

Лекция 3.
Move semantics
&& perfect forwarding

CS Club, Novosibirsk, 2019

Быстрые программы

- Программы на C++ ценят за их скорость

```
1.  string inverted(string const& str)
2.  {
3.      if (str.empty())
4.          return string();
5.
6.      string tmp(str);
7.      size_t beg = 0;
8.      size_t end = tmp.size() - 1;
9.
10.     while(beg < end)
11.         swap(tmp[beg++], tmp[end--]);
12.
13.     return tmp;
14. }
15.
16. //...
17. string a = "olleH";
18. string b = inverted(a);
```

- Сколько копирований строки в этом коде?

Быстрые программы

- Программы на C++ ценят за их скорость

```
1.  string inverted(string const& str)
2.  {
3.      if (str.empty())
4.          return string();
5.
6.      string tmp(str);
7.      size_t beg = 0;
8.      size_t end = tmp.size() - 1;
9.
10.     while(beg < end)
11.         swap(tmp[beg++], tmp[end--]);
12.
13.     return tmp;
14. }
15.
16. //...
17. string a = "olleH";
18. string b = inverted(a);
```

- Здесь будет два копирования*. Можно ли улучшить?

Быстрые программы

- Программы на C++ ценят за их скорость

```
1. string inverted(string const& str)
2. {
3.     string tmp(str);
4.
5.     if (!str.empty())
6.     {
7.         size_t beg = 0;
8.         size_t end = tmp.size() - 1;
9.
10.        while(beg < end)
11.            swap(tmp[beg++], tmp[end--]);
12.    }
13.
14.    return tmp;
15. }
16.
17. //...
18. string a = "olleH";
    string b = inverted(a);
```

- Да, если положиться на NRVO – одно копирование. Но есть ограничения на код.

Return Value Optimization

```
1. struct X
2. {
3.     X()          { cout << "X()" << endl; }
4.     X(X const&) { cout << "X(X const&)" << endl; }
5. };
```

- Сколько копирований происходит при вызове этих функций?

```
1. X foo()
2. {
3.     //...
4.     return X();
5. }
```

- Одно
- Ноль при **RVO**

```
1. X bar()
2. {
3.     X value;
4.     //...
5.     return value;
6. }
```

- Одно при RVO
- Ноль при **NRVO**

```
1. X qux()
2. {
3.     X value1;
4.     X value2;
5.
6.     bool cond = true;
7.     //..
8.     return cond
9.         ? value1
10.        : value2;
11.
12. }
```

- Одно

Lvalue vs Rvalue

- **Lvalue** – выражение, описывающее не временный объект, чаще всего именованный. У этого объекта можно взять адрес и, возможно, изменить его. Мнемоника: то, что может быть по левую сторону от знака '=' (и по правую).
- **Rvalue** – выражение, описывающее временный объект или return выражение. Мнемоника: то, что может быть только по правую сторону от знака '='.
- **Lvalue** и **Rvalue** – это value categories. Не то же самое, что lvalue и rvalue reference. А когда есть различия?

Lvalue vs Rvalue. Примеры

```
1. // L-value
2. string answ = "42";
3. answ = "13";
4.
5. //...
6. string& answer_ref() {return answ;}
7. answer_ref() = "7";
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
```

Lvalue vs Rvalue. Примеры

```
1. // L-value
2. string answ = "42";
3. answ = "13";
4.
5. //...
6. string& answer_ref() {return answ;}
7. answer_ref() = "7";
8.
9. // R-value
10. string answer()
11. {
12.     string answ = "7";
13.     return good_day() ? answ : "13";
14. }
15.
16. string const& a = answer(); // works fine
17. string&      b = answer(); // nope!
18. answer() = "13";           // nope!
```


Возврат значения

```
1. using dict_t = map<string, string>;  
2. dict_t get_dictionary(/*...*/) {/*...*/}  
3.  
4. //...  
5. dict_t en_ru = get_dictionary(/*...*/);
```

