Argument-dependent name lookup (ADL)

Известен также, как Koenig lookup.

```
namespace X
{
    struct A { ... };

    std::ostream& operator<<(
        std::ostream& out, const A& value) { ... }

    void foo(const A& value) { ... }
}

X::A a;

std::cout << a;
foo(a);
```

Компилятор ищет функцию в текущем пространстве имен и если не находит, то в пространствах имен аргументов. Если находит подходящую функцию в двух местах, то возникает ошибка.

Методы генерируемые компилятором неявно

```
struct A
{
    X x;
    Y y;
    // Конструктор
    A()
        : x(X())
        , y(Y())
    {}
    // Деструктор
    ~A()
    {}
    // Копирующий конструктор
    // A a1;
    // A a2 = a1;
    A(const A& copied)
        : x(copied.x)
        , y(copied.y)
    {}
    // Оператор копирования
    // A a1;
    // A a2;
    // a2 = a1;
    A& operator=(const A& copied)
    {
        x = copied.x;
        y = copied.y;
        return *this;
    }
    // Перемещающий конструктор
    // A a1;
    // A a2 = std::move(a1);
    A(A&& moved)
        : x(std::move(moved.x))
        , y(std::move(moved.y))
    {}
    // Оператор перемещения
    // A a1;
    // A a2;
    // a2 = std::move(a1);
    A& operator=(A&& moved)
    {
```

```
        x = std::move(moved.x);
        y = std::move(moved.y);
        return *this;
    }
};
```

Правило тройки (пятерки)

Если явно объявить один из следующих методов:

- деструктор
- конструктор копирования
- оператор копирования

(после C++11, еще два)

- конструктор перемещения
- оператор перемещения

То компилятор не будет генерировать остальные автоматически, поэтому если они вам нужны, вы должны реализовать их самостоятельно.

rvalue-ссылка и lvalue-ссылка

До стандарта C++11 было два типа значений:

1. lvalue
2. rvalue

> "Объект - это некоторая **именованная область памяти**; lvalue - это выражение, обозначающее объект. Термин "lvalue" произошел от записи присваивания E1 = E2, в которой левый (left - левый(англ.), отсюда буква l, value - значение) операнд E1 должен быть выражением lvalue."

Керниган и Ритчи

```
int a = 1;
int b = 2;
int c = (a + b);
int foo() { return 3; }
int d = foo();

1 = a; // left operand must be l-value
foo() = 2; // left operand must be l-value
(a + b) = 3; // left operand must be l-value
```

1. Ссылается на объект - lvalue
2. Если можно взять адрес - lvalue
3. Все что не lvalue, то rvalue

```
int a = 3;
a; // lvalue
int& b = a;
b; // lvalue, ссылается на a
int* c = &a;
*c; // lvalue, ссылается на a
void foo(int val)
{
    val; // lvalue
}
void foo(int& val)
{
    val; // lvalue, ссылается на val
}
int& bar() { return a; }
bar(); // lvalue, ссылается на a
```

```
3; // rvalue
(a + b); // rvalue
int bar() { return 1; }
bar(); // rvalue
```

lvalue-ссылка

Ссылка на lvalue.

```
int a = 3;
int& b = a;
```

```
int& a = 3; // ошибка
const int& a = 3; // ок
a; // const lvalue
```

Объект жив до тех пор, пока жива ссылающаяся на него константная ссылка.

rvalue-ссылка

```
#include <iostream>

int x = 0;

int val() { return 0; }
int& ref() { return x; }

void test(int&)
{
    std::cout << "lvalue\n";
}

void test(int&&)
{
    std::cout << "rvalue\n";
}

int main()
{
    test(0); // rvalue
    test(x); // lvalue
    test(val()); // rvalue
    test(ref()); // lvalue
    test(std::move(x)); // rvalue
    return 0;
}
```

> `std::move` приводит lvalue к rvalue

Копирование

Семантика: в результате копирования должна появится точная копия объекта.

```
int x = 3;
int y = x;
// x == y

String a;
String b = a;
String c;
c = a;
// a == b == c
```

```
class String
{
    size_t size_;
    char* data_;
```

```
public:
    ~String()
    {
        delete[] data_;
    }

    // String b1;
    // String b2 = b1;
    String(const String& copied)
        : data_(new char[copied.size_])
        , size_(copied.size_)
    {
        std::copy(copied.data_, copied.data_ + size_, data_);
    }

    // String b1;
    // String b2;
    // b2 = b1;
    String& operator=(const String& copied)
    {
        // Плохо
        delete[] data_;
        data_ = new char[copied.size_];
        size_ = copied.size_;
        std::copy(copied.data_, copied.data_ + size_, data_);
        return *this;
    }
};
```

```
String b1;
b1 = b1;

std::vector<String> words;
...
words[to] = words[from];
```

Проверяйте на присваивание самому себе.

```
String& operator=(const String& copied)
{
    if (this == &copied)
        return *this;
    // Плохо
    delete[] data_;
    data_ = new char[copied.size_];
    size_ = copied.size_;
    std::copy(copied.data_, copied.data_ + size_, data_);
    return *this;
}
```

Финальный вариант:

```
String& operator=(const String& copied)
{
    if (this == &copied)
        return *this;
    char* ptr = new char[copied.size_];
    delete[] data_;
    data_ = ptr;
    size_ = copied.size_;
    std::copy(copied.data_, copied.data_ + size_, data_);
    return *this;
}
```

> Подумайте, а стоит ли писать конструктор/оператор копирования самостоятельно?

