

# Smart Pointer & Reference Counting

## WP Deeply Embedded Sommer Semester 2015

Marian Triebe

`marian.triebe@haw-hamburg.de`

# Themen

- Motivation
- Shared Pointer (Smart Pointer)
- Shared Pointer und Arrays
- Intrusive Pointer (Reference Counting)
- Implementierung von Shared Pointern

# Motivation

- Keine Garbage Collection (GC) in C++
- Häufige Fehler: Memory Leaks, Use-after-free
- Motivation: Prüfen ob Objekte noch benötigt werden ohne Performance-Verlust

# Shared Pointer

- Boost Pointer
- STL Pointer
- Sind größtenteils per Textersetzung austauschbar
- Kamen zuerst nach boost, später in STL

# Die Basics

- RAW Pointer  $\text{int}^* p = \text{new int}(2);$
- Smart Pointer als Wrapper um RAW Pointer
- Dynamisch allozierter Speicher liegt auf dem Heap
- Frage: Ist der Unterschied zwischen Heap und Stack bewusst?

# Scoped / Unique Pointer

- Speicherfreigabe sobald Scope des "Owners" verlassen wird
- Nicht kopierbar
- Transfer of ownership (nur *unique\_ptr* mit *std::move*)
- Deleter Funktion nur bei *unique\_ptr*

Implementierungen:

- *std::unique\_ptr* in STL
- *boost::scoped\_ptr* in boost

Jedoch unterscheiden sich beide voneinander

# Scoped / Unique Pointer

- Wann sollte *boost::scoped\_ptr* verwendet werden?
  - Bei komplexerem Programflow mit mehreren Return Punkten
- Wann sollte *std::unique\_ptr* verwendet werden?
  - Wenn bspw. mit einer Bibliothek gearbeitet wird, die bei Aufruf Speicher allokiert
  - Kann auch wie *boost::scoped\_ptr* genutzt werden
  - Zusätzliche Möglichkeit des Transfer of Ownership

# boost::scoped\_ptr Beispiel

```
#include <iostream>
#include "boost/scoped_ptr.hpp"
using namespace std;

struct mystruct {
    mystruct() { cout << __PRETTY_FUNCTION__ << endl; }
    ~mystruct() { cout << __PRETTY_FUNCTION__ << endl; }
};

int main() {
    {
        boost::scoped_ptr<mystruct> ptr(new mystruct);
        // hier kann ptr benutzt werden
        // ...
    }
    return 0;
}
```



# unique\_ptr Beispiel (Transfer of Ownership)

```
#include <iostream>
#include <memory>
struct Foo {
    Foo() { std::cout << "Foo::Foo\n"; }
    ~Foo() { std::cout << "Foo::~~Foo\n"; }
    void bar() { std::cout << "Foo::bar\n"; }
};
void f(const Foo &foo) { std::cout << "f(const Foo&)\n"; }
int main()
{
    std::unique_ptr<Foo> p1(new Foo); // p1 owns Foo
    if (p1) { p1->bar(); }
    {
        std::unique_ptr<Foo> p2(std::move(p1)); // now p2 owns Foo
        f(*p2);
        p1 = std::move(p2); // ownership returns to p1
        std::cout << "destroying p2...\n";
    }
    if (p1) { p1->bar(); }
    // Foo instance is destroyed when p1 goes out of scope
}
```

# std::unique\_ptr Beispiel

```
std_unique_ptr.cpp - /Users/Hamdor/gitrepos/deeply_embedded/src - Atom

src
  std_unique_ptr.cpp x
    boost_1 1 #include <memory>
    boost_1 2 #include <iostream>
    boost_1 3
    boost_1 4 #include <netdb.h>
    boost_1 5 #include <arpa/inet.h>
    boost_1 6
    boost_1 7 using namespace std;
    Makefile 8
    std_uni 9 auto getaddrinfo(const string& host) {
    std_uni 10     struct addrinfo info;
    11     struct addrinfo* res = NULL;
    12     std::memset(&info, 0, sizeof(addrinfo));
    13     info.ai_family = AF_INET;
    14     info.ai_socktype = SOCK_STREAM;
    15     getaddrinfo(host.c_str(), NULL, &info, &res);
    16     //      typ      deleter
    17     return unique_ptr<addrinfo, decltype(freeaddrinfo)*> { res, freeaddrinfo };
    18 }
    19
    20 int main() {
    21     {
    22         auto uptr = getaddrinfo("localhost");
    23         // print list
    24         for (addrinfo* ptr = &(*uptr); ptr != nullptr; ptr=ptr->ai_next) {
    25             char address_buffer[INET_ADDRSTRLEN + 1] = { 0 };
    26             auto tmp = reinterpret_cast<sockaddr_in*>(ptr->ai_addr);
    27             auto ok = inet_ntop(tmp->sin_family, &tmp->sin_addr, address_buffer,
    28                               INET_ADDRSTRLEN);
    29             address_buffer[INET_ADDRSTRLEN] = '\0';
    30             if (ok) {
    31                 cout << address_buffer << endl;
    32             }
    33         }
    34     } // uptr wird nach verlassen dieses scopes gelöscht durch freeaddrinfo
    35 }
    36
```

