ADDISON-WESLEY

PEARSON Education

Konstruktoren, Destruktoren und Zuweisungsoperatoren

Fast jede Klasse, die Sie schreiben werden, wird einen oder mehrere Konstruktoren, einen Destruktor und einen Zuweisungsoperator besitzen. Das ist kein großes Wunder, denn dies sind Ihre Butterbrot-Funktionen, diejenigen Funktionen, die die grundlegenden Operationen für das Erzeugen und Initialisieren eines Objektes kontrollieren, für das Zerstören und ordnungsgemäße Abbauen eines Objektes und der Zuweisung eines neuen Wertes an das Objekt. Ein Fehler in diesen Funktionen wird weitreichende und sehr unangenehme Auswirkungen auf Ihre gesamte Klasse haben. Deshalb ist es von größter Wichtigkeit, daß Sie bei diesen Funktionen alles richtig machen. In diesem Abschnitt zeige ich Ihnen, wie Sie diese Funktionen, die das Rückgrat einer gut gestalteten Klasse darstellen, zusammenstellen können.

Lektion 11

Deklarieren Sie einen Copy-Konstruktor und einen Zuweisungsoperator für Klassen mit dynamisch allozierten Speicher

Betrachten Sie die folgende Klasse, die String-Objekte repräsentieren soll:

```
// eine schlecht entworfene String-Klasse
class String {
public:
  String(const char *value = 0);
```

```
  ~String();

private:
  char *data;
};

String::String(const char *value)
{
  if (value) {
    data = new char[strlen(value) + 1];
    strcpy(data, value);
  }
  else {
    data = new char[1];
    *data = '\0';
  }
}

inline String::~String() { delete [] data; }
```

Beachten Sie, daß in dieser Klasse kein Zuweisungsoperator und kein Copy-Konstruktor deklariert wurde. Wie Sie sehen werden, zieht dies einige unangenehme Konsequenzen nach sich.

Wenn Sie folgende Objekte definieren:

```
String a("Hello");
String b("World");
```

ergibt sich folgende Situation:

z01_03.eps

Innerhalb des Objektes a befindet sich ein Zeiger auf einen Speicherbereich, der den String »Hello« enthält. Davon unabhängig existiert ein Objekt b, daß einen Zeiger auf den String »World« enthält. Wenn Sie nun die Zuweisung

```
b = a;
```

durchführen, dann gibt es keinen benutzerdefinierten Zuweisungsoperator und kann demzufolge auch nicht aufgerufen werden. Statt dessen generiert C++ einen Aufruf für den Standard-Zuweisungsoperator (siehe Lektion 45). Dieser Standard-Zuweisungsoperator führt für alle Elemente von b eine Zuweisung mit den Werten von a aus. Für die Zeiger (a.data und b.data) bedeutet das nichts weiter als eine bitweise Kopie. Das Ergebnis dieser Zuweisung sieht so aus:

z01_04.eps

So wie die Dinge stehen, gibt es hierbei mindestens zwei Probleme. Als erstes ist der Speicher, auf den b zuvor gezeigt hat, niemals gelöscht worden und somit für immer verloren. Das ist ein klassisches Beispiel dafür, wie eine Speicherlücke entstehen kann. Zweitens enthalten sowohl a als auch b nun einen Zeiger, der auf den gleichen String zeigt. Wenn eines dieser beiden Objekte seinen Sichtbarkeitsbereich verläßt, wird dessen Destruktor den Speicherbereich freigeben, auf den der andere Zeiger immer noch zeigt. Dazu ein Beispiel:

```
String a("Hello");     // definiere und initialisiere a

{                      // öffnen eines neuen
                       // Sichtbarkeitsbereichs
  String b("World");   // definiere und initialisiere b

  ...

  b = a;               // führt den Standard-Zuweisungs-
                       // operator aus, der Speicher von b
                       // geht verloren

}                      // Schließen des Sichtbarkeits-
                       // bereiches führt zum Aufruf
                       // des Destruktors von b

String c = a;          // c.data ist nicht definiert, da
                       // a.data schon gelöscht wurde
```

Die letzte Anweisung in diesem Beispiel ruft den Copy-Konstruktor auf, der ebenfalls nicht für diese Klasse definiert ist und somit auf die gleiche Art und Weise wie der Zuweisungsoperator von C++ generiert wird (siehe wiederum Lektion 45), und zwar mit demselben Verhalten: bitweise Kopie des übergebenen Zeigers. Das führt zu dem gleichen Problem, außer daß keine Speicherlücke entsteht, da das zu initialisierende Objekt noch nicht auf irgendwelchen allozierten Speicher zeigen kann.

Für das obige Beispiel heißt das zum Beispiel, es entsteht keine Speicherlücke, wenn c.data mit dem Wert von a.data initialisiert wird, da c.data noch nirgendwo hin zeigt. Wie dem auch sei, nachdem c mit a initialisiert wurde, zeigen sowohl c.data als auch a.data auf den gleichen Speicherbereich, der somit zweimal gelöscht wird: Einmal, wenn c zerstört wird, und noch einmal, wenn a zerstört wird.

Der Fall mit dem Copy-Konstruktor ist trotzdem noch etwas anders gelagert, als der mit dem Zuweisungsoperator, da er Sie noch auf eine andere Weise einholen kann: Parame-

terübergabe via Wert. Natürlich zeigt die Lektion 22, daß man Objekte nur in den seltensten Fällen via Wert übergeben sollte, betrachten Sie aber dennoch einmal folgendes:

```
void doNothing(String localString) {}

String s = "The Truth Is Out There";

doNothing(s);
```

Alles sieht so harmlos aus, da localString aber via Wert übergeben wird, muß localString durch den (Standard-)Copy-Konstruktor mit s initialisiert werden. Folglich besitzt localString eine Kopie des *Zeigers*, der sich innerhalb von s befindet. Wenn doNothing mit der Ausführung fertig ist, verläßt localString den Sichtbarkeitsbereich der Funktion und somit wird der Destruktor für localString aufgerufen. Das Ergebnis ist Ihnen schon bekannt: s enthält einen Zeiger auf einen Speicherbereich, den der Destruktor von localString bereits gelöscht hat.

Übrigens ist das Verhalten, wenn delete für einen Zeiger aufgerufen wird, der bereits gelöscht wurde, nicht definiert, so daß selbst wenn s nicht noch mal benutzt wird, es trotzdem zu Problemen kommen könnte, wenn s den Sichtbarkeitsbereich verläßt.

Die Lösung all dieser Zeiger-Aliasing-Probleme besteht darin, daß Sie Ihre eigenen Versionen des Copy-Konstruktors und des Zuweisungsoperators schreiben, sobald Ihre Klasse auch nur einen einzigen Zeiger besitzt. Innerhalb dieser Funktionen können Sie entweder die Datenstrukturen, auf die gezeigt wird, kopieren, so daß jedes Objekt seine eigene Kopie hat, oder Sie können eine Art Referenzzähler implementieren, der darüber Buch führt, wie viele Zeiger gerade auf eine Datenstruktur zeigen. Der Ansatz mit dem Referenzzähler ist zwar komplizierter und erfordert auch zusätzliche Arbeit an den Konstruktoren und dem Destruktor, kann aber in einigen (jedoch bei weitem nicht allen) Programmen beträchtlich Speicherplatz sparen und die Ausführungsgeschwindigkeit wesentlich erhöhen.

Für einige Klassen bringt es mehr Ärger als Nutzen, wenn Sie den Copy-Konstruktor und den Zuweisungsoperator implementieren, insbesondere wenn Sie Grund zu der Annahme haben, daß Benutzer Ihrer Klasse gar keine Kopien erstellen oder Zuweisungen machen wollen. Die obigen Beispiele zeigen, daß der Verzicht auf diese Elementfunktionen ein sehr schlechtes Klassendesign darstellt, was machen Sie also, wenn das Schreiben dieser Funktion auch nicht besonders sinnvoll ist? Ganz einfach: Sie folgen der Regel aus der Überschrift dieser Lektion. Sie *deklarieren* diese Funktionen (am besten private), definieren (sprich implementieren) sie jedoch nicht. Damit verhindern Sie, daß Benutzer Ihrer Klasse diese Funktionen aufrufen können und daß der Compiler sie automatisch generiert. Für Einzelheiten dieses praktischen Tricks, siehe Lektion 27.

