SMART POINTERS

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school

AGENDA

1. Smart pointers
   - `std::unique_ptr<>`
   - `std::shared_ptr<>`
   - `std::weak_ptr<>`
2. Best practices
3. Implementation details
4. Efficiency

SMART POINTERS

SMART POINTERS

- A smart pointer manages a pointer to a heap allocated object
  - Deletes the pointed-to object at the right time
  - `operator->()` calls managed object methods
  - `operator.()` calls smart pointer methods
  - smart pointer to a base class can hold a pointer to a derived class
- STL smart pointers:
  - `std::unique_ptr<>`
  - `std::shared_ptr<>`
  - `std::weak_ptr<>`
  - `std::auto_ptr<>` - removed in C++17

std::unique_ptr<>

`std::unique_ptr<>`

TRAITS

`std::unique_ptr<>` USAGE

- Old style approach vs modern approach

```cpp
#include <iostream> // old-style approac

struct Msg {
    int getValue() { return 42; }
};

Msg* createMsg() {
    return new Msg{};
}

int main() {
    auto msg = createMsg();

    std::cout << msg->getValue();
    delete msg;
}
```

```cpp
#include <memory> // modern approach
#include <iostream>

struct Msg {
    int getValue() { return 42; }
};

std::unique_ptr<Msg> createMsg() {
    return std::make_unique<Msg>();
}

int main() {
    // unique ownership
    auto msg = createMsg();

    std::cout << msg->getValue();
}
```

`std::unique_ptr<>` USAGE

- Copying is not allowed
- Moving is allowed

```
std::unique_ptr<MyData> source(void);
void sink(std::unique_ptr<MyData> ptr);

void simpleUsage() {
    source();
    sink(source());
    auto ptr = source();
    // sink(ptr);         // compilation erro
    sink(std::move(ptr));
    auto p1 = source();
    // auto p2 = p1;      // compilation erro
    auto p2 = std::move(p1);
    // p1 = p2;           // compilation erro
    p1 = std::move(p2);
}
```

```
std::unique_ptr<MyData> source(void);
void sink(std::unique_ptr<MyData> ptr

void collections() {
    std::vector<std::unique_ptr<MyDat
    v.push_back(source());

    auto tmp = source();
    // v.push_back(tmp); // compilati
    v.push_back(std::move(tmp));

    // sink(v[0]);        // compilati
    sink(std::move(v[0]));

}
```

std::unique_ptr<> COOPERATION WITH RAW POINTERS

```cpp
#include <memory>

void legacyInterface(int*) {}
void deleteResource(int* p) { delete p; }
void referenceInterface(int&) {}

int main() {
    auto ptr = std::make_unique<int>(5);
    legacyInterface(ptr.get());
    deleteResource(ptr.release());
    ptr.reset(new int{10});
    referenceInterface(*ptr);
    ptr.reset(); // ptr is a nullptr
    return 0;
}
```

- `get()` - returns a raw pointer without releasing the ownership
- `release()` - returns a raw pointer and release the ownership
- `reset()` - replaces the manager object
- `operator*()` - dereferences pointer to the managed object

`std::make_unique()`

```cpp
#include <memory>

struct Msg {
    Msg(int i) : value(i) {}
    int value;
};

int main() {
    auto ptr1 = std::unique_ptr<Msg>(new Msg{5});
    auto ptr2 = std::make_unique<Msg>(5);   // equivalent to above
    return 0;
}
```

`std::make_unique()` is a factory function that produce `unique_ptrs`

- added in C++14 for symmetrical operations on unique and shared pointers
- avoids bare `new` expression

std::unique_ptr<T[]>

```
struct MyData {};

void processPointer(MyData* md) {}
void processElement(MyData md) {}

using Array = std::unique_ptr<MyData[]>;

void use(void)
{
    Array tab{new MyData[42]};
    processPointer(tab.get());
    processElement(tab[13]);
}
```

