PEARSON Education

ADDISON-WESLEY

Scott Meyers

Effektiv C++ programmieren

Dritte Ausgabe

55 Möglichkeiten, Ihre Programme und Entwürfe zu verbessern

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

3 Ressourcenverwaltung

Eine Ressource ist etwas, das Sie nach Gebrauch wieder an das System zurückgeben müssen. Tun Sie dies nicht, so kann dies schwer wiegende Folgen haben. Die am häufigsten genutzte Ressource in C++-Programmen ist dynamisch belegter Speicherplatz (wenn Sie Speicherplatz reservieren und niemals wieder freigeben, dann haben Sie ein Speicherleck). Speicherplatz ist jedoch nur eine Ressource unter vielen, die Sie im Auge behalten müssen. Andere gebräuchliche Ressourcen sind z. B. Dateideskriptoren, Mutex-Sperren, Zeichensätze und Pinsel in grafischen Benutzeroberflächen, Datenbankverbindungen und Netzwerk-Sockets. Egal welche Ressource Sie auch benutzen, Sie müssen sie immer nach Gebrauch wieder freigeben.

Dies per Hand zu erledigen, ist natürlich immer schwierig. Aber wenn Sie Ausnahmen bedenken, Funktionen mit mehreren Rückgabepfaden oder Wartungsprogrammierer, die Programme verändern, ohne sich der Auswirkungen ihrer Maßnahmen vollständig bewusst zu sein, wird Ihnen klar, dass eine ungeplante Ressourcenverwaltung nicht ausreicht.

Dieses Kapitel beginnt mit einem einfachen, objektorientierten Ansatz zur Ressourcenverwaltung, gestützt auf die C++-Sprachelemente der Konstruktoren, Destruktoren und Kopierbefehle. Die Erfahrung hat gezeigt, dass man durch konsequente und konzentrierte Durchführung dieses Ansatzes so gut wie alle Probleme der Ressourcenverwaltung lösen kann. Danach folgt eine explizite Besprechung der Speicherverwaltung. Die zuletzt genannten Themen vervollständigen die allgemeineren, da Objekte zur Speicherverwaltung natürlich wissen müssen, wie Speicherplatz richtig zu behandeln ist.

3.1 Tipp 13: Verwenden Sie Objekte zur Ressourcenverwaltung

Stellen Sie sich vor, Sie arbeiten mit einer Bibliothek zur Modellierung von Investitionen (z. B. Aktien, Anleihen usw.), in der die verschiedenen Investmenttypen von einer Basisklasse `Investment` erben:

```
class Investment{...}        // Basisklasse der Hierarchie der
                             // Investmenttypen
```

Nehmen Sie ferner an, dass spezielle Investmenttypen von der Bibliothek über eine Fabrik-Funktion (siehe auch Tipp 7) bereitgestellt werden:

```
Investment* createInvestment();   // Gibt Zeiger auf dynamisch
                                  // erstelltes Objekt in der
                                  // Investment-Hierarchie zurück;
                                  // die aufrufende Funktion muss
                                  // es wieder löschen
                                  // (Parameter der Einfachheit
                                  // halber weggelassen)
```

Der Kommentar erklärt, dass die Funktionen, die `createInvestment` aufrufen, dafür verantwortlich sind, dass das zurückgegebene Objekt nach Gebrauch wieder gelöscht wird. Betrachten Sie nun eine Funktion `f`, die dieser Anforderung entsprechend geschrieben wurde:

```
void f()
{
   Investment *plnv = createInvestment();  // Ruft Fabrik-Funktion auf
   ...                                     // plnv wird benutzt
   delete plnv;                            // Objekt wird freigegeben
}
```

Das sieht zwar gut aus, aber es gibt mehrere Möglichkeiten dafür, dass `f` das Investmentobjekt nicht wieder löscht, das die Funktion von `createInvestment` bekommen hat. Es kann ja sein, dass im Funktionsteil »...« ein `return`-Ausdruck auftaucht. Würde solch eine `return`-Anweisung ausgeführt, dann würde der `delete`-Ausdruck niemals erreicht werden können. Eine ähnliche Situation liegt vor, wenn `createInvestment` und `delete` in einer Schleife auftreten und diese Schleife vorzeitig durch eine `continue`- oder eine `goto`-Anweisung verlassen wird. Schließlich kann im Funktionsteil »...« eine Anweisung eine Ausnahme aufwerfen. In diesem Falle gelangt der Programmablauf ebenfalls nicht zur `delete`-Anweisung. Wie auch immer die `delete`-Anweisung übersprungen wird, Sie verlieren auf jeden Fall nicht nur den Speicherplatz, den das Objekt belegt, sondern auch jedwede andere Ressource, die von diesem Objekt beansprucht wird.

Natürlich kann sorgfältige und gewissenhafte Programmierung diese Art von Fehlern vermeiden, aber bedenken Sie nur, wie sehr der Programmtext sich mit der Zeit verändern kann. Bei der Wartung der Software könnte jemand eine `return`- oder eine `continue`-Anweisung hinzufügen, ohne sich der Auswirkung auf die Strategie der Ressourcenverwaltung der Funktion bewusst zu sein. Schlimmer noch: Im Funktionsteil »...« könnte eine Anweisung stehen, die niemals zuvor eine Ausnahme aufwarf, dies aber nun plötzlich nach der »Verbesserung« tut. Sie sollten sich also niemals darauf verlassen, dass `f` immer zur `delete`-Anweisung gelangt.

Um immer sicherzugehen, dass die Ressource, die `createInvestment` zurückgibt, wieder freigegeben wird, müssen Sie sie in ein Objekt einbetten, dessen Destruktor die Ressource automatisch freigibt, sobald der Programmablauf die Funktion `f` wieder verlässt. Tatsächlich ist dies fast der ganze Clou hinter diesem Tipp: Indem Sie Ressourcen in Objekte einbetten, können Sie sich ganz darauf verlassen, dass durch die automatischen Destruktoraufrufe von C++ die Ressourcen wieder freigegeben werden. (Den Rest des Clous behandeln wir in Kürze.)

