Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 5
06.10.2017

lvalue, xvalue, prvalue

```
int    f();
const int&  g();
int&& h();
// ...
int x = 5;
int *p = &x;

f();
g();
h();
*p;
14;
static_cast<int&&>(x);
std::move(5);
```

lvalue, xvalue, prvalue

```
int    f();
const int&  g();
int&& h();
// ...
int x = 5;
int *p = &x;

f();  // prvalue
g();  // lvalue
h();  // xvalue
*p;   // lvalue
14;   // prvalue
static_cast<int&&>(x); // xvalue
std::move(5);  // xvalue
```

std::move

```
class TSuperClass {
  public:
    // ...
    TSuperClass(const TSuperClass&);
    TSuperClass(TSuperClass&&);
    // ...
};
```

std::move

```
class TSuperClass {
  public:
    // ...
    TSuperClass(const TSuperClass&);
    TSuperClass(TSuperClass&&);
    // ...
};

class TMyType {
  public:
    TMyType(const TSuperClass val) : field(std::move(val));
  private:
    TSuperClass field;
}
```

В чем тут проблема?

std::forward

- std::move выполняет безусловное приведение своего аргумента к rvalue
- std::forward выполняет приведение только при соблюдении определенных условий.

std::forward

```
class A{};
void Do(const A& x) {
    std::cout << "call Do lvalue" << std::endl;
}
void Do(A&& x) {
    std::cout << "call Do rvalue" << std::endl;
}
template <typename T>
void call(T&& obj) {
    Do(obj);
}
int main() {
    A x;
    call(x);
    call(std::move(x));
}
```

std::forward

```
class A{};
void Do(const A& x) {
    std::cout << "call Do lvalue" << std::endl;
}
void Do(A&& x) {
    std::cout << "call Do rvalue" << std::endl;
}
template <typename T>
void call(T&& obj) {
    Do(obj);
}
int main() {
    A x;
    call(x);
    call(std::move(x));
}
```

```
call Do lvalue
call Do lvalue
```

std::forward

```
class A{};
void Do(const A& x) {
    std::cout << "call Do lvalue" << std::endl;
}
void Do(A&& x) {
    std::cout << "call Do rvalue" << std::endl;
}
template <typename T>
void call(T&& obj) {
    Do(std::forward<T>(obj));
}
int main() {
    A x;
    call(x);
    call(std::move(x));
}
```

```
call Do lvalue
call Do rvalue
```

Как работает std::forward

Шаблон с универсальной ссылкой

```
template <typename T>
void call(T&& obj);
```

- Если в качестве аргумента передается lvalue, то T выводится как lvalue-ссылка.
- Если в качестве аргумента передается rvalue, то T не является ссылкой.

Как работает std::forward

Шаблон с универсальной ссылкой

```
template <typename T>
void call(T&& obj);
```

- Если в качестве аргумента передается lvalue, то T выводится как lvalue-ссылка.
- Если в качестве аргумента передается rvalue, то T не является ссылкой.

```
int x;
call(x);
call(std::move(x));
```

Как работает std::forward

Шаблон с универсальной ссылкой

```
template <typename T>
void call(T&& obj);
```

- Если в качестве аргумента передается lvalue, то T выводится как lvalue-ссылка.
- Если в качестве аргумента передается rvalue, то T не является ссылкой.

```
int x;
call(x);  // T - int&
call(std::move(x));  // T - int
```

Свертывание ссылок

Стандарт определяет следующие правила свертки ссылок, применимые для определений typedef и decltype, а также параметров шаблонов:

- `A& & становится A&`
- `A& && становится A&`
- `A&& & становится A&`
- `A&& && становится A&&`

Свертывание ссылок

```
template <typename T>
struct A {
    typedef T&& TRef;
};
// ...
A<int&> x;
```

typedef int& && TRef; $\rightarrow$ typedef int& TRef;

Свертывание ссылок

Свертывание ссылок применяется при:

- инстанциировании шаблонов,
- генерации типа auto,
- typedef и using,
- decltype.

Универсальные ссылки и rvalue-ссылки

```
class A {
  public:
    template <typename T>
    void set(T&& x) {
        text = std::move(x);
    }
  private:
    std::string text;
};

int main() {
    A obj;
    std::string text = "123";
    obj.set(text); // text теперь пусто
}
```

Универсальные ссылки и rvalue-ссылки

Тогда так:

```
class A {
  public:
    void set(const std::string& x) {
        text = x;
    }
    void set(std::string&& x) {
        text = std::move(x);
    }

  private:
    std::string text;
};
```

Универсальные ссылки и rvalue-ссылки

Воспользуемся std::forward:

```
class A {
  public:
    template <typename T>
    void set(T&& x) {
        text = std::forward<T>(x);
    }
  private:
    std::string text;
};
```

Оптимизация

```
template <typename T>
MyType f(T&& obj) {  // универсальная ссылка
    obj.modify();
    return std::forward<T>(obj);
}
```

Без std::forward — всегда копия.

