Семантика перемещения II

Проблема

Допустим, хотим написать обертку, которая принимает функцию, ее аргумент и измеряет время работы функции на этих данных.

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg arg) {
    const auto start = std::clock();
    func(arg);
    return std::clock() - start;
}

void Function(int) { /* ... */ }
```

```
Function(5);  // Ok
RunningTime(Function, 5);  // Ok
```

А где проблема то?

Проблема

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg arg) {
    const auto start = std::clock();
    func(arg);
    return std::clock() - start;
}

void Function(int& value) { ++value; }
```

```
int x = 0;
Function(x);  // x == 1
RunningTime(Function, x);  // x == 1 ?!
```

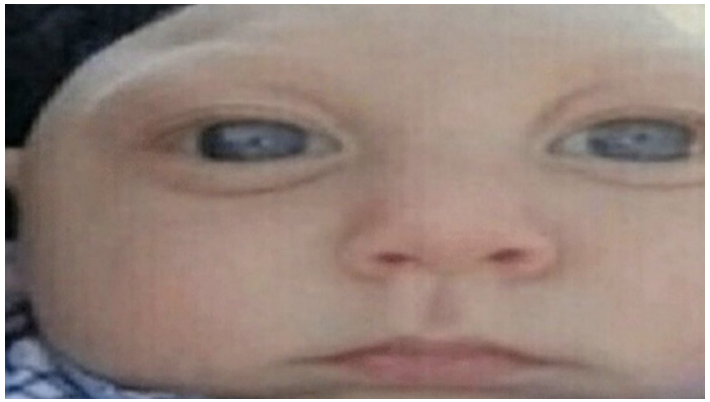
Если `Func` принимает аргумент по ссылке, то будем работать с локальной копией, а не с реально переданным объектом!

Решение (?)

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg& arg) {  // <-- давайте здесь добавим &
    const auto start = std::clock();
    func(arg);
    return std::clock() - start;
}

void Function(int& value) { ++value; }
```

```
int x = 0;
Function(x);  // x == 1
RunningTime(Function, x);  // x == 2
```

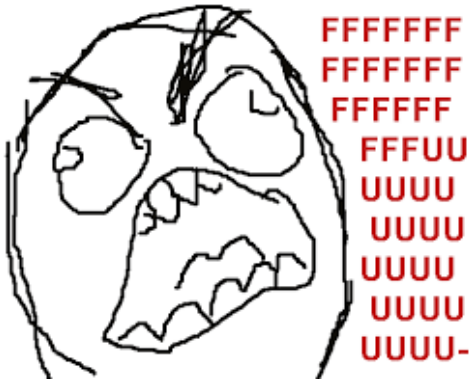


Решение (нет)

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg& arg) {  // <-- давайте здесь добавим &
    const auto start = std::clock();
    func(arg);
    return std::clock() - start;
}

void Function(int) { /* ... */ }  // <-- теперь хочу так
```

```
Function(5);  // Так вызывать можно
RunningTime(Function, 5);  // CE: А так теперь нельзя
```



В чем дело?

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg arg);
```

- `Arg` не умеет адаптироваться под фактический аргумент функции.

Хочется: если передали lvalue, то `arg` является lvalue-ссылкой на это значение; если передали rvalue, то `arg` - rvalue-ссылка на переданное значение.

- Конечно, можно написать перегрузки с lvalue-ссылкой и rvalue-ссылкой.

```
template <class Func, class Arg> clock_t RunningTime(Func, Arg&);
template <class Func, class Arg> clock_t RunningTime(Func, Arg&&);
```

Но это же дублирование кода...

Forwarding reference (универсальная ссылка)

Сворачивание ссылок

В процессе подстановки шаблонных параметров может возникнуть "ссылка на ссылку". Несмотря на то, что они запрещены стандартом, в этом случае действуют специальные правила.

```
template <class T>
void f(T& x, T&& y);

f<int&>(...);   // T = int&,  type(x) == int&, type(y) == int&
f<int&&>(...);  // T = int&&, type(x) == int&, type(y) == int&&
```

- Правила сворачивания ссылок при подстановке шаблонных параметров:

```
type&  &  == type&
type&  && == type&
type&& &  == type&
type&& && == type&&
```

Forwarding reference (универсальная ссылка)

Универсальная ссылка - это ссылка одного из следующих двух видов:

```
template <class T>
void Function(T&& x);  // <-- T&& - универсальная ссылка
```

```
auto&& x = /* ... */;  // <-- auto&& - универсальная ссылка
```

То есть универсальная ссылка имеет вид rvalue-ссылки (но ей не является!) и применяется только к шаблонным параметрам функции или к объявлениям `auto`.

Forwarding reference (универсальная ссылка)

То есть универсальная ссылка имеет вид обычной (без модификаторов) rvalue-ссылки (но ей не является!) и применяется только к шаблонным параметрам функции или к объявлениям `auto`.

