20 Verwaltetes und nicht verwaltetes C++

20.1 C++ unterstützt sowohl die verwaltete als auch die nicht verwaltete Programmierung

In Visual Studio .NET werden – stärker als in allen vorhergegangenen Visual Studio-Releases – alle unterstützten Programmiersprachen gleich behandelt. Früher standen bestimmte Elemente und Optionen nur C++-Programmierern, andere nur Visual Basic-Programmierern zur Verfügung. Diese Zeiten sind nun vorbei – jedenfalls so gut wie. Einige wenige Elemente, die nur C++-Programmierern vorbehalten sind, gibt es noch – beispielsweise den ATL-Server, der in Kapitel 10, »Internet-Programmierung«, beschrieben ist.

Wie bereits in der Einführung angesprochen, behandelt die erste Hälfte des Buches die »klassische Win32-Programmierung« – die Art von Visual C++-Programmierung, die Sie vor der Markteinführung von .NET ausübten. Der zugehörige Code kann weiterhin in Visual Studio kompiliert werden, irgendeine Form der Konvertierung oder Portierung ist nicht erforderlich. Dies gilt jedoch nur für C++, nicht für andere Programmiersprachen. Das »klassische Visual Basic« wird zum Beispiel nicht mehr von Visual Studio unterstützt.

Die zweite Hälfte des Buches beschäftigt sich mit der Visual C++ .NET-Programmierung – Programmierung mit Unterstützung durch die mächtige Common Language Runtime-Bibliothek, durch die automatische Speicherverwaltung, die statt Ihnen über den Speicher wacht, durch sprachenübergreifende Vererbung, durch Interoperabilität zwischen Sprachen und durch die unglaublich einfache Erstellung von Webdiensten.

Sie wissen es sicherlich schon oder erahnen es zumindest: Die Visual C++-Programmierung vor .NET unterscheidet sich wesentlich von der Visual C++-Programmierung nach .NET. Es ist schlichtweg nicht möglich, von einer im alten Stil geschriebenen C++-Klasse eine Visual Basic-Klasse abzuleiten, und Sie können auch nicht irgendwelchen Code hernehmen, der mit seiner eigenen Speicherverwaltung beschäftigt ist, und erwarten, dass diese Aufgabe plötzlich vom .NET Framework für Sie übernommen wird (insbesondere dann nicht, wenn dieser Legacy-Code `malloc` und `delete` verwendet, um den Speicher von Nicht-Objekten zu verwalten).

Dieses Kapitel beschäftigt sich mit den Unterschieden zwischen verwaltetem Code (im Wesentlichen .NET-kompatiblem Code) und nicht verwaltetem (klassischem) Code. Welche Art von Code Sie schreiben, bleibt Ihnen überlassen; Sie können sogar beide Formen in ein und derselben Anwendung verwenden.

20.2 Was ist verwalteter (managed) Code?

Sie entscheiden, ob die von Ihnen geschriebenen Klassen verwaltete Klassen sind oder nicht. (Die Klassen der Common Language Runtime sind alle verwaltet.) In Visual C++ .NET gibt es drei Arten von verwalteten Klassen:

- Klassen mit Speicherverwaltung (Garbage Collection)
- Werttyp-Klassen
- Klassen mit verwalteter Schnittstelle

Die meisten verwalteten Klassen sind Klassen mit Speicherverwaltung. Um eine Klasse zu einer Klasse mit Speicherverwaltung (und damit zu einer verwalteten Klasse) zu machen, deklarieren Sie sie wie folgt:

```
__gc class Employee
{
  // Inhalt der Klasse
};
```

20.2.1 Garbage Collection: Speicherverwaltung

Das Schlüsselwort `__gc` (beginnt mit zwei Unterstrichen) zeigt an, dass es sich um eine Klasse mit Speicherverwaltung handelt, das heißt, die Nutzung des Speichers durch die Klasse fällt nicht in Ihren Verantwortungsbereich, sondern in den des Frameworks. Wenn Sie `__gc` nicht verwenden, gilt standardmäßig das althergebrachte, noch immer bestehende C++-Verhalten.

Managed Extensions

Einige Leser werden sich vielleicht an die C++-Grundregel erinnern, wonach Unterstriche, obwohl sie in Variablennamen grundsätzlich erlaubt sind, nicht am Anfang des Variablennamens verwendet werden sollten. Die Hersteller von C++-Compilern tendieren nämlich dazu, ihrer Implementierung der Sprache eigene Schlüsselwörter hinzuzufügen, und diese Schlüsselwörter beginnen – gemäß einer Übereinkunft zwischen den Compiler-Herstellern – stets mit einem Unterstrich. Wenn Sie also Ihre Variablennamen nicht mit

einem Unterstrich beginnen, laufen Sie auch nicht Gefahr, mit den compilerspezifischen Schlüsselwörtern in Konflikt zu geraten.