Für viele Ressourcen wird zunächst auf dem Heap dynamisch Speicherplatz reserviert und dann werden sie lediglich in einem einzigen Block oder in einer einzigen Funktion verwendet. Wenn der Programmablauf diesen Block bzw. diese Funktion wieder verlässt, sollte dieser Speicherplatz wieder freigegeben werden. Das Objekt `auto_ptr` der Standardbibliothek ist wie geschaffen für diese Situation. `auto_ptr` ist ein zeigerähnliches Objekt (ein so genannter intelligenter Zeiger), dessen Destruktor automatisch die `delete`-Anweisung des Objekts aufruft, auf das `auto_ptr` zeigt. Das Speicherleck der Funktion `f` wird unter Benutzung von `auto_ptr` folgendermaßen gestopft:

```
void f()
{
   std::auto_ptr<Investment> plnv(createInvestment()); // Aufruf der
                                                       // Fabrik-Funktion

...                                                    // Benutzt plnv
                                                       // wie vorher

}                                                      // Löscht autom.
                                                       // plnv durch
                                                       // auto_ptr-
                                                       // Destruktor
```

Dieses einfache Beispiel zeigt die zwei Hauptaspekte des Gebrauchs von Objekten zur Ressourcenverwaltung:

- **Ressourcen werden angefordert und sofort an das verwaltende Objekt weitergeleitet.** Im vorherigen Beispiel wurde die Ressource, die von `createInvestment` zurückgegeben wurde, dazu benutzt, das `auto_ptr`-Objekt zu initialisieren. Das `auto_ptr`-Objekt verwaltet dann diese Ressource. Tatsächlich wird das Verfahren, Objekte zur Ressourcenverwaltung zu verwenden, »Ressourcenerwerb ist Initialisierung« (RAII – **R**esource **A**cquisition **I**s **I**nitialization) genannt, da es so häufig vorkommt, dass eine Ressource angefordert und im gleichen Zusammenhang ein Objekt zur Ressourcenverwaltung initialisiert wird. Manchmal werden angeforderte Ressourcen verwaltenden Objekten zugewiesen, anstatt sie zu initialisieren. In allen Fällen wird aber jede Ressource augenblicklich an ein Objekt zur Ressourcenverwaltung übergeben, und zwar in dem Moment, in dem die Ressource erworben wird.

- **Objekte zur Ressourcenverwaltung benutzen ihren Destruktor, um sicherzugehen, dass Ressourcen wieder freigegeben werden.** Weil Destruktoren automatisch aufgerufen werden, wenn ein Objekt zerstört wird (z.B. wenn der Gültigkeitsbereich eines Objekts verlassen wird), werden die Ressourcen korrekt wieder freigegeben, und zwar unabhängig davon, wie der Programmablauf den Block verlässt. Es können aber Schwierigkeiten auftreten, wenn die Freigabe von Ressourcen dazu führt, dass Ausnahmen ausgelöst werden. Aber dieses Problem wurde ausführlicher in Tipp 8 abgehandelt, also kümmern wir uns nun nicht weiter darum.

Da ein `auto_ptr` bei seiner Zerstörung automatisch das Objekt löscht, auf das er zeigt, ist es wichtig, dass auf ein Objekt nur ein `auto_ptr` zeigt. Sollte dies nicht der Fall sein, so würde das Objekt zweimal gelöscht werden, was schnell zu unkontrollierbarem Verhalten Ihres Programms führen würde. Um dies zu verhindern, haben `auto_ptr` eine ungewöhnliche Eigenschaft: Wenn sie kopiert werden (über einen Kopierkonstruktor oder Zuweisungsoperator), werden sie auf null gesetzt, während das Kopierziel die alleinige Kontrolle über die Ressource übernimmt.

```
std::auto_ptr<Investment>              // plnv1 zeigt auf das Objekt,
  plnv1(createInvestment());           // das von createInvestment
                                       // zurückgegeben wird

std::auto_ptr<Investment>plnv2(plnv1); // plnv2 zeigt nun auf das
                                       // Objekt; plnv1 ist nun null

plnv1 = plnv2;                         // Nun zeigt plnv1 auf das
                                       // Objekt und plnv2 ist null
```

Dieses merkwürdige Kopierverhalten und die Anforderung, dass auf Ressourcen, die von `auto_ptr` verwaltet werden, niemals mehr als ein `auto_ptr` zeigen darf, macht `auto_ptr` nicht gerade zum besten Werkzeug, um dynamisch erzeugte Ressourcen zu verwalten. Zum Beispiel erfordern STL-Container, dass ihr Inhalt ein »normales« Kopierverhalten aufweist. Daher sind Container aus `auto_ptr`-Elementen nicht zulässig.

Eine Alternative zu `auto_ptr` stellen intelligente Zähler mit Referenzzählung (RCSP – **R**eference-**C**ounting **S**mart **P**ointer) dar. Ein RCSP ist ein intelligenter Zeiger, der überwacht, wie viele Objekte auf eine Ressource zeigen, und automatisch diese Ressource löscht, wenn kein Objekt mehr auf sie zeigt. Somit verhalten sich RCSPs ähnlich wie die Garbage Collection. Anders als die Garbage Collection können RCSPs keine Kreisreferenzen aufbrechen (z.B. zwei ansonsten ungenutzte Objekte, die jeweils auf das andere Objekt zeigen).