- Что происходит:
 - Копируем тяжелое значение в новый объект при возврате из функции
 - Удаляем только что созданный временный объект

Возврат значения

```
1. using dict_t = map<string, string>;
2. dict_t get_dictionary(/*...*/) {/*...*/}
3.
4. //...
5. dict_t en_ru = get_dictionary(/*...*/);
```

- Что происходит:
 - Копируем тяжелое значение в новый объект при возврате из функции
 - Удаляем только что созданный временный объект
- Может, просто обменяться значениями?

```
1. // what a wonderful would be if (a bit pseudo code):
2. map& map(<rvalue map>) noexcept
3. {
4.     swap(*this, <rvalue map>);
5.     return *this;
6. }
```

Как отличить временный объект от не временного

- Поможет *rvalue reference*. Такая ссылка крайне полезна при перегрузке.

1.	<code>string low_case(string const& s);</code>
2.	<code>string low_case(string&& s);</code>

- Rvalue reference можно объявить переменную, но в 99% случаев используется как формальный аргумент функции (см. выше).

1.	<code>string&& a = answer(); // compiled, but rare code</code>
2.	<code>// extends lifetime of the object like 'string const&' does</code>

Преобразование & <-> const & <-> &&

- | | | | | | |
|----|------------------|------|------|--------|-------|
| 1. | implemented | : | void | foo | (T&); |
| 2. | NOT implemented: | void | foo | (T&&); | |

- Можно вызвать для Lvalue, но не для Rvalue.

- | | | | | | |
|----|------------------|------|------|--------|-------------|
| 1. | implemented | : | void | foo | (T const&); |
| 2. | NOT implemented: | void | foo | (T&&); | |

- Можно вызвать для Lvalue и для Rvalue, но нельзя их ОТЛИЧИТЬ.

- | | | | | | |
|----|------------------|------|------|-------------|--------|
| 1. | implemented | : | void | foo | (T&&); |
| 2. | NOT implemented: | void | foo | (T&); | |
| 3. | NOT implemented: | void | foo | (T const&); | |

- Можно вызвать для Rvalue, а вот вызов для Lvalue даст compile error.

Move constructor

```
1. // copy constructor
2. string(string const& other)
3.     : buf_ (new char[other.size() + 1]) // buf_ is 'char*'
4.     , size_(other.size())
5. {
6.     strcpy(buf_, other.buf_);
7. }
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
```

Move constructor

```
1. // copy constructor
2. string(string const& other)
3.     : buf_ (new char[other.size() + 1]) // buf_ is 'char*'
4.     , size_(other.size())
5. {
6.     strcpy(buf_, other.buf_);
7. }
8.
9. // move constructor, no need to copy in 'string s = answer()'
10. string(string&& other)
11.     : buf_ (other.buf_ ) // stealing pointer
12.     , size_(other.size())
13. {
14.     other.buf_ = nullptr; // some valid state
15.     other.size_ = 0;
16. }
17.
18.
19.
20.
21.
22.
23.
```

Move constructor

```
1. // copy constructor
2. string(string const& other)
3.     : buf_ (new char[other.size() + 1]) // buf_ is 'char*'
4.     , size_(other.size())
5. {
6.     strcpy(buf_, other.buf_);
7. }
8.
9. // move constructor, no need to copy in 'string s = answer()'
10. string(string&& other)
11.     : buf_ (other.buf_ ) // stealing pointer
12.     , size_(other.size())
13. {
14.     other.buf_ = nullptr;
15.     other.size_ = 0;
16. }
17.
18. // move constructor via delegating and swap (*)
19. string(string&& other) noexcept
20.     : string() // default constructor
21. {
22.     swap(other);
23. }
```

Move assignment operator

```
1. // swap trick
2. string& operator=(string const& other)
3. {
4.     string tmp(other);
5.     swap(tmp);
6.     return *this;
7. }
8.
9.
10.
11.
12.
13.
14.
15.
```


Move assignment operator

```
1. // swap trick
2. string& operator=(string const& other)
3. {
4.     string tmp(other);
5.     swap(tmp);
6.     return *this;
7. }
8.
9. // extended swap trick gives
10. // copy and move 'operator=' at the same time
11. string& operator=(string other) noexcept
12. {
13.     swap(other);
14.     return *this;
15. }
```

Rvalue reference – Rvalue?