Die compilerspezifischen Schlüsselwörter bezeichnet man als Extensions (Erweiterungen), da sie den Compiler-Herstellern dazu dienen, die Sprache um zusätzliche Elemente zu erweitern. Die in diesem Kapitel beschriebenen neuen Schlüsselwörter (__gc, __property, __box, __value) werden unter dem Begriff der »Managed Extensions« zusammengefasst.

Das folgende Beispiel veranschaulicht, was es bedeutet, wenn die Speicherbelegung für die Klasse nicht von Ihnen, sondern vom Framework übernommen wird. Eine normale Klasse wie Employee verwenden Sie wie folgt:

```
// Fragment eines größeren Programms
Employee * pe = new Employee(); // könnte Parameter an
                                // den Konstruktor über-
                                // geben
//
//   ... Methoden der Employee-Instanz verwenden
//
delete pe;
// Programm wird fortgesetzt
```

Den Speicher für die Instanzen von Employee reservieren Sie indirekt, indem Sie den new-Operator aufrufen, der den Speicher allokiert und den Konstruktor aufruft. Wenn Sie die Instanz nicht mehr benötigen, verwenden Sie den delete-Operator, der den Destruktor aufruft und den belegten Speicher wieder freigibt. Der freigegebene Speicher kann daraufhin von nachfolgenden Allokationen erneut genutzt werden.

Wenn Sie vergessen, den delete-Operator aufzurufen, erzeugen Sie ein *Speicherleck*. In Kapitel 12, »Bessere Anwendungen schreiben«, können Sie mehr über die eigenhändige Speicherverwaltung erfahren, die dem Programmierer viel Arbeit und Verdruss bereiten kann.

Zum Vergleich dazu eine Klasse mit Speicherverwaltung. Reservierung und Freigabe des Speichers ist nun nicht mehr Ihr Problem; der zugehörige Code sieht wie folgt aus:

```
// Fragment eines größeren Programms
Employee * pe = new Employee(); // könnte Parameter an
                                // den Konstruktor über-
                                // geben
//
//   ... Methoden der Employee-Instanz verwenden
//
// Programm wird fortgesetzt
```

Es besteht keine Notwendigkeit, den Zeiger zu löschen, wenn Sie die Instanz nicht mehr weiter benötigen. Die Instanz wird aufgelöst, wenn der Gültigkeitsbereich des Zeigers verlassen wird und es keinen anderen Zeiger gibt, der auf die Instanz verweist. Vielleicht wird die Instanz aber auch nicht aufgelöst – wenn Ihr Programm nur wenig Speicher allokiert hat, kann es gut sein, dass die Speicherbereinigung (Garbage Collector) überhaupt nicht in Gang gesetzt wird. Sie brauchen sich darüber aber in der Regel keine Gedanken zu machen, alles, was für Sie zählt, ist, dass Ihr Programm nicht aus Mangel an Arbeitsspeicher zum Stillstand kommt.

Wenn Sie möchten, dass Ihr Destruktor ausgeführt wird, etwa weil er eine Datei schließt oder weil er noch irgendeine andere Ressource als den belegten Speicher freigibt, gibt es verschiedene Wege, dies zu erreichen. Sie können den Destruktor direkt aufrufen:

```
pe->~Employee();
```

Sie können den Zeiger auf `0` (oder `NULL`) setzen – was für die Speicherbereinigung ein deutlicher Hinweis darauf ist, dass Sie die Instanz nicht weiter benötigen – und die Speicherbereinigung danach selbst starten:

```
pe = 0;
GC::Collect();
```

Sie können den Zeiger auch selbst löschen, wodurch der Destruktor aufgerufen und der Zeiger auf 0 gesetzt wird.

```
delete pe;
```

Bevor Sie sich jedoch zu sehr mit dieser Art von Code anfreunden, sollten Sie sich einen VB- oder C#-Programmierer vorstellen, der Ihre Klasse verwendet. Vermutlich wird er jeden dieser Ansätze mit Unbehagen betrachten. Die beste Lösung dürfte daher darin bestehen, für die Erledigung der Aufräumarbeiten eine eigene Methode zu schreiben, dieser einen passenden Namen zu geben, etwa `CloseFiles()` oder `ReleaseLocks()`, und zu hoffen, dass die anderen Programmierer nicht vergessen, sie aufzurufen. Dies mag zwar wie ein Rückschritt erscheinen, aber denken Sie nur einmal daran, dass Sie für Instanzen, die mit `new` allokiert wurden, ja sonst auch darauf hoffen müssen, dass die Programmierer nicht vergessen, `delete` zu verwenden.

