movable_ptr<T>

ASSIGNMENT #1

ADVANCED C++

Motivation

In managed runtimes (JVM, .NET) are heap objects often moved in memory

- Useful in garbage collection
- Transparent for the caller – all the references to an object are updated automagically

Let’s try the same thing C++ way!

- Standard way to move objects: `std::move`
- Encapsulation of pointer behaviour - smart pointers: `unique_ptr<T>`, `shared_ptr<T>`

`movable_ptr<T>`

- Keeps the track of an object even when it is moved
- Invalidated (`nullptr`) when the object is destructed
- Does **NOT** handle allocation/deallocation

Sample: Ideal

```
struct A {
    int val;
    A (int v) : val(v) {}
};

void fn() {
    A a1(42);
    movable_ptr<A> ptr = get_movable(a1);

    A a2 = std::move(a1);
    a1.val = 666;

    assert(ptr->val == 42);
}
```

PROBLEM:

How to force the custom object a1 to update ptr?
(as transparently as possible)

Sample: enable_movable_ptr<T>

```
struct A : public enable_movable_ptr<A> {
    int val;
    A (int v) : val(v) {}
};

void fn() {
    A a1(42);
    movable_ptr<A> ptr = get_movable(a1);

    A a2 = std::move(a1);
    a1.val = 666;

    assert(ptr->val == 42);
}
```

Encapsulates the pointer updates

Analogy to
std::enable_shared_from_this<T>
(but with completely different semantics)

Challenges #1

The tracked object must contain additional state (knowledge of all the movable pointers pointing to it) and behaviour (it must notify them all about the movement or deletion)

→ Additional class `enable_movable_ptr<T>`

- Usage: `class MyMovable : public enable_movable_ptr<MyMovable>`
- `MyMovable` can now be used by `movable_ptr<MyMovable>`
- `static_cast` possible from `enable_movable_ptr<MyMovable>*` to `MyMovable*`

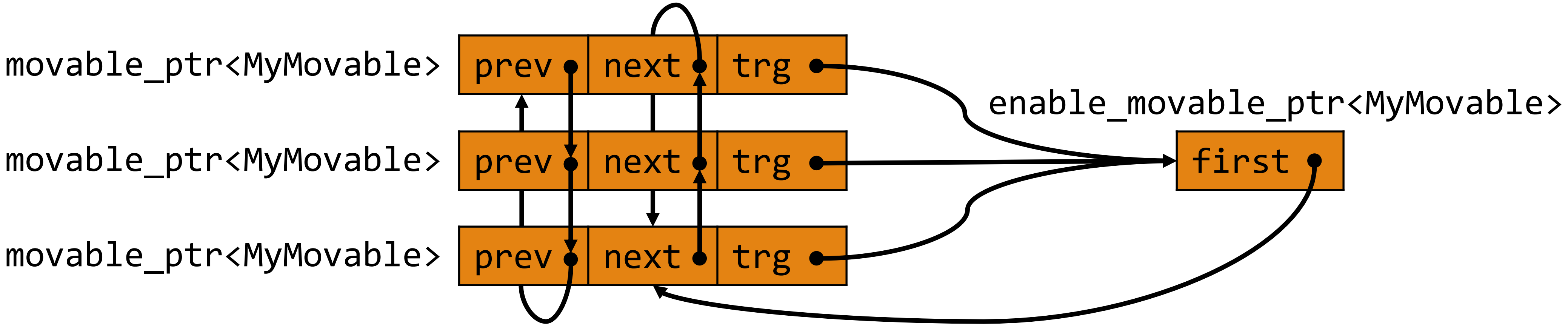
What if a movable_ptr<T> itself is moved or destructed?

→ Must propagate this information back to the tracked object

Challenges #2

We don’t want any additional allocations of helper objects (containers etc.) which would itself stay in the same place in the heap – overhead, killing the idea of moving, too high-level

→ Better to create an intrinsic data structure based on a linked list, e.g.:



Each its element must update the others in the case of its movement or deletion

Requirements #1

`enable_movable_ptr<T>`

- Must keep the track of all the instances of `movable_ptr<T>` pointing at it
- Parameterless constructor (to be easily usable as an ancestor)
- Destructor: reset all pointers
- Copy constructor: the same as parameterless constructor (pointers are not interested in copies)
- Move constructor: redirect all the pointers to the new location, remove the pointers from the old
- Copy/move assignment: the same as the respective constructor, but destroy the target (apart from the self-assignment):

```
A x, y;
movable_ptr<A> px = &x;
movable_ptr<A> py = &y;
x = y;
assert(px.get() == nullptr);
```