Der Zeiger `tr1::shared_ptr` aus TR1 ist ein solcher RCSP (siehe Tipp 54). Sie können `f` also auch folgendermaßen schreiben:

```
void f()
{
   ...
   std::tr1::shared_ptr<Investment>
     plnv1(createInvestment());          // Ruft Fabrik-Funktion auf
   ...                                   // Benutzt plnv wie vorher
}                                        // plnv wird automatisch
                                         // durch den Destruktor von
                                         // shared_ptr gelöscht
```

Dieser Code sieht fast so aus wie der, der `auto_ptr` verwendet, aber der Kopiervorgang von `shared_ptr` erfolgt sehr viel natürlicher:

```
void f()
{
   ...
   std::tr1::shared_ptr<Investment>     // plnv1 zeigt auf das
     plnv1(createInvestment());         // Objekt, das von
                                        // createInvestment zurückgegeben
                                        // wird

   std::tr1::shared_ptr<Investment>     // plnv1 und plnv2 zeigen
     plnv2(plnv1);                      // nun beide auf das Objekt

   plnv1 = plnv2;                       // dito - nichts hat sich
                                        // geändert

   ...
}                                       // plnv1 und plnv2 sind
                                        // zerstört und das Objekt,
                                        // auf das sie zeigten, ist
                                        // gelöscht
```

Da das Kopieren von `tr1::shared_ptr`-Zeigern wie erwartet funktioniert, können Sie sie in STL-Containern verwenden und auch in anderen Zusammenhängen, in denen das unorthodoxe Kopierverhalten von `auto_ptr` störend wirkt.

Lassen Sie sich aber nicht beirren. Dieser Tipp handelt nicht von `auto_ptr`, `tr1::shared_ptr` oder irgendeiner anderen Art von intelligenten Zeigern, sondern von der Wichtigkeit, Objekte zur Ressourcenverwaltung zu verwenden. `auto_ptr` und `tr1::shared_ptr` sind lediglich Beispiele für Objekte, die dies können. (Weitere Informationen über `tr1::shared_ptr` finden Sie in den Tipps 14, 18 und 54.)

`auto_ptr` und `tr1::shared_ptr` benutzen beide `delete` in ihren Destruktoren und nicht `delete[]`. (Tipp 16 erklärt den Unterschied.) Das bedeutet, dass es keine gute Idee ist, `auto_ptr` und `tr1::shared_ptr` mit dynamisch zugewiesenen Arrays zu verwenden. Bedauerlicherweise können Sie so etwas aber problemlos kompilieren:

```
std::auto_ptr<std::string>                    // Schlechte Idee! Das
  aps(new std:string[10]);                    // falsche delete wird
                                              // verwendet

std::tr1::shared_ptr<int>spi(new int[1024]); // Das gleiche Problem
```

Sie werden erstaunt sein zu erfahren, dass es nichts Derartiges wie `auto_ptr` oder `tr1::shared_ptr` für dynamisch zugewiesene Arrays gibt – nicht einmal in TR1. Das liegt daran, dass `vector` und `string` fast immer dynamisch zugewiesene Arrays ersetzen können. Falls Sie immer noch denken sollten, es wäre nett, wenn es ähnliche Klassen wie `auto_ptr` oder `tr1::shared_ptr` gäbe, dann werfen Sie einen Blick auf Boost (siehe Tipp 55). Sie werden erfreut sein, dort die Klassen `boost::scoped_array` und `boost::shared_array` zu finden, die genau das bieten, wonach Sie suchen.

Der Tenor dieses Tipps über die Verwendung von Objekten zur Ressourcenverwaltung deutet an, dass Sie etwas falsch machen, wenn Sie Ressourcen per Hand freigeben (z. B. indem Sie `delete` nicht in einer Klasse zur Ressourcenverwaltung verwenden). Vorgefertigte Klassen zur Ressourcenverwaltung wie `auto_ptr` und `tr1::shared_ptr` machen es Ihnen einfach, dem Ratschlag dieses Tipps zu folgen. Aber manchmal verwenden Sie Ressourcen, bei denen diese Klassen nicht das liefern können, was Sie benötigen. In diesem Fall müssen Sie sich Ihre eigenen Klassen zur Ressourcenverwaltung fertigen. Dies ist zwar nicht sonderlich schwierig, erfordert aber, einige Feinheiten zu beachten. Diese Feinheiten werden in den Tipps 14 und 15 behandelt.

Zum Schluss möchte ich Sie noch darauf hinweisen, dass der Rohzeiger als Rückgabetyp von `createInvestment` einer Einladung zu einem Speicherleck gleichkommt. Es ist nämlich sehr leicht zu vergessen, dass die aufrufende Funktion den Zeiger, den sie geliefert bekommt, mit `delete` wieder löschen muss. (Sogar wenn Sie `auto_ptr` oder `tr1::shared_ptr` verwenden, um die Ressource wieder freizugeben, müssen Sie immer noch daran denken, den Rückgabewert von `createInvestment` in einem solchen intelligenten Zeiger zu speichern.) Um dieses Problem zu umgehen, ist eine Schnittstellenumstellung von `createInvestment` vonnöten – dieses Problem behandle ich in Tipp 18.

Was Sie sich merken sollten

- Verwenden Sie RAII-Objekte, die Ressourcen über ihre Konstruktoren erwerben und sie wieder in ihren Destruktoren freigeben, um Speicherlecks zu verhindern.
- Zwei gebräuchliche RAII-Klassen sind `tr1::shared_ptr` und `auto_ptr`. `tr1::shared_ptr` ist gewöhnlich die bessere Wahl, da ihr Kopierverhalten intuitiv ist. Einen `auto_ptr`-Zeiger zu kopieren setzt diesen auf null.

3.2 Tipp 14: Bedenken Sie das Kopierverhalten in Klassen zur Ressourcenverwaltung

Tipp 13 stellt das Konzept »Ressourcenerwerb ist Initialisierung« (RAII – **R**esource **A**cquisition **I**s **I**nitialization) als Hauptaspekt der Klassen zur Ressourcenverwaltung vor und beschreibt, wie `auto_ptr` und `tr1::shared_ptr` diese Idee für heapbasierte Ressourcen realisieren. Jedoch sind nicht alle Ressourcen heapbasiert, so dass die intelligenten Zeiger `auto_ptr` und `tr1::shared_ptr` für manche zur Verwaltung ungeeignet sind. Von Zeit zu Zeit werden Sie sich also damit konfrontiert sehen, dass Sie sich Ihre eigenen Klassen zur Ressourcenverwaltung erstellen müssen.