20.2.2 Ein Zeiger auf ein verwaltetes Objekt ist kein Zeiger auf ein nicht verwaltetes Objekt

Die CLR-Bibliotheken sind voll gestopft mit nützlichen Klassen und Methoden. `System::Collections::Stack` zum Beispiel repräsentiert einen einfachen Stack mit Methoden wie `Push()` und `Pop()`. Wenn Sie einen Stack von `Emplo-`

yee-Instanzen anlegen möchten, muss Employee eine Klasse mit Speicherverwaltung sein. Statt eines Stacks von Instanzen, verwalten Sie dann einen Stack von Zeigern auf Employee-Instanzen.

(Die anderen Sprachen von Visual Studio .NET und die Dokumentation sprechen von Referenzen, doch in C++ sind es Zeiger.) Warum muss eine Klasse an der automatischen Speicherverwaltung teilnehmen, wenn man eine Instanz der Klasse (oder einen Zeiger auf eine Instanz der Klasse) an eine Framework-Methode übergeben möchte? Die Methoden des Frameworks erwarten einen oder mehrere Zeiger auf speicherverwaltete Instanzen. In manchen Fällen können Sie der Methode ohne Probleme etwas anderes übergeben, als sie erwartet, weil der Compiler eine passende Typumwandlung vornimmt. Zwischen Zeigern auf »klassische« und speicherverwaltete Objekte gibt es jedoch keine Typumwandlung – die Unterschiede sind einfach zu groß.

Der wichtigste Unterschied ist wohl, dass sich der numerische Wert eines Zeigers auf eine Instanz einer speicherverwalteten Klasse jederzeit ändern kann. Wenn die Speicherbereinigung läuft, kann sie die Instanz verschieben. Tut sie dies, aktualisiert sie sämtliche Zeiger, damit diese auf die neue Lokation verweisen. Folglich können Sie mit einem Zeiger auf ein verwaltetes Objekt keine Zeigerarithmetik oder -manipulation (Inkrementierung oder Dekrementierung) durchführen.

20.2.3 Boxing für klassische Zeiger

Stellen Sie sich vor, Sie hätten irgendwelche einfache Daten vorliegen, die Sie nicht in einer speicherverwalteten Klasse aufbewahren:

```
int i = 3;
```

Diesen Integer-Wert möchten Sie nun an eine der überladenen Versionen von System::Console::WriteLine() übergeben. Eine der überladenen Methoden nimmt einen Formatstring (ähnlich wie der Formatstring von printf, jedoch anders formatiert) und einen Zeiger auf eine Instanz einer speicherverwalteten Klasse entgegen. Ihr erster Versuch, diese Methode aufzurufen, könnte wie folgt aussehen:

```
System::Console::WriteLine("i is {0}",i);
```

Der Compiler gibt dafür folgende Fehlermeldung aus:

```
error C2665: 'System::Console::WriteLine' : Durch keine der 19
Überladungen kann Parameter 2 vom Typ 'int' konvertiert werden
```

Es liegt nahe, aus der Meldung die Schlussfolgerung zu ziehen, dass man der Methode einen Zeiger übergeben muss, und es wie folgt zu versuchen:

```
System::Console::WriteLine("i is {0}", &i);
```

Doch dieser Aufruf produziert im Wesentlichen die gleiche Fehlermeldung, nur dass nun keine Umwandlung von `int*` möglich ist.

Die Lösung liegt im *Boxing* (Schachteln) des Integers. Das Boxing erzeugt eine temporäre Instanz einer speicherverwalteten Klasse, legt den Integer darin ab und liefert Ihnen einen Zeiger auf die Instanz zurück, den Sie an den Code, der einen solchen Zeiger erwartet, übergeben können:

```
System::Console::WriteLine("i is {0}",__box(i));
```

Dies klappt ausgezeichnet und erschließt die Funktionalität der Framework-Klassen für Daten jeden beliebigen Typs, einschließlich der fundamentalen Datentypen, zu denen auch die Integer-Daten gehören.

20.2.4 Fixieren verwalteter Zeiger

Manchmal verursacht genau der umgekehrte Weg Probleme. Angenommen, Sie haben eine bereits vollständig implementierte Funktion, die irgendeine Art von Zeiger erwartet. Handelt es sich um Legacy-Code, ist dieser selbstverständlich nicht darauf eingerichtet, dass der Zeiger umherwandern kann; er erwartet einen klassischen Zeiger auf ein nicht verwaltetes Objekt. (Auch ein Zeiger auf eine Integer-Membervariable eines verwalteten Objekts kann verschoben werden, wenn die Speicherbereinigung die komplette Instanz des Objekts verschiebt.)

