ЛЕКЦИЯ 02
УМНЫЕ УКАЗАТЕЛИ. ИТЕРАТОРЫ

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

SHARED POINTER

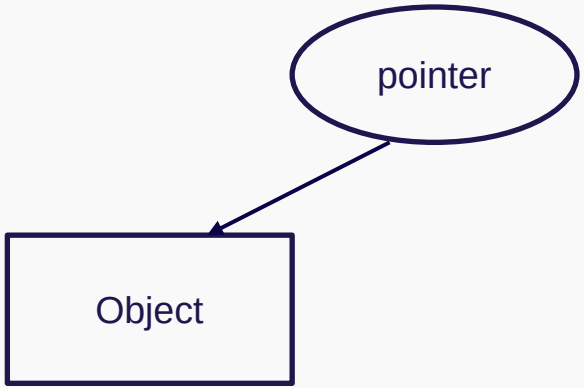
В прошлый раз мы пришли к концепции умного указателя, который обеспечивает уникальность объекта и владеет им, обеспечивая концепцию RAII.

SHARED POINTER

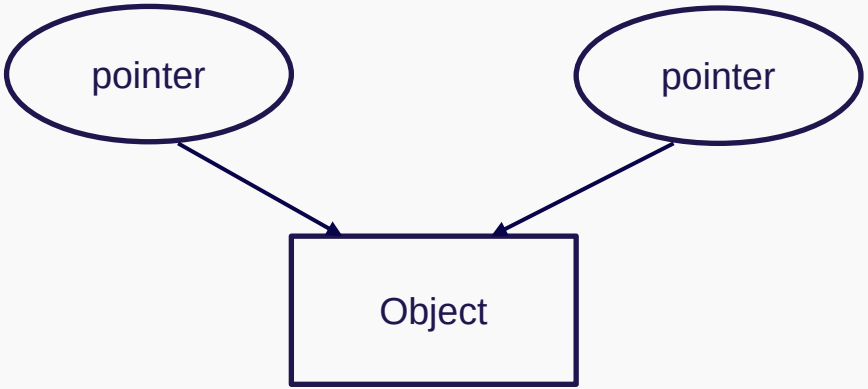
В прошлый раз мы пришли к концепции умного указателя, который обеспечивает уникальность объекта и владеет им, обеспечивая концепцию RAII.

Мы хотим изменить наш указатель так, чтобы он мог «разделять» владение ресурсом. То есть деструктор должен быть вызван только тогда, когда уничтожается последний указатель на ресурс.