Eine Sache zu der in dieser Lektion benutzten String-Klasse möchte ich noch loswerden. Im Konstruktor habe ich sorgfältig [] für beide new-Aufrufe verwendet, obwohl ich an

der einen Stelle nur ein einziges char brauche. Wie in Lektion 5 beschrieben, ist es unbedingt erforderlich, daß korrespondierende new- und delete-Aufrufe in derselben Form ausgeführt werden, weshalb meine new-Aufrufe alle konsistent sind. Das ist etwas, das Sie nie vergessen sollten. Achten Sie *immer* darauf, daß Sie delete mit [] aufrufen, wenn und nur wenn Sie new mit [] aufgerufen haben.

Lektion 12

Bevorzugen Sie Initialisierung gegenüber Zuweisung im Konstruktor

Betrachten Sie ein Template, das Klassen generiert, die es erlauben, Namen mit einem Zeiger auf irgendeinen Typ T zu assoziieren:

```
template<class T>
class NamedPtr {
public:
  NamedPtr(const string& initName, T *initPtr);
  ...

private:
  string name;
  T *ptr;
};
```

(In Hinblick auf die Aliasing-Probleme, die durch eine Zuweisung oder den Aufruf eines Copy-Konstruktors bei Klassen mit Zeiger-Datenelementen auftreten können (siehe Lektion 11), sollten Sie sich fragen, ob das Klassen-Template NamedPtr diese Funktionen nicht implementieren sollte. Kleiner Tip: es sollte – siehe Lektion 27.)

Wenn Sie den Konstruktor für NamedPtr schreiben, müssen Sie die Werte der übergebenen Parameter auf die entsprechenden Datenelemente transferieren. Dafür gibt es zwei Möglichkeiten. Die erste ist die Initialisierungsliste:

```
template<class T>
NamedPtr(T)::NamedPtr(const string& initName, T *initPtr)
: name(initName), ptr(initPtr)
{}
```

Die zweite ist, innerhalb des Konstruktorrumpfes Zuweisungen zu machen:

```
template<class T>
NamedPtr(T)::NamedPtr(const string& initName, T *initPtr)
```

```
{
  name = initName;
  ptr = initPtr;
}
```

Zwischen diesen beiden Möglichkeiten bestehen erhebliche Unterschiede.

Vom rein pragmatischen Gesichtspunkt aus betrachtet, gibt es Situationen, in denen die Initialisierungsliste benutzt werden *muß*. Insbesondere können Konstanten und Referenzen *nur* initialisiert und niemals zugewiesen werden. Wenn Sie also entscheiden, daß ein NamedPtr<T>-Objekt niemals seinen Namen oder seinen Zeiger ändern kann, werden Sie vielleicht den Ratschlag aus Lektion 21 befolgen und die Datenelemente als const deklarieren:

```
template<class T>
class NamedPtr {
public:
  NamedPtr(const string& initName, T *initPtr);
  ...

private:
  const string name;
  T * const ptr;
};
```

Diese Klassendefinition *erfordert*, daß Sie die Initialisierungsliste einsetzen, denn const-Datenelemente können ausschließlich initialisiert werden, niemals zugewiesen.

Sie würden ein völlig anderes Verhalten erhalten, wenn Sie entscheiden, daß ein NamedPtr<T>-Objekt eine *Referenz* auf einen existierenden Namen enthalten soll. Aber auch in diesem Fall müssen Sie die Initialisierungsliste benutzen, wenn Sie die Referenz in Ihrem Konstruktor initialisieren wollen. Sie können auch beides kombinieren, womit Sie NamedPtr<T>-Objekte erhalten, die einen Read-only-Zugriff (Nur-Lese-Zugriff) auf die Namen haben, die außerhalb der Klasse modifiziert werden könnten:

```
template<class T>
class NamedPtr {
public:
  NamedPtr(const string& initName, T *initPtr);
  ...

private:
  const string& name;   // muß durch die Initialisierungs-
                        // liste initialisiert werden
```

```
  T * const ptr;       // dito
};
```

Das ursprüngliche Klassen-Template enthielt jedoch keine const- oder Referenz-Datenelemente. Trotzdem sollten Sie die Initialisierungsliste gegenüber Zuweisungen im Konstruktorrumpf bevorzugen. Diesmal ist der Grund Effizienz. Wenn Sie die Initialisierungsliste benutzen, wird nur eine einzige string-Elementfunktion aufgerufen. Führen Sie hingegen eine Zuweisung im Konstruktorrumpf aus, werden zwei string-Elementfunktionen aufgerufen. Um dies zu verstehen, betrachten Sie was passiert, wenn Sie ein NamedPtr<T>-Objekt deklarieren.

Die Konstruktion eines solchen Objektes geschieht in zwei Schritten:

Initialisierung der Datenelemente (siehe auch Lektion 13).

Ausführen des Rumpfes des Konstruktors, der aufgerufen wurde.

(Für Objekte mit Basisklassen geschieht die Initialisierung der Basisklassen-Datenelemente und die Ausführung des Basisklassen-Konstruktors vor den beiden Schritten für die abgeleitete Klasse.)

Für die NamedPtr-Klasse bedeutet dies, daß, bevor der NamedPtr-Konstruktor überhaupt aufgerufen wird, *immer* schon ein Konstruktor für die Initialisierung des string-Elements name aufgerufen wurde. Die einzige Frage dabei ist, *welcher* string-Konstruktor dabei aufgerufen wurde?

Das hängt von der Initialisierungsliste des NamedPtr-Konstruktors ab. Wenn Sie kein Initialisierungsargument für name spezifizieren, wird der Default-Konstruktor aufgerufen, Wenn Sie dann später eine Zuweisung für name innerhalb des Konstruktors für NamedPtr ausführen, rufen Sie string::operator= auf. Es finden also zwei Aufrufe von string-Elementfunktionen statt: einer für den Default-Konstruktor und einer für den Zuweisungsoperator.

Wenn Sie der anderen Seite die Initialisierungsliste benutzen, um anzugeben, daß name mit initName initialisiert werden soll, wird name durch den Aufruf des Copy-Konstruktors der string-Klasse initialisiert – mit dem Aufwand von nur einem Funktionsaufruf.

Selbst für diese eine ordinäre string-Klasse, kann sich der Aufwand eines unnötigen Funktionsaufrufes durchaus bemerkbar machen. Und wenn Klassen größer und komplexer werden, werden es auch ihre Konstruktoren und somit der Aufwand zum Konstruieren eines Objektes. Wenn Sie es sich angewöhnen, die Initialisierungsliste einzusetzen, wann immer Sie können, erfüllen Sie nicht nur eine Bedingung für const-Datenelemente, sondern verkleinern auch die Gefahr, Datenelemente auf eine ineffiziente Art und Weise zu initialisieren.

Mit anderen Worten, Initialisierung mit Hilfe der Initialisierungsliste ist *immer* legal, *niemals* ineffizienter als eine Zuweisung innerhalb des Konstruktorrumpfes und häufig sogar *effizienter*. Außerdem vereinfachen Sie damit die Wartung einer Klasse, da bei

einer späteren Änderung eines Datenelements, das den Gebrauch einer Initialisierungsliste erforderlich macht, nichts weiter ändern müssen.

Es gibt allerdings auch Situationen, wo es Sinn macht, die Datenelemente einer Klasse durch Zuweisung und nicht durch die Initialisierungsliste zu initialisieren. Nämlich wenn Sie eine große Anzahl Datenelemente *eingebauter Typen* haben und diese alle mit dem gleichen Wert im Konstruktor initialisieren möchten. Die folgende Klasse käme dafür zum Beispiel in Frage:

```
class ManyDataMbrs {
public:
  // Default-Konstruktor
  ManyDataMbrs();

  // Copy-Konstruktor
  ManyDataMbrs(const ManyDataMbrs& x);

private:
  int a, b, c, d, e, f, g, h;
  double i, j, k, l, m;
};
```

Angenommen, Sie wollen alle ints mit 1 und alle doubles mit 0 initialisieren, auch wenn der Copy-Konstruktor verwendet wird. Mit einer Initialisierungsliste würde das so aussehen:

```
ManyDataMbrs::ManyDataMbrs()
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
  j(0), k(0), l(0), m(0)
{ ... }

ManyDataMbrs::ManyDataMbrs(const ManyDataMbrs& x)
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
  j(0), k(0), l(0), m(0)
{ ... }
```

Das ist mehr als nur unbequeme Schreibarbeit, sondern auf kurze Sicht fehleranfällig und auf lange Sicht schlecht wartbar.