Nehmen Sie zum Beispiel einmal an, Sie benutzen eine C-API zur Behandlung von Objekten des Typs `Mutex`, die die Funktionen `lock` und `unlock` enthält:

```
void lock(Mutex *pm);           // Sperrt den Mutex, auf den pm zeigt
void unlock(Mutex *pm);         // Entsperrt den Mutex
```

Um sicherzugehen, dass Sie niemals vergessen, einen gesperrten Mutex zu entsperren, möchten Sie vielleicht eine Klasse schreiben, die Sperren verwaltet. Die Grundstruktur solch einer Klasse wird durch das RAII-Prinzip vorgegeben, dass Ressourcen bei der Erzeugung angefordert und bei der Zerstörung wieder freigegeben werden:

```
class Lock{
public:
   explicit Lock(Mutex *pm)
   :mutexPtr(pm)
   {lock(mutexPtr);}              // Ressource anfordern

   ~Lock(){unlock(mutexPTR);}     // Ressource freigeben

private:
   Mutex *mutexPtr;
};
```

Kunden verwenden `Lock` in der RAII-typischen Art und Weise:

```
Mutex m;                    // Definiert den Mutex, den Sie benötigen

...
{                           // Erzeugt Block, um entscheidenden Abschnitt
                            // zu definieren

   Lock ml(&m);             // Sperrt den Mutex

   ...                      // Kritische Operationen werden
                            // ausgeführt
```

```
}                          // Am Ende des Blocks wird Mutex
                           // automatisch entsperrt
```

Das ist in Ordnung, aber was passiert, wenn ein `Lock`-Objekt kopiert werden soll?

```
Lock ml1(&m);              // Sperrt m
Lock ml2(ml1);              // Kopiert ml1 nach ml2 - was muss
                           // hier passieren?
```

Dies ist ein spezielles Beispiel einer grundsätzlichen Frage, die sich jeder Autor einer RAII-Klasse stellen muss: Was soll passieren, wenn ein RAII-Objekt kopiert wird? Meistens wählen Sie eine der folgenden Möglichkeiten aus:

- **Kopieren verbieten.** In vielen Fällen ergibt es keinen Sinn, wenn es erlaubt wird RAII-Objekte zu kopieren. Für eine Klasse wie `Lock` ist dies sehr wahrscheinlich, da es wenig gescheit ist, »Kopien« von Synchronisations-Objekten zu haben. Sollte es nicht sinnvoll sein ein RAII-Objekt zu kopieren, dann sollten Sie es verbieten. Tipp 6 erklärt, wie Sie dies tun: Sie deklarieren die Kopierbefehle als `private`. Für `Lock` sieht das in etwa so aus:

  ```
  class Lock: private Uncopyable{     // Kopieren verbieten -
  public:                            // siehe Tipp 6
     ...                             // Wie vorher
  };
  ```

- **Referenzzählen der zugrunde liegenden Ressource.** Manchmal ist es wünschenswert eine Ressource zu behalten, bis das letzte Objekt, das auf sie zugreift, gelöscht wurde. In diesem Falle erhöht die Kopie eines RAII-Objekts die Anzahl der Objekte, die auf die Ressource verweisen. So funktioniert das Kopieren bei `tr1::shared_ptr`.

  Häufig können Sie in RAII-Klassen dieses Kopieren mit Referenzzählung durch ein Exemplar eines `tr1::shared_ptr`-Zeigers einbauen. Wenn Sie z. B. in der Klasse `Lock` die Referenzzählung einsetzen wollen, können Sie den Typ von `mutexPtr` von `Mutex*` in `tr1::shared_ptr<Mutex>` ändern. Unglücklicherweise löscht `tr1::shared_ptr` standardmäßig die Ressource, auf die er zeigt, wenn der Referenzzähler auf null gesunken ist, aber das wollen Sie ja nicht. Wenn Sie fertig sind mit einem `Mutex`, dann wollen Sie ihn ja entsperren und nicht löschen.

  Glücklicherweise können Sie für `tr1::shared_ptr` eine Löschfunktion bzw. ein Funktionsobjekt angeben, die bzw. das aufgerufen wird, wenn der Referenzzähler auf null zurückgeht. (Diese Funktionalität gibt es für `auto_ptr` nicht – er löscht seinen Zeiger *immer*.) Die Löschfunktion ist ein optionaler, zweiter Parameter des Konstruktors von `tr1::shared_ptr`, mit dem der Programmtext wie folgt aussieht:

```
class Lock{
public:
   explicit Lock(Mutex *pm) // Initialisiert shared_ptr mit
                            // Mutex, auf den gezeigt wird,
   :mutexPtr(pm, unlock)    // und mit Funktion unlock
   {                        // als Löschfunktion
      lock(mutexPtr.get()); // Mehr Informationen über "get"
   }                        // in Tipp 15
private:
   std::tr1::shared_ptr<Mutex> mutexPtr;  // Benutzt
};                                        // shared_ptr statt
                                          // Rohzeiger
```

Beachten Sie bitte, dass in diesem Beispiel für die Klasse `Lock` kein Destruktor mehr deklariert wird. Das liegt daran, dass sie keinen mehr benötigt. In Tipp 5 wird erklärt, dass ein Klassendestruktor (egal, ob compilergeneriert oder benutzerdefiniert) automatisch die Destruktoren der nicht statischen Datenelemente einer Klasse aufruft. In diesem Beispiel ist dies `mutexPtr`. Aber der Destruktor von `mutexPtr` ruft automatisch die Löschfunktion von `tr1::shared_ptr` auf, wenn der Referenzzähler des Mutex auf null fällt – in diesem Fall ist die Löschfunktion `unlock`. (Es wäre sinnvoll, wenn Sie im Kommentar des Programms erwähnen würden, dass Sie den Destruktor nicht vergessen haben, sondern sich einfach auf das Standardverfahren des Compilers verlassen.)

- **Kopieren der zugrunde liegenden Ressource.** Manchmal können Sie so viele Kopien einer Ressource haben wie Sie wollen, wobei der einzige Grund, warum Sie eine Klasse zur Ressourcenverwaltung anlegen, darin besteht, dass jede Kopie nach Gebrauch wieder freigegeben werden soll. In diesem Fall sollte auch die verknüpfte Ressource mitkopiert werden, wenn Sie das Objekt zur Ressourcenverwaltung kopieren. Das heißt, dass das Kopieren eines Objekts zur Ressourcenverwaltung eine »Tiefenkopie« durchführt.