- During destruction
  - `std::unique_ptr<T>` calls `delete`
  - `std::unique_ptr<T[]>` calls `delete[]`
- `std::unique_ptr<T[]>` has additional `operator[]` for accessing array element
- Usually `std::vector<T>` is a better choice

EXERCISE: RESOURCED

1. Compile and run ResourceD application
2. Check memory leaks under valgrind
3. Fix memory leaks with a proper usage of `delete` operator
4. Refactor the solution to use `std::unique_ptr<>`
5. Use `std::make_unique()`

`std::shared_ptr<>`

std::shared_ptr<>

TRAITS

`std::shared_ptr<>` USAGE

- Copying and moving is allowed

```cpp
std::shared_ptr<MyData> source();
void sink(std::shared_ptr<MyData> ptr);

void simpleUsage() {
    source();
    sink(source());
    auto ptr = source();
    sink(ptr);
    sink(std::move(ptr));
    auto p1 = source();
    auto p2 = p1;
    p2 = std::move(p1);
    p1 = p2;
    p1 = std::move(p2);
}
```

```cpp
std::shared_ptr<MyData> source();
void sink(std::shared_ptr<MyData> ptr);

void collections() {
    std::vector<std::shared_ptr<MyData>> v;

    v.push_back(source());

    auto tmp = source();
    v.push_back(tmp);
    v.push_back(std::move(tmp));

    sink(v[0]);
    sink(std::move(v[0]));
}
```

`std::shared_ptr<>` USAGE CONT.

```cpp
#include <memory>
#include <map>
#include <string>

class Gadget {};
std::map<std::string, std::shared_ptr<Gadget>> gadgets;
// above wouldn't compile with C++03. Why?

void foo() {
    std::shared_ptr<Gadget> p1{new Gadget()};   // reference counter = 1
    {
        auto p2 = p1;                           // copy (reference counter == 2)
        gadgets.insert(make_pair("mp3", p2));   // copy (reference counter == 3)
        p2->use();
    }                                           // destruction of p2, reference counter = 2
}                                               // destruction of p1, reference counter = 1

int main() {
    foo();
    gadgets.clear();                            // reference counter = 0 - gadget is removed
}
```

`std::shared_ptr<>` CYCLIC DEPENDENCIES

- What happens here?

```
#include <memory>

struct Node {
    std::shared_ptr<Node> child;
    std::shared_ptr<Node> parent;
};

int main () {
    auto root = std::shared_ptr<Node>(new Node);
    auto child = std::shared_ptr<Node>(new Node);

    root->child = child;
    child->parent = root;
}
```

Memory leak!

CYCLIC DEPENDENCIES

`std::weak_ptr<>` TO THE RESCUE

TRAITS

- does not own an object
- observes only
- must be converted to `std::shared_ptr<>` to access the object
- can be created only from a `std::shared_ptr<>`

std::weak_ptr<> USAGE

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

struct Msg { int value; };

void checkMe(const std::weak_ptr<Msg> & wp) {
    std::shared_ptr<Msg> p = wp.lock();
    if (p)
        std::cout << p->value << '\n';
    else
        std::cout << "Expired\n";
}

int main() {
    auto sp = std::shared_ptr<Msg>{new Msg{10}};
    auto wp = std::weak_ptr<Msg>{sp};
    checkMe(wp);
    sp.reset();
    checkMe(wp);
}
```

```
> ./a.out
10
Expired
```

std::shared_ptr<> CYCLIC DEPENDENCIES

- How to solve this problem?