Sie können sich hier die Tatsache zunutze machen, daß es zwischen der Initialisierung und der Zuweisung für (nicht-const und nicht-Referenz-) Objekte eingebauter Typen keinen Unterschied gibt, so daß Sie problemlose die elementweise Initialisierungsliste mit einem Aufruf einer gemeinsamen Initialisierungsfunktion ersetzen können:

```
class ManyDataMbrs {
public:
```

```
  // Default-Konstruktor
  ManyDataMbrs();

  // Copy-Konstruktor
  //ManyDataMbrs(const ManyDataMbrs& x);

private:
  int a, b, c, d, e, f, g, h;
  double i, j, k, l, m;

  void init();  // Initialisierungsfunktion
                // für die Datenelemente
};

void ManyDataMbrs::init()
{
  a = b = c = d = e = f = g = h = 1;
  i = j = k = l = m = 0;
}

ManyDataMbrs::ManyDataMbrs()
{
  init();

...

}

ManyDataMbrs::ManyDataMbrs(const ManyDataMbrs& x)
{
  init();

  ...

}
```

Da diese Initialisierungsfunktion ein Implementationsdetail der Klasse ist, deklarieren Sie sie achtsam als private, nicht wahr?

Beachten Sie, daß static-Datenelemente *niemals* im Konstruktor einer Klasse initialisiert werden sollten. Statische Datenelemente werden nur einmal in einem Programm initialisiert, weshalb es keinen Sinn macht, sie jedesmal, wenn ein Objekt dieser Klasse erzeugt wird, wieder zu »initialisieren«. Das wäre außerdem auch sehr ineffizient: Warum den Aufwand für die Initialisierung eines Objektes mehrmals betreiben? Übrigens ist die

Initialisierung statischer Datenelemente so verschieden von der Initialisierung ihrer nicht-statischen Gegenstücke, daß eine ganze Lektion – Lektion 47 – diesem Thema gewidmet ist.

Lektion 13

Führen Sie die Datenelemente in der Initialisierungsliste in der Reihenfolge ihrer Deklaration auf

Zufriedene Pascal- und Ada-Programmierer sehnen sich in C++ oft nach der Möglichkeit beliebige Array-Grenzen definieren zu können, d. h. zum Beispiel von 10 bis 20 anstatt von 0 bis 10. Altgediente C-Programmierer bestehen darauf, daß jeder, der etwas auf sich hält, beim Zählen mit 0 anfängt, aber in C++ ist es ganz einfach, die begin-end-Menschen zufriedenzustellen. Alles, was Sie tun müssen, ist ein Array-Klassen-Template zu definieren:

```
template<class T>
class Array {
public:
  Array(int lowBound, int highBound);
  ...

private:
  vector(T) data;       // die Array-Daten werden in
                        // vector-Objekt abgespeichert,
                        // nähere Informationen dazu
                        // finden Sie in Lektion 49

  unsigned int size;    // Anzahl der Elemente im Array

  int lBound, hBound;   // untere und obere Grenze
};

template<class T>
Array(T)::Array(int lowBound, int highBound)
: size(highBound – lowBound + 1),
  lBound(lowBound), hBound(highBound),
  data(size)
{}
```

Ein kommerziell ausgereifter Konstruktor würde noch die übergebenen Parameter überprüfen, um sicherzustellen, daß highBound mindestens genausogroß wie lowBound ist, aber hier gibt es einen viel gemeineren Fehler: Selbst wenn Sie die Array-Grenzen völlig

richtig übergeben, haben Sie absolut keine Ahnung davon, wie viele Elemente data enthält.

»Wie kann das sein?« höre ich Sie rufen. »Ich habe doch size ordnungsgemäß initialisiert, bevor ich es an den Konstruktor von vector übergeben habe!« Leider haben Sie nicht – Sie haben es nur versucht. Die Spielregeln sind aber die, daß *Datenelemente in der Reihenfolge ihrer Deklaration* in der Klasse initialisiert werden. Die Reihenfolge, in der sie in der Initialisierungsliste stehen, spielt dabei nicht die kleinste Rolle. In den Klassen, die von Ihrem Array-Template generiert werden, wird immer zuerst data initialisiert, danach size, lBound und zum Schluß hBound. Immer.

So widernatürlich das auch zu sein scheint, gibt es dafür einen Grund. Betrachten Sie folgendes Szenario:

```
class Wacko {
public:
  Wacko(const char *s): s1(s), s2(0) {}
  Wacko(const Wacko& rhs): s2(rhs.s1), s1(0) {}

private:
  string s1, s2;
};

Wacko w1 = "Hello world!";
Wacko w2 = w1;
```

Wenn die Datenelemente in der Reihenfolge initialisiert werden würden, wie sie in der Initialisierungsliste stehen, würden die Datenelemente von w1 und w2 in unterschiedlicher Reihenfolge initialisiert werden. Erinnern Sie sich daran, daß die Destruktoren für die Datenelemente eines Objektes immer in der umgekehrten Reihenfolge ihrer Konstruktoren aufgerufen werden. Wäre die Initialisierung in der Reihenfolge des Auftretens in der Initialisierungsliste erlaubt, würde das bedeuten, daß der Compiler für *jedes Objekt* über die Reihenfolge, in der die Datenelemente dieses Objektes initialisiert wurden, Buch führen muß, um sicherzustellen, daß die Destruktoren in der richtigen Reihenfolge aufgerufen werden. Eine sehr kostspielige Angelegenheit. Um diesen zusätzlichen Aufwand zu vermeiden, ist die Reihenfolge für die Konstruktion und Destruktion für alle Objekte eines gegebenen Typs die gleiche, und die Reihenfolge der Datenelemente in der Initialisierungsliste wird ignoriert.

Wenn Sie es wirklich ganz genau nehmen wollen, werden eigentlich nur nicht-statische Datenelemente entsprechend dieser Regel initialisiert. Statische Datenelemente verhalten sich wie globale oder namespace-Objekte, die nur einmal initialisiert werden – weitere Details hierzu finden Sie in Lektion 47.

Weiterhin werden die Datenelemente einer Basisklasse vor den Datenelementen der abgeleiteten Klasse initialisiert. Wenn Sie also Vererbung benutzen, sollten Sie die Datenelemente der Basisklasse an den Anfang der Initialisierungsliste stellen.

(Wenn Sie *Mehrfach*vererbung einsetzen, werden die Datenelemente der Basisklassen in der Reihenfolge initialisiert, in der die einzelnen Basisklassen *geerbt* wurden. Die Reihenfolge in der Initialisierungsliste wird wiederum ignoriert. Wie dem auch sei, wenn Sie wirklich Mehrfachvererbung benutzen, müssen Sie sich um ganz andere, wichtigere Dinge Sorgen machen. Falls nicht, kann ich Ihnen nur wärmstens Lektion 43 ans Herz legen, die Ihnen die Aspekte von Mehrfachvererbung zeigt, die Sie unbedingt beachten sollten.)

Die Schlußworte für diese Lektion sind die folgenden: Wenn Sie verstehen wollen, was bei der Initialisierung Ihrer Objekte wirklich passiert, sollten Sie die Datenelemente in der Initialisierungsliste in der Reihenfolge anordnen, in der sie in der Klasse deklariert sind.

Lektion 14

Stellen Sie sicher, daß Basisklassen einen virtuellen Destruktor haben

Manchmal ist es für eine Klasse ganz praktisch, wenn Sie weiß, wie viele Objekte ihres Typs existieren. Die geradlinigste Möglichkeit dafür ist ein statisches Datenelement, das die erzeugten Objekte zählt. Dieses Datenelement wird mit 0 initialisiert, in den Konstruktoren der Klasse inkrementiert und im Klassendestruktor dekrementiert.

