Department I - C Plus Plus

Modern and Lucid C++ for Professional Programmers

Week 12 – Heap Memory Management

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 04.12.2018
HS2018

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

Recap Week 10

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

- **Define class templates completely in header files**
- **Member functions of class templates**
  - Either in class template directly
  - Or as inline function templates in the same header file
- **`static` member variables of a template class can be defined in header without violating ODR, even if included in several compilation units**
  - Since C++17 they can even be declared inside the class template, this requires the `inline` keyword

```
template <typename T>
struct staticmember {
  inline static int dummy{sizeof(T)};
};
```

`typename` for Dependent Names

```
using size_type = typename SackType::size_type;
```

- **Within the template definition you might use names that are directly or indirectly depending on the template parameter**
  - E.g. everything using `SackType::`
- **But you have to tell the compiler if one is a type**
  - In contrast to a variable or function name
- **When the `typename` keyword is required you should extract the type into a type alias**
- **Old spelling in `typedef`**

```
typedef typename SackType::size_type size_type;
```

- **Rule: Always use `this->` or the class `name::` to refer to inherited members in a template class**

```
this->bar
```

```
gotchas::bar
```

- **If the name could be a dependent name the compiler will not look for it when compiling the template definition**
- **Checks might only be made for dependent names at template usage (=template instantiation)**
  - That is sometimes the reason for lengthy error messages from template usages

- **How can we adapt a standard container by adding invariants or by extending their functionality?**
  - `SafeVector ->` no undetected out-of-bounds access
  - `IndexableSet ->` provide operator[]
  - `SortedVector ->` guarantee sorted order of elements
- **Template class inheriting from template base class**
  - And inherit ctors of standard container
  - Caution: no safe conversion to base class, no polymorphism

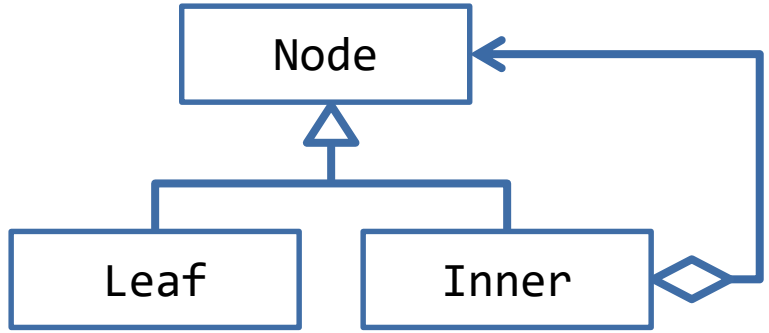
Heap Memory Management

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

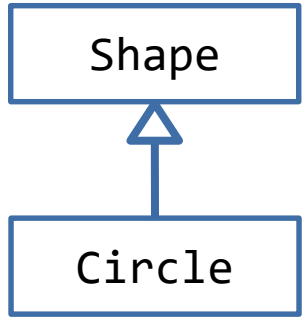
- **Stack memory is scarce**

- **It might be needed for creating object structures**
  - First think about library classes for your intended structure
  - Look into the Boost library collection if the STL is insufficient



- **For polymorphic factory functions to class hierarchies**
  - If you return a base-class "pointer" to one of its subclasses from functions

```
std::unique_ptr<Shape> circle = make_circle(x, y, r);
```



- **Always rely on library classes for managing it (if possible)**
- **Resource Acquisition Is Initialization (RAII) Idiom**
  - Allocation in the constructor
  - Deallocation in the destructor
  - Use RAII wrapper as value in local scope
  - Destructor will be called when the scoped is exited (`}`, `return` or exception)

```
struct RaiiWrapper {
  RaiiWrapper() {
    //Allocate Resource
  }
  ~RaiWrapper() {
    //Deallocate Resource
  }
};
```

- **Doooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooon't do that!**
- **C++ allows allocating objects on the heap directly**

```
auto ptr = new int{};
std::cout << *ptr << '\n';
delete ptr;
```