Betrachten Sie die einfache verwaltete Klasse aus Listing 20.1

Listing 20.1: Eine einfache verwaltete Klasse

```
__gc class Sample
{
public:
   int a;
   int b;
   Sample(int x, int y)
   {
      a = x;
      b = y;
   }
   void Report()
   {
      Console::WriteLine("a ist {0} und b ist {1}",
                         __box(a),__box(b));
   }
};
```

Sie könnten bemängeln, dass die Membervariablen `a` und `b` `public` sind, doch lassen Sie uns für einen Moment darüber hinwegsehen und wenden wir uns der folgenden Funktion zu, die vielleicht noch mit einer früheren Visual C++-Version geschrieben wurde:

```
void Equalize(int* a, int* b)
{
   int avg = (*a + *b)/2 ;
   *a = avg;
   *b = avg;
}
```

Nehmen wir weiter an, Sie möchten mithilfe dieser Funktion den Mittelwert der beiden Membervariablen einer Instanz der `Sample`-Klasse berechnen:

```
Sample* s = new Sample(2,4);
s->Report();

Equalize(&(s->a),&(s->b));
s->Report();
```

Dieser Code wird nicht kompiliert. Die Fehlermeldung lautet:

```
error C2664: 'Equalize' : Konvertierung des Parameters 1 von 'int __gc
*' in 'int *' nicht möglich
```

Obwohl es sich bei beiden um Zeiger handelt, kann ein Zeiger auf einen speicherverwalteten Typ nicht in einen Zeiger auf einen nicht verwalteten Typ umgewandelt werden. Sie können allerdings den Zeiger *fixieren*. Die Fixierung erzeugt einen neuen Zeiger, den Sie an die Funktion übergeben können, und sie stellt sicher, dass die Speicherbereinigung die Instanz (in unserem Falle `s`) für die Lebensdauer des fixierten Zeigers nicht verschiebt. Um den Zeiger zu fixieren, schreiben Sie:

```
int __pin* pa = &(s->a);
int __pin* pb = &(s->b);
Equalize(pa,pb);
```

Die beiden neuen Zeiger `pa` und `pb` sind Zeiger auf nicht verwaltete Typen, können also an `Equalize()` übergeben werden. Sie zeigen auf die Speicherbereiche, in denen die Membervariablen von `s` abgelegt sind, und die Speicherbereinigung wird die Instanz `s` nicht verschieben, bevor nicht der Gültigkeitsbereich von `pa` und `pb` verlassen wird.

Wenn Sie die Fixierung der Instanz schon früher aufheben möchten, setzen Sie `pa` und `pb` auf 0:

```
pa=0;
pb=0;
```

20.3 Können alle Klassen an der Speicherverwaltung teilhaben?

Nicht immer genügt es, der Klassendefinition das Schlüsselwort `__gc` voranzustellen, um aus der Klasse eine speicherverwaltete Klasse zu machen. Zusätzlich sind bestimmte Beschränkungen zu beachten, die teils die Definition, teils die Verwendung der Klasse betreffen.

Einige dieser Beschränkungen gelten für alle drei Arten von verwalteten Klassen, andere betreffen nur die speicherverwalteten Klassen.

20.3.1 Verwaltete Klassen können nur von verwalteten Klassen abgeleitet werden

Betrachten Sie die Klasse aus Listing 20.2, eine leicht abgewandelte Variation der weiter vorne eingeführten `Sample`-Klasse.

Listing 20.2: Eine abgeleitete speicherverwaltete Klasse

```
class A
{
protected:
   int a;
};

class B
{
protected:
   int b;
};
__gc class Sample: public A, public B
{
public:
   Sample(int x, int y)
   {
   a = x;
   b = y;
   }
   void Report()
   {
   Console::WriteLine("a ist {0} und b ist {1}",
                      __box(a),__box(b));
```

```
    }
};
```

Dieser Code sieht nach perfektem C++ aus. Er wäre es auch, wenn in der Definition von `Sample` nicht das Schlüsselwort `__gc` auftauchen würde. Wenn Sie den obigen Code eintippen und kompilieren lassen, erhalten Sie eine Fehlermeldung, die daher resultiert, dass verwaltete Klassen nicht von nicht verwalteten Klassen abgeleitet werden können.

Um das Problem zu beheben, müssen Sie – falls möglich – die Basisklasse zu einer speicherverwalteten Klasse machen oder auf die automatische Speicherverwaltung für die `Sample`-Klasse verzichten.

