Objektorientierte Programmierung für Dummies

Marcus Bäckmann

mitp-Verlag, ISBN 3-8266-2984-1

Nachtrag zu Kapitel 24

Die folgenden Teile erscheinen nicht im Buch, da sie aus Platzgründen gekürzt wurden – der Text würde eigentlich ab Seite 442 im Buch erscheinen... leider wurde die Sache ein wenig dick.

24Und wenn man die Kiste ausschaltet, ist dann alles weg?

In diesem Kapitel

- wird ein Verfahren für die Speicherung von Objektreferenzen vorgestellt

<Seiten 427 – 442>

24.2 Objekt-IDs zur Speicherung von Zeigern und Referenzen

Wenn Sie Geschwister haben, kennen Sie sicherlich das Problem, dass Sie Ihre Spielsachen immer teilen mussten. Den Objekten von heute geht's da manchmal nicht anders, nicht immer besitzt ein Objekt ein anderes exklusiv, oftmals muss es sich ein Objekt mit anderen teilen. Sie erkennen das daran, dass ein Objekt von einem anderen Objekt keine Instanz besitzt, sondern nur einen Zeiger oder eine Referenz.

Ausgehend von einem solchen Problem erfahren Sie nun, wie man auch in diesem Fall eine Speicherung durchführt und nachher die Objekte wieder einander richtig zuordnen kann.

Gehen Sie von der Problemstellung aus, dass es fünf Objekte der Klasse `A` gibt, aber zehn Objekte der Klasse `B`. Jeweils zwei – aber das könnte auch beliebig anders sein – Objekte von `B` teilen sich ein Objekt der Klasse `A`.

Speichert man nun die Objekte `B`, so könnte man in einem naiven Ansatz jedes Mal auch das Objekt `A` einbetten und mitspeichern, aber das entspricht nicht dem Abbild der Daten im Speicher, da auf diese Weise letztlich zehn Objekte der Klasse `A` auf wundersame Weise entstehen. Sozusagen eine wundersame Objektvermehrung.

Im folgenden Quellcode sehen Sie zunächst, wie eine Speicherung erfolgen kann, ohne dass die Beziehung zwischen `A` und `B` verändert wird. Auch hier wird zur Speicherung wieder `TinyXML` angewendet.

```
#include "tinyxml.h"

#include <vector>

#include <iostream>

using namespace std;

class A

{

public:
```

```
   A(int value)
      : m_value(value), m_ID(0)
   {}
   int getVal() const
   {
      return m_value;
   }
   int getID() const
   {
      return m_ID;
   }
   void save(TinyXML::oXMLStream& file)
   {
      if (m_ID == 0)
      {
         m_ID = m_IDCount;
         m_IDCount++;
      }
      file.writeTagOpen("A");
      file.writeTagLine("id", m_ID);
      file.writeTagLine("val", m_value);
      file.writeTagClose("A");
   }
private:
   static int m_IDCount;
   int   m_value;
   int   m_ID;
};
int A::m_IDCount(0);
```

```
class B
{
public:
   B(int value,
     A* pA)
     : m_value(value), m_pA(pA)
   {}
   int getValB() const
   {
      return m_value;
   }
   int getValA() const
   {
      return m_pA->getVal();
   }
   void save(TinyXML::oXMLStream& file)
   {
      file.writeTagOpen("B");
      file.writeTagLine("val", m_value);
      file.writeTagLine("refA", m_pA->getID());
      file.writeTagClose("B");
   }
   void print() const
   {
      cout << "B: " << m_value
           << "  A: " << m_pA->getVal()
           << " (id) " << m_pA->getID() << endl;
   }
private:
```

```
   int m_value;
   A*  m_pA;
};
int main()
{
   vector<A> vecA;
   for (int i = 0; i < 5; i++)
      vecA.push_back(A(i));
   vector<B> vecB;
   for (int j = 0; j < 10; j++)
      vecB.push_back(B(j * 100, &vecA[j / 2]));
   TinyXML::oXMLStream file("refs.xml");
   file.writeTagOpen("refdemo");
   file.writeTagLine("counta", (int)vecA.size());
   vector<A>::iterator ita = vecA.begin();
   while (ita != vecA.end())
   {
      ita->save(file);
      ++ita;
   }
   file.writeTagLine("countb", (int)vecB.size());
   vector<B>::iterator itb = vecB.begin();
   while (itb != vecB.end())
   {
      itb->save(file);
      ++itb;
   }
   file.writeTagClose("refdemo");
   for (int k = 0; k < vecB.size(); k++)
```

```
        vecB[k].print();
    return 0;
}
```

Listing 1: *KAP24/REFS.CPP*

In Abbildung 24.4 sehen Sie noch einmal deutlich, dass die Klassen `A` und `B` nicht mit einer Komposition, sondern mit einer Aggregation verbunden sind. Ein `B` besitzt immer genau ein `A`, aber ein `A` kann zu `0..*` Objekten der Klasse `B` gehören.

