Praxisorientierte Einführung in C++
Lektion: "Smart-Pointer"

Christof Elbrechter

Neuroinformatics Group, CITEC

June 26, 2014

Table of Contents

Nachteile des Speicher-Managements in C++

- Wie bereits oft gesehen: C++-Pointer bringen nicht nur Vorteile mit sich
  - Speicher muss explizit mit `new` oder `new []` alloziert werden
  - Speicher muss immer irgendwann explizit mit `delete` oder `delete []` freigegeben werden
  - Man darf mit- und ohne []-Klammern Versionen nicht verwechseln/mischen
  - Gefahr für Speicherlecks
  - Manchmal ist nicht klar, wer ein Objekt Freigeben muss

```
struct Button{
   void add(ActionListener *listener); // Ownership
   // ...
};
```

- Dabei ist der minimale Overhead eines *managed*-Pointers meißt zu vertreten

Nachteile des Speicher-Managements in C++

- Gleiches Problem entsteht, falls Funktionen Pointer zurückgeben/erzeugen

```
struct ImageReader{
   Image *grabNextImage();
   // ...
};
```

- Wem gehört das zurückgegebene Bild? Darf man es verändern? (OK: `const`-Hint) Muss man es freigeben?
- Ownership muss eindeutig in der Dokumentation geklärt werden
- Ansonsten besteht die Gefahr für
  - Speicherlecks (delete-Aufruf zu wenig)
  - Seg.-Faults (delete Aufruf zu viel)
- Größtes Problem: Dokumentation des Verhaltens ist beliebig und kein eigentliches Sprachkonstrukt

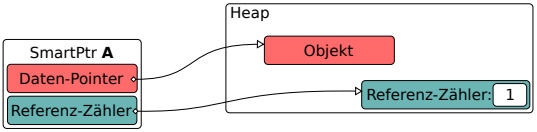
Smart-Pointer in C++

- Auch können oft nicht einfach Objekte übergeben/zurückgegeben werden
  - Großer Aufwand für tiefe Kopien
  - Vererbung und virtuelle Funktionen funktionieren nicht
- Ein Ausweg für all diese Probleme bieten sog. intelligente Pointer (Smart-Pointer)
- Aber Smart-Pointer sind keine spezielles Sprachkonstrukte
- Leider innerhalb der C++-Standard-Bibliothek keine vernünftige Implementation vorhanden
- Aber glücklicherweise generische Implementation mittels Template *relativ* simpel

Smart-Pointer in C++

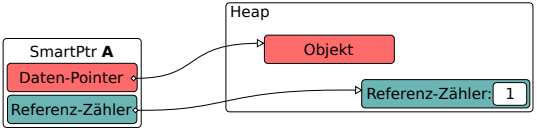
- Das SmartPtr Objekt kann Java-like verwendet werden
  (Ohne & und ohne *)
- Intern wird ein Referenzzähler verwendet, um zu ermitteln, wann der letzte `SmartPtr` für einen bestimmten Daten-Pointer verloren geht
- Wenn die letzte Referenz gelöscht wird, wird auch der Speicher auf dem Heap wieder freigegeben
- Da Smart-Pointer intern einen Pointer *wrappen*, funktionieren auch Vererbung und virtuelle Funktionen damit wie erwartet

SmartPtr Step-by-Step



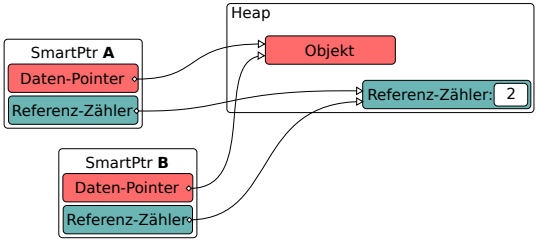
- Anstatt direkt mit dem Pointer auf einen bestimmten Typ zu arbeiten: Arbeite mit Objekten einer Smart-Pointer-Klasse
- ← **Schema**

SmartPtr Step-by-Step



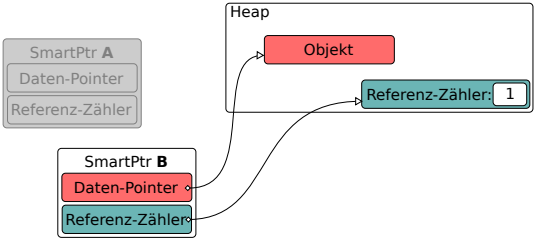
- Ein einfaches SmartPtr-Objekt, welches derzeit die einzige Referenz auf ein bestimmtes Datum darstellt
- `SmartPtr A(new X);`
- Referenz-Zähler ist 1

