# 4.28. Мотивировка умных указателей. Класс unique_ptr, его идея и реализация основных методов. Особенности поведения unique_ptr при попытке его скопировать. Функция make_unique, её реализация, пример использования, преимущество перед обычным конструктором unique_ptr

**Умный указатель** - инструмент, который позволяет автоматически освобождать динмаически выделенные ресурсы. std::unique_ptr и std::shared_ptr решают проблему автоматического очищения памяти при выходе указателя из области видимости (так как можно легко потерять delete, соответствующий какому-то new, например, если между new и delete бросится исключение и delete не вызовется, давайте вспомним про RAII и исключения в конструкторах, функциях).

**unique_ptr** должен использоваться, когда ресурс памяти не должен быть разделяемым (у этого указателя нет к-ра копирования), но он может быть передан другому unique_ptr.

unique_ptr **нельзя копировать**, соответствующий конструктор и оператор у них удалены, но зато можно мувать (поэтому вектор из UP корректен - для этого же нужны noexcept'ы (они не обязательны, но нужны, чтобы вектор гарантировал безопасность исключений)). Присвоить значение этому указателю можно только в момент объявления (т.е. инициализация умного указателя должна быть в момент объявления).

```
template<typename T, typename Deleter = std::default_delete<T>>
class unique_ptr {
 private:
  T* pointer;
 public:
  unique_ptr() {
    pointer = nullptr;
  }

  explicit unique_ptr(T* ptr): pointer(ptr) {}

  unique_ptr(const unique_ptr<T>& other) = delete;

  unique_ptr<T>& operator=(const unique_ptr<T>& other) = delete;

  unique_ptr(unique_ptr<T>&& other)  noexcept {
    pointer = std::move(other.pointer);
    other.pointer = nullptr;
  }

  unique_ptr& operator=(unique_ptr<T>&& other)  noexcept {
    delete pointer;
    pointer = std::move(other.pointer);
    other.pointer = nullptr;
  }

  ~unique_ptr() {
    delete pointer;
  }

  T& operator*() const {
    return *pointer;
  }

```

```
};
```

На самом деле, unique_ptr принимает два шаблонных параметра: второй—это typename Deleter, у которого определен оператор () и он вызывается (как функция) в деструкторе (дефолтный Deleter вызывает delete). Unique_ptr - легковесный, быстрый; первый пример класса, который мувать можно, но копировать нельзя.

**make_unique**

Какие у нас есть проблемы с unique_ptr?

1. Мы не можем полностью избавиться от использования new (см. сравнение).
2. Нужно явно перечислять аргументы типа шаблона.
3. Exception safety: см. код.

```
// Сравнение make_unique и unique_ptr
std::make_unique<int>(1);
std::unique_ptr<int>(new int(1));

// Применение make_unique

class Fraction
{
 private:
  int m_numerator = 0;
  int m_denominator = 1;

 public:
  Fraction(int numerator = 0, int denominator = 1) :
           m_numerator(numerator), m_denominator(denominator)
  {
  }

  friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
  {
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
  }
};

// exception safety:
void function(std::unique_ptr<A>(new A()), std::unique_ptr<B>(new B())) {
    ... }

void function(std::make_unique<A>(), std::make_unique<B>()) { ... }

// если B() создаёт исключение, а A - нет, то стандарт c++ не требует, чтобы
// первый объект был уничтожен; во втором же случае у нас временные объекты
// и стандарт c++ обязывает их уничтожить

int main()
{
  // Создаем объект с динамически выделенным Fraction с numerator = 7 и denominator = 9
  std::unique_ptr<Fraction> f1 = std::make_unique<Fraction>(7, 9);
  std::cout << *f1 << '\n'; // вывод: 7/9

  // Создаем объект с динамически выделенным массивом Fraction длиной 5.
  // Используем автоматическое определение типа данных с помощью ключевого слова auto
  auto f2 = std::make_unique<Fraction[]>(5);
  std::cout << f2[0] << '\n'; // вывод: 0/1

  return 0;
}
```

## 4.29. Идея реализации класса shared_ptr (без поддержки weak_ptr, без поддержки нестандартных аллокаторов и deleter'ов): конструкторы, деструктор, операторы присваивания. Функция make_shared, ее реализация, пример использования, преимущества перед обычным конструктором shared_ptr.