```
#include <memory>

struct Node {
    std::shared_ptr<Node> child;
    std::shared_ptr<Node> parent;
};

int main () {
    auto root = std::shared_ptr<Node>(new Node);
    auto child = std::shared_ptr<Node>(new Node);
    root->child = child;
    child->parent = root;
}
```

BREAKING CYCLE - SOLUTION

- Use `std::weak_ptr<Node>` in one direction

```cpp
#include <memory>
struct Node {
    std::shared_ptr<Node> child;
    std::weak_ptr<Node> parent;
};

int main () {
    auto root = std::shared_ptr<Node>(new Node);
    auto child = std::shared_ptr<Node>(new Node);
    root->child = child;
    child->parent = root;
}
```

```
==148== All heap blocks were freed -- no leaks are possible
```

`std::auto_ptr<>` - SOMETHING TO FORGET

- C++98 provided `std::auto_ptr<>`
- Few fixes in C++03
- Yet still it’s easy to use incorrectly…
- Deprecated since C++11
- Removed since C++17
- Do not use it, use `std::unique_ptr<>` instead

SMART POINTERS - SUMMARY

- `#include <memory>`
- `std::unique_ptr<>` for exclusive ownership
- `std::shared_ptr<>` for shared ownership
- `std::weak_ptr<>` for observation and breaking cycles

EXERCISE: RESOURCEFACTORY

1. Compile and run ResourceFactory application
2. Put comments in places where you can spot some problems
3. How to remove elements from the collection (`vector<Resource*> resources`)?
4. Check memory leaks
5. Fix problems

BEST PRACTICES

BEST PRACTICES

- Rule of 0, Rule of 5
- Avoid explicit `new`
- Use `std::make_shared()` / `std::make_unique()`
- Copying `std::shared_ptr<>`
- Use references instead of pointers

RULE OF 0, RULE OF 5

RULE OF 5

- If you need to implement one of those functions:
  - destructor
  - copy constructor
  - copy assignment operator
  - move constructor
  - move assignment operator
- It probably means that you should implement them all, because you have manual resources management.

RULE OF 0

- If you use RAII wrappers on resources, you don't need to implement any of Rule of 5 functions.

AVOID EXPLICIT `new`

- Smart pointers eliminate the need to use `delete` explicitly
- To be symmetrical, do not use `new` as well
- Allocate using:
  - `std::make_unique()`
  - `std::make_shared()`

USE `std::make_shared()` / `std::make_unique()`

- What is a problem here?

```
struct MyData { int value; };
using Ptr = std::shared_ptr<MyData>;
void sink(Ptr oldData, Ptr newData);

void use(void) {
    sink(Ptr{new MyData{41}}, Ptr{new MyData{42}});
}
```

- Hint: this version is not problematic

```
struct MyData { int value; };
using Ptr = std::shared_ptr<MyData>;
void sink(Ptr oldData, Ptr newData);

void use(void) {
    Ptr oldData{new MyData{41}};
    Ptr newData{new MyData{42}};
    sink(std::move(oldData), std::move(newData));
}
```

ALLOCATION DECONSTRUCTED

`auto p = new MyData(10);` means:

- allocate `sizeof(MyData)` bytes
- run `MyData` constructor
- assign address of allocated memory to `p`

> The order of evaluation of operands of almost all C++ operators (including the order of evaluation of function arguments in a function-call expression and the order of evaluation of the subexpressions within any expression) is **unspecified**.

UNSPECIFIED ORDER OF EVALUATION

- How about two such operations?

first operation (A)	**second operation (B)**
(1) allocate `sizeof(MyData)` bytes	(1) allocate `sizeof(MyData)` bytes
(2) run `MyData` constructor	(2) run `MyData` constructor
(3) assign address of allocated memory to `p`	(3) assign address of allocated memory to `p`

- Unspecified order of evaluation means that order can be for example:
  - A1, A2, B1, B2, C3, C3
- What if B2 throws an exception?

USE `std::make_shared()` / `std::make_unique()`

- `std::make_shared()` / `std::make_unique()` resolves this problem

```
struct MyData{ int value; };
using Ptr = std::shared_ptr<MyData>;
void sink(Ptr oldData, Ptr newData);

void use() {
    sink(std::make_shared<MyData>(41), std::make_shared<MyData>(42));
}
```

- Fixes previous bug
- Does not repeat a constructed type
- Does not use explicit `new`
- Optimizes memory usage (only for `std::make_shared()`)

COPYING `std::shared_ptr<>`