# Shared Pointer

- Sobald keine Shared Pointer mehr auf Speicher verweisen, wird Objekt gelöscht
- Nicht kopierbar
- Kein transfer of ownership (per `std::move`)
- Per Funktionsaufruf von `make_shared` erstellbar

Implementierungen:

- `std::shared_ptr`
- `boost::shared_ptr`

# Shared Pointer Beispiel

```
boost_shared_ptr.cpp - /Users/Hamdor/gitrepos/deeply_embedded/src - Atom

src
  .DS_Store
  boost_intrusive_ptr
  boost_intrusive_ptr.cpp
  boost_scoped_ptr
  boost_scoped_ptr.cpp
  boost_shared_ptr
  boost_shared_ptr.cpp
  Makefile
  std_unique_ptr
  std_unique_ptr.cpp
  std_unique_ptr_ownership
  std_unique_ptr_ownership.cpp
  std_weak_ptr
  std_weak_ptr.cpp

1 // Sourcefile from:
2 // http://en.cppreference.com/w/cpp/memory/shared_ptr
3 #include <iostream>
4 #include <memory>
5 #include <thread>
6 #include <chrono>
7 #include <mutex>
8
9 struct Base {
10     Base() { std::cout << " Base::Base()\n"; }
11     // Note: non-virtual destructor is OK here
12     ~Base() { std::cout << " Base::~Base()\n"; }
13 };
14 struct Derived: public Base {
15     Derived() { std::cout << " Derived::Derived()\n"; }
16     ~Derived() { std::cout << " Derived::~Derived()\n"; }
17 };
18
19 void thr(std::shared_ptr<Base> p) {
20     std::this_thread::sleep_for(std::chrono::seconds(1));
21     std::shared_ptr<Base> lp = p; // thread-safe, even though the
22                                 // shared use_count is incremented
23     {
24         static std::mutex io_mutex;
25         std::lock_guard<std::mutex> lk(io_mutex);
26         std::cout << "local pointer in a thread:\n"
27                 << " lp.get() = " << lp.get()
28                 << ", lp.use_count() = " << lp.use_count() << '\n';
29     }
30 }
31
32 int main() {
33     std::shared_ptr<Base> p = std::make_shared<Derived>();
34
35     std::cout << "Created a shared Derived (as a pointer to Base)\n"
36             << " p.get() = " << p.get()
37             << ", p.use_count() = " << p.use_count() << '\n';
38     std::thread t1(thr, p), t2(thr, p), t3(thr, p);
39     p.reset(); // release ownership from main
40     std::cout << "Shared ownership between 3 threads and released\n"
41             << " ownership from main:\n"
42             << " p.get() = " << p.get()
43             << ", p.use_count() = " << p.use_count() << '\n';
44     t1.join(); t2.join(); t3.join();
45     std::cout << "All threads completed, the last one deleted Derived\n";
46 }
47
```

boost\_shared\_ptr.cpp 7,17

UTF-8 C++ master 41, -7

# Weak Pointer

- Kann aus einem *shared\_ptr* erstellt werden
- Erhöht nicht den Referenzzähler (also nicht wie *shared\_ptr*)
- Kann einen *shared\_ptr* erstellen

Implementierungen:

- *std::weak\_ptr*
- *boost::weak\_ptr*

# Wann sollte `shared_ptr` oder `weak_ptr` verwendet werden?

- `shared_ptr`
  - Ist verantwortlich für das löschen des Objektes
  - Nicht definiertes Verhalten, wenn Objekt nicht existiert
  - Es muss sichergestellt werden, dass das Objekt existiert
- `weak_ptr`
  - Wenn nicht verantwortlich für löschen des Objektes
  - Definiertes Verhalten, wenn Objekt nicht existiert

