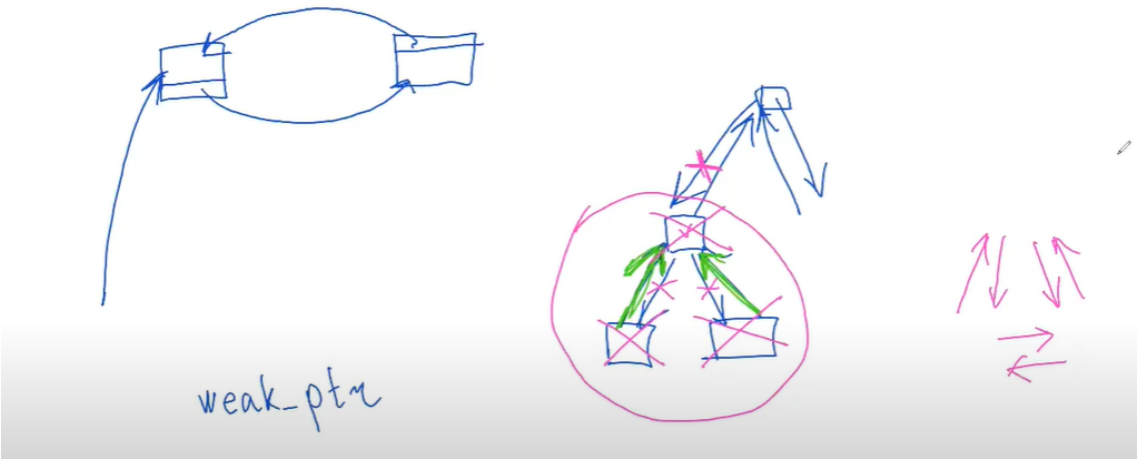
5.16 Проблема циклических shared_ptr. Класс weak_ptr как решение этой проблемы. Реализация основных методов weak_ptr: конструкторы, деструктор, операторы присваивания, методы expired() и lock(). Модификация реализации shared_ptr для поддержки weak_ptr.

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 поддерево 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 двустороняя циклическая зависмость, 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

```
 template<typename T>
 class weak_ptr{
 private:
         ControlBlock<T>* inner_block = nullptr;
 public:
         weak_ptr(const shared_ptr<T>& p): inner_block(p.inner_block){}; //
constructor

         weak_ptr(const weak_ptr<T>& other):inner_block(other.inner_block){};
//copy
constructor

         weak_ptr& operator = (const weak_ptr<T>& other){
             if (this != std::addressof(other)) {
                 inner_block = other.inner_block;
             }
             return *this;
         }

         weak_ptr(weak_ptr<T>&& other):
         inner_block(std::move(other.inner_block)){};

         weak_ptr& operator = (weak_ptr<T>&& other){
             if (this != std::addressof(other)) {
                 inner_block = std::move(other.inner_block);
             }
             return *this;
         }

         bool expired() const {
             return inner_block->shared_cnt == 0;
         }

         shared_ptr<T> lock() const {
             if (expired()) {
                 throw std::bad_weak_ptr();
             }
             return shared_ptr<T>(inner_block);

         }

         ~weak_ptr() {
             if(!inner_block) return;
             --inner_block->weak_cnt;
             if (inner_block->weak_cnt == 0 and inner_block->shared_cnt == 0)
{
                 delete inner_block;
             }
         }

 };
```

shared_ptr with weak_ptr

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

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

```
template<typename T>
 class shared_ptr{
 private:
     T* ptr;
     ControlBlock<T>* inner_block = nullptr;

     template<typename  U>
     friend class weak_ptr;

     template <typename U, typename ...Args>
     friend shared_ptr<U> make_shared(Args&& ...args);

     shared_ptr(ControlBlock<T>* cb) : inner_block(cb), ptr(cb->val){};
 public:
     explicit shared_ptr(T* pointer){
         inner_block = new ControlBlock<T>{1, pointer};
         ptr = pointer;
         if constexpr (std::is_base_of_v<enable_shared_from_this<T>, T>) {
             ptr->wptr - *this;
         }
     }

     shared_ptr(const shared_ptr<T>& other) {
         ptr = other.ptr;
         ++inner_block->shared_cnt;
     }

     shared_ptr(shared_ptr<T>&& other) {
         inner_block = std::move(other.inner_block);
         other.ptr = nullptr;
     }

     shared_ptr<T>& operator=(const shared_ptr<T>& other) & {
         shared_ptr<T> copy(other);
         std::swap(copy, *this);
         return *this;
     }
     shared_ptr<T>& operator=(shared_ptr<T>&& other) & noexcept {
         inner_block = std::move(other.inner_block);
         other.ptr = nullptr;
         return *this;
     }

     ~shared_ptr() {
         if (!inner_block) return; // no control block
         --inner_block->shared_cnt;
         if (inner_block->shared_cnt == 0) {
             delete ptr; // obj deleted, but not control block
             if (inner_block->weak_cnt == 0 ) {  //if false, then some weak_ptr are looking at obj
                 delete inner_block;
             }
         }
     }

 };
```

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

Как получить из тела метода структруры/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 бибилиотечной функции, которая генерит shared_ptr (который начинает делить владение объектом с уже созданными указателями) и возвращает его нам.

CRTP = Curiously Recursive Template Pattern

```
template<typename  T>
class enable_shared_from_this{
private:
    weak_ptr<T> ptr = nullptr;
protected:
    shared_ptr<T> shared_from_this() const {
        return ptr.lock();
    }
};

struct S : public enable_shared_from_this<S> {
    shared_ptr<S> getPointer() const {
        return shared_from_this();
    }
};
```

Определение структры S не требуется для работы enable_shared_from_this<S>.

Замечание: попытка вызвать shared_from_this в случае, когда объектом не владеет ни один shared_ptr является UB(с C++17 выкидывается исключение std::bad_weak_ptr).