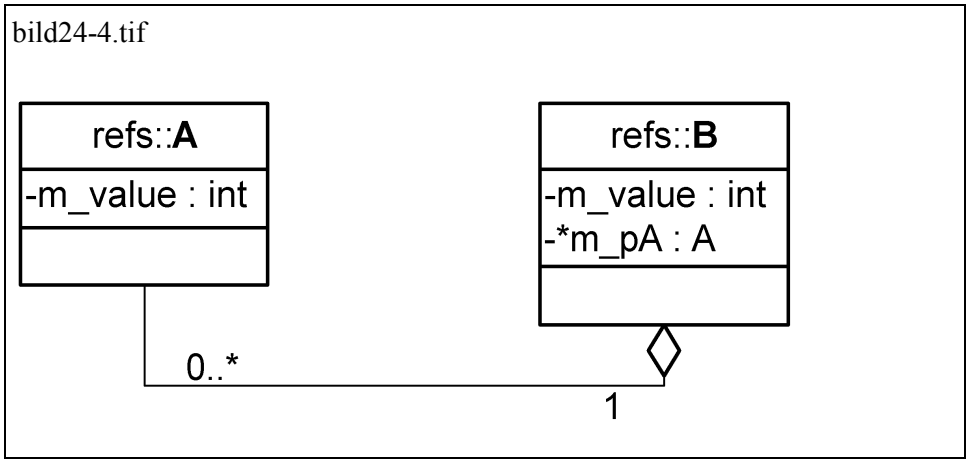

Abbildung 24.4: Objektbeziehung zwischen den Klassen A und B in REFS.CPP

Die Vorgehensweise bei der Speicherung läuft schrittweise wie folgt ab:

1. Speichere zunächst alle Objekte der Klasse `A` ab
2. Bei der Speicherung bekommt jedes Objekt der Klasse `A` einen eindeutigen Identifier (`m_ID`) zugewiesen, der zusätzlich zu den Objektdaten von `A` noch mitgespeichert wird
3. Speichere danach alle Objekte der Klasse `B` ab
4. Statt dass jedes Objekt von `B` nun `A::save()` aufruft, speichert das Objekt zusätzlich zu seinen eigenen Daten noch den Identifier des mit ihm verlinkten Objekts `A` ab. Der Identifier wird über `A::getID()` ermittelt.

Als Resultat speichert man nun tatsächlich fünf Objekte der Klasse `A` und zehn Objekte der Klasse `B`, wobei die Beziehung zwischen den beiden Objekten – im Speicher realisiert durch einen Zeiger – durch einen Identifier, eine Zahl, ausgedrückt wird.

Das Ergebnis in der XML-Datei sieht dann – ausschnittsweise – so aus:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<refdemo>
  <counta>5</counta>
  <A>
    <id>0</id>
    <val>0</val>
  </A>
  <A>
    <id>1</id>
    <val>1</val>
  </A>
...
  <countb>10</countb>
  <B>
    <val>0</val>
    <refA>0</refA>
  </B>
...
  <B>
    <val>900</val>
    <refA>4</refA>
  </B>
</refdemo>
```

Um einen Ladevorgang auszuführen, gehen Sie wie im folgenden Programm gezeigt vor.

```
#include "tinyxml.h"
#include <vector>
#include <iostream>
using namespace std;
class A
```

```
{
public:
   A(int value)
      : m_value(value), m_ID(0)
   {}
   A(TinyXML::iXMLStream& file)
   {
      if (file.findTagOpen("A"))
      {
         file.parseTagLine("id", m_ID);
         file.parseTagLine("val", m_value);
         file.findTagClose("A");
      }
   }
   int getVal() const
   {
      return m_value;
   }
   int getID() const
   {
      return m_ID;
   }
private:
   static int m_IDCount;
   int   m_value;
   int   m_ID;
};
int A::m_IDCount(0);
class B
```

```
{
public:
   B(int value,
     A* pA)
     : m_value(value), m_pA(pA)
   {}
   B(TinyXML::iXMLStream& file,
     vector<A>& vecA)
   {
      file.findTagOpen("B");
      file.parseTagLine("val", m_value);
      int id;
      file.parseTagLine("refA", id);
      for (int i = 0; i < vecA.size(); i++)
         if (vecA[i].getID() == id)
            m_pA = &vecA[i];
      file.findTagClose("B");
   }
   int getValB() const
   {
      return m_value;
   }
   int getValA() const
   {
      return m_pA->getVal();
   }
   void print() const
   {
      cout << "B: " << m_value
```