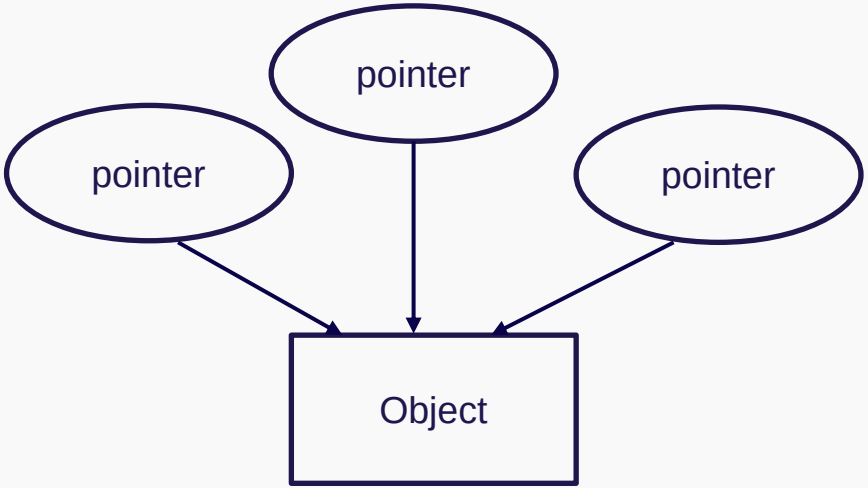
SHARED POINTER



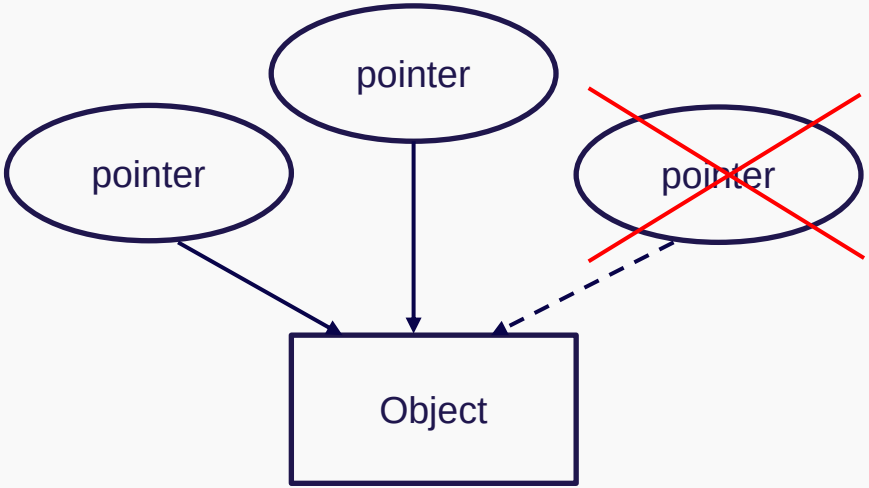
SHARED POINTER



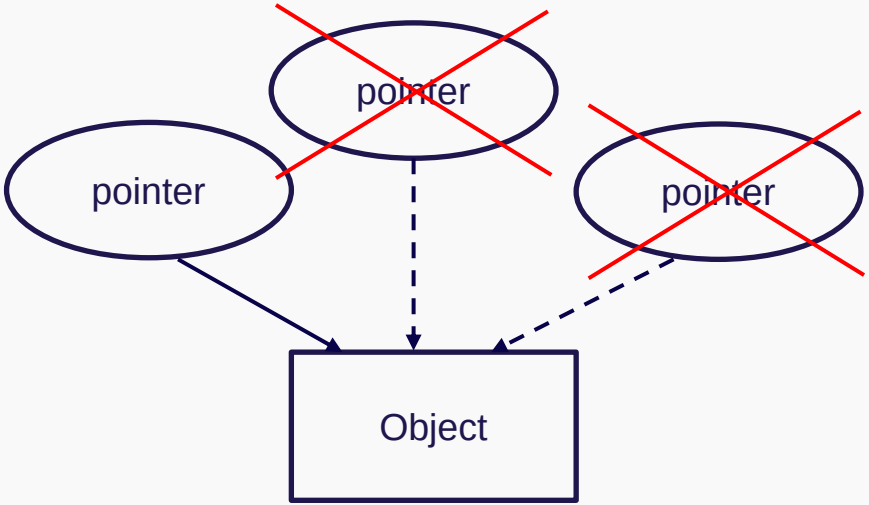
SHARED POINTER



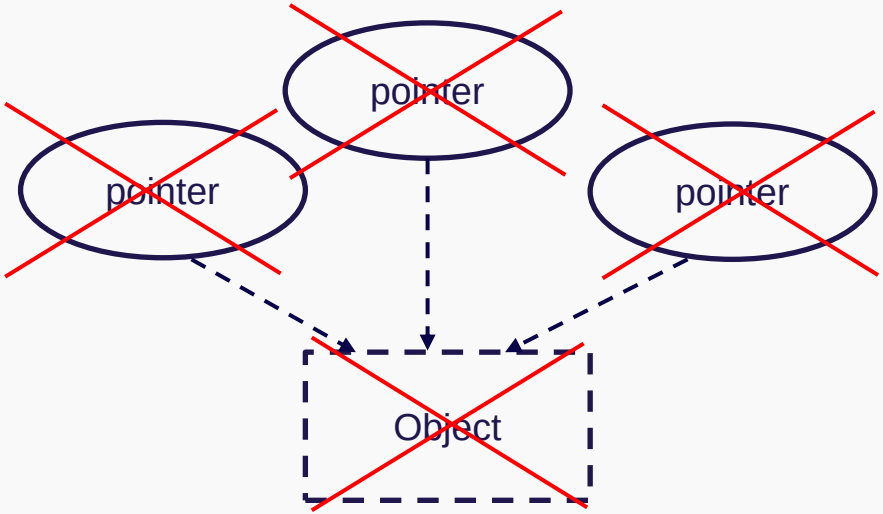
SHARED POINTER



SHARED POINTER



SHARED POINTER



SHARED POINTER

Возьмем за основу наш указатель и доработаем его.

```
template <typename T>
class PRD_pointer final {
private:
    T* p;
public:
    …
};
```

Сперва подумайте, какие конструкторы ему нужны?

SHARED POINTER

Возьмем за основу наш указатель и доработаем его.

```
template <typename T>
class PRD_pointer final {
private:
    T* p;
public:
    PRD_pointer(T* p);
    PRD_pointer(const PRD_pointer& rhs);
    PRD_pointer(PRD_pointer&& rhs);
};
```

SHARED POINTER

Возьмем за основу наш указатель и доработаем его.

```
template <typename T>
class PRD_pointer final {
private:
    T* p;
public:
    PRD_pointer(T* p);
    PRD_pointer(const PRD_pointer& rhs);
    PRD_pointer(PRD_pointer&& rhs);
};
```

Теперь нам как-то надо считать, сколько указателей еще указывают на наш ресурс. Какие предложения?

SHARED POINTER

Возьмем за основу наш указатель и доработаем его.

```
template <typename T>
class PRD_pointer final {
private:
    int counter;
    T* p;
public:
    PRD_pointer(T* p);
    PRD_pointer(const PRD_pointer& rhs);
    PRD_pointer(PRD_pointer&& rhs);
};
```

Завести счетчик может быть хорошей идеей, но…

SHARED POINTER

Что вы напишете в конструкторе копирования?

```
template <typename T>
PRD_pointer::PRD_pointer(const PRD_pointer& rhs) {