20.3.2 Für speicherverwaltete Klassen gibt es keine Mehrfachvererbung

Wenn Sie versuchen, den Beispielcode aus Listing 20.2 dadurch zu korrigieren, dass Sie das Schlüsselwort `__gc` vor die Klassendefinitionen von `A` und `B` stellen, erhalten Sie folgende Fehlermeldung:

```
error C2890: 'Sample' : Eine __gc-Klasse kann nur eine Nichtschnittstellen-
Basisklasse haben
```

In der Praxis wird die Mehrfachvererbung eher selten eingesetzt, ja die meisten C++-Programmierer verwenden sie aus verschiedenen Gründen grundsätzlich nicht. Sollten Sie zu diesen Programmierern gehören, dürfte der Verzicht auf die Mehrfachvererbung kein großer Verlust für Sie sein. Wenn Sie bestehenden Code vorliegen haben, in dem die Mehrfachvererbung genutzt wird, ist es am besten, den Code unverändert zu lassen und ihn von neuem, verwaltetem Code aus anzusprechen. Wie das geht, werden Sie in Kürze erfahren.

Für die Ableitung von verwalteten Schnittstellen gibt es dagegen keine Beschränkung. Sie können von so vielen verwalteten Schnittstellen ableiten, wie Sie wollen. Wie dies geht, wird weiter hinten im Abschnitt »Verwaltete Schnittstellen« beschrieben.

20.3.3 Zusätzliche Beschränkungen für verwaltete Klassen

Verwaltete Klassen unterliegen noch weiteren Einschränkungen:

- Es ist nicht möglich, mithilfe des `friend`-Schlüsselwortes einer Klasse den Zugriff auf die `private`-Elemente einer verwalteten Klasse einzuräumen.

- Die Membervariablen der Klasse dürfen keine Instanzen von nicht verwalteten Klassen sein (es sei denn, alle Memberfunktionen der nicht verwalteten Klasse wären statisch.)

Für Klassen mit Speicherverwaltung gelten darüber hinaus noch weitere Beschränkungen:

- Die Operatoren `&` und `new` können nicht überschrieben werden.
- Sie können keinen Kopierkonstruktor implementieren.

Auf den ersten Blick erscheinen diese Einschränkungen recht hart. Doch überlegen Sie einmal, warum Sie üblicherweise einen Kopierkonstruktor schreiben? Weil Ihr Destruktor irgendetwas Destruktives tut, etwa belegten Arbeitsspeicher freigibt. Im Falle einer speicherverwalteten Klasse ist es jedoch wahrscheinlich, dass die Klasse gar keinen Destruktor besitzt, und es folglich auch nicht erforderlich ist, einen speziellen Kopierkonstruktor zu implementieren.

Wenn Sie so weit sind, dass Sie eine speicherverwaltete Klasse benutzen wollen, stoßen Sie direkt auf die nächste Beschränkung. Instanzen von speicherverwalteten Klassen dürfen nicht auf dem Stack erzeugt werden.

```
Sample s(2,4);
s.Report();
```

Wenn Sie es wie in den obigen Zeilen doch versuchen, ernten Sie eine Fehlermeldung:

```
error C3149: 'Sample' : Ungültige Verwendung des verwalteten Typs 'Sample';
haben Sie ein '*' vergessen?
```

Stattdessen müssen alle Instanzen speicherverwalteter Klassen auf dem Heap erzeugt werden:

```
Sample* s = new Sample(2,4);
```

Auf diese Weise wird die Instanz zur Verwaltung an die Speicherbereinigung übergeben. Der numerische Wert von `s` (das heißt die Speicheradresse, an der die `Sample`-Instanz zu finden ist) kann sich danach ändern, doch wen kümmert das? Sie sicherlich nicht. Schließlich ist dies ja der Hauptgrund, warum Sie einen speicherverwalteten Typ verwenden.

20.4 Werttypen

Vielen Programmierern kommen Zweifel, wenn sie hören, dass sie von speicherverwalteten Klassen keine Instanzen auf dem Stack erzeugen können. Schließlich gibt es im »klassischen« C++ eine Reihe von Vorteilen, die mit dem Erzeugen von Objekten auf dem Stack verknüpft sind:

- Das Objekt wird automatisch aufgelöst, wenn sein Gültigkeitsbereich verlassen wird.
- Die Kosten für die Allokation auf dem Stack sind etwas geringer als für die Reservierung auf dem Heap.
- Es kann zur Fragmentierung des Heaps kommen (was meist mit einem Performance-Abfall verbunden ist), wenn Sie viele kurzlebige Objekte allokieren und freigeben.

In verwaltetem C++ (sprich in C++ mit Managed Extensions) sorgt die Speicherbereinigung für die Auflösung der Objekte. Sie kann sogar den Heap defragmentieren. Auf der anderen Seite muss man sehen, dass die Speicherbereinigung einen gewissen Overhead verursacht und dass die höheren Kosten für die Allokation auf dem Heap natürlich bestehen bleiben. Für bestimmte Arten von Objekten kann die Verwendung einer Werttyp-Klasse anstelle einer speicherverwalteten Klasse daher die bessere Wahl sein.

