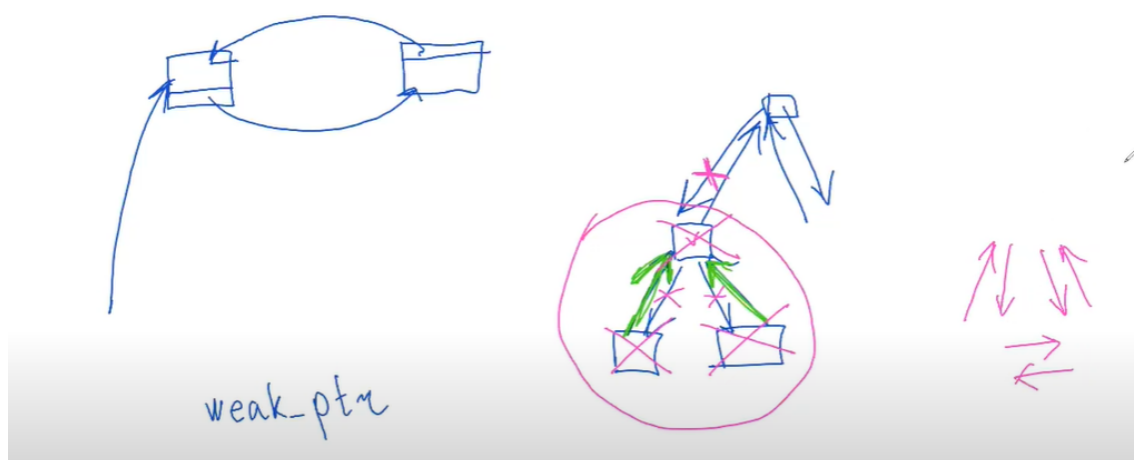


5.16 Проблема циклических shared_ptr. Класс weak_ptr как решение этой проблемы. Реализация основных методов weak_ptr: конструкторы, деструктор, операторы присваивания, методы expired() и lock(). Модификация реализации shared_ptr для поддержки weak_ptr.

Еще одна проблема - возможная циклическая зависимость (другие языки со сборкой мусора тоже ею страдают).

Допустим мы реализуем двоичное дерево и в какой-то момент хотим удалить поддереву этого дерева. Логично предположить, что все указатели должны удалиться, однако из-за того что внутри поддерева сын указывает на родителя, а родитель на сына, будут оставаться объекты указывающие на вершинку - следовательно, вершинка не удалится, и так со всеми вершинками в удаляемом дереве. Т.о. произошла утечка памяти. Решение проблемы - **weak_ptr**



Это сущность, которая как и shared_ptr хранит указатель на некий объект, однако он им не владеет (просто смотрит) .

Его можно спросить две вещи:

- 1 Не умер ли еще объект на который ты указываешь
- 2 Создай новый shared_ptr на объект, на который ты смотришь

Сам weak_ptr разыменовывать нельзя.

Если объект убит, а мы попросили shared_ptr то это будет UB/RE.

Как weak_ptr решает проблему? Можно сделать указатели на родителей слабыми(зеленые стрелки на рисунке)

Тогда если мы обнуляем указатель идущий от родителя к вершине (стрелка, зачеркнутая жирным розовым крестиком), вершина понимает что на нее указывает 0 shared_ptr и 2 weak_ptr. Но weak_ptr-ы не считаются и объект уничтожается. Следом умирают указатели, идущие от вершины к детям (указатели, идущие в противоположную сторону от зеленых), и уничтожаются самые нижние вершины

Правило: если есть двусторонняя циклическая зависимость, то хотя бы один указатель (связь) в каждой вершине в ней должен быть weak_ptr-ом. Тогда если хотя бы одна связь из цикла будет разрушена, будет разрушен и весь цикл, причем в правильном порядке

