# Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 3
02.10.2019

# Ссылки и копии

(C++ Core Guidelines)
**F.16: For "in"parameters, pass cheaply-copied types by value and others by reference to const**
Both let the caller know that a function will not modify the argument, and both allow initialization by rvalues. What is “cheap to copy” depends on the machine architecture, but two or three words (doubles, pointers, references) are usually best passed by value. When copying is cheap, nothing beats the simplicity and safety of copying, and for small objects (up to two or three words) it is also faster than passing by reference because it does not require an extra indirection to access from the function.

# Ссылки и копии

(C++ Core Guidelines)
**F.20: For "out"output values, prefer return values to output parameters**
A return value is self-documenting, whereas a & could be either in-out or out-only and is liable to be misused. This includes large objects like standard containers that use implicit move operations for performance and to avoid explicit memory management.
If you have multiple values to return, use a tuple or similar multi-member type.

# Строки

```
void f(const std::string& s);
```

Что если нужно вызвать эту функцию от подстроки или от `char*`?

# Строки

```
void f(const std::string& s);
```

Что если нужно вызвать эту функцию от подстроки или от `char*`?

```
void f(std::string_view s);
```

# std::string_view

```
std::string str = "James while John had had had had had had had
// плохо: создаем новую строчку
std::cout << str.substr(10, 10) << std::endl;

// норм: никакого копирования
std::string_view view = str;

// string_view::substr возвращает новый string_view
std::cout << view.substr(10, 10) << std::endl;
```

# Временные объекты

```
#include <iostream>
#include <vector>

void Do(std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

void DoConst(const std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

int main() {
    Do(std::vector<int>(10));
    DoConst(std::vector<int>(10));
}
```

# Временные объекты

```
#include <iostream>
#include <vector>

void Do(std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

void DoConst(const std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

int main() {
    Do(std::vector<int>(10));  // error
    DoConst(std::vector<int>(10));
}
```

# Временные объекты

```
#include <iostream>
#include <vector>

void Do(std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

void Do(const std::vector<int> &v) {
    std::cout << "const v.size() == " << v.size() << std::endl;
}

int main() {
    std::vector<int> v(10);
    Do(v);
    Do(std::vector<int>(10));
}
```

# Временные объекты

```
#include <iostream>
#include <vector>

void Do(std::vector<int> &v) {
    std::cout << "v.size() == " << v.size() << std::endl;
}

void Do(const std::vector<int> &v) {
    std::cout << "const v.size() == " << v.size() << std::endl;
}

int main() {
    std::vector<int> v(10);
    Do(v);                          // v.size() == 10
    Do(std::vector<int>(10)); // const v.size() == 10
}
```

# Типы ссылок

```
void f(T&& x);  // rvalue-ссылка

T&& x = T();  // rvalue-ссылка

auto&& y = x; // не rvalue-ссылка

template<typename T>
void f(T&& x);  // не rvalue-ссылка

template<typename T>
void f(std::vector<T>&& x);  // rvalue-ссылка
```

# Типы ссылок

При инициализации универсальной ссылки определяется, какую ссылку она представляет – rvalue/lvalue.

```
template<typename T>
void f(T&& x);  // не rvalue-ссылка

T a;
f(a); // lvalue-ссылка
f(std::move(a)); // rvalue-ссылка
```

# Типы ссылок

```
template<typename T>
void f(std::vector<T>&& x);

std::vector<int> v;
f(v); // error
```

# Типы ссылок

```
template<typename T>
void f(std::vector<T>&& x);

std::vector<int> v;
f(v); // error
```

```
template<typename T>
void f(const T&& x); // rvalue-ссылка
```

# Типы ссылок

- **lvalue**
  - обычные ссылки, константные ссылки, ...
  - это не временный объект и не тот объект, который вскоре будет уничтожен.
- **xvalue**
  - объект который вот-вот должен быть уничтожен (expired)
  - пример — то, что приходит в оператор перемещения
- **prvalue**
  - pure, настоящее rvalue
  - пример — результат вызова функции, у которой возвращаемое значение – не ссылка.

**glvalue** — lvalue или xvalue
**rvalue** — xvalue, или временный объект, или значение, не ассоциированное с объектом.

# xvalue: два примера

1.
```
int&& f(){   return 3; }
// ...
f();
```

2.
```
static_cast<int&&>(7);
std::move(7);
```

# Типы ссылок: практическое правило

1. Если у выражения можно взять адрес — это **lvalue**
2. Если тип выражения — `T&` или `const T&`, — это **lvalue**
3. Иначе это **rvalue**. Обычно — литералы, результат вызова функций и т. п.

# emplace_back

```
int main() {
    std::vector<A> as;
    // as.push_back(A(3));
    as.reserve(5);
    as.emplace_back(1);
    as.emplace_back(2);
    as.emplace_back(3);
    as.emplace_back(4);
    as.emplace_back(5);
}
```

# 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?