- **However, if done manually you are responsible for deallocation and risk undefined behavior!**
  - Memory leaks
  - Dangling pointers
  - Double deletes
- **No garbage collection happens, it is your responsibility**

`#include <memory>`


```
std::unique_ptr<X> factory(int i) {
  return std::make_unique<X>(i);
}
```

- **`std::unique_ptr<T>` obtained with `std::make_unique<T>()`**
- **`std::shared_ptr<T>` obtained with `std::make_shared<T>()`**
- **`std::make_unique<T>()` and `std::make_shared<T>()` are factory functions**
- **With these smart pointers you don't have to call `delete ptr;` yourself**
- **Still: Always prefer storing a value locally as value-type variable (stack-based or member)**

- **Used for unshared heap memory**
  - Or for local stuff that must be on the heap
    (rarely needed, e.g. for large instances and limited stack space)
  - Can be returned from a factory function
- **Only a single owner exists**
- **It can wrap to-be-freed pointers from C functions when interfacing legacy code**
- **Not best for class hierarchies**
  - Use `std::shared_ptr<Base>` instead (`unique_ptr` base classes need virtual destructor)

- **A `std::unique_ptr` cannot be copied**

```cpp
#include <iostream>
#include <memory>
#include <utility>

std::unique_ptr<int> create(int i) {
  return std::make_unique<int>(i);
}

int main() {
  std::cout << std::boolalpha;
  auto pi = create(42);
  std::cout << "*pi = " << *pi << '\n';
  std::cout << "pi.valid? " << static_cast<bool>(pi) << '\n';
  auto pj = std::move(pi);
  std::cout << "*pj = " << *pj << '\n';
  std::cout << "pi.valid? " << static_cast<bool>(pi) << '\n';
}
```

Transfer of ownership through return by value

Explicit transfer of ownership from lvalue

`std::unique_ptr<T>` for Naked C Pointers (1/2)

- **Some C functions return pointers that must be deallocated with the function `free(ptr)`**
- **We can use `std::unique_ptr` to ensure that**
- **Example: `__cxa_demangle()` is such a function**

```
std::string demangle(std::string const & name) {
  auto cleanup = [] (char * ptr){
    free(ptr);
  };
  std::unique_ptr<char, decltype(cleanup)> toBeFreed {
    __cxxabiv1::__cxa_demangle(name.c_str(), 0, 0, 0), cleanup};
  std::string result(toBeFreed.get());
  return result;
}
```

- **If there is an exception in the context of that pointer, `free` will be called on the returned pointer**
  - No memory leak

`std::unique_ptr<T>` for Naked C Pointers (2/2)

- **A `std::unique_ptr` storing the address of the deleter function/lambda has an extra pointer and thus is twice the size**
  - Better provide a deleter type as template argument, which implies no space overhead

```
struct free_deleter {
  template<typename T>
  void operator()(T * p) const {
    free(const_cast<std::remove_const_t<T> *>(p));
  }
};

template<typename T>
using unique_C_ptr = std::unique_ptr<T, free_deleter>;

std::string plain_demangle(char const * name) {
  unique_C_ptr<char> toBeFreed{__cxxabiv1::__cxa_demangle(name, 0, 0, 0)};
  std::string result(toBeFreed.get());
  return result;
}
```

Guidelines for `std::unique_ptr`

- **As member variable:**
  - To keep a polymorphic reference instantiated by the class or passed in as `std::unique_ptr` and transferring ownership
- **As local variable:**
  - To implement RAII
  - Can provide custom deleter function as second template argument to type that is called on destruction
- **`std::unique_ptr<T> const p{new T{}}; // local`**
  - Cannot transfer ownership
  - Cannot leak!

