Лекция 3. Семантика перемещения

ИУ8

October 2, 2018

Содержание

На текущей лекции

- rvalue ссылки
- семантика перемещения
- прямая передача и универсальные ссылки

Мнение эксперта

Perhaps the most significant new feature in C++11 is rvalue references (*Scott Meyers*)

Rvalue ссылки, пожалуй, самая главная возможность C++11

Рассмотрим операции копирования

```
class String {
  char * data = nullptr;
 public:
  String& operator=(const String& rhs) {
    if (this != &rhs) {
      delete[] data;
      data = new char[strlen(rhs.data) + 1];
      strcpy(data, rhs.data);
      data[strlen(rhs.data)] = 0;
    }
    return *this;
  }

  String(const String& rhs) {
    data = new char[strlen(rhs.data) + 1];
    strcpy(data, rhs.data);
    data[strlen(rhs.data)] = 0;
  }
};
```

Как в действительности выглядит копирование строк

Пример

```
1  void update(std::string& data);
2  // ...
3  const int N = 1 << 23;
4  std::vector<std::string> pull(N);
5  for (int i = 0; i < N; ++i) {
6
7    std::string data = get_data_by_index(i);
8    update(data);
9    pull[i] = data; // copy data to vector.
10
11 } // |data| is destroyed.
```

Пример

```
1  void update(std::string& data);
2  // ...
3  const int N = 1 << 23;
4  std::vector<std::string> pull(N);
5  for (int i = 0; i < N; ++i) {
6
7    std::string data = get_data_by_index(i);
8    update(data);
9    pull[i] = move(data); // MOVE data to vector.
10
11 } // |data| is destroyed, but |data| is already empty.
```

Как выглядит перемещение

Остается определить как компилятор может понимать, когда происходит копирование объекта, а когда перемещение. Для этого в стандарт был включен специальный тип ссылок на объекты - rvalue-ссылки.

Определения

lvalue значение-это выражение, которое ссылается на область памяти и к которому применима операция получения адреса выражения

rvalue значение-это НЕ **lvalue** выражение

Критерий rvalue

If it has a name, then it is an lvalue. Otherwise, it is an rvalue.

Реализация операций перемещения

```
1  class String {
2    char * data = nullptr;
3   public:
4    String& operator=(const String& rhs);
5    String& operator=(String&& rhs) {
6      if (this != &rhs) {
7        std::swap(data, rhs.data);
8      }
9      return *this;
10   }
11
12   String(const String& rhs);
13   String(String&& rhs) {
14       std::swap(data, rhs.data);
15   }
16 };
```

std::move

Как сообщить компилятору, что объект может быть перемещен? Для этого надо привести объект к rvalue-ссылке. Этим занимается функция *std::move*.

std::move

Исходный код C++14

```
1 template<typename T>
2 decltype(auto) move(T&& arg) {
3   using ReturnType = remove_reference_t<T>&&;
4   return static_cast<ReturnType>(arg);
5 }
```

- std::move выполняет безусловное примедение агрумента к rvalue ссылке
- std::move ничего не перемещает!
- std::move во время выполнения ничего не делает!
- std::move не генерирует выполняющийся код!

Какие объекты компилятор может перемещать без последсвий и без помощи разработчика?
Временные объекты, которые вот-вот будут уничтожены. К таким объектам можно причислить объект возвращаемый функцией (критерий: нет имени - значит rvalue-ссылка).

Пример

```
1 void update(std::string& data);
2 // ...
3 const int N = 1 << 23;
4 std::vector<std::string> pull(N);
5 for (int i = 0; i < N; ++i) {
6
7   pull[i] = get_data_by_index(data, i);
8
9 }
```

Семантика перемещения

Выводы

Семантика перемещения позволяет компиляторам заменять дорогостоящие операции копирования менее ресурсозатратными перемещениями. Перемещающие конструкторы и перемещающие операторы присваивания предоставляют контроль над семантикой перемещения.

Семантика перемещения позволяет также создавать типы, которые могут реализовывать только операцию перемещения, такие как std::unique_ptr, std::future или std::thread.

Не используйте объект после того, как его переместили в другой!