Die fundamentalen Datentypen wie `int` werden auch als *Werttypen* bezeichnet, da sie auf dem Stack allokiert werden. Wenn Sie möchten, können Sie auch kleinere Klassen, die Sie selbst schreiben, als Werttyp-Klassen definieren. Gleiches gilt für Strukturen. Wenn zum Beispiel Ihre Klasse nur wenige Membervariablen enthält und die gewünschte Lebensdauer kein Hindernis darstellt, ist sie ein guter Kandidat für eine Werttyp-Klasse.

Listing 20.3 definiert `Sample` als Werttyp-Klasse.

Listing 20.3:
Eine Werttyp-Klasse

```
__value class Sample
{
public:
   int a;
   int b;
   Sample(int x, int y)
   {
   a = x;
   b = y;
   }
   void Report()
   {
   Console::WriteLine("a ist {0} und b ist {1}"
                      ,__box(a),__box(b));
```

```
    }
};
```

Wenn Sie Instanzen von `Sample` erzeugen und verwenden wollen, müssen Sie diese auf dem Stack allokieren, nicht auf dem Heap:

```
Sample s(2,4);
s.Report();
```

Werttyp-Klassen sind verwaltet, aber nicht speicherverwaltet. Alle Beschränkungen, die weiter oben für verwaltete Klassen aufgeführt wurden, gelten auch für Werttyp-Klassen. Zusätzlich ist zu beachten:

- Sie können keinen Kopierkonstruktor implementieren.
- Werttyp-Klassen können weder von speicherverwalteten Klassen noch von Werttyp-Klassen oder nicht verwalteten Klassen abgeleitet werden. Sie können aber von beliebig vielen verwalteten Schnittstellen erben.
- Werttyp-Klassen dürfen keine virtuellen Methoden enthalten, außer denen, die sie von `System::ValueType` erben (und möglicherweise überschreiben).
- Von einer Werttyp-Klasse dürfen keine Klassen abgeleitet werden.
- Werttyp-Klassen können nicht mit dem Schlüsselwort `__abstract` als abstrakte Klassen deklariert werden.

Die besten Kandidaten für Werttyp-Klassen sind kleine Klassen, deren Objekte nicht allzu lange existieren und nicht von Methode zu Methode weitergegeben werden (wodurch, über verschiedene Codeabschnitte verteilt, ein Heer von Verweisen auf die Objekte erzeugt würde).

:-) TIPP

Leser, die sich bereits auf dem einen oder anderen Weg über Visual Studio .NET informiert haben, haben vielleicht auch gehört, dass `structs`*, Werttypen und Klassen Verweistypen sind oder dass Strukturen auf dem Stack und Objekte auf dem Heap erzeugt werden. Diese Aussagen beziehen sich auf C#. In C++ ist es Ihre Entscheidung, ob Sie Instanzen von Werttypen (Klassen oder* `structs`*) auf dem Stack oder Instanzen von speicherverwalteten Typen (Klassen oder* `structs`*) auf dem Heap allokieren.*

Ein weiterer Vorzug der Werttyp-Klassen besteht darin, dass die Instanzen prinzipiell stets initialisiert werden. Wenn Sie eine Instanz von `Sample` auf dem Stack anlegen, können Sie Parameter an den Konstruktor übergeben. Tun Sie dies nicht, werden die Membervariablen automatisch mit Null initialisiert. Dies ist selbst dann der Fall, wenn Sie einen Konstruktor mit Argumenten, aber keinen Konstruktor ohne Argumente definiert haben. Sehen Sie sich die beiden folgenden Zeilen an:

```
Sample s2;
s2.Report();
```

Wenn dieser Code ausgeführt wird, meldet er:

```
a ist 0 und b ist 0
```

Weil die Klasse `Sample` verwaltet wird, werden ihre Elemente automatisch initialisiert.

20.5 Verwaltete Schnittstellen

Verwaltete Schnittstellen können Sie sich als Repräsentationen von COM-Schnittstellen vorstellen. Das Schlüsselwort `__interface` erlegt einer Klasse eine Reihe von Beschränkungen auf, das Schlüsselwort `__gc` packt noch einige drauf. Klassen, die diese Beschränkungen erfüllen, können als zusätzliche Basisklassen für speicherverwaltete Klassen dienen, wodurch verwaltetes C++ in den Genuss einer nützlichen (und im Allgemeinen sicheren) Form der Mehrfachvererbung kommt.