SmartPtr Step-by-Step



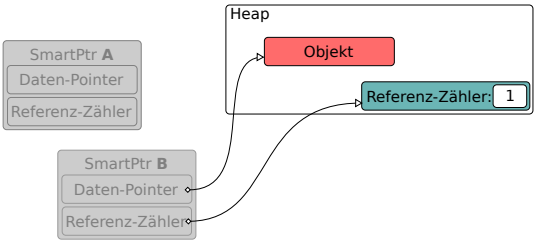
- Kopie des `SmartPtrs` erstellen
- `SmartPtr B = A;`
- Referenz-Zähler wird inkrementiert
- Nicht das Datum, sondern der Pointer darauf wird kopiert

SmartPtr Step-by-Step



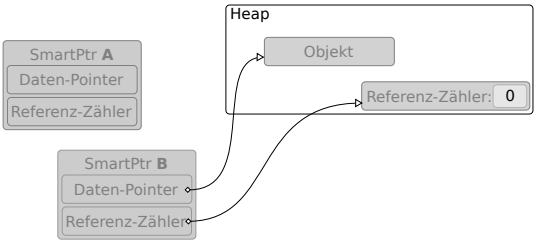
- Falls irgendeine Referenz verloren geht, wird Referenz-Zähler dekrementiert
- Nur wenn der Referenz-Zähler 0 wird, wird auch das Datum gelöscht
- Hier: Referenz-Zähler ist immer noch 1 $\Rightarrow$ nichts wird gelöscht

SmartPtr Step-by-Step



- Falls jedoch auch die letzte Referenz verloren geht, wird der das Objekt freigegeben
- Auch der Speicher für den Referenzzähler muss freigegeben werden

SmartPtr Step-by-Step



- Falls jedoch auch die letzte Referenz verloren geht, wird der das Objekt freigegeben
- Auch der Speicher für den Referenzzähler muss freigegeben werden

SmartPointer Implementation

- Es existiert eine Vielzahl von Implementationen von Smart-Pointer Klassen
- Besonders prominent: `boost::shared_ptr`[1]
- I.d.R. realisiert durch generisches Klassen-Template

SmartPtr.h

```
tempate<class T>
class SmartPtr{
   // ...
};
```

[1] http://www.boost.org/

Exkurs: Pointer- und Dereferenz-Operator

- In der Lektion *Operatoren* wurden diese ausgelassen, da sie ohne Eine Klasse wie SmartPointer relativ umbrauchbar sind und selten verwendet werden
- Dereferenzierungsoperator: `operator*()`
- Zeigeroperator: `operator->()`
- Damit: Klassen, die sich ein wenig wie Zeiger verhalten

Beispiel

```
struct X {  int i; };
struct WannaBePointerToX {
   X m_x;
   X* operator->() { return &m_x; }
   X& operator*() { return m_x; }
};
int main() {
  WannaBePointerToX x;
  x->i = 1;
  std::cout << (*x).i << std::endl;
}
```

Fokus: Zeiger-Operator

- Warum muss man -> nicht zweimal verwenden (`x->->i`)?

```
int main() {
   WannaBePointerToX x;
   x->i = 1;
}
```

Pointe-Operator

- Wenn Klasse `X` den `operator->()` definiert, dann wird bei Zugriff über diesen Operator auf dem Rückgabewert aufgerufen
- Solange, bis *echter* Pointer zurückgegeben wird

Proof-of-concept

```
struct X {
   int i;
};

struct WannaBePointerToX {
   X m_x;
   X* operator->() { return &m_x; }
};
struct WannaBePointerToPointerToX {
   WannaBePointerToX m_x;
   WannaBePointerToX operator->() { return m_x; }
};

int main() {
   WannaBePointerToPointerToX x;
   x->i = 1;
}
```

Eine SmartPtr Implementation

- Eine spezielle Implementation wird hier nun vorgestellt
- Vereinfachte Version des `boost::shared_ptr` Klassen-Templates
- Diese funktioniert nur mit Objekten auf dem Heap – nicht mit Arrays (da delete und nicht delete []) verwendet wird
- Ist aber leicht erweiterbar
- Komplette Implem. in einem Header