  Einige Implementierungen des Standardtyps `string` bestehen aus Zeigern auf den Heap-Speicher, in dem die Symbole, aus denen die Zeichenkette besteht, gespeichert werden. Solche `string`-Objekte enthalten einen Zeiger auf den Heap. Wenn nun ein `string`-Objekt kopiert wird, dann wird gleich beides kopiert, der Zeiger und auch der Speicherinhalt, auf den er weist. Bei solchen Zeichenketten tritt eine »Tiefenkopie« auf.

- **Besitz der zugrunde liegenden Ressource übertragen.** In seltenen Fällen möchten Sie sicherstellen, dass lediglich ein RAII-Objekt auf eine Ausgangsressource verweist und der Besitz beim Kopieren des RAII-Objekts von der Kopierquelle auf das Kopierziel wechselt. Wie in Tipp 13 beschrieben, ist dies das Kopierverhalten von `auto_ptr`.

Die Kopierfunktionen (der Kopierkonstruktor und der Zuweisungsoperator) können zwar vom Compiler generiert sein (Tipp 5 beschreibt deren Standardverhalten), wenn sie sich aber nicht so verhalten, wie Sie wünschen, müssen Sie die Funktionen selbst schreiben. In einigen Fällen möchten Sie vielleicht auch verallgemeinerte Versionen dieser Funktionen anbieten. Solche Versionen werden im Tipp 45 vorgestellt.

Was Sie sich merken sollten

- Das Kopieren eines RAII-Objekts führt immer auch zum Kopieren der Ressource, die vom Objekt verwaltet wird. Daher bestimmt das Kopierverhalten der Ressource das des RAII-Objekts.
- Das übliche Kopierverhalten von RAII-Klassen untersagt das Kopieren und führt eine Referenzzählung durch. Abweichende Verhalten sind aber auch möglich.

3.3 Tipp 15: Bieten Sie in Klassen zur Ressourcenverwaltung einen Zugriff auf Rohressourcen an

Klassen zur Ressourcenverwaltung sind wunderbar. Sie sind Ihr Bollwerk gegen Speicherlecks; ein gut entwickeltes System zeichnet sich durch die Abwesenheit von Speicherlecks aus. In einer idealen Welt könnten Sie sich beim Umgang mit Ressourcen blind auf solche Klassen verlassen und müssten sich niemals die Hände mit dem direkten Zugriff auf Rohressourcen schmutzig machen. Die Welt ist aber leider nicht ideal. Viele APIs beziehen sich direkt auf Ressourcen. Solange Sie also solchen APIs nicht abschwören (was nicht sehr praktisch wäre), müssen Sie Objekte zur Ressourcenverwaltung umgehen und sich von Zeit zu Zeit mit Rohressourcen beschäftigen.

Tipp 13 zum Beispiel führt den Gedanken ein, intelligente Zeiger wie `auto_ptr` oder `tr1::shared_ptr` einzusetzen, um das Ergebnis einer Fabrik-Funktion wie `createInvestment` aufzunehmen:

```
std::tr1::shared_ptr<Investment>plnv(createInvestment());   // Aus Tipp 13
```

Nehmen Sie einmal an, dass eine Funktion, die Sie im Zusammenhang mit Investment-Objekten benutzen wollen, wie folgt aussieht:

```
int daysHeld(const Investment *pi);  // Gibt die Anzahl der Tage zurück,
                                     // die die Anlage gehalten wurde
```

Diese Funktion rufen Sie folgendermaßen auf:

```
int days = daysHeld(plnv);           // Fehler!
```

Das Programm kann nicht kompiliert werden: `daysHeld` will einen rohen `Investment*`-Zeiger, Sie übergeben der Funktion aber ein Objekt vom Typ `tr1::shared_ptr<Investment>`.

Sie brauchen also ein Verfahren, das ein Objekt der RAII-Klasse (in diesem Fall `tr1::shared_ptr`) in die Rohressource konvertiert, die es enthält (z.B. in den zugrunde liegenden `Investment*`-Zeiger). Es gibt zwei gebräuchliche Verfahren, dies zu tun: die explizite Konvertierung und die implizite Konvertierung.

`tr1::shared_ptr` und `auto_ptr` liefern beide eine `get`-Elementfunktion, um eine explizite Konvertierung durchzuführen, also um den Rohzeiger (eine Kopie davon) innerhalb des Objekts für den intelligenten Zeiger zurückzugeben:

```
int days = daysHeld(plnv.get());    // Prima, übergibt den Rohzeiger
                                    // in plnv an daysHeld
```

Wie bei ziemlich allen Klassen für intelligente Zeiger sind auch die Dereferenzierungsoperatoren (Operator `->` und Operator `*`) von `tr1::shared_ptr` und `auto_ptr` überladen, was die implizite Konvertierung der zugrunde liegenden Rohzeiger ermöglicht:

```
class Investment{                       // Ursprungsklasse für Hierarchie
public:                                 // der Investment-Typen
   bool isTaxFree() const;
   ...
};
Investment* createInvestment();         // Fabrik-Funktion

std::tr1::shared_ptr<Investment>        // tr1::shared_ptr soll eine
   pi1(createInvestment());             // Ressource verwalten

bool taxable1 = !(pi1->isTaxFree());    // Zugang zu Ressource per ->
...
std::auto_ptr<Investment>pi2(createInvestment());   // auto_ptr soll
                                                    // eine Ressource
                                                    // verwalten

bool taxable2 = !((*pi2).isTaxFree());  // Zugang zu Ressource per *
...
```