20.5.1 Verwaltete Schnittstellen deklarieren

Sie möchten Ihre eigene Schnittstelle schreiben? Hier ein Beispiel:

```
__gc __interface ISomething
{
   void UsefulMethod();
};
```

- Beachten Sie die Doppelpackung von Extension-Schlüsselwörtern: Sie markieren Ihre Schnittstelle sowohl mit `__gc` als auch mit `__interface`. Den Schnittstellennamen mit einem großen *I* zu beginnen, ist zwar nicht obligatorisch (wird nicht vom Compiler vorgeschrieben), aber gute Tradition, und ich kann Ihnen nur empfehlen, sich ebenfalls an diese Konvention zu halten. Verwaltete Schnittstellen unterliegen sämtlichen Beschränkungen, die bereits für verwaltete Klassen aufgelistet wurden, und noch einigen weiteren:
- Alle Elemente der Schnittstelle müssen `public` sein.
- Schnittstellen dürfen keine Membervariablen enthalten. (Programmierer, die bereits in Java Schnittstellen definiert haben, könnten annehmen, dass sie statische Membervariablen verwenden können, doch dem ist nicht so.)
- Schnittstellen dürfen ihre Methoden nicht implementieren. (Sie müssen ausnahmslos rein virtuell sein.)

- Schnittstellen dürfen keinen Konstruktor oder Destruktor enthalten.
- Schnittstellen dürfen keine Operatoren überladen.
- Schnittstellen können nur von anderen verwalteten Schnittstellen abgeleitet werden.

Durch die Deklaration mit dem Schlüsselwort `__interface` ändern sich die Standardeinstellungen für die Klasse. Normale C++-Klassen haben als voreingestelltes Zugriffsrecht `private`. Betrachten Sie folgende Klasse:

```
class Point
{
private:
   int x;
   int y;
};
```

Sie hätten die Definition auch folgendermaßen schreiben können:

```
class Point
{
   int x;
   int y;
};
```

Das standardmäßig geltende Zugriffsrecht ist `private`. Das standardmäßig geltende Zugriffsrecht für Schnittstellen ist dagegen `public`.

Die Funktionen (oder Methoden) normaler C++-Klassen sind nicht virtuell, es sei denn, Sie deklarieren sie als virtuell. Wenn Sie eine Funktion als virtuell deklariert haben, die Funktion aber nicht in der Klasse implementieren wollen, müssen Sie dies durch die etwas merkwürdige Syntax `=0` am Ende der Funktionsdeklaration mitteilen:

```
virtual void SomeFunction(int a, int b) =0 ;
```

Wenn Sie jedoch eine Klasse als Schnittstelle markieren, sind alle Methoden automatisch rein virtuell.

Aus den beiden beschriebenen Voreinstellungen folgt, dass die oben definierte `ISomething`-Schnittstelle auch wie folgt hätte geschrieben werden können:

```
__gc __interface ISomething
{
public:
   virtual void UsefulMethod()=0;
};
```

Die verkürzte Notation ist bequem, aber vergessen Sie nicht, welche Auswirkung das Schlüsselwort `__interface` für die Standardeinstellungen der Klasse haben kann.

20.5.2 Ableitung von verwalteten Schnittstellen

Wenn eine speicherverwaltete Klasse von einer verwalteten Schnittstelle abgeleitet wird, muss sie alle Methoden der Schnittstelle implementieren. Code, der einen Zeiger auf eine Instanz der Schnittstelle erwartet, akzeptiert ohne Murren auch Zeiger auf Klassen, die von der Schnittstelle abgeleitet sind. Auf diese Weise sorgt der Compiler für Typensicherheit beim Umgang mit Schnittstellen, was recht angenehm ist.

Hier ein Beispiel: Stellen Sie sich ein Bankkonto vor. Sicher besitzen Sie selbst eines, und vermutlich wird das Geld auf Ihrem Konto verzinst. Die meisten Guthaben, insbesondere Sparbücher, werden verzinst. Nehmen wir weiter an, Sie bekämen jeden Monat per Post die Kontoauszüge zugeschickt oder Sie würden – im Falle eines Sparbuches – Besuche in Ihrer Filiale dazu nutzen, das Sparbuch aktualisieren zu lassen. Wie könnte dies als C++-Code aussehen?

Ich persönlich würde mit einer Klasse `CAccount` beginnen, in der alle Gemeinsamkeiten von Bankkonten, wie zum Beispiel der Kontostand, zusammengefasst sind. Des Weiteren gehören zu jedem Konto ein Besitzer und eine Aufzeichnung der vorgenommenen Transaktionen. Danach würde ich untergeordnete Klassen, sprich abgeleitete Klassen, erzeugen, die `CAccount` erben. Für den Anfang dürften `CSavingAccount` und `CStatementSavingAccount` reichen. Statt nun aber diesen Klassen auf traditionelle Weise Methoden hinzuzufügen, würde ich einige Schnittstellen definieren: `IStatement` für Bankkonten, die Kontoauszüge (statements) ausdrucken, `IPassbook` für Konten, die Sparbücher (passbooks) führen, und `IInterestEarning` für diejenigen, die Zinsen (interests) einbringen. Listing 20.4 zeigt die Basisklassen und Schnittstellen dieses Modells.