    p = rhs.p;
    //а что тут напишете?
};
```

Завести счетчик может быть хорошей идеей, но…

SHARED POINTER

Что вы напишете в конструкторе копирования?

```
template <typename T>
PRD_pointer::PRD_pointer(const PRD_pointer& rhs) {
    p = rhs.p;
    //а что тут напишете?
};
```

Счетчик в качестве поля делает его индивидуальным для каждого отдельного указателя. И проблема контроля счетчиков возникает при появлении хотя бы трех указателей на один ресурс.

SHARED POINTER

Возьмем за основу наш указатель и доработаем его.

```
template <typename T>
class PRD_pointer final {
private:
    static int counter;
    T* p;
public:
    PRD_pointer(T* p);
    PRD_pointer(const PRD_pointer& rhs);
    PRD_pointer(PRD_pointer&& rhs);
};
```

Завести статический счетчик может быть отличной идеей, но…

SHARED POINTER

Возьмем за основу наш указатель и доработаем его.

```
template <typename T>
class PRD_pointer final {
private:
    int* counter;
    T* p;
public:
    PRD_pointer(T* p);
    PRD_pointer(const PRD_pointer& rhs);
    PRD_pointer(PRD_pointer&& rhs);
};
```

Завести указатель на счетчик будет решением нашей проблемы

SHARED POINTER

Тогда что вы напишете в конструкторе копирования?

```
template <typename T>
PRD_pointer::PRD_pointer(const PRD_pointer& rhs) {
    p = rhs.p;
    //а что тут напишете?
};
```

SHARED POINTER

Тогда что вы напишете в конструкторе копирования?

```
template <typename T>
PRD_pointer::PRD_pointer(const PRD_pointer& rhs) {
    p = rhs.p;
    (*counter)++;
};
```