Копирование и наследование

```
struct A
{
    A() {}
```

```
    A(const A& a) {}
    virtual A& operator=(const A& copied)
        { return *this; }
};

class B
    : public A
{
public:
    B() {}

    B(B& b)
        : A(b)
    {
    }

    A& operator=(const A& copied) override
    {
        A::operator=(copied);
        return *this;
    }
};
```

Срезка

```
void foo(A a)
{
    // Срезанный до A объект
}

B a;
foo(a);
```

Нежелательное копирование

```
void send(std::vector<char> data)
{
    ...
}
```

```
void print(const std::vector<char>& data)
```

Используйте передачу по ссылке!

Явный запрет копирования

До C++11:

```
class Noncopyable
{
    Noncopyable(const Noncopyable&);
    Noncopyable& operator=(const Noncopyable&);
};

class Buffer
    : private Noncopyable
{
};
```

`boost::noncopyable` устроен именно так.

C++11:

```
class Buffer
{
    Buffer(const Buffer&) = delete;
    Buffer& operator=(const Buffer&) = delete;
};
```

Явное указание компилятору сгенерировать конструктор и оператор копирования

```
class Buffer
{
public:
    Buffer(const Buffer&) = default;
    Buffer& operator=(const Buffer&) = default;
};
```

Перемещение

Семантика: в результате перемещения в объекте, куда происходит перемещение, должна появиться точная копия перемещаемого объекта, оригинальный объект после этого остается в неопределенном, но корректном состоянии.

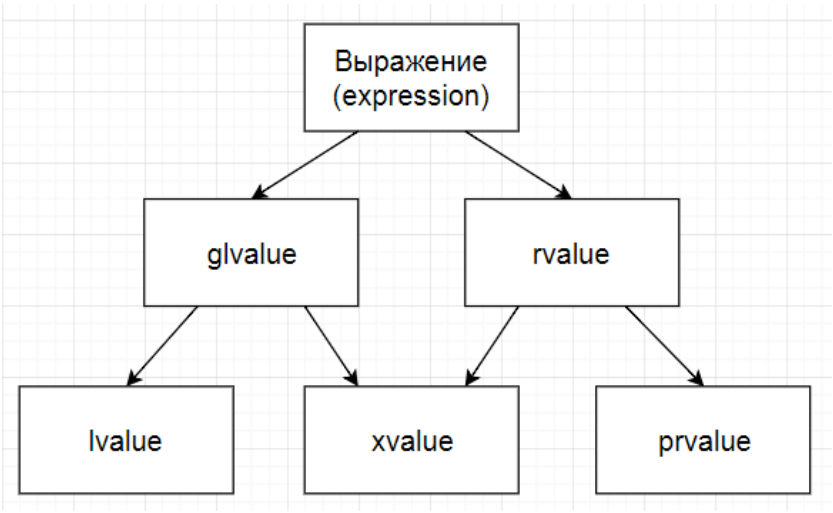
Передача владения

```
class unique_ptr
{
    T* data_;
};
```

Производительность

```
class Buffer
{
    char* data_;
    size_t size_;
};
```

lvalue и rvalue начиная с C++11



glvalue ("generalized" lvalue)

Выражение, чьё вычисление определяет сущность объекта.

prvalue ("pure" rvalue)

Выражение, чьё вычисление инициализирует объект или вычисляет значение операнда оператора, с соответствии с контекстом использования.

xvalue ("eXpiring" value)

Это glvalue, которое обозначает объект, чьи ресурсы могут быть повторно использованы (обычно потому, что они находятся около конца своего времени жизни).

lvalue

Это glvalue, которое не является xvalue.

rvalue

Это prvalue или xvalue.

Пример

lvalue

Выражение является lvalue, если ссылается на объект уже имеющий имя доступное вне выражения.

```
int a = 3;
a; // lvalue
int& b = a;
b; // lvalue
int* c = &a;
*c; // lvalue

int& foo() { return a; }
foo(); // lvalue
```

xvalue

1. Результат вызова функции возвращающей rvalue-ссылку

```
int&& foo() { return 3; }
foo(); // xvalue
```

2. Явное приведение к rvalue

```
static_cast<int&&>(5); // xvalue
std::move(5); // эквивалентно static_cast<int&&>
```

3. Результат доступа к нестатическому члену, объекта xvalue значения

```
struct A
{
    int i;
};

A&& foo() { return A(); }

foo().i; // xvalue
```

prvalue

Не принадлежит ни к lvalue, ни к xvalue.

```
int foo() { return 3; }
foo(); // prvalue
```

rvalue

Все что принадлежит к xvalue или prvalue.

glvalue

Все что принадлежит к xvalue или lrvalue.