Vielleicht haben Sie eine militärische Anwendung ins Auge gefaßt, in der eine Klasse zum Repräsentieren der feindlichen Ziele so aussehen könnte:

```
class EnemyTarget {
public:
  EnemyTarget() { ++numTargets; }
  EnemyTarget(const EnemyTarget&) { ++numTargets; }
  ~EnemyTarget() { --numTargets; }

  static unsigned int numberOfTargets()
  { return numTargets; }

  virtual bool destroy();          // Zerstörungsversuch
                                   // des EnemyTarget-
                                   // Objektes
                                   // erfolgreich?
```

```
private:
  static unsigned int numTargets;   // Objektzähler
};

// statische Datenelemente müssen außerhalb
// einer Klasse definiert werden,
// sie werden standardmäßig mit 0 initialisiert
unsigned int EnemyTarget::numTargets;
```

Mit dieser Klasse werden Sie zwar keinen Vertrag mit dem Ministerium für Verteidigung bekommen, aber für unsere Bedürfnisse reicht sie völlig aus. Die Anforderungen des Ministeriums für Verteidigung dürften um einiges höher sein, was wir allerdings nur hoffen können...

Nehmen wir an, daß ein feindlicher Panzer ein spezielles Feindobjekt ist. Dies modellieren Sie völlig natürlich (siehe Lektion 35) mit einer öffentlich von EnemyTarget abgeleiteten Klasse. Da Sie sowohl an der Anzahl feindlicher Panzer als auch an der Gesamtanzahl feindlicher Objekte interessiert sind, verwenden Sie in der abgeleiteten Klasse den gleichen Kunstgriff wie in der Basisklasse:

```
class EnemyTank: public EnemyTarget {
public:
  EnemyTank() { ++numTanks; }

  EnemyTank(const EnemyTank& rhs)
  : EnemyTarget(rhs)
  { ++numTanks; }

  ~EnemyTank() { --numTanks; }

  static unsigned int numberOfTanks()
  { return numTanks; }

  virtual bool destroy();

private:
  static unsigned int numTanks;   // Objektzähler für
                                  // die Panzer
};
```

Lassen Sie uns nun annehmen, daß Sie irgendwo in Ihrer Applikation ein EnemyTank-Objekt dynamisch mit new erzeugen und es später mit delete wieder loswerden:

```
EnemyTarget *targetPtr = new EnemyTank;

...

delete targetPtr;
```

Alles, was Sie bis jetzt gemacht haben, scheint völlig in Ordnung zu sein. Beide Klassen machen im Destruktor rückgängig, was Sie im Konstruktor angestellt haben, und Ihr Programm, in dem Sie sorgfältig das Objekt, welches Sie mit new erzeugt haben, mittels delete wieder löschen, nachdem Sie es nicht mehr brauchen, ist bestimmt auch richtig. Und trotzdem geschieht hier etwas sehr Beunruhigendes: Das Programmverhalten ist *undefiniert* – Sie können sich nicht sicher sein, *was* passieren wird.

Der C++-Standard trifft zu diesem Punkt eine ungewöhnlich klare Aussage: Wenn Sie versuchen, ein Objekt einer abgeleiteten Klasse durch einen Basisklassen-Zeiger zu löschen und diese Basisklasse *keinen virtuellen* Destruktor besitzt (was der Fall für EnemyTarget ist), dann ist das Ergebnis dieser Operation nicht definiert. Das heißt, Ihre Compiler könnten jeden beliebigen Code generieren, den sie wollen – Code zum Formatieren Ihrer Festplatte, um zweideutige E-Mails an Ihren Chef zu schicken, der Konkurrenz den Quellcode Ihrer Programme zu faxen oder was auch immer.

(Was zur Laufzeit meistens passiert ist, daß der Destruktor der abgeleiteten Klasse nicht aufgerufen wird. In diesem Beispiel würde das bedeuten, daß numTanks beim Löschen von targetPtr nicht wieder angepaßt wird. Die Anzahl der Panzer wäre damit falsch, eine ziemlich unvorteilhafte Aussicht für Soldaten, die auf genaue Informationen über das Schlachtfeld angewiesen sind.)

Um dieses Problem zu vermeiden, müssen Sie einfach nur den Destruktor von EnemyTarget *virtuell* machen. Die virtual-Deklaration des Destruktor führt zu einem wohldefinierten Verhalten, und es passiert genau das, was Sie wollen: Sowohl der Destruktor von EnemyTank als auch der Destruktor von EnemyTarget wird aufgerufen, bevor der Speicher für das Objekt deallociert wird.

Jetzt beinhaltet EnemyTarget eine virtuelle Funktion, was für Basisklassen im allgemeinen immer der Fall ist. Schließlich ist der Zweck einer virtuellen Funktion, das Verhalten abgeleiteter Klassen anzupassen (siehe Lektion 36), weshalb eigentlich alle Basisklassen virtuelle Funktionen enthalten.

Wenn eine Klasse *keine* virtuellen Funktionen enthält, ist das oft ein Zeichen dafür, daß diese Klasse nicht als Basisklasse gedacht ist. Wenn eine Klasse nicht als Basisklasse eingesetzt werden soll, ist es auch keine gute Idee, den Destruktor dieser Klasse als virtual zu deklarieren. Betrachten Sie das folgende Beispiel, welches auf einem Beispiel aus dem ARM (siehe Lektion 50) basiert:

```
// Klasse für die Repräsentation eines 2D-Punktes
class Point {
```

```
public:
  Point(short int xCoord, short int yCoord);
  ~Point();

private:
  short int x, y;
};
```

Wenn ein short int 16 Bit belegt, paßt ein Point-Objekt in ein 32-Bit-Register. Weiterhin kann ein Point-Objekt als ein 32-Bit-Wert an eine Funktion, die in einer anderen Sprache wie C oder FORTRAN geschrieben wurde, übergeben werden. Wenn Sie den Destruktor von Point virtuell machen, ändert sich diese Situation gewaltig.

Die Implementation virtueller Funktionen erfordert, daß Objekte einige Zusatzinformationen mit sich herumtragen, die zur Laufzeit herangezogen werden können, um zu entscheiden, welche virtuelle Funktion für dieses Objekt aufgerufen werden muß. Bei den meisten Compilern, ist diese Zusatzinformation ein Zeiger, der auch kurz als vptr (»virtual table pointer«) bezeichnet wird. vptr zeigt auf ein Array mit Zeigern auf Funktionen, das als vtbl (»virtual table«) bezeichnet wird. Zu jeder Klasse mit virtuellen Funktionen gehört eine solche vtbl. Wenn eine virtuelle Funktion für ein Objekt aufgerufen wird, bestimmt sich der eigentliche Funktionsaufruf daraus, daß dem vptr-Zeiger des Objektes zu einer vtbl gefolgt wird, in der dann der Zeiger auf die gefragte Funktion ermittelt wird.

Die Details dazu, wie die Implementation virtueller Funktionen aussieht, sind aber gar nicht so wichtig. Was *wirklich* wichtig ist, ist die Tatsache, daß, wenn die Point-Klasse eine virtuelle Funktion enthält, sich die Größe dieses Typs implizit *verdoppelt*, von zwei 16-Bit-Zahlen auf zwei 16-Bit-Zahlen und dem 32-Bit-Zeiger vptr! Ein Point-Objekt wird nicht länger in einen 32-Bit-Register passen und ein Point-Objekt in C++ sieht nicht mehr so aus, wie die gleiche Struktur, wenn Sie in einer anderen Sprache wie C deklariert worden wäre, da diese »Fremdsprachen« kein Gegenstück zu vptr besitzen. Daraus folgt, daß Sie Point-Objekte auch nicht länger an Funktionen übergeben können, die in anderen Sprachen geschrieben sind, wenn Sie nicht den vptr-Zeiger explizit ausgleichen, was wiederum ein Implementationsdetail und damit nicht portabel ist.

Die Schlußfolgerung aus alldem ist, daß unbegründetes Deklarieren aller Destruktoren als virtual genauso falsch ist wie die Vorgehensweise, Destruktoren niemals als virtual zu deklarieren. In der Praxis fassen viele Leute dies wie folgt zusammen: Deklarieren Sie einen virtuellen Destruktor für eine Klasse, wenn und nur wenn diese Klasse wenigstens eine virtuelle Funktion besitzt.