SHARED POINTER

Тогда что вы напишете в конструкторе копирования?

```
template <typename T>
PRD_pointer::PRD_pointer(const PRD_pointer& rhs) {
    p = rhs.p;
    (*counter)++;
};

template <typename T>
PRD_pointer::PRD_pointer(T*  p)  :  p(p),  counter(new
int(1) {};
```

SHARED POINTER

Осталось написать деструктор

```
template <typename T>
PRD_pointer::~PRD_pointer() {
    //что напишете?
};
```

SHARED POINTER

Осталось написать деструктор

```
template <typename T>
PRD_pointer::~PRD_pointer() {
     (*counter)--;
     if (*counter == 0) {
          delete p;
          delete counter;
};
```

SHARED POINTER

Осталось написать деструктор

```
template <typename T>
PRD_pointer::~PRD_pointer() {
     (*counter)--;
     if (*counter == 0) {
          delete p;
          delete counter;
};
```

Аналогично допишем операторы присваивания (копирующий и перемещающий)

SHARED POINTER

Осталось написать деструктор

```
template <typename T>
PRD_pointer::~PRD_pointer() {
     (*counter)--;
     if (*counter == 0) {
          delete p;
          delete counter;
};
```

Аналогично допишем операторы присваивания (копирующий и перемещающий).

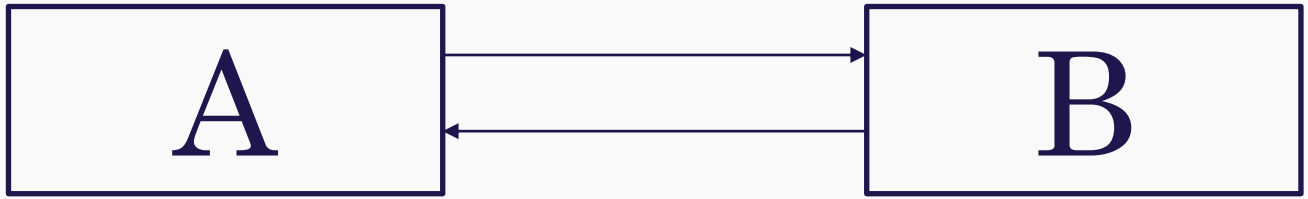
Насколько хорош наш указатель?

ПРОВЕРИМ НА ПРОЧНОСТЬ

```
struct S {
      int x;
}
{
PRD_pointer p1(new S(1));
PRD_pointer p2 = p1;
PRD_pointer p3(new S(3));
p3 = p1;
}
```

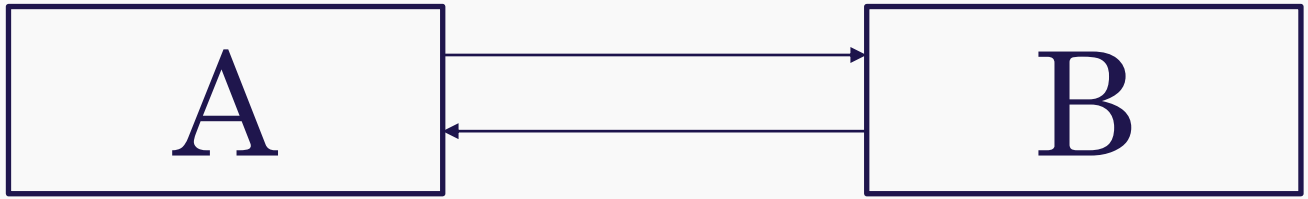
Всё ли тут хорошо?