Da es manchmal notwendig ist, die Rohressource innerhalb eines RAII-Objekts zu bekommen, bieten einige Entwickler von RAII-Klassen eine implizite Konvertierungsfunktion an. Schauen Sie z.B. einmal folgende RAII-Klasse für Zeichensätze einer C-API an:

```
FontHandle getFont();             // Aus der C-API - Parameter
                                  // der Einfachheit halber weggelassen
void releaseFont(FontHandle fh);  // Von der C-API
```

```
class Font{                          // RAII-Klasse
public:
   explicit Font(FontHandle fh)      // Ressource anforden
   :f(fh)                            // Benutzt "Übergabe als Wert",
   {}                                // da die C-API dies tut

   ~Font(){releaseFont(f);}          // Ressource freigeben

private:
   FontHandle f;                     // Die Rohressource für die Schriftart
}
```

Angenommen, es gibt eine große C-API für Schriftarten, die sich ausschließlich mit `FontHandle`s beschäftigt, dann werden Sie häufig von `Font`-Objekten in `FontHandle`s konvertieren müssen. Die Klasse `Font` kann eine explizite Konvertierungsfunktion enthalten, z.B. `get`:

```
class Font{
public:
   ...
   FontHandle get() const { return f; }  // Explizite Konvertierungs-
                                         // funktion
   ...
};
```

Unglücklicherweise bringt dies mit sich, dass Kunden jedes Mal `get` aufrufen müssen, wenn sie mit der API kommunizieren möchten:

```
void changeFontSize(FontHandle f, int newSize);  // Von der C-API

Font f(getFont());
int newFontSize;

...

changeFontSize(f.get(), newFontSize);            // Explizite Konvertierung
                                                 // Font in FontHandle
```

Einige Programmierer finden es derart abstoßend, solche Konvertierungen ausdrücklich zu verlangen, dass sie es vermeiden, Klassen dieser Art zu verwenden. Dies wiederum erhöht die Gefahr, dass die Zeichensätze leckschlagen. Aber das ist genau der Grund, warum die Klasse `Font` entworfen wurde.

Die Alternative dazu besteht darin, `Font` eine implizite Konvertierungsfunktion für seine `FontHandle` mitzugeben:

```
class Font{
public:
   ...
   operator FontHandle() const    // Implizite Konvertierungsfunktion
   { return f;}
   ...
};
```

Das vereinfacht den Aufruf in der C-API enorm:

```
Font f(getFont());
int newFontSize;
...
changeFontSize(f, newFontSize);     // Konvertiert implizit
                                    // Font in FontHandle
```

Der Kehrseite ist aber, dass implizite Konvertierungen die Fehleranfälligkeit erhöhen. Zum Beispiel könnte ein Kunde versehentlich ein `FontHandle` erstellen, obwohl ein `Font` beabsichtigt war:

```
Font f1(getFont());
...
FontHandle f2 = f1;              // Hoppla! Ein Font-Objekt sollte
                                 // kopiert werden, stattdessen
                                 // wurde f1 implizit in
                                 // seinen zugrunde
                                 // liegenden FontHandle konvertiert
                                 // und dann kopiert
```

Nun wird in dem Programm ein `FontHandle` von dem `Font`-Objekt `f1` verwaltet, aber dieser `FontHandle` ist für den direkten Gebrauch auch als `f2` zugänglich. Das ist fast nie gut. Wenn `f1` zum Beispiel gelöscht wird, dann wird der Zeichensatz freigegeben und `f2` zeigt auf ein ungültiges Objekt.

Ob Sie nun die explizite Konvertierung einer RAII-Klasse in ihre zugrunde liegende Ressource (z.B. durch die Elementfunktion `get`) zulassen oder die implizite, hängt von den jeweiligen Aufgaben ab, für die die RAII-Klasse entworfen wurde, sowie von den Umständen, unter denen sie benutzt wird. Der beste Entwurf ist wahrscheinlich der, der dem Ratschlag aus Tipp 18 am ehesten gerecht wird: Es muss einfach sein, die Schnittstelle korrekt zu bedienen, und es muss schwierig sein, sie falsch zu bedienen. Oft ist eine explizite Konvertierungsfunktion wie `get` vorzuziehen, da dies die Möglichkeiten zur unbeabsichtigten Typkonvertierung minimiert. Manchmal jedoch macht die Unkompliziertheit der impliziten Konvertierung diesen Weg attraktiver.

Es ist Ihnen wahrscheinlich schon aufgefallen, dass Funktionen, die einen Rohzeiger zurückgeben, dem Prinzip der Kapselung widersprechen. Das mag zwar richtig sein,

ist aber nicht die große Katastrophe, nach der es zuerst aussieht. RAII-Klassen sind nicht dazu da, etwas einzukapseln; sie existieren, um sicherzustellen, dass etwas Bestimmtes geschieht – nämlich die Freigabe von Ressourcen. Falls gewünscht, können Sie die Kapselung der Ressource dieser Primärfunktionalität aufsetzen – dies ist aber nicht notwendig. Darüber hinaus kombinieren einige RAII-Klassen die echte Kapselung der Implementierung mit der sehr lockeren Kapselung der zugrunde liegenden Ressource. Zum Beispiel kapselt `tr1::shared_ptr` seinen kompletten Referenzzählungsmechanismus, bietet aber immer noch leichten Zugang zu seinem Rohzeiger, den der intelligente Zeiger enthält. Wie die meisten gut gestalteten Klassen versteckt diese Klasse vor den Kunden alles, was sie nicht zu sehen brauchen, bietet aber gleichzeitig Zugang zu den Dingen, auf die die Kunden wirklich zugreifen müssen.

Was Sie sich merken sollten

- APIs brauchen häufig Zugang zu Rohressourcen. Deshalb sollte jede RAII-Klasse eine Möglichkeit bieten, an die Ressource zu gelangen, die sie verwaltet.
- Der Zugang kann durch explizite oder implizite Konvertierung geschehen. Im Allgemeinen ist die explizite Konvertierung sicherer, die implizite Konvertierung ist aber für die Kunden bequemer.

3.4 Tipp 16: Verwenden Sie für einander entsprechende Anwendungen von new und delete die gleiche Form

Was ist hier falsch?