- Не объявляйте объекты константными, если хотите иметь возможность их перемещать
- std::move не гарантирует, что приведенный объект будет иметь право быть перемещенным

Еще пример

```
1  struct Engine {
2    Engine(const std::string& model)
3      : model_(model) {}
4
5
6   private:
7    std::string model_;
8  };
9
10 struct Car {
11   Car(const std::string& engine_model)
12     : engine_(engine_model) {}
13
14
15  private:
16   Engine engine_;
17 };
18
19 std::string engine_model;
20 std::cin >> engine_model;
21 Car car(engine_model); // call Car(const std::string&);
```

Еще пример

```
1  struct Engine {
2    Engine(const std::string& model)
3      : model_(model) {}
4    Engine(std::string&& model)
5      : model_(std::move(model)) {}
6   private:
7    std::string model_;
8  };
9
10 struct Car {
11   Car(const std::string& engine_model)
12     : engine_(engine_model) {}
13   Car(std::string&& engine_model)
14     : engine_(std::move(engine_model)) {}
15  private:
16   Engine engine_;
17 };
18
19 std::string get_engine_model_from_file();
20
21 Car car(get_engine_model_from_file()); // call Car(std::string&&);
```

В предыдущих примерах вызываются два различных конструктора.
Если в конструтор передается lvalue ссылка, то в классе Engine поле *model_* будет хранить копию входного аргумента.
Если в конструтор передается rvalue ссылка, то входной аргумент переместится в поле *model_* класса Engine.

Но возникает проблема: как реализовать **обобщенный код**, который использовать перемещение там, где это возможно, и копирование во всех остальных случаях.

Задача

```
struct Car {
  Car(const std::string& engine_model);
  Car(std::string&& engine_model);
  // ...
};

struct Airplane {
  Airplane(const std::vector<int>& params);
  Airplane(std::vector<int>&& params);
  // ...
};
```

Требуется написать обобщенную функцию make, которая создавала бы объекты типа Car или Airplane, и которая использовала бы перемещение там где это мозможно.

```
template <typename T, typename Arg>
T make(Arg arg);
```

Ошибочный путь

Конечно, можно реализовать две функции make. Например,

```
1  template <typename T, typename Arg>
2  T make(const Arg& arg) {
3    return T(arg);
4  }
5
6
7  template <typename T, typename Arg>
8  T make(Arg&& arg) {
9    return T(std::move(arg));
10 }
```

Но это плохой способ.

Right way

Чтобы решить поставленную задачу, необходимо воспользовать универсальными ссылками.

```
template <typename T, typename Arg>
T make(Arg&& arg) {
  return T(std::forward<Args>(arg));
};
```

Это, так называемая, прямая (идеальная) передача аргументов (англ. термин perfect forward).
Воспользовавшись функцией std::forward и универсальной ссылкой Arg&&, конструктор типа T принимает или lvalue ссылку, или rvalue в зависимости от вызывающего кода.

```
auro car = make<Car>(get_engine_model_from_file());
// call Car(std::string&&),
// because get_engine_model_from_file() returns rvalue

std::vector<int> v = {1, 4, 3};
auto air = make<Airplane>(v);
// call Airplane(const std::vector<int>&),
// because v - is lvalue
```

Прямая передача

```
1 template<class T>
2 T&& forward(remove_reference_t<T> & a) {
3   return static_cast<T&&>(a);
4 }
```

- std::forward выполняет приведение к rvalue ссылке, тогда и только тогда, когда входной аргумент rvalue ссылка
- std::forward ничего не передает!
- std::forward во время выполнения ничего не делает!
- std::forward не генерирует выполнающийся код!

Прямая передача

Выводы

Прямая передача делает возможным написание **шаблонов функций**, которые принимают произвольные аргументы и передают их другим функциям так, что целевые функции получают **в точности** те же аргументы, что и переданные исходным функциям.

Rvalue-ссылки делают возможными как семантику перемещения, так и прямую передачу.

lifehack

see example.cpp

The end

Самостоятельное изучение

- Правило свертывания ссылок
- rvalue-ссылки
- универсальные ссылки

Список литературы

- С. Мэйерс - Эффективный и современный C++
- https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers
- http://thbecker.net/articles/rvalue_references/section_01.html
- https://accu.org/index.php/journals/227
- http://alenacpp.blogspot.ru/2008/02/rvo-nrvo.html
- https://www.ibm.com/developerworks/community/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/RVO_V_S_std_move?lang=en