Если нам нужно иметь несколько указателей на один и тот же объект, то для этого воспользуемся **std::shared_ptr**. Внутри shared_ptr есть счетчик, который показывается, сколько копий у этого указателя существуют и указывают на то же что и он (решает проблему многократного удаления по одному и тому же указателю).

В этом случае оба умных указателя в равной мере управляют обычным указателем. Освобождение памяти произойдет в момент, когда последний shared_ptr, обладающий общим ресурсом, покинет область видимости. Оператор и конструктор копирования разрешены. shared_ptr предоставляет больше возможностей, но увеличивается и расход памяти, и время доступа.

```
template<typename T>
class shared_ptr {
 private:
  // на самом деле, не очень эффективно хранить ptr и counter как 2 отдельных указателя
  // отсюда ускорение при использовании make_shared, так как
  // при его использовании эти два указателя будут лежать рядом
  T* ptr = nullptr;
  size_t* counter = nullptr;
 public:
  // shared_ptr(T* ptr): ptr(ptr)
  // shared_ptr(shared_ptr other): ptr(other.ptr), count(++other.count)
  // реализация сверху НЕ ОК, т.к. никто не гарантировал, что shared ptr'ов 2

  // static count тоже не работает: он получится один на ВЕСЬ класс
  // вне зависимости от T

  // правильное решение: иметь указатель на счётчик
  shared_ptr() {}

  shared_ptr(T* ptr): ptr(ptr), counter(new size_t(1)) {}

  shared_ptr(const shared_ptr& other): ptr(other.ptr), counter(other.counter
   ) {
    ++*counter;
  }

  shared_ptr(shared_ptr&& other): ptr(other.ptr), counter(other.counter) {
    other.ptr = nullptr;
    other.counter = nullptr;
  }

  // + операторы присваивания, но с ними понятно

  // деструктор
  ~shared_ptr() {
    if (*counter > 1) {
      --*counter;
      return;
```

```
    }
    delete ptr;
    delete counter;
  }

  // разыменование
  T& operator*() const { return *ptr; }
  T& operator->() const { return ptr; }

  size_t use_count() const { return *count; }

};
```

Аргументация для **make_shared** совпадает с аргументацией для 4.28 (не использовать больше никогда new/delete, например), но теперь здесь добавляется скорость работы программы (см. комментарий в коде). Пишем make_shared!

```
template <typename ...Args>
unique_ptr<T> make_unique(Args&& ...args) {
  return unique_ptr<T>(new T(std::forward<Args>(args)...));
}


template <typename T, typename ...Args>
shared_ptr<T> make_shared(Args&& ...args) {
  auto p = new ControlBlock<T>(1, std::forward<Args>(args)...);
  return p;
}

int main(){
  auto p = std::make_unique<int>(5); //copy elision called, no copy-ctor here
```

**Пример использования:**

```
// Пример использования
class Item
{
 public:
  Item() { std::cout << "Item acquired\n"; }
  ~Item() { std::cout << "Item destroyed\n"; }
};

int main()
{
  // Выделяем Item и передаем его в std::shared_ptr
  auto ptr1 = std::make_shared<Item>();
  {
    auto ptr2 = ptr1; // создаем ptr2 из ptr1, используя семантику копирования

    std::cout << "Killing one shared pointer\n";
  } // ptr2 выходит из области видимости здесь, но ничего больше не происходит

  std::cout << "Killing another shared pointer\n";

  return 0;
} // ptr1 выходит из области видимости здесь, и выделенный Item также уничтожается здесь
```