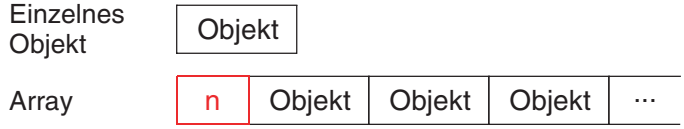
```
Std::string *stringArray = new std::string[100];
...
delete stringArray;
```

Es scheint alles in Ordnung zu sein. Zum `new` gibt es ein `delete`. Aber immer noch ist etwas falsch. Das Programmverhalten ist undefiniert. Mindestens 99 von 100 `string`-Objekten, auf die `stringArray` zeigt, werden nicht ordentlich entfernt, weil ihre Destruktoren wahrscheinlich niemals aufgerufen werden.

Wenn Sie einen *new-Ausdruck* anwenden (z.B. bei der dynamischen Erzeugung eines Objekts durch `new`), geschehen zwei Dinge. Als Erstes wird Speicherplatz reserviert (mit der Funktion `operator new` – siehe Tipps 49 und 51). Zweitens werden ein oder mehrere Konstruktoren für diesen Speicherplatz aufgerufen. Wenn Sie einen *delete-Ausdruck* ausführen (z.B. `delete`), passieren zwei andere Dinge: Ein oder mehrere Destruktoren werden für den Speicherplatz aufgerufen, dann wird dieser Speicherplatz freigegeben (durch die Funktion `operator delete` – siehe Tipp 51). Für `delete`

stellt sich aber die große Frage: Wie viele Objekte befinden sich im Speicher, die zu löschen sind? Die Antwort auf diese Frage bestimmt, wie viele Destruktoren aufgerufen werden müssen.

Tatsächlich ist die Frage aber noch einfacher: Zeigt der Zeiger, der gelöscht werden soll, lediglich auf ein einzelnes Objekt oder auf ein Array von Objekten? Dies ist eine heikle Frage, denn die Speicheranordnung eines einzelnen Objekts ist grundlegend anders als die eines Arrays. Im Besonderen enthält der Speicher des Arrays gewöhnlich die Größe des Arrays, was es `delete` einfacher macht, zu wissen, wie viele Destruktoren aufgerufen werden müssen. Der Speicher für ein einzelnes Objekt besitzt keine solche Information. Sie können sich die verschiedenen Anordnungen in etwa wie folgt vorstellen, wobei `n` die Größe des Felds angibt:



Dies ist natürlich nur ein Beispiel. Die Compiler müssen dies nicht so implementieren, viele tun es jedoch.

Wenn Sie `delete` auf einen Zeiger anwenden, kann es die Größe des Arrays nur erfahren, wenn Sie es ihm mitteilen. Benutzen Sie im Aufruf von `delete` Klammern, dann nimmt `delete` an, dass auf ein Array verwiesen wird. Andernfalls glaubt `delete`, dass auf ein einzelnes Objekt verwiesen wird:

```
std::string *stringPtr1 = new std::string;
std::string *stringPtr2 = new std::string[100];
...
delete stringPtr1;              // Löscht ein einzelnes Objekt
delete[] stringPtr2;            // Löscht ein Feld von Objekten
```

Was würde passieren, wenn Sie die »`[]`«-Form auf `stringPtr1` anwenden? Das Resultat ist unbestimmt, wird aber auf keinen Fall angenehm sein. Ausgehend von der obigen Speicheranordnung würde `delete` etwas Speicher auslesen und diesen Speicherinhalt als Größe des Arrays interpretieren. Dann würde `delete` ebenso viele Destruktoren aufrufen – ungeachtet der Tatsache, dass der folgende Speicherinhalt nicht nur kein Array enthält, sondern wahrscheinlich nicht einmal vom gleichen Datentyp ist wie das Objekt, das eigentlich gelöscht werden soll.

Was würde passieren, wenn Sie die »`[]`«-Form nicht auf `stringPtr2` anwenden? Das Resultat ist auch unbestimmt. Aber Sie können sehen, dass in diesem Fall zu wenige Destruktoren aufgerufen werden. Des Weiteren ist es auch unbestimmt (und manch-

mal auch schädlich) für native Datentypen wie `int`, obwohl solche Datentypen gar keinen Destruktor besitzen.

Die Faustregel ist sehr einfach: Wenn Sie `[]` in einer `new`-Anweisung gebrauchen, müssen Sie `[]` auch in der entsprechenden `delete`-Anweisung benutzen. Verwenden Sie `[]` nicht in der `new`-Anweisung, dann dürfen Sie `[]` auch nicht in der zugehörigen `delete`-Anweisung einsetzen.

Dies ist eine besonders wichtige Regel, die Sie im Kopf haben sollten, wenn Sie eine Klasse schreiben, die einen Zeiger zur dynamischen Speicherreservierung und mehrere Konstruktoren enthält. Dann müssen Sie nämlich aufpassen, dass Sie die *gleiche Form* von `new` in allen Konstruktoren zur Initialisierung der Zeigerelemente verwenden. Tun Sie dies nicht, wie wollen Sie dann wissen, welche Form von `delete` Sie in den Destruktoren verwenden müssen?

Diese Regel ist auch für die `typedef`-Anweisung wichtig. Sie bedeutet nämlich, dass der Autor der `typedef`-Anweisung dokumentieren sollte, welche Form von `delete` angewendet werden muss, wenn `new` verwendet wird, um Objekte des `typedef`-Typs zu erzeugen. Betrachten Sie z.B. folgende Typdefinition:

```
typedef std::string AddressLines[4];   // Die Adresse einer Person
                                       // besteht aus vier Zeilen,
                                       // die jeweils aus Strings
                                       // bestehen
```

Da `AddressLines` ein Feld ist, muss der folgende Gebrauch von `new` ...

```
std::string *pal = new AddressLines;   // Beachten Sie, dass
                                       // "new AddressLines" ebenso
                                       // einen string* liefert wie
                                       // "new string[4]"
```

... mit der Array-Form von `delete` korrespondieren:

```
delete pal;                            // Undefiniert!
delete[] pal;                          // Richtig
```

Um eine solche Verwirrung zu vermeiden, sollten Sie von Typdefinitionen für Arraytypen absehen. Dies dürfte nicht allzu schwierig sein, denn die C++-Standardbibliothek liefert mit `string` und `vector` zwei vorgefertigte Datentypen, die den Bedarf nach dynamisch erzeugten Arrays praktisch auf null senken. In unserem Beispiel könnte `AddressLines` auch als `vector` von `string`s definiert werden, d.h. vom Typ `vector<string>` sein.