`std::shared_ptr<T>` and `std::make_shared<T>()`

`#include <memory>`

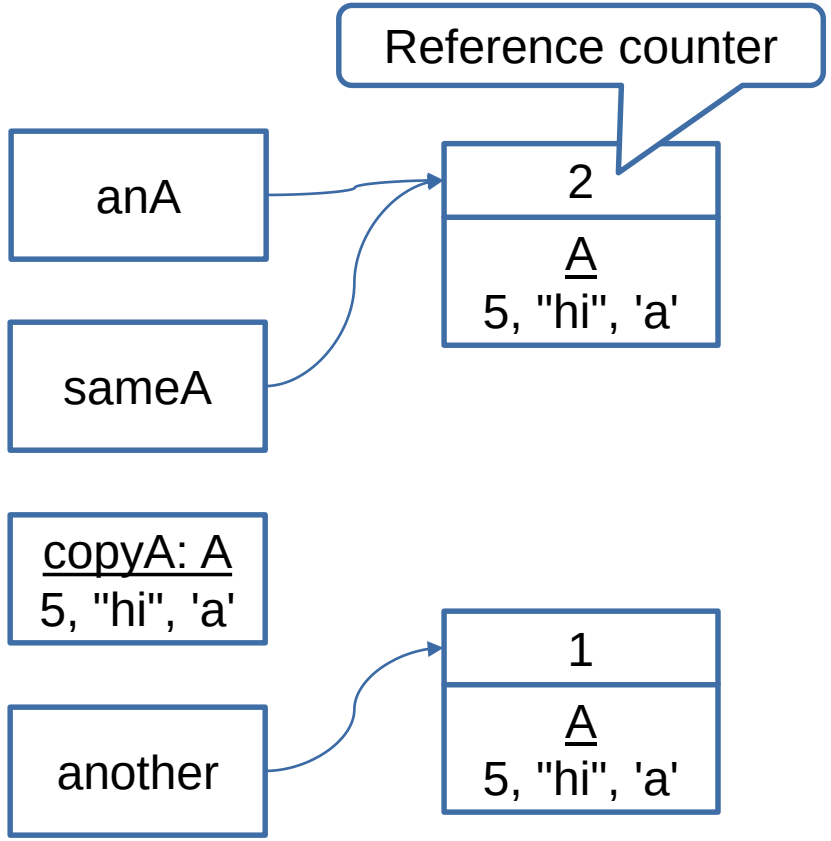
- **`std::unique_ptr` allows only one owner and cannot be copied, but only returned by value**
- **`std::shared_ptr` works more like Java's references**
  - It can be copied and passed around
  - The last one ceasing to exist deletes the object
- **You create `std::shared_ptr` and associated objects of type `T` using `std::make_shared<T>(...)`**
- **`std::make_shared<T>` allows all `T`'s public constructor's parameters to be used**

```
struct Article {
Article(std::string title, std::string content);
  //..
};

Article cppExam{"How to pass CPl?", "In order to pass the C++ exam, you have to..."};

std::shared_ptr<Article> abcPtr = std::make_shared<Article>("Alphabet", "ABCDEFGHIJKLMNOPQRSTUVXYZ");
```

- If you really need heap-allocated objects, because you create your own object networks you can use `std::shared_ptr<T>`
- If you need to support run-time polymorphic container contents or class members that can not be passed as reference, e.g., because of lifetime issues
- Factory functions returning `std::shared_ptr` for heap allocated objects
- But first check if alternatives are viable:
  - (const) references as parameter types or class members (to surviving objects!)
  - Plain member objects or containers with plain class instances

- **If you really need to keep something explicitly on the heap, use a factory**

```
struct A {
  A(int a, std::string b, char c);
};

auto createA() {
  return std::make_shared<A>(5, "hi", 'a');
}

int main(){
  auto anA = createA();
  auto sameA = anA; //second pointer to same object
  A copyA{*sameA}; //copy ctor.
  auto another = std::make_shared<A>(copyA); //copy ctor on heap
}
```