```
void foo(std::shared_ptr<MyData> p);

void bar(std::shared_ptr<MyData> p) {
    foo(p);
}
```

- requires counters incrementing / decrementing
- atomics / locks are not free
- will call destructors

CAN BE BETTER?

COPYING `std::shared_ptr<>`

```
void foo(const std::shared_ptr<MyData> & p);

void bar(const std::shared_ptr<MyData> & p) {
    foo(p);
}
```

- as fast as pointer passing
- no extra operations
- not safe in multithreaded applications

USE REFERENCES INSTEAD OF POINTERS

- What is the difference between a pointer and a reference?
  - reference cannot be empty
  - reference, once assigned cannot point to anything else
- Priorities of usage (if possible):
  - `(const) T&`
  - `std::unique_ptr<T>`
  - `std::shared_ptr<T>`
  - `T*`

EXERCISE: LIST

Take a look at `List.cpp` file, where simple (and buggy) single-linked list is implemented.

- `void add(Node* node)` method adds a new `Node` at the end of the list.
- `Node* get(const int value)` method iterates over the list and returns the first Node with matching `value` or `nullptr`

1. Compile and run List application
2. Fix memory leaks without introducing smart pointers
3. Fix memory leaks with smart pointers. What kind of pointers needs to be applied and why?
4. (Optional) What happens when the same Node is added twice? Fix this problem.