```
template <class T>
void Function(const T&& x);  // <-- не универсальная ссылка! (const)
```

```
const auto&& x = /* ... */;  // <-- не универсальная ссылка! (const)
```

```
template <class T>
class Stack {
    // ...
    void Push(T&& value);  // <-- не универсальная ссылка!
                           // (не является шаблонным параметром функции)

    template <class U>
    void Push(U&& value);  // <-- универсальная ссылка
};
```

Правила вывода для универсальных ссылок

```
template <class T>
void Function(T&& arg);
```

- `cv` квалификаторы не отбрасываются.
- При передаче *lvalue* в качестве аргумента тип `T` выводится как lvalue-ссылка.
- При передаче *rvalue* в качестве аргумента тип `T` выводится как нессылочный.

```
const int cx = 0;

Function(cx);  // [T = const int&], type(arg) == const int&
Function(0);   // [T = int], type(arg) == int&&
```

Универсальные ссылки: примеры

```
template <class T>
void Function(T&& arg);
```

```
int x = 0;
const int cx = x;
int&& rx = 0;
const int&& crx = 0;

Function(x);  // [T = ???], type(arg) == ???
Function(cx);
Function(rx);
Function(crx);
Function(0);
Function(std::move(x));
Function(std::move(cx));
Function<int&>(x);
Function<int&&>(0);
Function<int>(0);
```

Универсальные ссылки: примеры

```
template <class T>
void Function(T&& arg);
```

```
int x = 0;
const int cx = x;
int&& rx = 0;
const int&& crx = 0;

Function(x);                // [T = int&],        type(arg) == int&
Function(cx);               // [T = const int&],  type(arg) == const int&
Function(rx);               // [T = int&],        type(arg) == int&
Function(crx);              // [T = const int&],  type(arg) == const int&
Function(0);                // [T = int],         type(arg) == int&&
Function(std::move(x));     // [T = int],         type(arg) == int&&
Function(std::move(cx));    // [T = const int],   type(arg) == const int&&
Function<int&>(x);          // [T = int&],        type(arg) == int&
Function<int&&>(0);         // [T = int&&],       type(arg) == int&&
Function<int>(0);           // [T = int],         type(arg) == int&&
```

Решение проблемы

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg&& arg);
```

Теперь все (почти) работает

```
void Function(int);

Function(0);
RunningTime(Function, 0);  // ok: [Arg = int, type(arg) = int&&]
```

```
void Function(int& value) { ++value; }

int x = 0;
Function(x);  // x == 1;
RunningTime(Function, x);  // ok: [Arg = int&, type(arg) = int&], x == 2
```

Новая проблема

Вернемся к реализации `RunningTime`

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg&& arg) {
    const auto start = std::clock();
    func(arg);
    return std::clock() - start;
}
```

```
void FunctionL(int&);
```

```
FunctionL(x) /* Ok */; RunningTime(FunctionL, x) /* Ok */;
```

```
FunctionL(0) /* CE */; RunningTime(FunctionL, 0) /* Ok */;
```

Новая проблема

Вернемся к реализации `RunningTime`

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg&& arg) {
    const auto start = std::clock();
    func(arg);
    return std::clock() - start;
}
```

```
void FunctionR(int&&);
```

```
FunctionR(x) /* CE */; RunningTime(FunctionR, x) /* CE */;
```

```
FunctionR(0) /* Ok */; RunningTime(FunctionR, 0) /* CE */;
```

Новая проблема

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg&& arg) {
    const auto start = std::clock();
    func(arg);
    return std::clock() - start;
}
```

```
void FunctionL(int&);
void FunctionR(int&&);
```

```
FunctionL(0) /* CE */; RunningTime(FunctionL, 0) /* Ok */;
FunctionR(0) /* Ok */; RunningTime(FunctionR, 0) /* CE */;
```

1. Наблюдается ассиметричность: хотелось бы, чтобы там, где было CE, оставалось CE, и аналогично для Ok
2. Невозможно использовать `FunctionR`!

Новая проблема

Дело в том, что внутри `RunningTime` `arg` всегда является `lvalue`!

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg&& arg) {
    const auto start = std::clock();
    func(arg);  // <-- arg - это lvalue!
    return std::clock() - start;
}
```

```
void FunctionL(int&);
void FunctionR(int&&);
```

```
FunctionL(0) /* CE */; RunningTime(FunctionL, 0) /* Ok */;
FunctionR(0) /* Ok */; RunningTime(FunctionR, 0) /* CE */;
```