- Как быть с именованной Rvalue ссылкой?

```
1. string(string&& other)
2.     : vec_(other.vec_) // vec_ is 'vector<char>'
3. { // doesn't work as expected
4. }
```

- Здесь Rvalue reference – не Rvalue, т.к. указывает на именованный (!) объект.
- Объект `other` можно изменять как и любой *lvalue*.
- Как же его передать дальше как rvalue?

std::move

- Как объяснить компилятору, что объект все-таки rvalue?
- `std::move` не ‘перемещает’ объект, а только изменяет его тип, делая его rvalue category.

```
1. template< class T >
2.     typename std::remove_reference<T>::type&& move(T&& t);
3.
4.     //...
5.
6.     string(string&& other)
7.         : vec_(std::move(other.vec_)) // vec_ is 'vector<char>'
8.     { // now works fine!
9.     }
```

Move constructor vs other constructors

- Move семантика является родной для RAII
 - Копирование часто невозможно, а передача владения допустима
 - Пример: `std::unique_ptr`
- Move конструктор не создается автоматически, если пользователем определен сору конструктор, или любой оператор присваивания или деструктор – т.е. пользователь определил действия с нетривиальным состоянием.
- Move конструктор ‘выключает’ автоматическую генерацию других конструкторов (но не в старых MSVC), но в C++11 есть удобный прием:

1.	<code>struct data</code>
2.	<code>{</code>
3.	<code> data() = default;</code>
4.	<code> data(data&& other){/*...*/}</code>
5.	<code> //...</code>
6.	<code>};</code>

Возврат из функции значения vs &&

- Rvalue reference – лишь ссылка. Вернув ее на временный объект, получите undefined behavior.
- Возвращайте значение! Контринтуитивно, да? Move семантика обеспечит ‘условно бесплатное’ копирование. А если возможно, сработает RVO/NRVO.
- Можно вернуть && на поле (как и обычную ссылку). Но тогда удаление поля будет зависеть от контекста вызова. Почему?

Возврат больших объектов по значению

- Теперь можно возвращать большие объекты также, как и встроенные типы без потери производительности

```
1. void send(vector<double> const& data);
2.
3. void      extract_data (vector<double>& arr); // old
4. vector<char> extracted_data();              // new
5.
6. void send_data()
7. {
8.     // old
9.     vector<char> data;
10.    extract_data(data);
11.    send(data);
12.
13.    // new
14.    send(extracted_data());
15. }
```

Нужен ли теперь `swap(T&, T&)`?

- Если ваш класс обладает `move` семантикой, то нет необходимости реализовывать отдельный `swap` для него – подойдет обобщенный.
- Но будьте осторожны с `swap-trick`'ом в этом случае. Что не так?

```
1.  template<class T>
2.      void std::swap(T& lhs, T& rhs)
3.  {
4.      T tmp = move(lhs);
5.      lhs  = move(rhs);
6.      rhs  = move(tmp);
7.  }
```

Передача по значению

- В пример с функцией `inverted` по-прежнему осталось лишнее копирование при передаче параметра. Что делать?
- Общее правило – если делаете копию внутри и тип перемещаем, рассмотрите передачу по значению

```
1.  string inverted(string str)
2.  {
3.      if (str.empty())
4.          return string();
5.
6.      // ...
7.
8.      return str;
9.  }
```

- Вызов `inverted(str)` приведет к копированию
- `inverted(str + "test")` или `inverted(move(str))` – к перемещению.
- Бинго! Можно сократить количество копирований до нуля!