Requirements #2

`movable_ptr<T>`

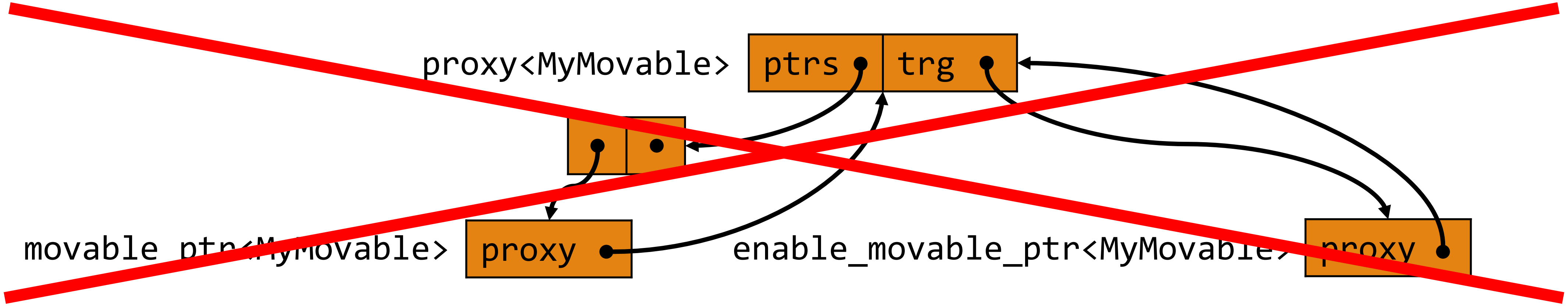
- Expect that T inherits from `enable_movable_ptr<T>`
- Parameterless constructor: empty pointer (its `get()` returns `nullptr`)
- Constructor from `T*` (must add itself to the pointers tracked by the target)
- Destructor, copy/move constructor, copy/move operator = (to notify other pointers and/or the target)
- Standard smart pointer operations: `reset()`, `reset(T*)`, `get()`, operators `*`, `->` , `==`, `!=`, `bool`
  - See the slides in the end for operator overloading
  - `==` and `!=` compare the pointers, not the objects themselves

`movable_ptr<T> get_movable(enable_movable_ptr<T>&)`

- Standalone utility function
- `auto px = get_movable(x);`
- `std::make_pair(get_movable(x), get_movable(y))`

Requirements #3

DO NOT allocate additional memory, such as by using a centralized proxy which never moves:



Although elegant, the resulting overhead is too high (memory allocation, each dereference takes two steps etc.), it also does not fit in the idea of movability

Requirements #4

DO NOT destroy the target object when there are no movable pointers referencing it

- We are **NOT** implementing `std::shared_ptr<T>`

The code of all tests (available in ReCodEx) must be compilable by both GCC and Visual Studio

- No warnings in your code during compilation
- MSVC outside Windows: https://godbolt.org (with option `/W3` for warnings)

Code correctness and readability

- Concise code
- `const`, `noexcept` etc.
- All the mentioned good practices – consistent formatting, identifier names, …

Evaluation

Upload `movable_ptr.hpp` to ReCodEx

- All tests use custom `main.cpp` file with `main` (along with `CompactableGraph.hpp` and `.cpp`)
- All tests are publicly available, as well as their output (see the task in ReCodEx)
- Resulting points form the basis for the manual evaluation

Manual evaluation

- Will be performed after the deadline
- Will check mainly the requirements not checked by ReCodEx (e.g. readability, no allocation, no compiler warnings, …)
- Can modify the points from ReCodEx, usually by deducing some of them

Hints

Check the first two tests for the precise semantics

Mind the Rule of Five
- Most of the semantics will be contained in constructors, destructors and asignments anyway
- Beware of the self-assignment

Don‘t overcomplicate the solution
- Shorter code is usually more readable
- Think out the operations properly before implementing them
- The original solution has less than 250 lines

Different behaviour in VS and ReCodEx is often caused by omitting to initialize memory
- Debugging in GCC on Linux using Visual Studio: WSL, remote (or local virtual) Linux

Encapsulate the behaviour to private fields and methods
- `friend` construct might be useful

friend

A friend declaration (anywhere inside a class) allows access to private/protected members from another class or a function outside of the class. There are (at least) the following forms:

friend class X; // befriending a non-template class X

template<typename U> **friend** class Y; // befriending all instantiations of a template class Y

friend class Y<T>; // befriending a specific instantiation of a template class Y

friend T1 f(T2,T3); // befriending a non-template (non-member) function

template<typename V> **friend** T g(V); // befriending a template function g

It is recommended that the befriended class/function be previously declared (although it is not mandatory). When befriending a template, the template arguments in the friend declaration must match the template declaration.

Friendship is not symmetric nor transitive nor inherited. However, it applies to nested classes.

Operator overloading

```
template< typename T> class movable_ptr {
  T& operator*() const;
  T* operator->() const;
  bool operator!() const;
  bool operator==(const movable_ptr<T>&) const;
  bool operator!=(const movable_ptr<T>&) const;
};
```

When implementing a kind of smart pointer, several overloaded operators are required to mimic the behaviour of plain pointers.

The tricky part is the operator-> which shall (by definition) return another “pointer” – the compiler repeats calling operator-> until a plain pointer appears on which the built-in behaviour is applied. In other words, an user-defined operator-> never deals with the member name being accessed.

Note that it is implementer’s responsibility to make operators == and != consistent.

Note that this assignment does **not** require implementing read-only smart-pointers, i.e. pointers whose operator* return const T&.