Das ist eine gute Regel, die auch meistens richtig ist, aber leider kann Sie das Problem mit dem nicht-virtuellen Destruktor auch dann einholen, wenn es in der Klasse keine virtuelle Funktion gibt. In Lektion 13 wird zum Beispiel ein Klassen-Template für Arrays mit benutzerdefinierten Grenzen betrachtet. Nehmen wir einmal an, daß Sie ein

abgeleitetes Klassen-Template für Klassen, die benannte Arrays zur Verfügung stellen (d. h. jedes Array hat einen Namen) schreiben wollen:

```
template<class T>     // Basisklassen-Template
class Array {         // (aus Lektion 13)
public:
  Array(int lowBound, int highBound);
 ~Array();

private:
  vector(T) data;
  unsigned int size;
  int lBound, hBound;
};

template<class T>
class NamedArray: public Array(T) {
public:
  NamedArray(int lowBound, int highBound,
             const string& name);
  ...

private:
  string arrayName;
};
```

Wenn Sie irgendwo in Ihrem Programm einen Zeiger auf ein NamedArray in einen Zeiger auf ein Array umwandeln und für diesen Array-Zeiger dann delete aufrufen, befinden Sie sich sofort wieder im Reich des undefinierten Verhaltens:

```
NamedArray<int> *pna =
  new NamedArray<int>(10, 20, "Impending Doom");

Array<int> *pa;

...

pa = pna;      // NamedArray<int>* -> Array<int>*

...

delete pa;     // undefiniert! (Fügen Sie hier die
               // Musik aus "Twilight Zone" ein),
               // In der Praxis wird pa->arrayName
```

```
// meistens zu einer Speicherlücke
// führen, da der NamedArray-Teil
// von *pa niemals zerstört wird
```

Diese Situation tritt öfter auf, als Sie vielleicht denken, da es nicht ungewöhnlich ist, daß Sie eine Klasse nehmen, die schon etwas kann (in diesem Fall Array), und von dieser Klasse dann eine neue Klasse ableiten, die das gleiche und noch etwas mehr kann. NamedArray ändert nichts am Verhalten von Array – es erbt alle Funktionen, ohne Sie neu zu definieren – es fügt nur etwas zusätzliche Funktionalität hinzu. Allerdings entsteht dabei auch das Problem mit dem nicht-virtuellen Destruktor.

Zum Schluß möchte ich noch erwähnen, daß es durchaus sinnvoll sein kann, in bestimmten Klassen einen rein virtuellen Destruktor zu deklarieren. Rufen Sie sich ins Gedächtnis, daß rein virtuelle Funktionen zu abstrakten Klassen führen – Klassen, die nicht instanziiert werden können (d. h. Klassen, von denen keine Objekte erzeugt werden können). Vielleicht werden Sie es einmal mit einer Klasse zu tun bekommen, die abstrakt sein soll, für die Sie aber keine einzige rein virtuelle Funktion haben. Was tun? Tja, da eine abstrakte Klasse als Basisklasse fungieren soll und da Basisklassen einen virtuellen Destruktor haben sollen und da rein virtuelle Funktionen eine abstrakte Klasse erzeugen, ist die Lösung ganz einfach: Deklarieren Sie einen rein virtuellen Destruktor für die Klasse, die abstrakt sein soll.

Hier ein Beispiel:

```
class AWOV {          // AWOV = "Abstract w/o Virtuals"
                      // abstrakte Klasse ohne
                      // virtuelle Funktionen
public:
  virtual ~AWOV() = 0;  // Deklarieren eines
                      // rein virtuellen Destruktors
};
```

Die Klasse hat damit eine rein virtuelle Funktion und ist somit abstrakt. Außerdem hat sie einen virtuellen Destruktor, so daß Sie sich über das Problem mit dem nicht-virtuellen Destruktor kein Sorgen machen müssen. Allerdings gibt es dabei noch eine Kleinigkeit zu beachten: Sie müssen eine *Definition* für den rein virtuellen Destruktor bereitstellen:

```
AWOV::~AWOV() {}     // Definition des
                     // rein virtuellen Destruktors
```

Sie benötigen diese Definition deshalb, da die Funktionsweise für virtuelle Destruktoren meistens so ist, daß zunächst der Destruktor der eigentlichen abgeleiteten Klasse aufgerufen wird und dann die Destruktoren aller Basisklassen. Das bedeutet, daß Ihre Compiler einen Aufruf für ~AWOV generieren werden, obwohl die Klasse abstrakt ist. Deshalb müssen Sie sicherstellen, daß es für diese Funktion einen Funktionsrumpf gibt.

Falls nicht, wird sich Ihr Linker über ein fehlendes Symbol beschweren, und dann müssen Sie sich Ihren Quellcode noch mal vornehmen und es erstellen.

Sie können in dieser Funktion tun und lassen, was Sie wollen, aber wie im Beispiel ist es nicht ungewöhnlich, wenn Sie nichts zu tun haben. In diesem Fall sind Sie vielleicht versucht, den Aufwand für den Aufruf einer leeren Funktion zu vermeiden, indem Sie den Destruktor inline deklarieren. Das ist eine völlig einleuchtende Strategie, aber da ist eine Sache, die Sie dabei beachten sollten.

Da Ihr Destruktor virtuell ist, muß seine Adresse in die vtbl der Klasse eingetragen werden. Da inline-Funktionen normalerweise nicht als eigenständige Funktionen existieren (das ist nämlich die Bedeutung von inline, nicht wahr?), müssen spezielle Maßnahmen ergriffen werden, um die Adresse einer inline-Funktion bereitstellen zu können. Lektion 33 erzählt Ihnen dazu die ganze Geschichte, aber die Moral ist: Wenn Sie einen virtuellen Destruktor inline deklarieren, vermeiden Sie wahrscheinlich den Aufwand eines Funktionsaufrufes, wenn er ausgeführt werden soll, aber Ihre Compiler müssen auch immer noch eine zusätzliche Kopie des Destruktors irgendwo ablegen.

Lektion 15

Lassen Sie operator= eine Referenz auf *this zurückliefern

Bjarne Stroustrup, der Erfinder von C++, hat eine Menge Schwierigkeiten auf sich genommen, um es zu ermöglichen, daß benutzerdefinierte Datentypen das Verhalten der eingebauten Typen so gut wie nur irgend möglich nachahmen können. Deshalb können Sie jetzt Operatoren überladen, Umwandlungsfunktionen schreiben, können die Kontrolle über Zuweisung und Copy-Konstruktor übernehmen usw. Nach soviel Anstrengungen von seiner Seite ist das mindeste was Sie tun können, seine Bemühungen aufzugreifen.

Damit kommen wir zur Zuweisung. Mit den eingebauten Typen können Sie Zuweisungen in einer Kette hintereinander schreiben:

```
int x, y, z;

x = y = z = 0;
```

Folglich sollten Sie auch Zuweisungen für benutzerdefinierte Typen verketten können:

```
string x, y, z;  // string ist ein "benutzerdefinierter"
                 // Typ aus der Standard C++ Library
                 // (siehe Lektion 49)

x = y = z = "Hello";
```

Wie das Schicksal nun mal so spielt, ist der Zuweisungsoperator rechtsassoziativ, womit die Verkettung wie folgt ausgewertet wird:

```
x = (y = (z = "Hello"));
```

Es lohnt sich, dies einmal in der äquivalenten funktionalen Form zu schreiben. Wenn Sie nicht gerade ein begeisterter LISP-Programmierer sind, wird Ihnen dieses Beispiel klarmachen, wie schön es ist, Infixoperatoren definieren zu können:

```
x.operator=(y.operator=(z.operator=("Hello")));
```

Diese Form illustriert sehr gut, daß das Argument für x.operator= und y.operator= jeweils der Rückgabewert des zuvor aufgerufenen operator= ist. Das bedeutet, daß der Rückgabewert von operator= als Argument für die Funktion selbst akzeptiert werden muß. Der Standard-operator= einer Klasse C besitzt folgende Deklaration (siehe Lektion 45):

```
C& C::operator=(const C&);
```

Sie werden fast immer dieser Konvention, daß operator= eine Referenz auf ein Objekt der Klasse sowohl erwartet als auch zurückgibt, folgen wollen. Wenngleich Sie auch von Zeit zu Zeit operator= überladen werden, damit er auch andere Argumente akzeptiert. Zum Beispiel stellt die string-Klasse aus der Standard C++ Library zwei verschiedene Zuweisungsoperatoren zur Verfügung:

```
string&                    // Zuweisung eines strings
operator=(const string& rhs);   // an einen string

string&                    // Zuweisung eines char*
operator=(const char *rhs);     // an einen string
```

Beachten Sie, daß trotz des Überladens der Rückgabewert eine Referenz auf ein Objekt der Klasse string bleibt.