Реализация

```

1  template<typename T>
2  class weak_ptr{
3  private:
4      ControlBlock<T>* inner_block = nullptr;
5  public:
6      weak_ptr(const shared_ptr<T>& p): inner_block(p.inner_block){}; //
constructor
7
8      weak_ptr(const weak_ptr<T>& other):inner_block(other.inner_block){};
//copy
constructor
9
10     weak_ptr& operator = (const weak_ptr<T>& other){
11         if (this != std::addressof(other)) {
12             inner_block = other.inner_block;
13         }
14         return *this;
15     }
16
17     weak_ptr(weak_ptr<T>&& other):
18     inner_block(std::move(other.inner_block)){};
19
20     weak_ptr& operator = (weak_ptr<T>&& other){
21         if (this != std::addressof(other)) {
22             inner_block = std::move(other.inner_block);
23         }
24         return *this;
25     }
26
27     bool expired() const {
28         return inner_block->shared_cnt == 0;
29     }
30
31     shared_ptr<T> lock() const {
32         if (expired()) {
33             throw std::bad_weak_ptr();
34         }
35         return shared_ptr<T>(inner_block);
36     }
37
38
39     ~weak_ptr() {
40         if(!inner_block) return;
41         --inner_block->weak_cnt;
42         if (inner_block->weak_cnt == 0 and inner_block->shared_cnt == 0)
{
43             delete inner_block;
44         }
45     }
46
47 };

```

shared_ptr with weak_ptr

```
template <typename U>
struct ControlBlock {
    T object;
    size_t count;

    template <typename... Args>
    ControlBlock(size_t count, Args&&... args);
};
```

```
1  template<typename T>
2      class shared_ptr{
3  private:
4      T* ptr;
5      ControlBlock<T>* inner_block = nullptr;
6
7      template<typename U>
8      friend class weak_ptr;
9
10     template <typename U, typename ...Args>
11     friend shared_ptr<U> make_shared(Args&& ...args);
12
13     shared_ptr(ControlBlock<T>* cb) : inner_block(cb), ptr(cb->val){};
14 public:
15     explicit shared_ptr(T* pointer){
16         inner_block = new ControlBlock<T>{1, pointer};
17         ptr = pointer;
18         if constexpr (std::is_base_of_v<enable_shared_from_this<T>, T>) {
19             ptr->wptr = *this;
20         }
21     }
22
23     shared_ptr(const shared_ptr<T>& other) {
24         ptr = other.ptr;
25         ++inner_block->shared_cnt;
26     }
27
28     shared_ptr(shared_ptr<T>&& other) {
29         inner_block = std::move(other.inner_block);
30         other.ptr = nullptr;
31     }
32
33     shared_ptr<T>& operator=(const shared_ptr<T>& other) & {
34         shared_ptr<T> copy(other);
35         std::swap(copy, *this);
36         return *this;
37     }
38     shared_ptr<T>& operator=(shared_ptr<T>&& other) & noexcept {
39         inner_block = std::move(other.inner_block);
40         other.ptr = nullptr;
41         return *this;
42     }
43
44     ~shared_ptr() {
45         if (!inner_block) return; // no control block
46         --inner_block->shared_cnt;
47         if (inner_block->shared_cnt == 0) {
48             delete ptr; // obj deleted, but not control block
49             if (inner_block->weak_cnt == 0 ) { //if false, then some
weak_ptr are looking at obj
50                 delete inner_block;
51             }
52         }
53     }
54
55     };
```

5.17 Класс `enable_shared_from_this`, описание проблемы, которую он решает. Реализация этого класса.

Как получить из тела метода структуры/класса умный указатель на самого себя? Если возникает такая ситуация - т.е мы хотим чтобы класс поддерживал возможность возвращать `shared_ptr` на себя, то мы не в полях заводим `weak_ptr`, а обращаемся к некоторой библиотечной функции, которая генерит `shared_ptr` (который начинает делить владение объектом с уже созданными указателями) и возвращает его нам.

CRTP = Curiously Recursive Template Pattern

См. часть реализации в `shared_ptr` (конструктор)

```
1  template<typename T>
2  class enable_shared_from_this{
3  private:
4      weak_ptr<T> ptr = nullptr;
5  protected:
6      shared_ptr<T> shared_from_this() const {
7          return ptr.lock();
8      }
9  };
10
11  struct S : public enable_shared_from_this<S> {
12      shared_ptr<S> getPointer() const {
13          return shared_from_this();
14      }
15  };
```

Определение структуры `S` не требуется для работы `enable_shared_from_this<S>`.