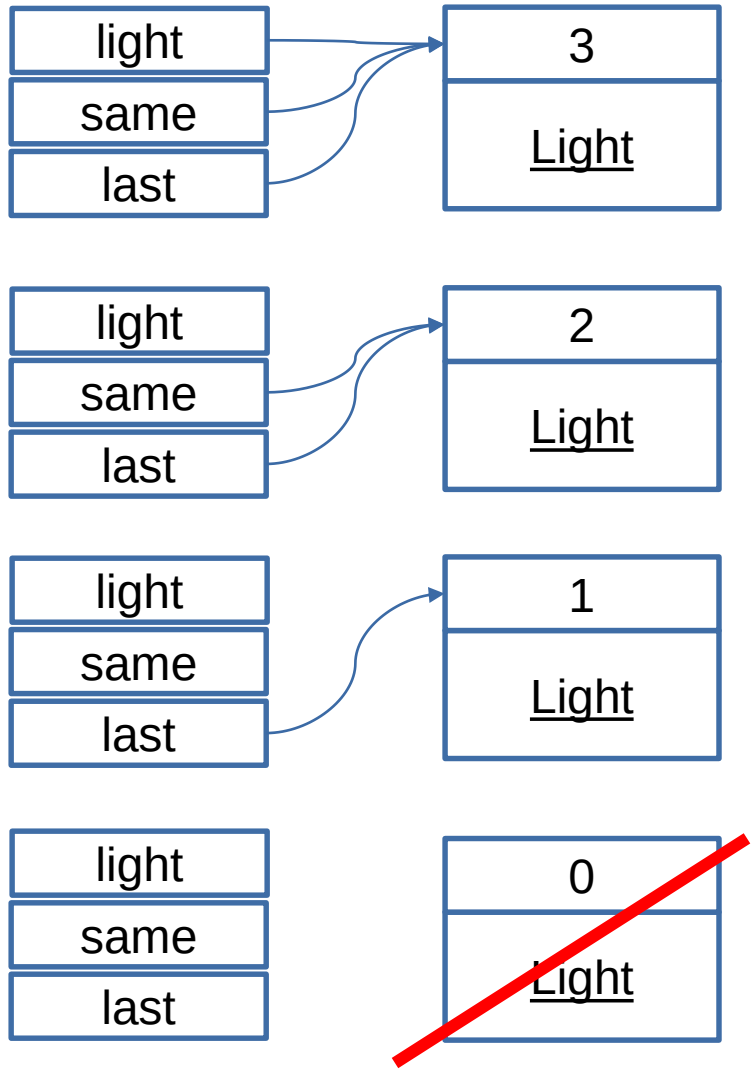


- **Use `std::ostream`, just as an example for a base class**
  - And a very primitive factory function.
  - The concrete type is required as template argument for `make_shared`

```
std::shared_ptr<std::ostream> os_factory(bool file) {
  using namespace std;
  if (file) {
    return make_shared<ofstream>("hello.txt");
  } else {
    return make_shared<ostringstream>();
  }
}

int main(){
  auto out = os_factory(false);
  if (out) {
    (*out) << "hello world\n";
  }
  auto fileout = os_factory(true);
  if (fileout) {
    (*fileout) << "Hello, world!\n";
  }
}
```

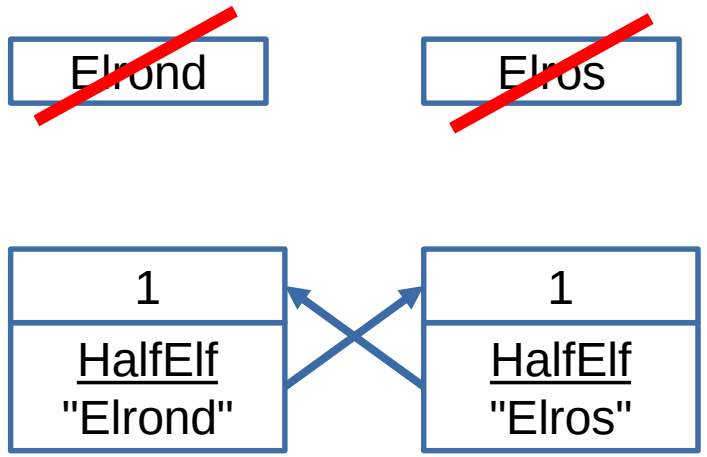
- Last `std::shared_ptr` handle destroyed/reset will delete the allocated object

```
struct Light {
  Light() {
    std::cout << "Turn on\n";
  }
  ~Light() {
    std::cout << "Turn off\n";
  }
};

int main() {
  auto light = std::make_shared<Light>();
  auto same = light;
  auto last = same;
  light.reset();
  same.reset();
  last.reset();
}
```