Ein verbreiteter Fehler unter neuen C++-Programmierern ist die Ansicht, daß operator= void zurückliefern sollte. Eine Entscheidung, die zunächst vernünftig erscheint, bis Sie bemerken, daß Sie damit die Möglichkeit von Zuweisungsketten verhindern. Lassen Sie es also bleiben.

Ein anderer verbreiteter Fehler ist, daß operator= eine Referenz auf ein const-Objekt zurückliefert, wie hier:

```
class Widget {
public:
  ...                              // Rückgabe
  const Widget& operator=(const Widget& rhs);  // einer
```

```
  ...                                   // const-
};                                      // Referenz
```

Gewöhnlich ist die Motivation dafür zu verhindern, daß Benutzer der Klasse solche albernen Dinge tun können:

```
Widget w1, w2, w3;

...

(w1 = w2) = w3;     // Zuweisung von w2 an w1, dann w3 an
                    // das Ergebnis der Zuweisung! (Wenn
                    // Widget's operator= eine const-
                    // Referenz zurückliefert, würde dieser
                    // Ausdruck nicht kompiliert werden.)
```

So verrückt das auch sein mag, ist es dennoch nicht so verrückt, daß es für die eingebauten Typen verboten wäre:

```
int i1, i2, i3;

...

(i1 = i2) = i3;     // legal! Zuweisung von i2 an i1,
                    // dann Zuweisung von i3 an i1!
```

Ich weiß zwar nicht, von welchem praktischen Nutzen so etwas sein sollte, aber wenn es für ints funktioniert, soll es auch für meine Klasse funktionieren – und auch für Ihre Klassen. Warum grundlos von den Konventionen für die eingebauten Typen abweichen?

Innerhalb eines Zuweisungsoperators, der den Aufrufkonventionen des Standard-operator= folgt, gibt es offenbar zwei Kandidaten, die für den Rückgabewert in Frage kommen: das Objekt auf der linken Seite der Zuweisung (dasjenige, auf das this zeigt) und das Objekt auf der rechten Seite (dasjenige, welches als Parameter übergeben wurde). Welches ist das richtige?

Hier sind beide Möglichkeiten für eine String-Klasse (ein Klasse, für die Sie ganz bestimmt einen Zuweisungsoperator schreiben möchten, wie in Lektion 11 erklärt):

```
String& String::operator=(const String& rhs)
{

  ...

 return *this;     // Rückgabe einer Referenz auf
                   // das linke Objekt der Zuweisung
```

```
}

String& String::operator=(const String& rhs)
{

  ...

  return rhs;       // Rückgabe einer Referenz auf
                    // das rechte Objekt der Zuweisung
}
```

Das kommt Ihnen vielleicht wie eine Entscheidung zwischen sechs und einen halben Dutzend vor, aber es gibt einige wichtige Unterschiede.

Als erstes werden Sie die Version, die rhs zurückgibt, gar nicht kompilieren können. rhs ist nämlich eine Referenz auf einen const-String, operator= gibt aber eine Referenz auf einen String zurück. Ihre Compiler werden Ihnen nie den Kummer antun, den Sie hätten, wenn sie erlauben würden, eine Referenz auf ein nicht-konstantes Objekt zurückzugeben, das selber konstant ist. Das läßt sich aber leicht umgehen – deklarieren Sie operator= einfach so:

```
String& String::operator=(String& rhs) { ... }
```

Leider würde so Ihr Programmcode nicht mehr kompiliert werden können! Betrachten Sie noch einmal den letzten Teil der ursprünglichen Zuweisungskette:

```
x = "Hello";    // das gleiche wie x.operator=("Hello");
```

Da das rechte Objekt der Zuweisung nicht den richtigen Typ besitzt – es ist ein char* – müssen Ihre Compiler mittels des String-Konstruktor ein temporäres String-Objekt für den Aufruf des Zuweisungsoperators erzeugen. Der dabei generierte Code müßte in etwa zu dem folgenden äquivalent sein:

```
const String temp("Hello");  // Erzeugen eines temporären
                             // String-Objektes

x = temp;                    // Übergabe des temporären
                             // Objektes an den Zuweisungs-
                             // operator
```

Ihre Compiler würden Ihnen auch tatsächlich ein solches temporäres Objekt erzeugen (vorausgesetzt, der Konstruktor ist nicht explicit – siehe Lektion 19), aber beachten Sie, daß das temporäre Objekt ein const-Objekt ist. Das ist wichtig, denn dadurch wird verhindert, daß Sie aus Versehen ein temporäres Objekt an eine Funktion übergeben, die Ihre Parameter verändert. Wenn das nämlich erlaubt wäre, würden Programmierer sehr überrascht sein, daß nur das vom Compiler erzeugte, temporäre Objekt verändert

worden ist und *nicht* das eigentliche Objekt, das als Argument beim Aufruf der Funktion übergeben wurde. (Das weiß ich ganz genau, denn frühere Versionen von C++ erlaubten es, temporäre Objekte zu erzeugen, zu übergeben und verändern zu lassen. Die Folgen waren eine Menge überraschter Programmierer.)

Nun wissen Sie, warum der obige Programmcode nicht kompiliert werden würde, wenn Sie denn String::operator= so deklarieren, daß er eine nicht-konstante Referenz auf einen String erwartet: Es ist niemals erlaubt, ein const-Objekt an eine Funktion zu übergeben, die den entsprechenden Parameter nicht auch als const deklariert hat. Das ist einfache const-Korrektheit.

Sie befinden sich also in der glücklichen Lage, nicht die Qual der Wahl zu haben: Sie werden Ihre Zuweisungsoperatoren immer so definieren wollen, daß er eine Referenz auf das linke Objekt der Zuweisung, auf *this zurückgibt. Wenn Sie irgend etwas anderes tun, verhindern Sie Zuweisungsketten, implizite Typumwandlung oder beides.

Lektion 16

Weisen Sie allen Datenelementen im operator= etwas zu

Lektion 45 erläutert, daß C++ automatisch einen Zuweisungsoperator für Ihre Klasse generiert, wenn Sie selber keinen deklarieren, und Lektion 11 beschreibt, warum dieser automatisch generierte Zuweisungsoperator meistens nicht sehr hilfreich ist. Sie fragen sich jetzt vielleicht, wie Sie das Beste beider Welten bekommen können, indem Sie C++ den Standard-Zuweisungsoperator generieren lassen und selektiv nur die Teile überschreiben, die Sie nicht wollen. Das geht aber leider nicht. Wenn Sie die Kontrolle über irgendeinen Teil des Zuweisungsprozesses haben wollen, müssen Sie die gesamte Angelegenheit selbst in die Hand nehmen.

In der Praxis bedeutet das, daß Sie *jedem* Datenelement Ihres Objektes etwas zuweisen müssen, wenn Sie den Zuweisungsoperator selbst schreiben:

```
template<class T>  // Klassen-Template aus Lektion 12
class NamedPtr {   // für Zeiger, die mit einem Namen
                   // assoziiert sind
public:
  NamedPtr(const string& initName, T *initPtr);
  NamedPtr& operator=(const NamedPtr& rhs);

private:
  string name;
  T *ptr;
};
```

```
template<class T>
NamedPtr(T)& NamedPtr(T)::operator=(const NamedPtr(T)& rhs)
{
  if (this == &rhs)
    return *this;      // siehe Lektion 17

  // Zuweisung an alle Datenelemente
  name = rhs.name;     // Zuweisung an name

  *ptr = *rhs.ptr;     // Zuweisung an ptr, weist das zu,
                       // worauf gezeigt wird, nicht den
                       // Zeiger selbst

  return *this;        // siehe Lektion 15
}
```

Es ist nicht weiter schwer, sich daran zu erinnern, wenn die Klasse zum ersten Mal geschrieben wird. Aber es ist genauso wichtig, daß der Zuweisungsoperator jedesmal aktualisiert wird, wenn ein neues Datenelement zur Klasse hinzugefügt wird. Wenn Sie sich zum Beispiel entschließen, daß das NamedPtr-Template sich die Zeit merken soll, wann das letzte Mal sein Name verändert wurde, müssen Sie ein neues Datenelement in das Klassen-Template aufnehmen und das erfordert, daß Sie sowohl den/die Konstruktor(en) als auch den Zuweisungsoperator aktualisieren müssen. Im ganzen Durcheinander, wenn Sie eine Klasse erweitern und neue Elementfunktionen hinzufügen, kann es leicht passieren, daß Sie diese Sache vergessen.