Идея решения

Хочется:

- Если в `arg` передали *lvalue*, то вызывается `func(arg)`
- Если в `arg` передали *rvalue*, то вызывается `func(std::move(arg))`

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg&& arg) {
    const auto start = std::clock();
    if ( <в arg передали lvalue> ) {
        func(arg);
    } else {
        func(std::move(arg));
    }
    return std::clock() - start;
}
```

А можем ли мы как-то определить, что передали в `arg`?

Идея решения

А можем ли мы как-то определить, что передали в `arg`*?*

Да! Универсальные ссылки по-разному выводят `Arg` при передаче *lvalue* и *rvalue*:

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg&& arg) {
    const auto start = std::clock();
    if ( <Arg - lvalue-ссылка> ) {  // Arg == type&
        func(arg);
    } else {
        func(std::move(arg));
    }
    return std::clock() - start;
}
```

`std::forward`

Решение: `std::forward`

В C++ есть специальная функция, которая осуществляет "условный `std::move`" (делает `std::move`, если аргумент был принят как *rvalue*, и не делает, если был принят как *lvalue*).

```
std::forward<Arg>(arg) <=> if (Arg == type&) arg; else std::move(arg)
```

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg&& arg) {
    const auto start = std::clock();
    func(std::forward<Arg>(arg));  // <-- обязательно явно передавать Arg!
    return std::clock() - start;
}
```

Решение: `std::forward`

Теперь все прекрасно работает

```
template <class Func, class Arg>
clock_t RunningTime(Func func, Arg&& arg) {
    const auto start = std::clock();
    func(std::forward<Arg>(arg));  // <-- обязательно явно передавать Arg!
    return std::clock() - start;
}
```

```
void FunctionL(int&);
void FunctionR(int&&);
```

```
FunctionL(x) /* Ok */; RunningTime(FunctionL, x) /* Ok */;
FunctionL(0) /* CE */; RunningTime(FunctionL, 0) /* CE */;
```

```
FunctionR(x) /* CE */; RunningTime(FunctionR, x) /* CE */;
FunctionR(0) /* Ok */; RunningTime(FunctionR, 0) /* Ok */;
```

Решение: `std::forward`

Можем обобщить решение на произвольное число аргументов

```
template <class Func, class... Args>
clock_t RunningTime(Func func, Args&&... args) {
    const auto start = std::clock();
    func(std::forward<Args>(args)...);
    return std::clock() - start;
}
```

```
void FunctionLR(int&, int&&);
```

```
FunctionLR(x, 0) /* Ok */; RunningTime(FunctionLR, x, 0) /* Ok */;

FunctionLR(0, x) /* CE */; RunningTime(FunctionLR, 0, x) /* CE */;

FunctionLR(x, x) /* CE */; RunningTime(FunctionLR, x, x) /* CE */;

FunctionLR(0, 0) /* CE */; RunningTime(FunctionLR, 0, 0) /* CE */;
```

Примеры применения

`std::make_unique`, `std::make_shared`

Для создания умных указателей на объекты в динамической области памяти вместо `new` можно воспользоваться функциями `std::make_unique` и `std::make_shared`:

```
std::unique_ptr<int> uptr(new int(11));
std::shared_ptr<int> sptr(new int(0));
```

```
auto uptr = std::make_unique<int>(11);
auto sptr = std::make_shared<int>(0);
```

Методы `emplace`, `emplace_back`, `emplace_front`

Стандартные контейнеры C++ (например, `std::vector`, `std::deque`, `std::map` и т.д.) помимо обычных методов вставки (`push_back`, `push_front`, `insert`, ...) имеют методы вида `emplace`.

```
template <class... Args>
void std::vector<T>::emplace_back(Args&&... args);
```

В отличие от обычной вставки `emplace` принимает не объект, а *параметры конструктора*, с которыми нужно создать объект в нужном месте контейнера. Как правило, это более эффективно, чем передавать временный объект.