Eine SmartPtr Implementation

SmartPtr.h

```
template<class T> class SmartPtr{
  T   *elem;  // Daten Pointer
  int *refc;  // Referenzzaehler
  void inc();  // Eine Referenz mehr
  void dec();  // Eine Referenz weniger
public:
  SmartPtr():  elem(0),refc(0){}
  SmartPtr(T *data): elem(data), refc(data?new int(1):0){}
  SmartPtr(const SmartPtr &o): elem(0),refc(0){
    *this = o;
  }
  SmartPtr &operator=(const SmartPtr& o);
  ~SmartPtr(){  dec(); }
  T &operator* ()              { return *elem;  }
  const T &operator* () const  { return *elem;  }
  T *get ()                    { return elem;   }
  const T *get () const        { return elem;   }

  T *operator-> ()             { return elem;   }
  const T *operator-> () const { return elem;   }

  operator bool() const        { return !!elem; }
};
```

Eine SmartPtr Implementation

```
template<class T>
SmartPtr<T> &SmartPtr<T>::operator=(const SmartPtr<T>& o){
   if(o.elem == elem) return *this;   // Selbstzuweisung
   dec(); // dekrementiere Referenzzaehler fuer aktuelles Objekt. Loesche
          // das Objekt und den aktuellen Referenzzaehler falls notwendig
   elem = o.elem;   // kopiere Daten
   refc = o.refc;
   inc();           // this ist neue Referenz auf elem
                    // -> inkrementiere Referenzzaehler
   return *this;
}
template<class T>
void SmartPtr<T>::inc() { // Falls refc und elem nicht
   if(refc) (*refc)++;    // 0 sind: erhoehe Referenz-Zaehler
}
template<class T>
void SmartPtr<T>::dec() {
   if(!refc) return;      // tue nichts falls refc 0 ist
   if( (--(*refc)) == 0){ // dekrementiere *refc
      delete elem;   // falls nun 0: loesche elem und refc
      delete refc;
   }
}
```

Anwendungsbeispiel

```
struct Rect {
  int x,y,w,h,
};

// relativ gaengig: Ptr-Postfix
typedef SmartPtr<Rect> RectPtr;

int main(){
   RectPtr p1(new Rect);
   RectPtr p2;
   p1 = p2;
   p1 = p1 = p2 = p2 = RectPtr(new Rect);
   p1 = RectPtr(new Rect);
   p1 = RectPtr(0);

   if(p1){
      p1->x = p1->y;         // Direkter Zugriff auf die Member
   }
   if(p1) p1.get()->w = 7;  // Zugriff via Daten-Pointer
   if(p1 && p2) *p1 = *p2;  // Zugriff auf Objekte als Referenz
}
```

Bermerkungen

- Smart-Pointer: Gutes Konzept gegen Memory-Leaks
- Nur verwenden falls sinnvoll
- Für sehr kleine Objekte (z.B. `struct Point{int x,y;}`) besser direkt mit Objekten arbeiten
- Vor allem für Objekte, die viele Resourcen benötigen
- Es existiert eine etwas schwächere Implementation in der STL im header `<memory>`: die template-Klasse `std::auto_ptr`
- `std::auto_ptr`: kein Referenz-Zähler $\Rightarrow$ Ownership wird bei jeder Zuweisung und jedem Konstruktor-Aufruf an den lvalue übergeben
- **Exception-safe-Programming:**
  - Hier sind Smart-Pointer besonders interessant
  - `std::auto_ptr` sind aber auch ausreichend
  - $\Rightarrow$ wird in der Lektion *Exceptions* noch mal aufgegriffen

C++-11

- Im Header `memory` ist dann auch ein `std::shared_ptr<T>` Klassentemplate
- Equivalent zu boost's `shared_ptr<T>` template
- Zusätzlich hier: `std::weak_ptr<T>`
  - Ähnelt dem Konzept des shared pointers – ihm gehört das Objekt aber nicht
  - Ist er die einzige Referenz auf ein Objekt, so wird dieses zerstört
  - Für den Zugriff auf das dahinterliegende Objekt, wird temporär ein `std::shared_ptr` erstellt
  - Dieser ist dann entweder NULL oder er ist gültig, so lange er existiert
- und `std::unique_ptr`, welcher nicht kopiert werden kann
  - wird oft für Sigeltons und so verwendet