

# Programmieren in C++

---

Christian Lang (Lac)

11. Oktober 2019

# Smart-Pointer

---

- Konzept: Memory-Ownership
- Smart-Pointer
- Verwendungs-Empfehlungen

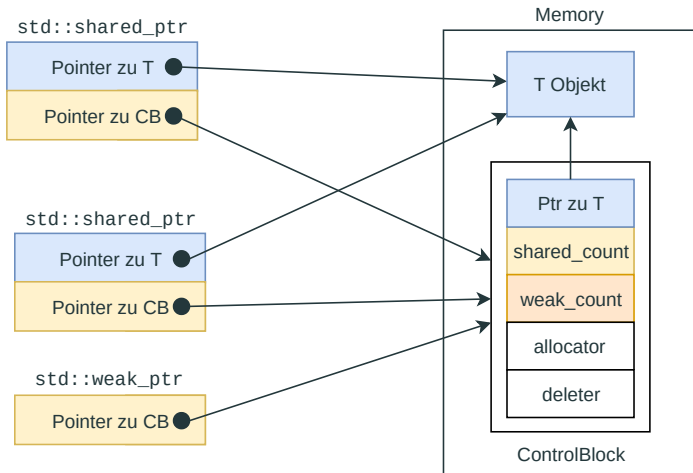
# Konzept: Memory-Ownership

- das Konzept von **Besitz** kann für Memory-Management verwendet werden
- Java verwendet **normale** Referenzen als **shared ownership** Konzept
- Pointer in C/C++ haben **keine Ownership-Semantik**

## Smart-Pointer

- existieren seit **C++11** in der Standard-Library
- Ausnutzen von **RAII**
- Einfache **Handhabung**
- Vorteile gegenüber Garbage-Collection:
  - Memory **sofort** freigeben, wenn nicht mehr benötigt
  - **kein undeterministisches** Verhalten wie bei GC

- Reference-Counting → shared ownership
- zentraler Management Teil → “ControlBlock”
- wenn `shared_count == 0` wird gemanagte Instanz zerstört

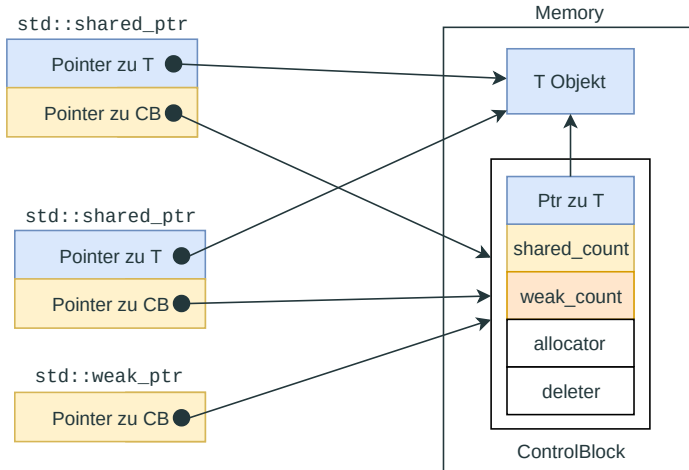


## Beispiel: std::shared\_ptr

```
1  #include <memory>
2
3  {
4      std::shared_ptr<Point> p1 = std::make_shared<Point>();
5      // shared_count == 1
6      {
7          std::shared_ptr<Point> p2;
8          p2 = p1;
9          // shared_count == 2
10
11         auto p3 = p1;
12         // shared_count == 3
13     }
14     // shared_count == 1
15 }
16 // shared_count == 0 -> Point wird im dtor von p1 zerstört
```

## std::weak\_ptr

- aufbrechen von **zyklischen** Abhängigkeiten
- erhöht nicht den **shared\_count** sondern **weak\_count**
- wenn **weak\_count == 0** wird auch CB zerstört



## Beispiel: std::weak\_ptr

```
1  {
2      std::weak_ptr<Point> weak;
3      {
4          auto shared = std::make_shared<Point>();
5          weak = shared;
6          // shared_count == 1
7          // weak_count == 1
8          {
9              std::shared_ptr<Point> usable = weak.lock();
10             // shared_count == 2
11         }
12         // shared_count == 1
13     }
14     // shared_count == 0 -> Point wird im dtor von shared zerstört
15 }
16 // weak_count == 0 -> CB wird in dtor von weak zerstört
```



- unique ownership
- benötigt **keinen** ControlBlock
- dieselbe **Performance** wie roher Pointer
- kann **Move-Semantik** verwenden

```
1  std::unique_ptr<Point> u1 = std::make_unique<Point>();
2  std::unique_ptr<Point> u2 = std::move(u1);
3  // u1 ist nullptr
4
5  std::shared_ptr shared = std::move(u2);
6  // u2 ist nullptr
7
8  shared = nullptr;
9  // Point wird zerstört
```

## `std::make_shared` und `std::make_unique`

- verwenden **Perfect Forwarding** für Konstruktor-Aufruf
- optimiert interne Memory-Allozierung für T und ControlBlock
- kontrolliertes Handling von **Exceptions** in Konstruktoren
- **nie mehr** ein `new` nötig
- meistens mit **auto** kombinieren
- auch **const** möglich

```
1  auto shared = std::make_shared<T>();  
2  std::weak_ptr<T> weak = shared;  
3  auto use = weak.lock();  
4  auto unique = std::make_unique<T>();  
5  auto const_shared = std::make_shared<const T>();
```

# Verwendung von Smart-Pointer

- Verwendung wie rohe Pointer mit `*` und `->`

```
1  // all
2  .reset();           // neu setzten
3  .get();             // rohen Pointer abholen
4  .operator bool();   // Casting zu bool
5
6  // std::shared_ptr
7  .use_count()        // wie viele aktive User?
8
9  // std::weak_ptr
10 .lock()              // konvertieren zu shared
11 .expired();          // shared_count schon 0?
12
13 // unique_ptr
14 .release();          // Management stoppen
```

# Best-Practices für Smart-Pointer

- Vorallem für **Infrastruktur** (z.B. Dependency-Injection)
- ControlBlock ist mittels **Mutex** synchronisiert → `unique_ptr` ist **lock-free**
- Normale Attribute **nicht** mittels Smart-Pointer (möglichst immer statisches Memory)

```
1 class Example {
2     ...
3     private:
4         std::shared_ptr<Point> origin;    // schlecht
5         std::unique_ptr<Point> origin;    // besser
6         Point origin;                    // gut
7     };
```

- APIs (Methoden etc.) sollen **selbst-motiviert** designed werden

```
1 void Calc(std::shared_ptr<Point> p);    // schlecht
2 void Calc(const Point& p);              // gut
3 // Calc will p nur lesen und nicht Ownership bekommen
```