Практическое правило (Скотт Мейерс)

1. Можно взять адрес - lvalue
2. Ссылается на lvalue (T&, const T&) - lvalue
3. Иначе rvalue

> Как правило rvalue соответствует временным объектам, например, возвращаемым из функций или создаваемых в результате неявных приведений типов. Также это большинство литералов.

Классификация

Есть имя	Может быть перемещено	Тип
да	нет	glvalue, lvalue
да	да	rvalue, xvalue, glvalue
нет	да	rvalue, prvalue

Еще примеры

```
void foo(int) {}
void foo(int&) {}
void foo(int&&) {}
```

```
void foo(int) {} // <-- этот?
void foo(int&) {} //  // <-- или этот?
void foo(int&&) {}

int x = 1;
foo(x); // lvalue
```

```
int x = 1;
int& y = x;
foo(y); // lvalue

void foo(int) {} // <-- этот?
void foo(int&) {} // <-- или этот?
void foo(int&&) {}
```

```
foo(1); // rvalue

void foo(int) {} // <-- этот?
void foo(int&) {}
void foo(int&&) {} // <-- или этот?
```

```
int bar() { return 1; }
foo(bar()); // rvalue

void foo(int) {} // <-- этот?
void foo(int&) {}
void foo(int&&) {} // <-- или этот?
```

```
foo(1 + 2); // rvalue

void foo(int) {} // <-- этот?
void foo(int&) {}
void foo(int&&) {} // <-- или этот?
```

Конструктор/оператор перемещения

```
class Buffer
{
    size_t size_;
    char* data_;
public:
    ~Buffer()
    {
        delete data_;
    }

    // Buffer b1;
    // Buffer b2 = std::move(b1);
    Buffer(Buffer&& moved)
        : data_(moved.data_)
```

```
        , size_(moved.size_)
    {
        moved.data_ = nullptr;
        moved.size_ = 0;
    }

    // Buffer b1;
    // Buffer b2;
    // b2 = std::move(b1);
    Buffer& operator=(Buffer&& moved)
    {
        if (this == &moved)
            return *this;
        delete[] data_;
        data_ = moved.data_;
        size_ = moved.size_;
        moved.data_ = nullptr;
        moved.size_ = 0;
        return *this;
    }
};
```

Явное указание компилятору сгенерировать конструктор и оператор перемещения

```
class Buffer
{
public:
    Buffer(Buffer&&) = default;
    Buffer& operator=(Buffer&&) = default;
};
```

Явное указание компилятору запретить перемещение

```
class Buffer
{
public:
    Buffer(Buffer&&) = delete;
    Buffer& operator=(Buffer&&) = delete;
};
```

Perfect forwarding

Задача: передать аргумент не создавая временных копий и не изменяя типа передаваемого аргумента.

```
void bar(T&) {}
void bar(T&&) {}
```

```
void foo(T x)
{
    // копия
    bar(x);
}
```

```
void foo(T& x)
{
    // Может приводить к ошибкам
    // компиляции, если x – rvalue:
    // foo(T());
    bar(x);
}

void foo(T&& x)
{
    // ок, но пришлось написать перегрузку
    bar(std::move(x);
}
```

Решение:

```
void foo(T&& x)
{
    bar(std::forward<T>(x));
}
```

Return value optimization (RVO)

Позволяет сконструировать возвращаемый объект в точке вызова.

```
Server makeServer(uint16_t port)
{
    Server server(port);
    server.setup(...);
    return server;
}

Server s = makeServer(8080);
```

Не мешайте компилятору:

```
Server&& makeServer(uint16_t port)
{
    Server server(port);
    server.setup(...);
    return std::move(server); // так не надо
}
```

Copy elision

Оптимизация компилятора разрешающая избегать лишнего вызова копирующего конструктора.

```
struct A
{
    explicit A(int) {}
    A(const A&) {}
};

A y = A(5); // Копирующий конструктор вызван не будет
```

> В копирующих конструкторах должна быть логика отвечающая только за копирование.

Шаблоны

Шаблоны классов

```
class Matrix
{
    double* data_;
};
```

```
class MatrixDouble
{
    double* data_;
};

class MatrixInt
{
    int* data_;
};
```

```
template <class T>
class Matrix
{
    T* data_;
};
```

```
Matrix<double> m;
Matrix<int> m;
```

Шаблоны функций

```
template <class T>
void printLine(const T& value)
{
    std::cout << value << '\n';
}
```

```
printLine<int>(5);
```

Компилятор может самостоятельно вывести тип шаблона в зависимости от аргументов вызова.

```
printLine(5);
```

class или typename

```
template <class T>
void printLine(const T& value)
{
}
```

```
template <typename T>
void printLine(const T& value)
{
}
```

Никакой разницы.

Практическая часть

Написать класс для работы с большими целыми числами. Размер числа ограничен только размером памяти. Нужно поддержать семантику работы с обычным int:

```
BigInt a = 1;
BigInt b = a;
BigInt c = a + b + 2;
```

Реализовать оператор вывода в поток, сложение, вычитание, унарный минус, все операции сравнения.

std::vector и другие контейнеры использовать нельзя - управляйте памятью сами.

EOF