C++ Parent-Shared-Ptr

Detlef Wilkening

14.01.2016

- **Shared-Ptr**
  - Kennt jeder
  - Nutzt jeder
  - Und das ist gut so
- **Aber es gibt ein paar Features, die nicht so bekannt sind**
  - Eins ist der sogenannte „aliasing constructor“
  - Da er fast immer im Zusammenhang mit Objekten benutzt wird, die in andere eingebettet sind (die Parents) heißt er auch: Parent-Shared-Ptr

- **Worum geht es?**
- **Es gibt einen Shared-Ptr Konstruktor, der**
  - Dessen Typisierung nicht dem Objekt entspricht, das er mit verwaltet
  - Sondern er verwaltet ein ganz anderes Objekt
  - Für das Objekt seiner Typisierung (auf das er zeigt) ist er gar nicht zuständig

- **shared_ptr<T>**
- ```
  template<class Y>
          shared_ptr(const shared_ptr<Y>&, T*) noexcept;
  ```

- **Schauen wir uns mal ein Beispiel an**
  ```

```
class A
{
public:
   A() { cout << "Konstruktor A" << endl; }
   ~A() { cout << "Destruktor A" << endl; }

   A(const A&) = delete;
   A& operator=(const A&) = delete;

   A(A&&) = delete;
   A& operator=(A&&) = delete;

   void fa() { cout << "A::fa()" << endl; }
};
```

```
class B
{
public:
   B() { cout << "Konstruktor B" << endl; }
   ~B() { cout << "Destruktor B" << endl; }

   B(const B&) = delete;
   B& operator=(const B&) = delete;

   B(B&&) = delete;
   B& operator=(B&&) = delete;

   void fb() { cout << "B::fb()" << endl; }
};
```

- **Was passiert hier?**

```
int main()
{
   shared_ptr<A> pa = make_shared<A>();
   shared_ptr<B> pb(pa, new B());

   pa->fa();
   pb->fb();
}
```

- **Was passiert hier?**
  - **Dynamisches A wird von den Shared-Pointern verwaltet**
  - **Dynamisches B nicht**
  - **Auch wenn man über „pb“ auf das B-Objekt zugreifen kann**
  - **=> Der Aliasing-Shared-Pointer verwaltet die Lebensdauer eines anderen Objekts, als man über ihn erreichen kann**

```
int main()
{
   shared_ptr<A> pa = make_shared<A>();      // Konstruktor A
   shared_ptr<B> pb(pa, new B());            // Konstruktor B

   pa->fa();                                 // A::fa
   pb->fb();                                 // B::fb
}                                            // Destruktor A
```

- **Wollen wir uns mal näher ansehen – was passiert hier?**

```
int main()
{
   shared_ptr<A> pa = make_shared<A>();
   shared_ptr<B> pb(pa, new B());

   pa->fa();
   pb->fb();

   cout << "pa reseten" << endl;
   pa.reset();

   cout << "pb reseten" << endl;
   pb.reset();

   cout << "Ende" << endl;
}
```

- **Wollen wir uns mal näher ansehen – was passiert hier?**
  - **Man sieht – „pb“ verwaltet wirklich „A“ – und niemand kümmert sich um „B“**

```
int main()
{
   shared_ptr<A> pa = make_shared<A>();        // Konstruktor A
   shared_ptr<B> pb(pa, new B());              // Konstruktor B

   pa->fa();                                   // A::fa
   pb->fb();                                   // B::fb

   cout << "pa reseten" << endl;               // pa reseten
   pa.reset();

   cout << "pb reseten" << endl;               // pa reseten
   pb.reset();                                 // Destruktor A

   cout << "Ende" << endl;                     // Ende
}
```

- **Was soll das?**
  - Sie haben ein Objekt „B“ als Teil eines größeren Objekts „A“
    - Attribut „B b;“ in Klasse „A“
  - Sie müssen ein „B“ Objekt-Zeiger in eine Funktion reichen
    - Die dort vielleicht auch noch gespeichert wird

  - Die Lebensdauer von „B“ wird aber über den Parent „A“ bestimmt
  - Wenn jetzt alle A-Shared-Pointer entfernt werden, dann wird „A“ gelöscht
  - Und mit „A“ auch „B“
  - „B“ muß aber am „Leben bleiben“, da das Child „B“ noch woanders referenziert und genutzt wird

  - Also benötigt man einen Shared-Pointer für ein „B“, der aber die Lebensdauer von „A“ mit verwaltet
  - => Einen Shared-Pointer mit Aliasing-Konstruktor