```
            << "  A: " << m_pA->getVal()
            << " (id) " << m_pA->getID() << endl;
   }
private:
   int m_value;
   A*  m_pA;
};
int main()
{
   vector<A> vecA;
   vector<B> vecB;
   TinyXML::iXMLStream file("refs.xml");
   file.findTagOpen("refdemo");
   int count;
   file.parseTagLine("counta", count);
   for (int i = 0; i < count; i++)
      vecA.push_back(A(file));
   file.parseTagLine("countb", count);
   for (int j = 0; j < count; j++)
      vecB.push_back(B(file, vecA));
   file.findTagClose("refdemo");
   for (int k = 0; k < vecB.size(); k++)
      vecB[k].print();
   return 0;
}
```

Listing 1: *KAP24/REFS2.CPP*

Vergewissern Sie sich zunächst erst einmal, dass beide Programme in der Tat die gleiche Bildschirmausgabe erzeugen:

```
B: 0  A: 0 (id) 0
```

```
B: 100  A: 0 (id) 0

B: 200  A: 1 (id) 1

B: 300  A: 1 (id) 1

B: 400  A: 2 (id) 2

B: 500  A: 2 (id) 2

B: 600  A: 3 (id) 3

B: 700  A: 3 (id) 3

B: 800  A: 4 (id) 4

B: 900  A: 4 (id) 4
```

Da es sich um getrennte Programme handelt, kann damit der Beweis geführt werden, dass über die Datei auch nach dem Ladevorgang wieder genau die gleiche Objektstruktur hergestellt wurde. Operation geglückt, Patient lebt.

Wie im letzten Abschnitt gesehen, wird auch hier der Ladevorgang mit Hilfe eines speziellen Konstruktors ausgeführt, der das Objekt direkt aus einer Datei heraus erzeugen kann. Neu ist aber, dass der Lade-Konstruktor von `B` auch noch einen Referenz auf alle vorhandenen `A`-Objekte mitbekommt. Dies muss so sein – denn bei der Erzeugung von `B` bekommt das Objekt nur den Identifier von `A` mitgeteilt. Über diesen Identifier muss aus dem Pool der vorhandenen `A`-Objekte noch das richtige herausgesucht werden. Und natürlich wird der so ermittelte Zeiger dann dauerhaft im Objekt der Klasse `B` abgelegt.

Zusammenfassend beherrschen Sie nun die wesentlichen Hilfsmittel und Tricks, um Ihre Objekte dauerhaft auf Festplatte zu verewigen. Zur Not tut's auch mal eine Diskette.

REMEMBER

- Müssen Sie Objekte speichern, die andere Objekte nur in Form einer Aggregation (also nicht exklusiv) besitzen, so speichern Sie zunächst diese anderen Objekte in der Datei, vergeben Sie für jedes dieser Objekte einen eindeutigen Identifier (ID). Danach erst sichern Sie Ihr eigentliches Objekt und schreiben als Verweis auf das andere Objekt nur dessen ID in die Datei.
- Zur vereinfachten Darstellung wird beim Ladevorgang hier keine Fehlerprüfung vorgenommen.
- In einem eigenen Programm müssen Sie natürlich die Methoden von `A` und `B` aus den beiden Beispielprogrammen REFS.CPP und REFS2.CPP kombinieren. Aus Gründen der Übersichtlichkeit wurde jeweils nur Laden oder Speichern implementiert.

- Die Vorgehensweise erfolgt analog, wenn `B` statt einem Zeiger auf `A*` eine Referenz auf `A&` besitzt.

REMEMBER

- Wichtig ist, dass der Identifier innerhalb der entstehenden Datei eindeutig ist. Es gibt auch andere Möglichkeiten, einen Identifier zu ermitteln, man könnte auch eindeutige Objektnamen oder Hash-Codes verwenden. Häufig reicht aber bereits eine einfache `int`-ID aus, da man damit auf einem 32-Bit-System immerhin bereits 2 Milliarden Objekte eindeutig zuordnen kann.

- Falls Ihr Programm beim Laden auch ein Append, also ein Anhängen an bestehende Daten, beherrschen soll, müssen Sie ein wenig trickreicher vorgehen. Bevor Sie den Ladevorgang beginnen, müssen zunächst alle bereits vorhandenen Objekt-IDs zurückgesetzt werden (auf einen Wert, der niemals vorkommen kann). Denn sonst gibt es in `vecA` doppelte IDs und die eindeutige Zuordnung geht verloren.

- Sie könnten den statischen Zähler für die Objekt-IDs auch bei jedem Speichern neu auf `0` setzen und die ID-Vergabe immer wieder neu vornehmen. Wie erwähnt, wichtig ist hier nur die Eindeutigkeit innerhalb der Datei.

- Sind Sie schon aufgeregt? Im nächsten Kapitel kommt die Klasse `Wolf` und deletet Objekte der Klasse `Sheep`. Blutige Sache, spannender als *Beben*.