Richtig Freude kommt erst auf, wenn Vererbung mit ins Spiel kommt, da der Zuweisungsoperator einer abgeleiteten Klasse auch die Zuweisungen an die Datenelemente aus der Basisklasse durchzuführen hat! Betrachten Sie dazu folgendes:

```
class Base {
public:
  Base(int initialValue = 0): x(initialValue) {}

private:
  int x;
};

class Derived: public Base {
public:
  Derived(int initialValue)
  : Base(initialValue), y(initialValue) {}

  Derived& operator=(const Derived& rhs);
```

```
private:
  int y;
};
```

Die logische Formulierung des operator= der Derived-Klasse würde so aussehen:

```
// fehlerhafter Zuweisungsoperator
Derived& Derived::operator=(const Derived& rhs)
{
  if (this == &rhs) return *this;   // siehe Lektion 17

  y = rhs.y;                    // Zuweisung an das
                                // einzige Datenelement
                                // der Derived-Klasse

  return *this;                 // siehe Lektion 15
}
```

Leider ist das nicht richtig, da das Datenelement x aus dem Base-Teil des Derived-Objektes durch diesen Zuweisungsoperator nicht behandelt wird. Betrachten Sie zum Beispiel das folgende Codefragment:

```
void assignmentTester()
{
  Derived d1(0);     // d1.x = 0, d1.y = 0
  Derived d2(1);     // d2.x = 1, d2.y = 1

  d1 = d2;           // d1.x = 0, d1.y = 1!
}
```

Wie Sie sehen, wird der Base-Teil von d1 nicht verändert.

Die einfachste Möglichkeit dieses Problem zu lösen wäre, eine Zuweisung an x im Derived::operator= durchzuführen. Leider ist das nicht legitim, da x ein private-Datenelement von Base ist. Statt dessen müssen Sie eine explizite Zuweisung des Base-Teils von Derived innerhalb des Zuweisungsoperators der Derived-Klasse vornehmen.

Und das machen Sie wie folgt:

```
// richtiger Zuweisungsoperator
Derived& Derived::operator=(const Derived& rhs)
{
  if (this == &rhs) return *this;
```

```
  Base::operator=(rhs);     // Aufruf von
                            // this->Base::operator=
  y = rhs.y;

  return *this;
}
```

Hierbei rufen Sie explizit Base::operator= auf. Dieser Aufruf, wie alle Aufrufe von Elementfunktionen innerhalb anderer Elementfunktionen, verwendet implizit *this als linken Operanden. Das Ergebnis dieses Aufrufes ist, daß der Base::operator= tut, was auch immer bei der Zuweisung des Base-Teils von *this getan werden muß – genau das, was Sie wollten.

Leider lehnen einige Compiler (inkorrekterweise) diese Art des Aufrufes des Zuweisungsoperators der Base-Klasse ab, wenn dieser Zuweisungsoperator automatisch generiert wurde (siehe Lektion 45). Um solche widerspenstigen Compiler zu beschwichtigen, müssen Sie Derived::operator= so implementieren:

```
Derived& Derived::operator=(const Derived& rhs)
{
  if (this == &rhs) return *this;

  static_cast<Base&>(*this) = rhs;  // Aufruf von
                                    // operator= für den
                                    // Base-Teil von *this
  y = rhs.y;

  return *this;
}
```

Dieses Monstrum wandelt *this in eine Referenz auf ein Base-Objekt um, und führt dann die Zuweisung an das Ergebnis von diesem Cast durch. Damit wird nur eine Zuweisung an den Base-Teil des Derived-Objektes bewirkt. Und jetzt aufgepaßt! Es ist entscheidend, daß der Cast zu einer *Referenz* auf ein Base-Objekt führt und nicht zu einem Base-Objekt selbst. Wenn Sie *this zu einem Base-Objekt casten, rufen Sie letztendlich den Copy-Konstruktor der Base-Klasse auf und das neue Objekt, was Sie damit erzeugen würden, wäre das Ziel der Zuweisung; *this bliebe unverändert – und das ist wohl kaum das, was Sie wollen.

Egal welche der beiden Möglichkeiten Sie verwenden, wenn Sie die Zuweisung für den Base-Teil des Derived-Objektes durchgeführt haben, ist die Arbeit des Zuweisungsoperators der Derived-Klasse noch nicht beendet. Sie müssen noch allen neuen Datenelementen von Derived etwas zuweisen.

Ein ähnliches Vererbungsproblem entsteht oft, wenn Sie den Copy-Konstruktor einer abgeleiteten Klasse implementieren. Werfen Sie einen Blick auf den folgenden Code, der das Analogon des eben behandelten Codes für den Copy-Konstruktor darstellt:

```
class Base {
public:
  Base(int initialValue = 0): x(initialValue) {}
  Base(const Base& rhs): x(rhs.x) {}

private:
  int x;
};

class Derived: public Base {
public:
  Derived(int initialValue)
  : Base(initialValue), y(initialValue) {}

  Derived(const Derived& rhs)  // fehlerhafterer
  : y(rhs.y) {}                // Copy-Konstruktor

private:
  int y;
};
```

Die Klasse Derived enthält einen der schlimmsten Fehler, den Sie in C++ machen können: der Base-Teil wird nicht mit kopiert, wenn ein Derived-Objekt mittels des Copy-Konstruktors konstruiert wird. Natürlich wird der Base-Teil eines solchen Derived-Objektes ordnungsgemäß initialisiert, aber durch den *Default*-Konstruktor der Base-Klasse. Das Datenelement x wird mit 0 initialisiert (dem Default-Parameter des Default-Konstruktors), unabhängig davon, welchen Wert das Datenelement x in dem zu kopierenden Objekt besitzt!

Um dieses Problem zu vermeiden, muß der Copy-Konstruktor der Derived-Klasse sicherstellen, daß der Copy-Konstruktor der Base-Klasse anstelle des Default-Konstruktors der Base-Klasse aufgerufen wird. Das ist schnell getan. Geben Sie einfach den Initialisierungswert für Base in der Initialisierungsliste des Copy-Konstruktors der Derived-Klasse an:

```
class Derived: public Base {
public:
  Derived(const Derived& rhs): Base(rhs), y(rhs.y) {}

  ...
```

};

Wenn ein Programmierer jetzt ein Derived-Objekt durch Kopieren eines existierenden Objektes dieses Typs erzeugt, wird auch der Base-Teil mitkopiert.

Lektion 17

Prüfen Sie in operator= auf Zuweisung an sich selbst

Eine Zuweisung an sich selbst tritt auf, wenn Sie so etwas tun:

```
class X { ... };

X a;

a = a;     // a wird sich selbst zugewiesen
```

Das sieht ziemlich unsinnig aus, ist aber völlig legal, deshalb sollten Sie auch keinen Augenblick daran zweifeln, daß Programmierer so etwas tun. Und was noch wichtiger ist, Zuweisung an sich selbst kann auch in verkleideter Form auftreten:

```
a = b;
```

Wenn b ein anderer Name für a ist (zum Beispiel eine Referenz, die mit a initialisiert wurde), so handelt es sich hierbei auch um eine Zuweisung an sich selbst, obwohl es zunächst gar nicht so aussieht. Dies ist ein Beispiel für *Aliasing*: zwei oder mehrere Namen stehen für ein und dasselbe Objekt. Wie Sie am Ende dieser Lektion sehen werden, kann Aliasing in den unterschiedlichsten Formen auftreten. Dies sollten Sie immer berücksichtigen, wenn Sie irgendeine Funktion schreiben.

Es gibt zwei herausragende Gründe dafür, warum gerade in Zuweisungsoperatoren besondere Sorgfalt für mögliches Aliasing an den Tag gelegt werden muß. Einer ist Effizienz. Wenn Sie eine Zuweisung an sich selbst am Anfang eines Zuweisungsoperators feststellen, können Sie sofort zum Aufrufer zurückkehren und sparen so unter Umständen eine Menge Arbeit, die Sie sonst beim Durchlaufen des Zuweisungsoperators bewältigen müßten.