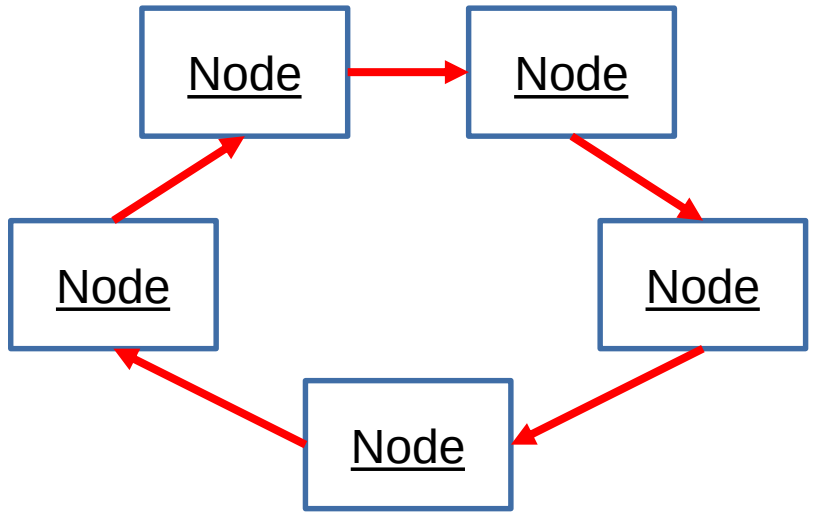


- Last `std::shared_ptr` handle destroyed will delete allocated object

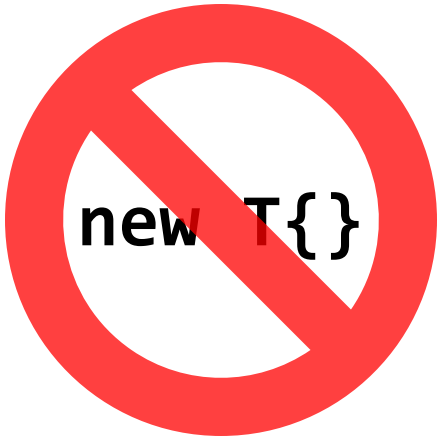
```
using HalfElfPtr = std::shared_ptr<struct HalfElf>;

struct HalfElf {
  explicit HalfElf(std::string name) : name{name}{}
  std::string name{};
  std::vector<HalfElfPtr> siblings{};
};

void middleEarth() {
  auto elrond = std::make_shared<HalfElf>("Elrond");
  auto elros = std::make_shared<HalfElf>("Elros");
  elrond->siblings.push_back(elros);
  elros->siblings.push_back(elrond);
}
```



- Last `std::shared_ptr` handle destroyed will delete allocated object
- If instances of a class hierarchy are always represented by a `std::shared_ptr<base>` but created through `std::make_shared<concrete>()` the destructor no longer needs to be virtual
  - `std::shared_ptr` memorizes concrete destructor for deletion on construction time in `std::make_shared<concrete>`
- `std::shared_ptr` can lead to object cycles no longer cleared, because of circular dependency
  - `std::weak_ptr` breaks such cycles



- Prefer `std::unique_ptr/std::shared_ptr` for heap-allocated objects over `T *`
- Use `std::vector` and `std::string` instead of heap-allocated arrays

