Сибирь II

RAII

Указатели – хорошо или нет?

Идиома RAII

Resourse acquisition is initialisation

А зачем оно надо?

Умные указатели

`std::unique_ptr`

Осуществляет единственное владение объектом

- Работает как обычный указатель, но является только перемещаемым
- По умолчанию ресурсы удаляются с помощью `delete`, но можно создавать свои удалители
- Легко преобразуется в `shared_ptr`

```
std::unique_ptr<int[]> vec_ptr(new int[3]{1, 2, 3});
std::cout << vec_ptr[0] << vec_ptr[1] << vec_ptr[2] << std::endl;
```

`std::shared_ptr`

Позволяет “расшаривать” объект

- Аналогичен “сборке мусора”
- Копируемый, работает с помощью *счётчика ссылок*, память для которого должна выделяться динамически
- Требует атомарных операций для счетчика ссылок
- По умолчанию ресурсы удаляются с помощью `delete`, но можно создавать свои удалители

`std::weak_ptr`

Дополнение к `shared_ptr` Не может быть разыменован и проверен на “нулевость”

Семантика перемещения

Чтобы экономить ресурсы (но не всегда)

```
class String {
 public:
  String() {...}
  String(const String&) {...}
  String(String&&) {...} // конструктор копирования

  String& operator=(const String&) {...}
  String& operator=(String&&) {...}

};
```

Отличие `rvalue` **от** `lvalue`

rvalue как правило указывают на объекты, которые могут быть перемещены, концептуально – на временные объекты

- type& – lvalue
- type&& – rvalue(как правило)

- если для объекта можно получить адрес – lvalue
- иначе – rvalue

Параметры функций – всегда lvalue

```
class Widget {
  public:
  Widget(Widget&& rhs); // rhs -- lvalue, хотя
                        // и имеет тип rvalue
};
```

Важно – при выводе типов (шаблон или `auto&&`**) мы получаем *универсальную ссылку***

`std::move`

Ничего не перемещает

*`std::move` лишь выполняет **безусловное** приведение к rvalue, чтобы указать, что объект предназначен для перемещения*

```
class Widget {
  public:
  Widget(Widget&& rhs)
  : name(std::move(rhs.name)), p(std::move(rhs.p)) {}

  private:
  std::string name;
  std::shared_ptr<Data> p;
};
```

`std::forward<T>`

Ничего не передаёт

*`std::forward` лишь выполняет **условное** приведение к rvalue*

```
class Widget {
  public:
  template<class T>
  void setName(T&& newName) { name = std::forward<T>(newName); }
};
```

- Применяйте std::move к rvalue, std::forward – к универсальным ссылкам
- Не надо выполнять перемещение к возвращаемому из функции значению