Forwarding argument problem

- Как передать параметры внутрь конструктора объекта так, как если бы фабрики не было, а был бы прямой вызов конструктора?

```
1.  template<typename T, typename Arg>
2.  shared_ptr<T> factory(Arg arg)
3.  {
4.      return shared_ptr<T>(new T(arg));
5.  }
```

Forwarding argument problem (2)

- Типовое решение:

```
1.  template<typename T, typename Arg>
2.  shared_ptr<T> factory(Arg& arg)
3.  {
4.      return shared_ptr<T>(new T(arg));
5.  }
6.
7.  template<typename T, typename Arg>
8.  shared_ptr<T> factory(Arg const & arg)
9.  {
10.     return shared_ptr<T>(new T(arg));
11. }
```

- Есть ли проблемы у такого решения?

Forwarding argument problem (2)

- Типовое решение:

```
1.  template<typename T, typename Arg>
2.  shared_ptr<T> factory(Arg& arg)
3.  {
4.      return shared_ptr<T>(new T(arg));
5.  }
6.
7.  template<typename T, typename Arg>
8.  shared_ptr<T> factory(Arg const & arg)
9.  {
10.     return shared_ptr<T>(new T(arg));
11. }
```

- Есть ли проблемы у такого решения?
 - А если параметр не один – комбинаторный взрыв.
 - Никак не воспользоваться move семантикой.

Новые правила для ссылок в C++11

- Reference collapsing rules
 - A& & becomes A&
 - A& && becomes A&
 - A&& & becomes A&
 - A&& && becomes A&&

```
1. template<typename T>  
2. void foo(T&& arg);
```

- Особые правила вывода типов при передаче &&
 - Вызов для lvalue на A, T становится A&, arg становится A&
 - Вызов для rvalue на A, T становится A, arg становится A&&
- Это называется *universal reference*

Perfect forwarding

- Немного магии:

```
1.  template<typename T, typename Arg>
2.  shared_ptr<T> factory(Arg&& arg)
3.  {
4.      return shared_ptr<T>(new T(std::forward<Arg>(arg)));
5.  }
6.
7.
8.
9.
10.
11.
12.
```

Perfect forwarding

- Немного магии:

```
1.  template<typename T, typename Arg>
2.  shared_ptr<T> factory(Arg&& arg)
3.  {
4.      return shared_ptr<T>(new T(forward<Arg>(arg)));
5.  }
6.
7.  // where
8.  template<class S>
9.  S&& forward(typename remove_reference<S>::type& a) noexcept
10. {
11.     return static_cast<S&&>(a);
12. }
```

- Попробуйте произвести вызов функции factory для lvalue и rvalue объектов. Profit!
- А зачем remove_reference?

Variadic templates, анонс

- А что же делать в случае многих параметров?

Variadic templates, анонс

- А что же делать в случае многих параметров?

```
1.  template<class T, class... Args>
2.  std::shared_ptr<T> factory(Args&&... args)
3.  {
4.      return std::shared_ptr<T>(
5.          new T(std::forward<Args>(args)...));
6.  }
7.
8.  int main()
9.  {
10.     auto x = factory(string("str"));
11.     return 0;
12. }
13.
```

- Но это уже другая история...

std::move_if_noexcept (*)

- Move конструктор и оператор присваивания обладают семантикой `swar` функции. От них не ожидают исключений.
- STL контейнеры для обеспечения строгой гарантии безопасности исключений требуют явного указания, что используемый тип не генерирует исключений в `move` конструкторе и присваивании, в частности `vector` в `resize`.

1.	<code>my_type(my_type&& other) noexcept; // since C++11</code>
2.	<code>my_type operator=(my_type&& other) noexcept; // since C++11</code>

- В MSVC поддерживается `noexcept` лишь с версии VS2015.

Спасибо за внимание!
Есть ли вопросы?