Was Sie sich merken sollten

- Wenn Sie `[]` in einer `new`-Anweisung gebrauchen, müssen Sie `[]` auch in der entsprechenden `delete`-Anweisung benutzen. Verwenden Sie `[]` nicht in der `new`-Anweisung, dann dürfen Sie `[]` auch nicht in der zugehörigen `delete`-Anweisung einsetzen.

3.5 Tipp 17: Speichern Sie mit new erzeugte Objekte in intelligenten Zeigern eigenständiger Anweisungen

Stellen Sie sich vor, wir haben eine Funktion, die unsere Verarbeitungspriorität anzeigt, und eine zweite, die ein dynamisch erzeugtes `Widget` nach seiner Priorität verarbeitet.

```
int priority();
void processWidget(std::tr1::shared_ptr<Widget> pw, int priority);
```

Dem Ratschlag folgend, Objekte zur Ressourcenverwaltung zu benutzen (siehe Tipp 13), verwendet `processWidget` einen intelligenten Zeiger (hier `tr1::shared_ptr`) für das dynamisch erzeugte `Widget`, das die Funktion abarbeiten soll.

Betrachten Sie nun folgenden Aufruf von `processWidget`:

```
processWidget(new Widget, priority());
```

Warten Sie! Vergessen Sie diesen Aufruf. Er lässt sich nicht kompilieren. Der Konstruktor von `tr1::shared_ptr`, der einen Rohzeiger verlangt, ist explizit, also gibt es keine implizite Konvertierung vom Rohzeiger, der vom Ausdruck »`new Widget`« geliefert wird, zu `tr1::shared_ptr`, der von `processWidget` verlangt wird. Das Folgende jedoch wird kompiliert:

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget), priority());
```

Obwohl wir hier überall Objekte zur Ressourcenverwaltung benutzen, führt dieser Aufruf überraschenderweise zu einem Ressourcenleck. Es ist aufschlussreich, zu sehen, wie das geschieht.

Bevor Compiler die Funktion `processWidget` aufrufen können, müssen sie zuerst die Argumente auswerten, die als Parameter übergeben werden. Das zweite Argument ist lediglich ein Aufruf der Funktion `priority`, aber das erste (»`std::tr1::shared_ptr<Widget>(new Widget)`«) besteht aus zwei Teilen:

- Ausführen des Ausdrucks »`new Widget`« und
- Aufruf des Konstruktors `tr1::shared_ptr`.

Bevor `processWidget` aufgerufen werden kann, muss ein Compiler also für die folgenden drei Vorgänge Code generieren:

- Aufruf von `priority`,
- Ausführung von »`new Widget`« und
- Aufruf des Konstruktors `tr1::shared_ptr`.

C++-Compilern wird eine gewisse Freiheit eingestanden, in welcher Reihenfolge sie diese Vorgänge abarbeiten wollen. (Das ist anders als bei anderen Sprachen wie z.B. Java und C#, bei denen Funktionsparameter immer in einer vorgegebenen Reihenfolge ausgewertet werden.) Der Ausdruck »`new Widget`« muss ausgeführt werden, bevor der Konstruktor `tr1::shared_ptr` aufgerufen werden kann, weil das Ergebnis des Ausdrucks dem Konstruktor als Argument übergeben wird. Der Aufruf von `priority` kann als Erstes, Zweites oder Drittes erfolgen. Sollte sich der Compiler entschließen, `priority` als Zweites aufzurufen (was ihn dazu befähigt, effizienteren Code zu erzeugen), dann erhalten wir folgende Reihenfolge:

1. Ausführung von »`new Widget`«,
2. Aufruf von `priority`,
3. Aufruf des Konstruktors `tr1::shared_ptr`.

Bedenken Sie aber, was passiert, wenn der Aufruf von `priority` eine Ausnahme aufwirft. In diesem Fall geht der Zeiger von »`new Widget`« verloren, weil er nicht im Zeiger `tr1::shared_ptr` gespeichert wird, den wir zum Abdichten des Ressourcenlecks eingeplant hatten. Ein Leck im Aufruf von `processWidget` kann auftauchen, wenn in der Zeit zwischen der Erzeugung der Ressource (durch »`new Widget`«) und ihrer Übergabe an das Objekt zur Ressourcenverwaltung eine Ausnahme auftritt.

Solche Probleme zu vermeiden, ist recht einfach: Benutzen Sie einen separaten Ausdruck, um das `Widget` zu erzeugen, und speichern Sie es in einem intelligenten Zeiger. Übergeben Sie letzteren dann an `processWidget`:

```
std::tr1::shared_ptr<Widget>pw(new Widget);  // Speichern Sie das neu
                                             // erzeugte Objekt in
                                             // einem intelligenten Zeiger
                                             // in einem separaten
                                             // Ausdruck

processWidget(pw, priority());               // Dieser Aufruf leckt nicht
```

Dies funktioniert, weil Compilern weniger Spielraum gegeben wird, Anweisungen untereinander zu ordnen, als den Ablauf innerhalb der Anweisungen zu ändern. In diesem neu gestalteten Code befinden sich der Ausdruck »`new Widget`« und der Aufruf

des Konstruktors `tr1::shared_ptr` in einer anderen Anweisung als der Aufruf von `priority`, was den Compiler daran hindert, den Aufruf von `priority` dazwischenzuschieben.

Was Sie sich merken sollten

- Speichern Sie neu erzeugte Objekte in intelligenten Zeigern in separaten Anweisungen. Anderenfalls kann es zu tückischen Ressourcenlecks kommen, wenn Ausnahmen aufgeworfen werden.