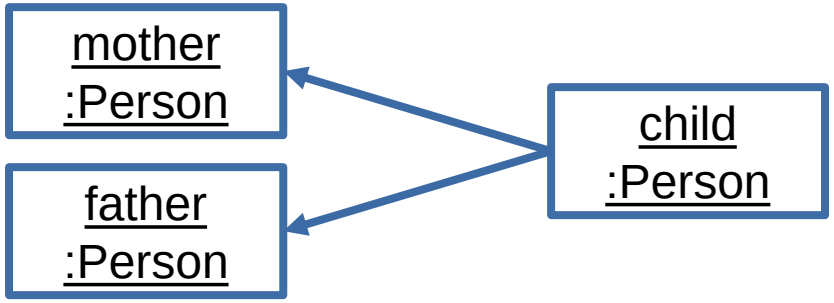


Use `nullptr`

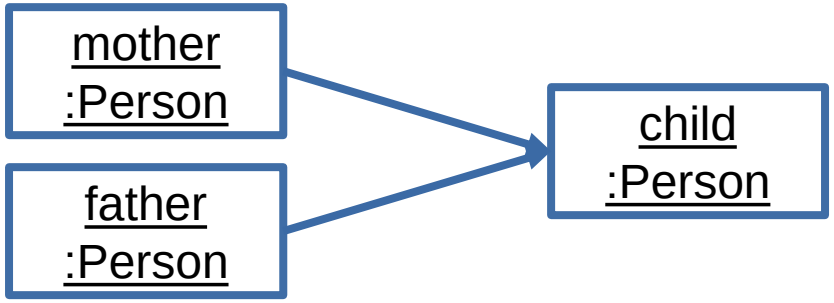
Example: Parents and Children

Creating cycles with std::shared_ptr
Breaking cylcles with std::weak_ptr
Acquiring std::shared_ptr to this with std::enable_shared_from_this

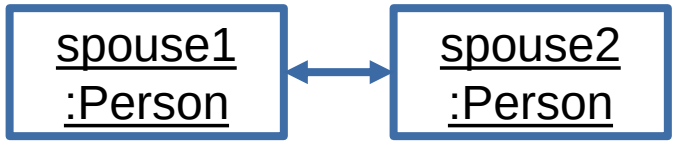
- **Create a class `Person` that represents a person**
  - Each `Person` knows about its parents (father/mother) if they are still alive



  - Each `Person` knows about its children

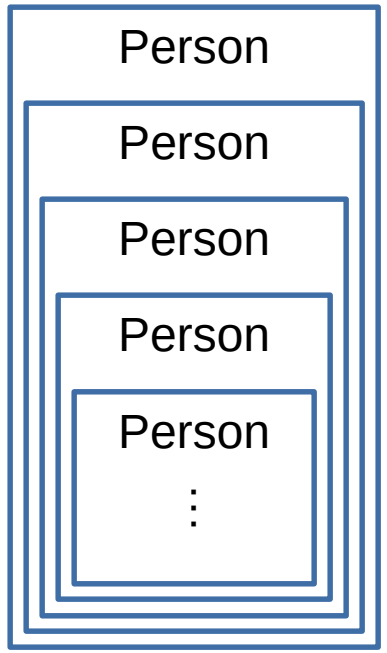


  - Each `Person` can be married



- **This results in cycles between spouses and between parents and children!**

- **Observation: You cannot use direct members (`Persons` as value members)**
  - This would incur copying `Persons`
  - The `Person` class would be recursive and therefore infinite



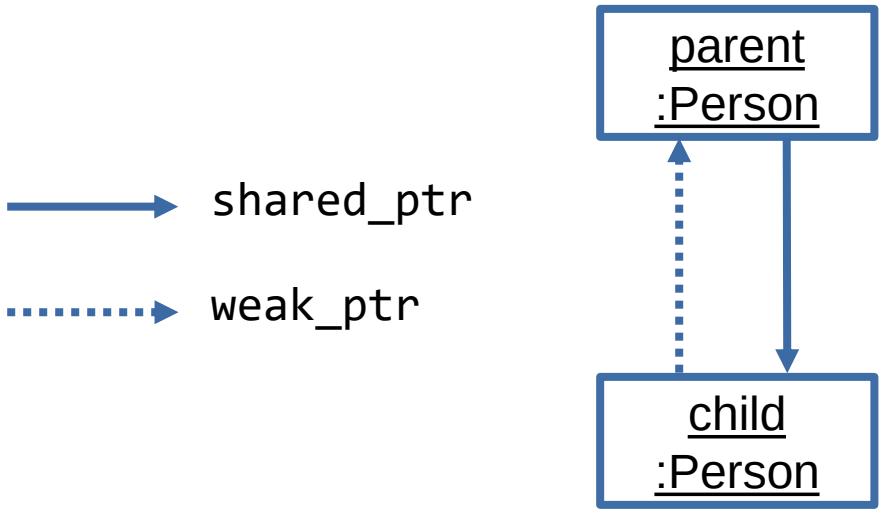

```
struct Matryoshka {
  Matryoshka nested;
};
```