```
std::vector<HeavyObject> v;

v.push_back(HeavyObject(3, "aba", false));  // создание + copy/move

v.emplace_back(3, "aba", false); // создание сразу в нужном месте
```

Упрощенная реализация `emplace`

```
template <class T>
template <class... Args>
void Stack<T>::Emplace(Args&&... args) {
    buffer_[size++] = T(std::forward<Args>(args)...);
}
```

В данной реализации объект, к сожалению, не "создается на месте", как того хочется, присваивается в ячейку слева.

Для непосредственного создания объекта в определенном месте в памяти стоит воспользоваться *размещающей формой* `new` (https://ru.wikipedia.org/wiki/New_(C%2B%2B)#Placement_new)

Может быть, поговорим об этом в одной из следующих лекций

Шаблонный конструктор

Следует с осторожностью использовать универсальные ссылки в конструкторах класса, так как это может привести к неожиданному поведению

```
class A {
    A() = default;
    A(const A&) { std::cout << "Copy"; };

    template <class T>
    A(T&&) { std::cout << "Something went wrong..."; }
};
```

```
A a;
A b(a);
```

```
Something went wrong...
```

Шаблонный конструктор

Следует с осторожностью использовать универсальные ссылки в конструкторах класса, так как это может привести к неожиданному поведению

```
class A {
    A() = default;
    A(const A&) { std::cout << "Copy"; };

    template <class T>
    A(T&&) { std::cout << "Something went wrong..."; }
};
```

```
A a;
A b(a);
```

```
Something went wrong...
```

При подстановке `[T = A&]`, получаем более точное соответствие чем `const A&`

Шаблонный конструктор

Существует несколько решений этой проблемы.

1. Написать перегрузку конструктора копирования под всевозможные ситуации:

```
class A {
    A() = default;
    A(const A&) { std::cout << "Copy"; };
    A(A& other) : A(std::as_const(other)) {}
    A(const A&& other) : A(other) {}

    template <class T>
    A(T&&) { std::cout << "Something went wrong..."; }
};
```

2. Поверить, что следующий код работает корректно

```
template <class T,
          class = std::enable_if_t<!std::is_same_v<std::decay_t<T>, A>>>
A(T&&) { std::cout << "Something went wrong..."; }
```

Copy Elision

Внимание на экран (песочница):

https://godbolt.org/z/qebhq3

https://godbolt.org/z/PhandE

Copy Elision

Copy Elision - оптимизация, позволяющая избежать копирования/перемещения объектов, при передаче временных (иногда локальных) объектов по значению.

При выполнении данной оптимизации конструкторы копирования и перемещения игнорируются, даже, если они имеют "побочные эффекты" или объявлены как удаленные или приватные.

```
class A {
    A(const A&) = delete;
    A(A&&) = delete;

public:
    A() = default;
};

A f() { return {}; }
A a = A();  // Ok: вызывается только к-р по умолчанию
A b = f();  // Ok: вызывается только к-р по умолчанию
```

Copy elision

Основные контексты проявления *copy elision*:

- Инициализация объекта с помощью *prvalue* выражения того же типа (с точностью до `const` и `volatile`)

```
A a = A(); A b(A());
```

- *Return Value Optimization (RVO)*: Возврат из функции *prvalue* выражения того же типа, что и тип возвращаемого значения (с точностью до `cv`)

```
A f() { return {}; }
```

- *Named Return Value Optimization (NRVO)*: То же, что и RVO, но в `return` выражении стоит локальная переменная не являющаяся аргументом функции.

```
A f() { A a; return a;}
```

Не обязательный copy elision (до C++17)

- До C++11 (C++98, C++03) *copy elision* не был частью стандарта C++. Компиляторы реализовывали эту оптимизацию в обход правил языка.
- В C++11 *copy elision* был включен в стандарт в качестве оптимизации, которую *могут* реализовывать компиляторы, но *не обязаны* (non-mandatory copy elision).
- Чтобы отключить эти оптимизации необходимо было передать дополнительный флаг компиляции `-fno-elide-constructors`

Обязательный copy elision (C++17)

- В C++17 *copy elision* стал частью языка и теперь он *гарантируется* в некоторых ситуациях (mandatory copy elision)
- *Copy elision* происходит при инициализации объекта с помощью *prvalue* (1 пункт) и RVO (2 пункт). На это даже не может повлиять флаг `-fno-elide-constructors`
- В остальных контекстах (в том числе NRVO) стандарт оставляет все на совести компилятора (ничего не гарантируется).

Резюме

- Универсальная ссылка позволяет одной шаблонной функции принимать аргумент по lvalue- или rvalue- ссылке в зависимости от переданного значения
- Для "прямой" передачи аргументов функции далее (без изменения категории значения) можно воспользоваться "условным `std::move`" - `std::forward`
- Семантика прямой передачи в ряде случаев позволяет писать более эффективный и безопасный код
- Copy elision - оптимизация компилятора, которая позволяет избежать лишних копирований/перемещений при работе с временными объектами
- Важно помнить, что copy elision происходит даже, если вы явно запретили использование конструктора копирования и конструктора перемещения