# Shared Pointer und Arrays

- `boost::shared_ptr` kann erst ab Version 1.53 für Arrays verwendet werden
- `std::shared_ptr` kann immer für Arrays verwendet werden
- Jedoch muss ein *custom deleter* verwendet werden

```
// Array mit 20 Ints,      alloktion      custom deleter
std::shared_ptr<int> sp(new int[20], std::default_delete<int[]>());

// Custom deleter kann auch per Lambda spezifiziert werden
std::shared_ptr<int> sp(new int[20], [](int* p) { delete[] p; } );

// Unique PTR ruft automatisch korrekten deleter auf
std::unique_ptr<int[]> uptr(new int[20]);
```

# weak\_ptr Beispiel

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

std::weak_ptr<int> gw;

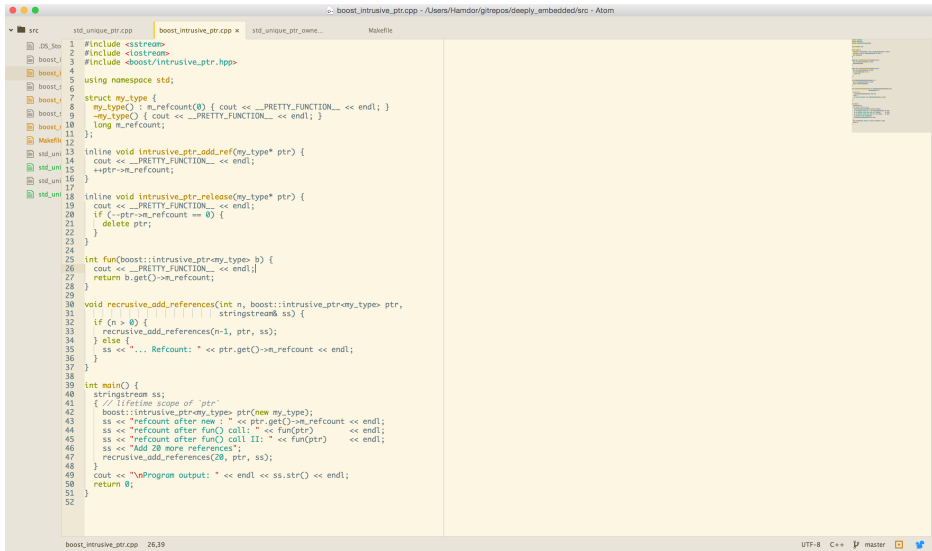
void f() {
    if (auto spt = gw.lock()) {
        std::cout << *spt << "\n";
    } else {
        std::cout << "gw is expired\n";
    }
}

int main() {
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;
        f();
    }
    f();
}
```



- Referenzzähler in Klasse eingebaut (oft durch Vererbung/Interface)
- Referenzzähler ist normaler unsigned Integer
- Gleichviel Speicherverbrauch wie RAW-Pointer
- Kein Overhead (im Vergleich zum *shared\_ptr*)
- Wird auch oft ohne Boost implementiert, damit keine Boost-dependency

# boost::intrusive\_ptr Beispiel