- **We need an indirection (Smart Pointer)**

```
struct Matryoshka {
  std::shared_ptr<Matryoshka> nested;
};
```

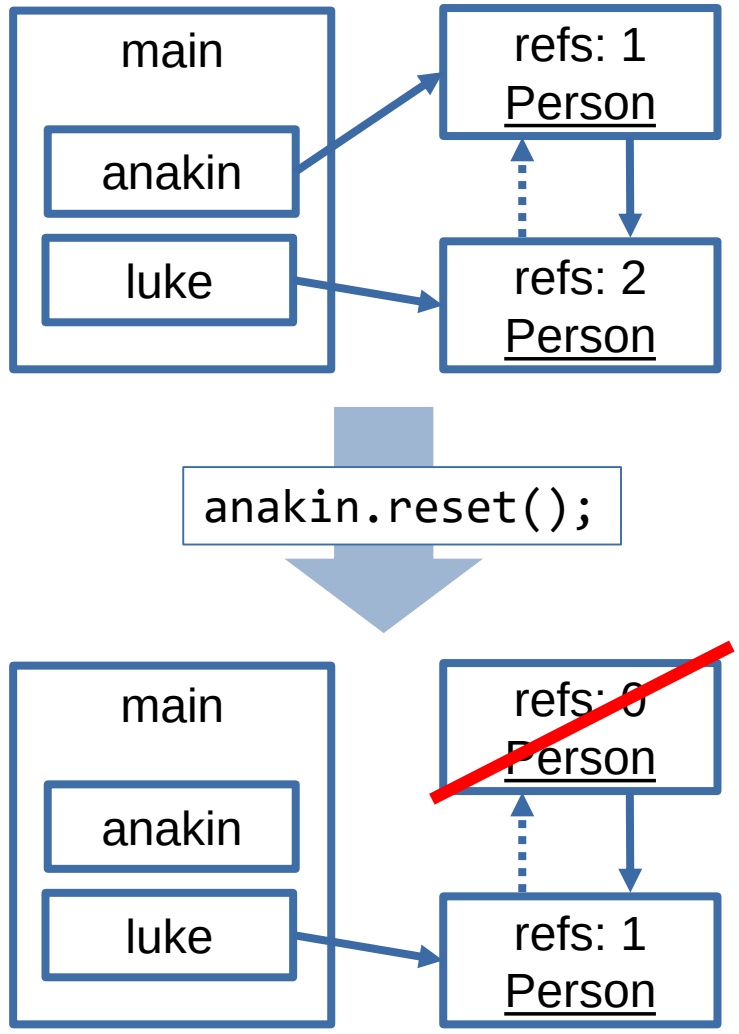
- **The `std::shared_ptr` cycles need to be broken**
- **`std::weak_ptr` does not allow direct access to the object**
  - With `lock()` a `std::shared_ptr` to the object can be acquired

```
struct Person {
  std::shared_ptr<Person> child;
  std::weak_ptr<Person> parent;
};
```