Оптимизация

```
template <typename T>
MyType f(T&& obj) {  // универсальная ссылка
    obj.modify();
    return std::forward<T>(obj);
}
```

Без std::forward — всегда копия.

```
MyType f() {
    MyType obj;
    return std::move(obj);
}
```

Но это лишнее! Почему?

Return value optimization

Устранение временного объекта для создание возвращаемого функцией значения.
Вместо

```
MyType f() {
    MyType obj;
    return std::move(obj);
}
```

правильнее

```
MyType f() {
    MyType obj;
    return obj;
}
```

Return value optimization

Устранение временного объекта для создание возвращаемого функцией значения.
Вместо

```
MyType f() {
    MyType obj;
    return std::move(obj);
}
```

правильнее

```
MyType f() {
    MyType obj;
    return obj;
}
```

Когда не работает RVO?

Перегрузка

```
void Do(std::set<std::string>& strings,
        const std::string& str) {
    std::cout << str << std::endl;
    strings.emplace(str);
}

int main() {
    std::set<std::string> strings;
    std::string s1("text");
    Do(strings, s1);
    Do(strings, "some");
    Do(strings, std::string("string"));
    return 0;
}
```

Перегрузка

Перепишем на универсальную ссылку:

```
template <typename T>
void Do(std::set<std::string>& strings, T&& str) {
    std::cout << str << std::endl;
    strings.emplace(std::forward<T>(str));
}

int main() {
    std::set<std::string> strings;
    std::string s1("text");
    Do(strings, s1);
    Do(strings, "some");
    Do(strings, std::string("string"));
    return 0;
}
```

Перегрузка

```
void Do(std::set<std::string>& strings, int x) {
    strings.emplace(std::to_string(x));
}
```

Перегрузка

```
void Do(std::set<std::string>& strings, int x) {
    strings.emplace(std::to_string(x));
}
```

И внезапно ломается код:

```
short x = 2;
Do(strings, x);
```

Перегрузка

```
void Do(std::set<std::string>& strings, int x) {
    strings.emplace(std::to_string(x));
}
```

И внезапно ломается код:

```
short x = 2;
Do(strings, x);
```

Функции с универсальными ссылками могут выполнить инстанциирование с точным соответствием практически любому типу.

Перегрузка: еще пример

```
class A {
  private:
    std::string text;
  public:
    template <typename T>
    explicit A(T&& str) : text(std::forward<T>(str)) {}

    explicit A(int x) : text(std::to_string(x)) {}
};

int main() {
    A x("123");
    auto copyX(x);
}
```

Перегрузка: еще пример

Класс после инстанциирования

```
class A {
  private:
    std::string text;
  public:
    explicit A(A& str) : text(std::forward<A&>(str)) {}
    A(const A& rhs); // сгенерировано компилятором
    explicit A(int x) : text(std::to_string(x)) {}
};
```

Прямая передача

```
template <typename T>
void fwd(T&& x) {
    f(std::forward<T>(x));
}
```

Целевая функция f должна получить в точности те же объекты, которые переданы функции fwd.

Прямая передача

```
template <typename T>
void fwd(T&& x) {
    f(std::forward<T>(x));
}
```

Целевая функция f должна получить в точности те же объекты, которые переданы функции fwd.

```
void f(const std::vector<int>& v);
f({0, 1, 0, 1});     // ok
fwd({0, 1, 0, 1});   // error
```

Прямая передача

```
template <typename T>
void fwd(T&& x) {
    f(std::forward<T>(x));
}
```

Целевая функция f должна получить в точности те же объекты, которые переданы функции fwd.

```
void f(const std::vector<int>& v);
f({0, 1, 0, 1});     // ok
fwd({0, 1, 0, 1});   // error

auto x = {0, 1, 0, 1}; // std::initializer_list<int>
fwd(x);              // ok
```

Перемещающие операции

Перемещающий конструктор и перемещающий оператор присваивания:

- генерируются только при необходимости;
- выполняют «почленное перемещение»;
- не генерируются при явном объявлении;
- не являются независимыми;
- не генерируются при явном объявлении копирующих операций или деструктора.

Перемещающие операции

Если все-таки нужно сгенерировать?

Перемещающие операции

Если все-таки нужно сгенерировать?

```
class A {
  public:
    A(A&&) = default;
    A& operator(A&&) = default;
    virtual ~A() { ...}
};
```

Некоторые выводы

- Перемещение — новая ключевая идея C++ — обычно используется для оптимизации копирования.
- std::move ничего не перемещает, std::forward ничего не передает.
- Не объявляйте объекты константными, если нужно выполнять перемещение из них.
- Применяйте std::move к rvalue-ссылкам, а std::forward к универсальным ссылкам.
- Перегрузка для универсальных ссылок может привести к неприятным эффектам (конструкторы с прямой передачей соответствуют неконстантным lvalue обычно лучше копирующих конструкторов)
- Большинство стандартных типов в C++11 перемещаемы, например, контейнеры STL.
- Некоторые типы только перемещаемы, например, объекты потоков, std::thread, std::unique_ptr.