ПРОБЛЕМА



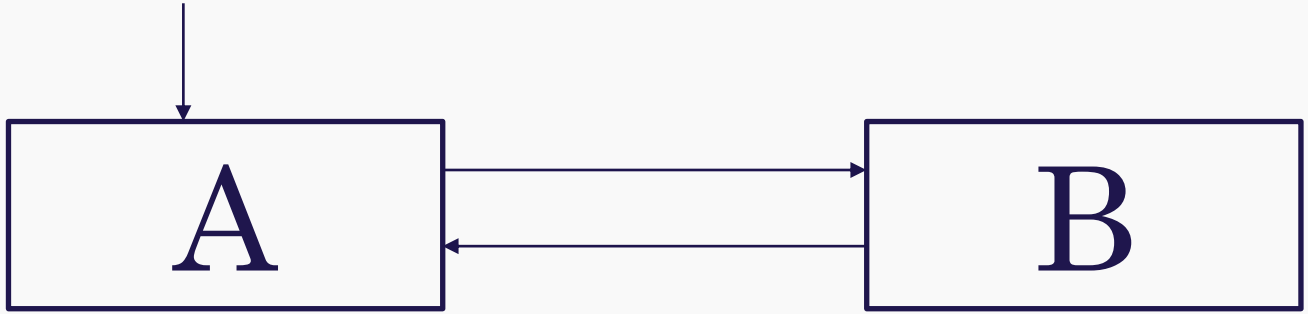
ПРОБЛЕМА

Как уничтожить эти объекты?



ПРОБЛЕМА

Как уничтожить эти объекты?



РЕШЕНИЕ

Ослабить одну из связей между объектами

WEAK POINTER

Слабый указатель – «наблюдатель» за ресурсом и им не владеет. Но может на него «смотреть».

WEAK POINTER

Слабый указатель – «наблюдатель» за ресурсом и им не владеет. Но может на него «смотреть».

- Слабый указатель может быть создан только от shared_ptr.

WEAK POINTER

Слабый указатель – «наблюдатель» за ресурсом и им не владеет. Но может на него «смотреть».

- Слабый указатель может быть создан только от shared_ptr.
- Слабый указатель не может уничтожить объект, но может освободить захваченную память
- Слабый указатель не может обратиться к объекту, получить его значение и, тем более, его изменить.

WEAK POINTER

Введение нового объекта в концепцию требует значительной переработки известного нам shared_ptr и внесения кардинальных изменений. Что нужно изменить?

WEAK POINTER

Введение нового объекта в концепцию требует значительной переработки известного нам shared_ptr и внесения кардинальных изменений. Что нужно изменить?

- Для подсчета weak_ptr и shared_ptr необходимо создать прокси-класс, который в себе объединит эти счетчики и сам ресурс

WEAK POINTER

Введение нового объекта в концепцию требует значительной переработки известного нам shared_ptr и внесения кардинальных изменений. Что нужно изменить?

- Для подсчета weak_ptr и shared_ptr необходимо создать прокси-класс, который в себе объединит эти счетчики и сам ресурс
- Переписать с учетом нового класса конструкторы и операторы присваивания.

WEAK POINTER

Введение нового объекта в концепцию требует значительной переработки известного нам shared_ptr и внесения кардинальных изменений. Что нужно изменить?

- Для подсчета weak_ptr и shared_ptr необходимо создать прокси-класс, который в себе объединит эти счетчики и сам ресурс
- Переписать с учетом нового класса конструкторы и операторы присваивания.
- Переписать деструктор

WEAK POINTER

Введение нового объекта в концепцию требует значительной переработки известного нам shared_ptr и внесения кардинальных изменений. Что нужно изменить?

- Для подсчета weak_ptr и shared_ptr необходимо создать прокси-класс, который в себе объединит эти счетчики и сам ресурс
- Переписать с учетом нового класса конструкторы и операторы присваивания.
- Переписать деструктор

ВАЖНО! Если на ресурс не указывают shared_ptr, но указывают weak_ptr, то объект должен быть уничтожен, но память не должна быть освобождена. Подумайте над реализацией этого.

WEAK POINTER

Введение нового объекта в концепцию требует значительной переработки известного нам shared_ptr и внесения кардинальных изменений. Что нужно изменить?