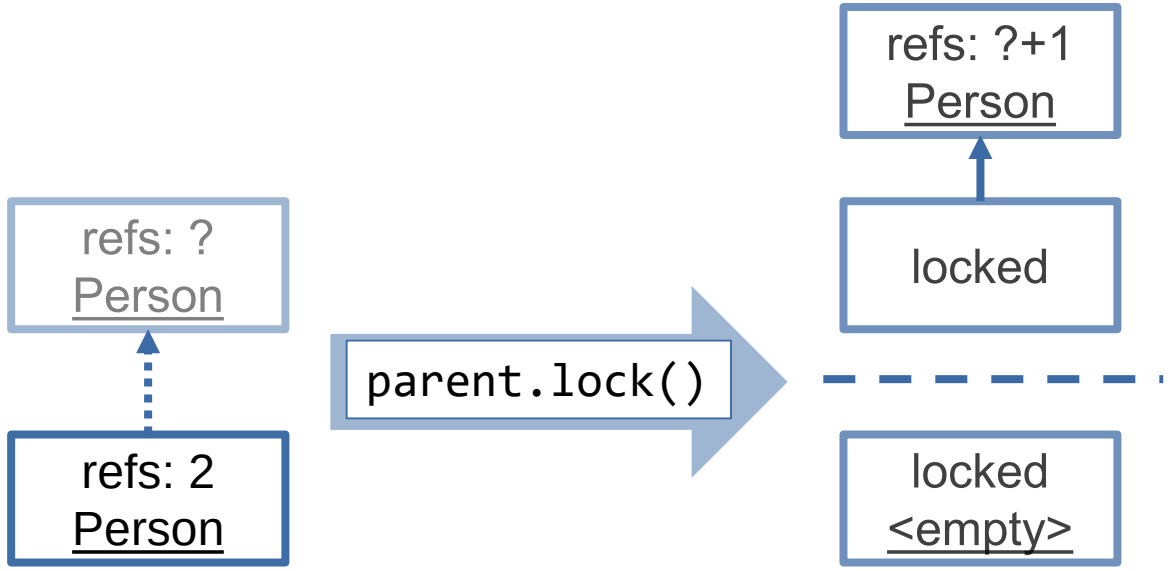
- **The `std::shared_ptr` cycles need to be broken**

```
struct Person {
  std::shared_ptr<Person> child;
  std::weak_ptr<Person> parent;
};

int main() {
  auto anakin = std::make_shared<Person>();
  auto luke = std::make_shared<Person>();
  anakin->child = luke;
  luke->parent = anakin;
  //...
}
```



- **A `std::weak_ptr` does not know whether the pointee is still alive**
  - `std::weak_ptr::lock()` returns a `std::shared_ptr` that either points to the alive pointee or is empty

```
struct Person {
  std::shared_ptr<Person> child;
  std::weak_ptr<Person> parent;

  void Person::acquireMoney() const {
    auto locked = parent.lock();
    if (locked) {
      begForMoney(*locked);
    } else {
      goToTheBank();
    }
  }
};
```



- **It would be nice if parents could spawn their own children**
  - We need a `std::weak_ptr/std::shared_ptr<Person>` to the this object, to assign `child.parent`

```
struct Person {
  std::shared_ptr<Person> child;
  std::weak_ptr<Person> parent;

 auto spawn() {
    child = std::make_shared<Person>();
    child->parent = ???;
    return child;
  }
};
```

```
struct Person : std::enable_shared_from_this<Person> {
  std::shared_ptr<Person> child;
  std::weak_ptr<Person> parent;

  auto spawn() {
    child = std::make_shared<Person>();
    child->parent = weak_from_this();
    return child;
  }
};
```

Curiously Recurring Template Pattern (CRTP)

- **Publicly deriving from `std::enable_shared_from_this<T>` provides the member functions `weak_from_this()` and `shared_from_this()`**
  - It internally stores a `std::weak_ptr` to the this object

- **Smart pointers can be stored in standard containers, like `std::vectors`**
- **An alias for a `Person` pointer that can be used in the type itself requires a forward declaration**

```
using PersonPtr = std::shared_ptr<struct Person>;

struct Person {
  //...
private:
  std::vector<PersonPtr> children;
  std::weak_ptr<Person> mother;
  std::weak_ptr<Person> father;
};
```

- **Be careful when creating object structures with `std::shared_ptr` and avoid circular object dependencies**
  - This requires deliberate breaking to get rid of the instantiated objects
- **Use `std::weak_ptr` consequently**
- **Possible approach:**
  - Keep all "living" objects in a separate data structure as `std::shared_ptr` and model dependencies through `std::weak_ptr`
  - Removing from "live list" destroys object
  - Memory released when last `std::weak_ptr` expires
- **Copying/destroying `std::shared_ptr` is slow due to atomic counter**