In Lektion 16 wird zum Beispiel klargestellt, daß ein richtig implementierter Zuweisungsoperator einer abgeleiteten Klasse die Zuweisungsoperatoren aller Basisklassen aufrufen muß. Diese Basisklassen könnten wiederum selbst abgeleitete Klassen sein, so daß das Überspringen des Funktionsrumpfes des Zuweisungsoperators einer abgeleiteten Klasse eine große Anzahl von Funktionsaufrufen sparen kann.

Ein noch viel wichtigerer Grund, auf Zuweisung an sich selbst zu prüfen, ist, die Korrektheit Ihrer Programme zu gewährleisten. Wie Sie sich erinnern, müssen Zuweisungsoperatoren normalerweise die von einem Objekt allozierten Ressourcen freigeben (d. h. die alten Werte verwerfen), bevor sie die Ressourcen für die neuen Werte allozieren können. Bei einer Zuweisung an sich selbst kann die Freigabe der alten Werte verheerende Folgen haben, da die alten Werte höchstwahrscheinlich für die Initialisierung der neuen Werte benötigt werden.

Betrachten Sie die Zuweisung für String-Objekte, bei der der Zuweisungsoperator nicht auf Zuweisung an sich selbst prüft:

```
class String {
public:
  String(const char *value = 0);  // die Definition
                                   // dieser Funktion
                                   // finden Sie in
                                   // Lektion 11

  ~String();                       // die Definition
                                   // dieser Funktion
                                   // finden Sie in
                                   // Lektion 11

  String& operator=(const String& rhs);

private:
  char *data;
};

// ein Zuweisungsoperator, der nicht auf
// Zuweisung an sich selbst prüft
String& String::operator=(const String& rhs)
{
  delete [] data;     // veralteten Speicher löschen

  // neuen Speicher allozieren und
  // die Daten von rhs hineinkopieren
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);

  return *this;       // siehe Lektion 15
}
```

Was passiert nun hier:

```
String a = "Hello";

a = a;              // das gleiche wie a.operator=(a)
```

Innerhalb des Zuweisungsoperators scheinen *this und rhs verschiedene Objekte zu sein, aber in diesem Fall handelt es sich um zwei verschiedene Namen für das gleiche Objekt. Das kann so veranschaulicht werden:

z01_05.eps

Die erste Aktion, die der Zuweisungsoperator durchführt, ist der Aufruf von delete für das Datenelement data. Dadurch entsteht die folgende Situation:

z01_06.eps

Wenn der Zuweisungsoperator nun versucht, strlen auf rhs.data anzuwenden, ist das Ergebnis nicht definiert, denn rhs.data wurde gelöscht als data gelöscht wurde. data, this->data und rhs.data sind nämlich alle ein und derselbe Zeiger! Von hier an kann der Rest eigentlich nur noch schiefgehen.

Aber Sie wissen ja jetzt, daß der Ausweg aus diesem Dilemma darin besteht, auf Zuweisung an sich selbst zu prüfen und sofort zurückzukehren, wenn solch eine Zuweisung entdeckt wird. Unglücklicherweise ist es einfacher, über eine solche Überprüfung zu reden als sie zu schreiben, denn Sie sind nun gezwungen herauszubekommen, was es bedeutet, daß zwei Objekte »gleich« sind.

Diese Frage, mit der Sie nun konfrontiert sind, ist in der technischen Literatur als das Problem der *Objektidentität* bekannt, und es ist sehr bekannt in objektorientierten Kreisen. Dieses Buch ist nicht der richtige Ort, um über Objektidentität zu diskutieren, aber ich möchte Ihnen trotzdem die beiden grundlegenden Ideen zu diesem Problem vorstellen.

Der erste Ansatz ist zu sagen, zwei Objekte sind gleich (haben die gleiche Identität), wenn sie den gleichen Wert haben. Zum Beispiel wären zwei String-Objekte genau dann gleich, wenn Sie die gleiche Folge von Zeichen repräsentieren:

```
String a = "Hello";
String b = "World";
String c = "Hello";
```

Hier besitzen a und c denselben Wert und werden somit als gleich betrachtet. b ist zu beiden verschieden. Wenn Sie diese Definition der Identität zugrunde legen, könnte Ihr Zuweisungsoperator so aussehen:

```
String& String::operator=(const String& rhs)
{
  if (strcmp(data, rhs.data) == 0) return *this;
```

```
  ...

}
```

Normalerweise wird die Wertegleichheit durch den operator== festgelegt, so daß der Zuweisungsoperator für eine Klasse C, der die Wertegleichheit als Identitätskriterum verwendet, die folgende allgemeine Form hat:

```
C& C::operator=(const C& rhs)
{
  // prüfe auf Zuweisung an sich selbst
  if (*this == rhs)     // erfordert, daß der
    return *this;       // operator== existiert

  ...

}
```

Beachten Sie, daß diese Funktion Objekte vergleicht (via operator==) und keine Zeiger. Wenn Sie die Wertegleichheit benutzen, um Objektidentität herauszufinden, spielt es keine Rolle, ob die zwei Objekte den gleichen Speicher belegen oder nicht, es zählen nur die Werte, die sie repräsentieren.

Der andere Ansatz setzt die Objektidentität mit der Identität der Speicheradressen gleich. Mit dieser Definition sind zwei Objekte genau dann gleich, wenn und nur wenn sie dieselbe Adresse besitzen. Diese Definition ist in C++-Programmen gebräuchlicher, wahrscheinlich deshalb, weil sie einfach zu implementieren ist und sehr schnell ermittelt werden kann, was beides nicht immer der Fall ist, wenn Objektidentität auf der Gleichheit der Werte der Objekte beruht. Ein allgemeiner Zuweisungsoperator, der auf der Gleichheit von Adressen basiert, sieht etwa so aus:

```
C& C::operator=(const C& rhs)
{
  // prüfe auf Zuweisung an sich selbst
  if (this == &rhs) return *this;

  ...

}
```

Das ist für die allermeisten Programme völlig ausreichend.

Wenn Sie einen ausgefeilteren Mechanismus benötigen, um zu bestimmen, ob zwei Objekte gleich sind, müssen Sie diesen selbst implementieren. Der meist genutzte Ansatz

hierfür basiert auf einer Elementfunktion, die irgendeine Art Objektidentifizierer zurückliefert:

```
class C {
public:
  ObjectID identity() const;   // siehe auch Lektion 36

  ...

};
```

Wenn Sie nun zwei Zeiger a und b auf Objekte dieser Klasse haben, sind diese Objekte genau dann identisch, wenn und nur wenn a->identity() == b->identity() ist. Natürlich sind Sie dafür verantwortlich, den operator== für ObjectID zu schreiben.

Die Probleme mit Aliasing und Objektidentität sind keineswegs nur auf den Zuweisungsoperator beschränkt; das ist nur eine Funktion, in der Sie besonders wahrscheinlich mit ihnen zu kämpfen haben. Sobald Sie es mit Referenzen oder Zeigern zu tun haben, können sofort zwei beliebige Namen für Objekte sich in Wirklichkeit auf ein und dasselbe Objekt beziehen. Hier sind ein paar weitere Situationen, in denen Aliasing einige seiner vielen Gesichter zeigt:

```
class Base {
  void mf1(Base& rb);          // rb und *this könnten das
                               // gleiche Objekt sein
  ...

};

void f1(Base& rb1,Base& rb2);   // rb1 und rb2 könnten das
                               // gleiche Objekt sein

class Derived: public Base {
  void mf2(Base& rb);          // rb und *this könnten das
                               // gleiche Objekt sein
  ...

};

int f2(Derived& rd, Base& rb);  // rd und rb könnten das
                               // gleiche Objekt sein
```

Diese Beispiele benutzten Referenzen, aber Zeiger würden es genauso tun.

Wie sie sehen, kann sich Aliasing hinter den verschiedensten Masken verbergen. Sie können es also nicht einfach vergessen und hoffen, niemals damit in Berührung zu kommen. Na gut, vielleicht können *Sie*, aber die meisten von uns können nicht. Auf die Gefahr hin, meine Metaphern durcheinanderzubringen, ist das ein klarer Fall dafür, daß eine Unze Vorbeugung ihr Gewicht in Gold wert ist. Jedesmal, wenn Sie eine Funktion definieren, in der Aliasing auftreten könnte, *müssen* Sie diese Möglichkeit auch in Betracht ziehen, wenn Sie den Code schreiben.