5. (Optional) Create `EmptyListError` exception (deriving from `std::runtime_error`). Add throwing and catching it in a proper places.

IMPLEMENTATION DETAILS

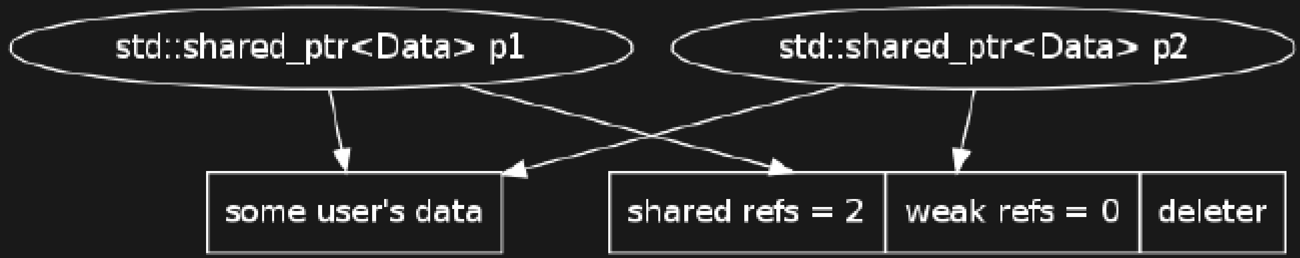
IMPLEMENTATION DETAILS – `std::unique_ptr<>`

- Just a holding wrapper
- Holds an object pointer
- Constructor copies a pointer
- Call proper delete in destructor
- No copying
- Moving means:
  - Copying original pointer to a new object
  - Setting source pointer to `nullptr`
- All methods are inline

IMPLEMENTATION DETAILS - `std::shared_ptr<>`

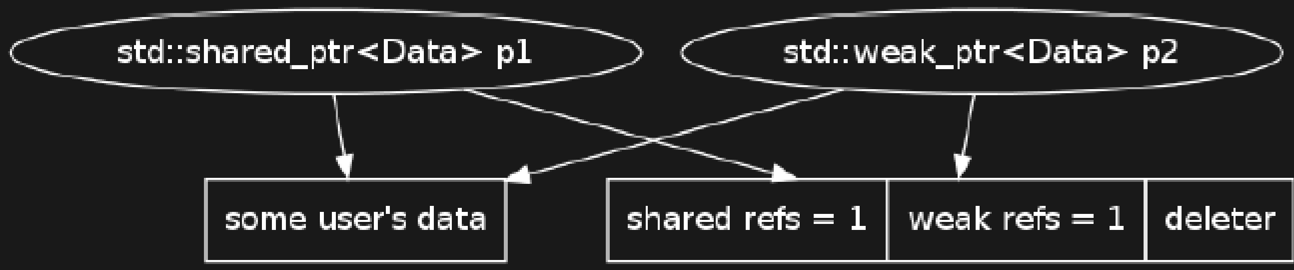
IMPLEMENTATION DETAILS - `std::shared_ptr<>`

- Copying means:
  - Copying pointers to the target
  - Incrementing `shared-refs`



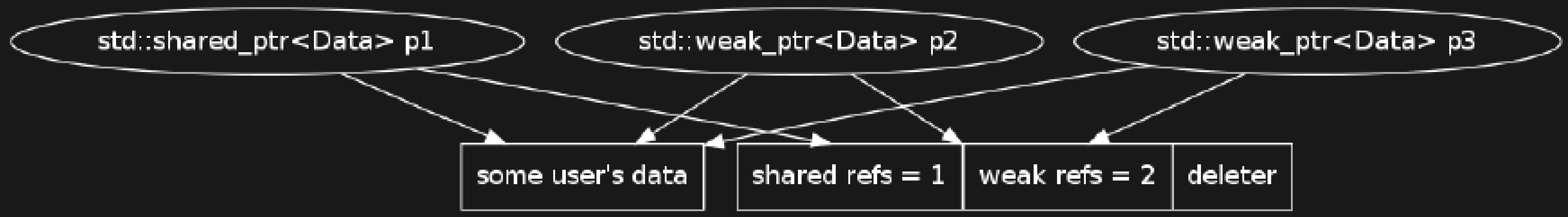
IMPLEMENTATION DETAILS - `std::weak_ptr<>`

- Holds an object pointer
- Holds 2 reference counters:
  - shared pointers count
  - weak pointers count
- Destructor:
  - decrements `weak-refs`
  - deletes reference counters when `shared-refs == 0` and `weak-refs == 0`



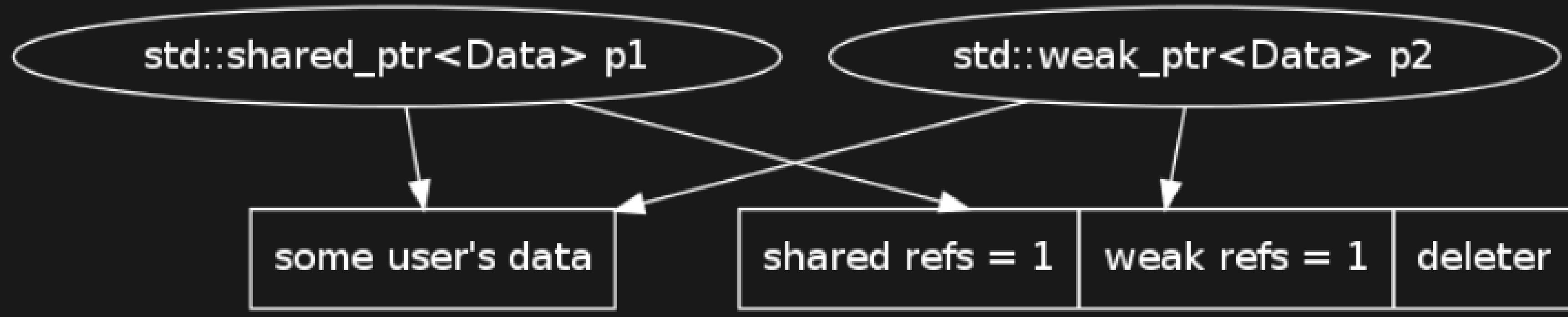
IMPLEMENTATION DETAILS – `std::weak_ptr<>`

- Copying means:
  - Copying pointers to the target
  - Incrementing `weak-refs`



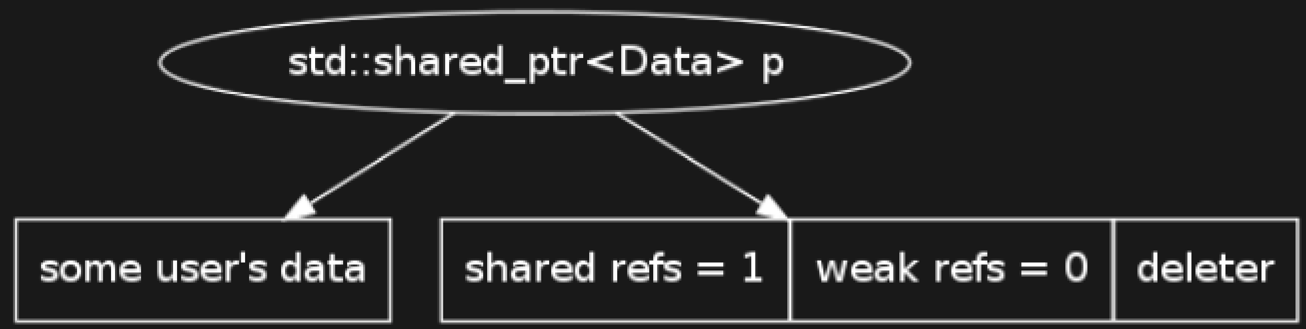
std::weak_ptr<> + std::shared_ptr<>

- Having a shared pointer and a weak pointer



MAKING A `std::shared_ptr<>`

- `std::shared_ptr<Data> p{new Data};`



EFFICIENCY

RAW POINTER