```
1 #include <sstream>
2 #include <iostream>
3 #include <boost/intrusive_ptr.hpp>
4
5 using namespace std;
6
7 struct my_type {
8     my_type() : m_refcount(0) { cout << __PRETTY_FUNCTION__ << endl; }
9     ~my_type() { cout << __PRETTY_FUNCTION__ << endl; }
10     long m_refcount;
11 };
12
13 inline void intrusive_ptr_add_ref(my_type* ptr) {
14     cout << __PRETTY_FUNCTION__ << endl;
15     ++ptr->m_refcount;
16 }
17
18 inline void intrusive_ptr_release(my_type* ptr) {
19     cout << __PRETTY_FUNCTION__ << endl;
20     if (--ptr->m_refcount == 0) {
21         delete ptr;
22     }
23 }
24
25 int fun(boost::intrusive_ptr<my_type> b) {
26     cout << __PRETTY_FUNCTION__ << endl;
27     return b.get()->m_refcount;
28 }
29
30 void recursive_add_references(int n, boost::intrusive_ptr<my_type> ptr,
31                               stringstream& ss) {
32     if (n > 0) {
33         recursive_add_references(n-1, ptr, ss);
34     } else {
35         ss << "... Refcount: " << ptr.get()->m_refcount << endl;
36     }
37 }
38
39 int main() {
40     stringstream ss;
41     { // lifetime scope of 'ptr'
42         boost::intrusive_ptr<my_type> ptr(new my_type);
43         ss << "refcount after new : " << ptr.get()->m_refcount << endl;
44         ss << "refcount after fun() call: " << fun(ptr) << endl;
45         ss << "refcount after fun() call II: " << fun(ptr) << endl;
46         ss << "Add 20 more references";
47         recursive_add_references(20, ptr, ss);
48     }
49     cout << "\nProgram output: " << endl << ss.str() << endl;
50     return 0;
51 }
52
```

## Vorteile gegenüber Shared Pointern

- Kann auch verwendet werden wenn eine benutzte Bibliothek keine Shared Pointer unterstützt
- Light-weight im Vergleich zu Shared Pointer
- Geeignet für Performance und Speicher kritische Systeme

## Nachteile gegenüber Shared Pointer

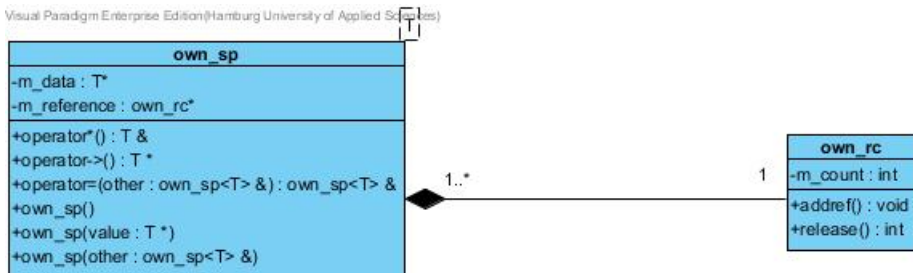
- Hat immer einen Referenz Counter, kann nicht beliebig abgeschaltet werden
- D.h. nur für Heap Objekte Interessant

# Implementierung von Shared Pointern

- Template Klasse die intern beliebigen Typen speichern kann
- Besteht aus 2 Objekten im Heap
- 1) Referenzzähler
- 2) Pointer auf die Daten
- Shared Pointer auf gleichen PTR teilen sich gleichen Referenzzähler

# Implementierung von Shared Pointern

Visual Paradigm Enterprise Edition (Hamburg University of Applied Sciences)



# Implementierung von Shared Pointern

```
own_sp.hpp
1 // Implementierung eines Eigenen Shared Pointers
2 #ifndef OWN_SP_HPP
3 #define OWN_SP_HPP
4
5 #include "own_rc.hpp"
6
7 template<typename T>
8 class own_sp {
9 private:
10     T* m_data; // RAW Pointer auf Daten
11     own_rc m_reference; // RAW Pointer auf Referenzzähler
12 public:
13     own_sp(): m_data(nullptr), m_reference(new own_rc) {
14         m_reference->addref();
15     }
16     own_sp(T* value) : m_data(value), m_reference(new own_rc) {
17         m_reference->addref();
18     }
19
20     // Copy CTOR
21     own_sp(const own_sp<T>& other)
22         : m_data(other.m_data), m_reference(other.m_reference) {
23         m_reference->addref();
24     }
25
26     ~own_sp() {
27         if(m_reference->release() == 0) {
28             delete m_data;
29             delete m_reference;
30         }
31     }
32
33     T& operator*() {
34         return *m_data;
35     }
36
37     T* operator->() {
38         return m_data;
39     }
40
41     // Assignment operator
42     own_sp<T>& operator = (const own_sp<T>& other) {
43         if (this != &other) { // Self assignment ignorieren...
44             if(m_reference->release() == 0) {
45                 delete m_data;
46                 delete m_reference;
47             }
48             m_data = other.m_data;
49             m_reference = other.m_reference;
50             m_reference->addref();
51         }
52         return *this;
53     }
54
55 };
56
57 src/own_shared_ptr.cpp 17,1
```

```
own_rc.hpp
1 #ifndef OWN_RC_HPP
2 #define OWN_RC_HPP
3
4 class own_rc {
5 private:
6     int m_count;
7 public:
8     void addref() {
9         ++m_count;
10     }
11
12     int release() {
13         return --m_count;
14     }
15 };
16
17 #endif // OWN_RC_HPP
18
```

```
own_shared_ptr.cpp
1 #include "own_sp.hpp"
2 #include <iostream>
3
4 struct my_struct {
5     int value = 0;
6     my_struct() { std::cout << __PRETTY_FUNCTION__ << std::endl; }
7     ~my_struct() { std::cout << __PRETTY_FUNCTION__ << std::endl; }
8 };
9
10 int main() {
11     own_sp<my_struct> sp = own_sp<my_struct>(new my_struct);
12     sp->value = 42;
13     std::cout << sp->value << std::endl;
14     std::cout << "sp sollte gleich gelöscht werden..." << std::endl;
15     return 0;
16 }
17
```

Source Codes:

*<https://github.com/Hamdor/WPDE>*

Gute Hilfen rund um C++:

*<http://en.cppreference.com/>*