- Для подсчета weak_ptr и shared_ptr необходимо создать прокси-класс, который в себе объединит эти счетчики и сам ресурс
- Переписать с учетом нового класса конструкторы и операторы присваивания.
- Переписать деструктор

ВАЖНО! Если на ресурс не указывают shared_ptr, но указывают weak_ptr, то объект должен быть уничтожен, но память не должна быть освобождена. Подумайте над реализацией этого.

ИТЕРАТОРЫ

Итератор – это объект, который ведет себя, как указатель и умеет перемещаться по элементам контейнера.

ИТЕРАТОРЫ

Итератор – это объект, который ведет себя, как указатель и умеет перемещаться по элементам контейнера. Но это совсем верно. Позже мы уточним определение.

ТРЕБОВАНИЯ К RANGE-BASED ОБХОДУ

Объект, возвращаемый `begin()` должен поддерживать:

- инкремент
- разыменование
- сравнение на неравенство

ТРЕБОВАНИЯ К RANGE-BASED ОБХОДУ

Объект, возвращаемый `begin()` должен поддерживать:

- инкремент
- разыменование
- сравнение на неравенство

```
for (; __begin != __end; ++__begin) {
      range_declaration = *__begin;
```

ТРЕБОВАНИЯ К RANGE-BASED ОБХОДУ

Объект, возвращаемый `begin()` должен поддерживать:

- инкремент
- разыменование
- сравнение на неравенство

```
for (; __begin != __end; ++__begin) {
      range_declaration = *__begin;
```

Эти требования входят в статический интерфейс прямого итератора

ТРЕБОВАНИЯ К RANGE-BASED ОБХОДУ

Объект, возвращаемый `begin()` должен поддерживать:

- инкремент
- разыменование
- сравнение на неравенство

```
for (; __begin != __end; ++__begin) {
      range_declaration = *__begin;
```

Эти требования входят в статический интерфейс прямого итератора

Можно заметить, что всем этим требованиям отвечают обычные указатели

ТРЕБОВАНИЯ К RANGE-BASED ОБХОДУ

Объект, возвращаемый `begin()` должен поддерживать:

- инкремент
- разыменование
- сравнение на неравенство

```
for (; __begin != __end; ++__begin) {
      range_declaration = *__begin;
```

Эти требования входят в статический интерфейс прямого итератора

Можно заметить, что всем этим требованиям отвечают обычные указатели

Очень важно: итератор это не какой-то класс и не наследник какого-то класса, это что угодно с таким интерфейсом

СВОЙСТВА УКАЗАТЕЛЕЙ

- Создание по умолчанию, копирование, копирующее присваивание
- Разыменовывание как rvalue и доступ к полям по разыменовыванию
- Разыменовывание как lvalue и присваиванию значения элементу под ним
- Инкремент и постинкремент за O(1)
- Сравнимость на равенство и неравенство за O(1)
- Декримент и постдекремент за O(1)
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за O(1)
- Многократный проход по одной и той же последовательности

OUTPUT ИТЕРАТОРЫ

- Создание по умолчанию, копирование, копирующее присваивание
- Разыменовывание как rvalue и доступ к полям по разыменовыванию
- Разыменовывание как lvalue и присваиванию значения элементу под ним
- Инкремент и постинкремент за O(1)
- Сравнимость на равенство и неравенство за O(1)
- Декримент и постдекремент за O(1)
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за O(1)
- Многократный проход по одной и той же последовательности

INPUT ИТЕРАТОРЫ

- Создание по умолчанию, копирование, копирующее присваивание
- Разыменовывание как rvalue и доступ к полям по разыменовыванию
- Разыменовывание как lvalue и присваиванию значения элементу под ним
- Инкремент и постинкремент за O(1)
- Сравнимость на равенство и неравенство за O(1)
- Декримент и постдекремент за O(1)
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за O(1)
- Многократный проход по одной и той же последовательности