```cpp
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<Data *> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        auto p = new Data;
        v.push_back(std::move(p));
    }
    for (auto p: v)
        delete p;
}
```

UNIQUE POINTER

```cpp
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::unique_ptr<Data>> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        std::unique_ptr<Data> p{new Data};
        v.push_back(std::move(p));
    }
}
```

SHARED POINTER

```cpp
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::shared_ptr<Data>> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        std::shared_ptr<Data> p{new Data};
        v.push_back(std::move(p));
    }
}
```

SHARED POINTER - make_shared

```
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::shared_ptr<Data>> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        auto p = std::make_shared<Data>();
        v.push_back(std::move(p));
    }
}
```

WEAK POINTER

```cpp
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::shared_ptr<Data>> vs;
    std::vector<std::weak_ptr<Data>> vw;
    vs.reserve(size);
    vw.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        std::shared_ptr<Data> p{new Data};
        std::weak_ptr<Data> w{p};
        vs.push_back(std::move(p));
        vw.push_back(std::move(w));
    }
}
```

MEASUREMENTS

- gcc-4.8.2
- compilation with `—std=c++11 —O3 —DNDEBUG`
- measuring with:
  - time (real)
  - htop (mem)
  - valgrind (allocations count)

RESULTS

test name	time [s]	allocations	memory [MB]
raw pointer	0.54	10 000 001	686
unique pointer	0.56	10 000 001	686
shared pointer	1.00	20 000 001	1072
make shared	0.76	10 000 001	914
weak pointer	1.28	20 000 002	1222

CONCLUSIONS

- RAII
  - acquire resource in constructor
  - release resource in destructor
- Rule of 5, Rule of 0
- Smart pointers:
  - `std::unique_ptr` – primary choice, no overhead, can convert to `std::shared_ptr`
  - `std::shared_ptr` – introduces memory and runtime overhead
  - `std::weak_ptr` – breaking cycles, can convert to/from `std::shared_ptr`
- Create smart pointers with `std::make_shared()` and `std::make_unique()`
- Raw pointer should mean „access only" (no ownership)
- Use reference instead of pointers if possible

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school