```
struct A
{
   A() { cout << "Konstruktor A" << endl; }
   ~A() { cout << "Destruktor A" << endl; }

   A(const A&) = delete;
   A& operator=(const A&) = delete;

   A(A&&) = delete;
   A& operator=(A&&) = delete;

   void fa() { cout << "A::fa()" << endl; }

   B b;
};
```

- **Was passiert hier?**

```
int main()
{
   shared_ptr<A> pa = make_shared<A>();

   shared_ptr<B> pb(pa, &pa->b);

   pa->fa();
   pb->fb();

   cout << "pa reseten" << endl;
   pa.reset();

   cout << "pb reseten" << endl;
   pb.reset();


   cout << "Ende" << endl;
}
```

- **Was passiert hier?**
  - Man sieht – am Ende ist alles gut…

```
int main()
{
   shared_ptr<A> pa = make_shared<A>();        // Konstruktor B
                                               // Konstruktor A
   shared_ptr<B> pb(pa, &pa->b);

   pa->fa();                                   // A::fa()
   pb->fb();                                   // B::fb()

   cout << "pa reseten" << endl;               // pa reseten
   pa.reset();

   cout << "pb reseten" << endl;               // pb reseten
   pb.reset();                                 // Destruktor A
                                               // Destruktor B

   cout << "Ende" << endl;                     // Ende
}
```

- **Es gibt noch ein Feature vom Alasing-Konstruktor**
  - Wird ein Alasing-Konstruktor mit einem „leeren“ Shared-Pointer aufgerufen
  - Dann ist der neue Shared-Pointer
    - bzgl. Lebensdauer-Verwaltung auch leer
      - Daher „use_count()“ ist „0“
    - aber sein Pointer ist nicht leer, sondern kann genutzt werden

- **Hier ein Beispiel, mit den ersten „A“ und „B“ Klassen**
  - Daher kein „B“ Attribut in „A“

```
int main()
{
   B b;                                  // Konstruktor B
   shared_ptr<A> pa;
   shared_ptr<B> pb(pa, &b);

   cout << "pa use-count: " << pa.use_count() << endl;     // -> 0
   cout << "pb use-count: " << pb.use_count() << endl;     // -> 0
   cout << "Adr b:  " << &b << endl;                       // -> ABCD
   cout << "pb.get: " << pb.get() << endl;                 // -> ABCD

   //pa->fa();                           // wäre Absturz
   pb->fb();                             // B::fb()

}                                        // Destruktor B
```

- **Anwendung**
  - Man hat ein lokales Objekt, z.B. vom Typ „B“
  - Die Schnittstelle einer Funktion erwartet einen SharedPtr<B>

```
void fct(const shared_ptr<B>&);
```

```
B b;          // Lokales Objekt
fct(b);       // Wie?
```

```
void fct(const shared_ptr<B>&);
```

```
B b;
shared_ptr<A> pa;
shared_ptr<B> pb(pa, &b);
fct(pb);                    // So!
```

- **Und seit wann gibt es dieses schöne Feature?**
  - Laut Artikel „std::shared_ptr's secret constructor“ von Anthony Williams
    - Link siehe nächste Folie

- **C++ Standard**
  - C++11
- **GCC**
  - Seit mindestens GCC 4.3
- **Microsoft Visual Studio**
  - Seit mindestens Visual Studio 2010
- **Boost**
  - Seit mindestens Boost 1.35.0

- **Links**
  - https://www.justsoftwaresolutions.co.uk/cplusplus/shared-ptr-secret-constructor.html
  - http://en.cppreference.com/w/cpp/memory/shared_ptr/shared_ptr
  - http://www.cplusplus.com/reference/memory/shared_ptr/shared_ptr/

- **ISO C++14 Standard ISO/IEC 14882:2014(E)**
  - 20.8.2.2.1 shared_ptr constructors
    - Punkte 13-16
  - template<class Y> shared_ptr(const shared_ptr<Y>& r, T* p) noexcept;
    - Effects:
      Constructs a shared_ptr instance that stores p and shares ownership with r.
    - Postconditions: get() == p && use_count() == r.use_count()
    - [Note: To avoid the possibility of a dangling pointer, the user of this constructor must ensure that p remains valid at least until the ownership group of r is destroyed. — end note]
    - [Note: This constructor allows creation of an empty shared_ptr instance with a non-null stored pointer. — end note]

Fragen?