $+=$ |
| <u><math>X operator+(int, X)</math></u> |  |
| <u><math>X operator+(X, int)</math></u> |  |

OK

1. operator+(x) - error!

В классе должен быть не-explicit к-р, чтобы можно было выполнить  $(1 + 1)$  неявное пользовательское преобразование

дополнительно: оператор приведения operator int () const {...} \*

\* может возникнуть неоднозначность выбора (к-р или int())

## Семантика перемещения

lvalue vs rvalue // по-простому: справа / слева от = изначально так  
свойство выражений, а не тип...

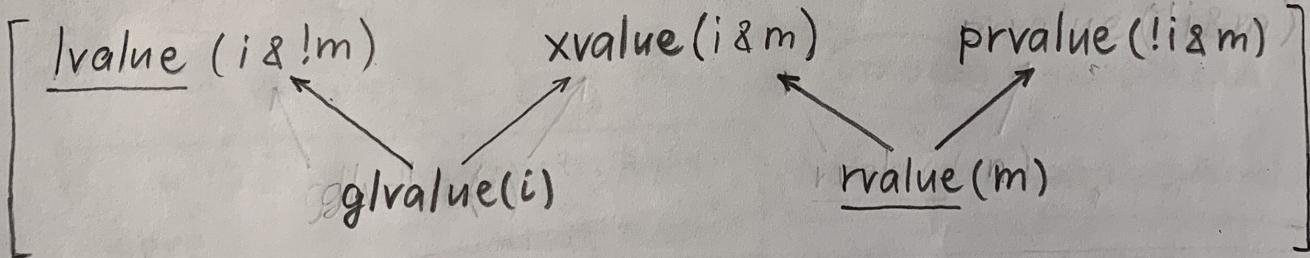
lvalue - объект, который занимает идентифицируемое место в памяти, можно получить его адрес, а  
rvalue - нельзя получить адрес (условно всё остальное)

**Пример** (live) разница lvalue и rvalue

foo(); - даёт временный объект, свойство выражения - rvalue  
const int & x = foo(); - даёт временный объект с продлённым временем жизни, но свойство выражения - lvalue

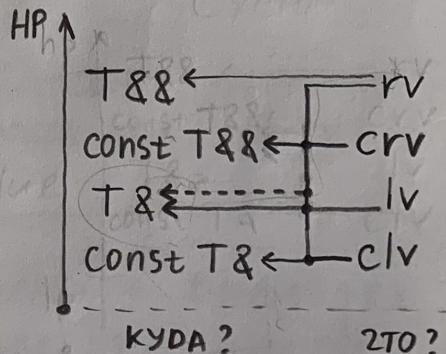
**Классификация выражений C++11:**

- Идентифицируемость (i)
  - Перемещаемость (m)
- } СВ-ВА выражения!



Rvalue - ссылки

- T& - на любое не-const lvalue;
- const T& - на любое выражение;
- T&& - на не-const xvalue, prvalue;
- const T&& - не на lvalue;



**Пример** references.cpp - использование

**Пример** overload.cpp - перегрузка (правила)

Дополнительно: правила свертывания ссылок (auto/template)

Copy или move : ЗАЧЕМ ЭТО НУЖНО на примере массива

$T x = y$  ↗ Копирование - для встроенных типов  
↗ перемещение - начиная с C++ 11  
(для сложных типов) - ИМП

```
{  
    T x;  
    return x;  
}
```

- функция

здесь x больше не нужен,  $\Rightarrow$  ЗАЧЕМ его копировать? можно переместить

void f(T x);

Свойства копирования:

- эквивалентность }  $x$  и  $y$
- независимость }

void swap(X& a, X& b)

```
{  
    const X tmp = a;  
    a = b;  
    b = tmp;  
}
```

копирование - много РАБОТЫ

КАК выполнять перемещение?

ЕСТЬ специальные перемещающие операции, которые выполняют непосредственную работу по перемещению.

ЗДЕСЬ : operator=(...) → КУДА = откуда - подсказка

мы имеем "откуда"-lvalue,  
 $\text{std}::\text{move}$  делает rvalue- ссылку  
выполняет ТОЛЬКО приведение

lvalue или xvalue/prvalue  
copy move

→ показывает, что объект может быть перемещён  
но сб-во xvalue, т.к. осталось ЧИЯ

\*  $\text{std}::\text{move}$  оптимизируется в RVO / copy elision  
В PREDE случаев нет необходимости вызывать  
 $\text{std}::\text{move}$  явно - есть оптимизация RVO / copy elision  
 $\text{operator+}(\dots) \{ \dots \text{return result;} \}$  - можно переместить  
 $X&$   $\text{operator+}(\dots) \{ X& x = * \text{new } X; \dots; \text{return } x; \}$  - old

Кто выполняет работу по copy / move? -

$X(\text{const } X\&)$  // опасно     $X(X\&)$  } копирование  
 $X\& \text{operator} = (\text{const } X\&)$  // +this }  
 $X(X\&&)$  } без const, иначе не работает  
 $X\& \text{operator} = (X\&&)$

move работает как copy, если нельзя переместить

---

Раньше мы не реализовывали самостоятельный все специальные функции-глобы, но использовали их версии.  
Никогда компилятор генерирует не совсем нужное:  
deep copy vs shallow copy на примере массива

---

Правила генерации специальных функций-глобов:

1. Если используются (к-р по умолчанию, если нет др.);
2. Копирующие независимы друг от друга, но зависят от перемещающихся;
3. Перемещающие зависят друг от друга, от копирующих и от деструктора.

default - реализация по умолчанию

delete - не генерировать реализацию

---

Пример String.cpp - учебная реализация строкового класса

Пример      C      String