FORWARD ИТЕРАТОРЫ

- Создание по умолчанию, копирование, копирующее присваивание
- Разыменовывание как rvalue и доступ к полям по разыменовыванию
- Разыменовывание как lvalue и присваиванию значения элементу под ним
- Инкремент и постинкремент за O(1)
- Сравнимость на равенство и неравенство за O(1)
- Декримент и постдекремент за O(1)
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за O(1)
- Многократный проход по одной и той же последовательности

BIDIRECTIONAL ИТЕРАТОРЫ

- Создание по умолчанию, копирование, копирующее присваивание
- Разыменовывание как rvalue и доступ к полям по разыменовыванию
- Разыменовывание как lvalue и присваиванию значения элементу под ним
- Инкремент и постинкремент за O(1)
- Сравнимость на равенство и неравенство за O(1)
- Декримент и постдекремент за O(1)
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за O(1)
- Многократный проход по одной и той же последовательности

RANDOM ACCESS ИТЕРАТОРЫ

- Создание по умолчанию, копирование, копирующее присваивание
- Разыменовывание как rvalue и доступ к полям по разыменовыванию
- Разыменовывание как lvalue и присваиванию значения элементу под ним
- Инкремент и постинкремент за O(1)
- Сравнимость на равенство и неравенство за O(1)
- Декримент и постдекремент за O(1)
- Индексирование квадратными скобками, сложение с целыми, сравнение на больше и меньше за O(1)
- Многократный проход по одной и той же последовательности

ОПРЕДЕЛЕНИИ КАТЕГОРИИ ИТЕРАТОРОВ

- Используется класс характеристик

```
typename iterator_traits<Iter>::iterator::category
```

- Возможное значения
- input_iterator_tag
- output_iterator_tag
- forward_iterator_tag: public input_iterator_tag
- bidirectional_iterator_tag: public forward_iterator_tag
- random_access_iterator_tag: public bidirectional_iterator_tag

ПИШЕМ СВОЙ ИТЕРАТОР

В итераторе нам нужно определить пять фундаментальных подтипов:

- `iterator_category` – категория нашего итератора
- `difference_type` – тип для хранения разности итераторов
- `value_type` – тип значений, по которым итерируемся
- `reference` – тип ссылки на значение, по которым мы итерируемся
- `pointer` – тип указателя на значения, по которым мы итерируемся

ВРЕМЯ ДЛЯ ПРАКТИКИ

Вместе с преподавателем напишем простой итератор для линейного контейнера

ДОМАШНЕЕ ЗАДАНИЕ

Написать свой shared_ptr и weak_ptr. Помимо оговоренного на лекции определить следующие методы:

1. `operator bool` – приводит пустой указатель к `false`, а не пустой – к `true` (только у `shared_ptr`)
2. `get` – возвращает указатель на управляемый объект (только у `shared_ptr`)
3. `swap` – обмен управляемыми объектами между указателями

Это минимум на оценку УДОВЛЕТВОРИТЕЛЬНО

ДОМАШНЕЕ ЗАДАНИЕ

Написать свой shared_ptr и weak_ptr. Помимо оговоренного на лекции определить следующие методы :

4. `reset` – заменяет объект, которым владеет
5. `use_count` – возвращает количество shared_ptr, владеющих ресурсом
6. операторы сравнения (только у `shared_ptr`)
7. `unique` (возвращает `true`, если `shared_ptr` единственный владеет ресурсом и `false` – если не единственный, только для `shared_ptr`)

Это минимум на оценку ХОРОШО

ДОМАШНЕЕ ЗАДАНИЕ

Написать свой shared_ptr и weak_ptr. Помимо оговоренного на лекции определить следующие методы :

8. `expired` – метод возвращает true, если объект уничтожен и `false`, если нет (только для `weak_ptr`)
9. `lock` – возвращает `shared_ptr` на объект (только для `weak_ptr`)

Это минимум на оценку ОТЛИЧНО

ВРЕМЯ ДЛЯ ВАШИХ ВОПРОСОВ

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. Скотт Мейерс, Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2017. – 300 с.
3. Скотт Мейерс, Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2016. – 298 с.