Listing 20.4: Basisklassen und Schnittstellen für das Bank-Beispiel

```
__gc __interface IPassbook
{
   void Print();
};


__gc __interface IStatement
{
   void Print();
};

__gc __interface IInterestEarning
{
```

```
   void IncreaseBalanceByInterest();
};

__gc class CAccount
{
 // alles, was den verschiedenen Arten von Bankkonten
 // gemeinsam ist: Kontostand, Besitzer etc.
protected:
   int balance; // in Cent, um Rundungsfehler zu
                // vermeiden
};
```

Die Schnittstellen definieren jeweils nur eine einzelne Methode. (In einem realen Modell würden sie vermutlich mehrere, miteinander verwandte Methoden definieren.) Die Implementierung dieser Methoden bleibt den Klassen überlassen, die diese Schnittstellen erben. Listing 20.5 zeigt eine radikal vereinfachte Version von `CSavingAccount`.

Listing 20.5: Mögliche Implementierung von `CSavingAccount`

```
__gc class CSavingAccount: public CAccount,
                           public IPassbook,
                           public IInterestEarning
{
public:
   void Print()
   {
   Console::WriteLine("Sparbuch: Einlage beträgt {0}
➥Cent", __box(balance));
   // ein echtes Buchhaltungssystem würde die
   // Aufzeichnungen der Transaktionen durchgehen und
   // noch nicht ausgedruckte Transaktionen ins Sparbuch
   // eintragen und dann als "gedruckt" markieren
   }
   void IncreaseBalanceByInterest()
   {
   balance += 100;  // 1 Euro pro Monat
   // ein echtes Buchhaltungssystem würde die
   // Aufzeichnungen der Transaktionen durchgehen und
   // dabei die monatlichen und täglichen Zinsen
   // berechnen und zum Schluss den Kontostand um den
   // errechneten Betrag erhöhen.
   }
private:
   // andere Membervariablen und -funktionen
};
```

Diese Klasse erbt von einer speicherverwalteten Klasse und zwei verwalteten Schnittstellen. Alle Methoden aus den Schnittstellen werden von ihr implementiert. Listing 20.6 zeigt eine sehr einfache Implementierung von `CStatementSavingAccount`:

Listing 20.6: Mögliche Implementierung von `CStatementSavingAccount`

```
class CStatementSavingAccount: public CAccount,
                               public IStatement,
                               public IInterestEarning
{
public:
   void Print()
   {
   Console::WriteLine("Kontoauszug: Konto steht bei {0}
➥Cent", __box(balance));
   // ein echtes Buchhaltungssystem würde die
   // Aufzeichnungen der Transaktionen durchgehen und
   // die letzten Transaktionen als Kontoauszüge
   // ausdrucken und dann das Datum für den letzten
   // Ausdruck der Kontoauszüge aktualisieren
   }
   void IncreaseBalanceByInterest()
   {
   balance += 100;  // 1 Euro pro Monat
   // ein echtes Buchhaltungssystem würde die
   // Aufzeichnungen der Transaktionen durchgehen und
   // dabei die monatlichen und täglichen Zinsen
   // berechnen und zum Schuss den Kontostand um den
   // errechneten Betrag erhöhen.
   }
private:
   // andere Membervariablen und -funktionen
};
```

Nachdem die beiden Arten von Konten implementiert sind, kann ich eine einfache `main()`-Funktion aufsetzen, die die Konto-Klassen verwendet und testet. Listing 20.7 enthält ein Beispiel für ein solches Testprogramm.

Listing 20.7: Testprogramm für das Bank-Beispiel

```
int main(void)
{

   CSavingAccount* save = new CSavingAccount();
   save->Print();
   save->IncreaseBalanceByInterest();
   save->Print();

   Console::WriteLine("-------------");

   CStatementSavingAccount* statement =
                         new CStatementSavingAccount();
   statement->Print();
   statement->IncreaseBalanceByInterest();
   statement->Print();

   return 0;

}
```

Wenn Sie diesen Code kompilieren und ausführen, erzeugt er erwartungsgemäß folgende Ausgabe:

```
Sparbuch: Einlage beträgt 0 Cent
Sparbuch: Einlage beträgt 100 Cent
-------------
Kontoauszug: Konto steht bei 0 Cent
Kontoauszug: Konto steht bei 100 Cent
```

Besonders aufregend ist das nicht und es demonstriert auch nicht die eigentliche Leistungsfähigkeit der Schnittstellenprogrammierung. Betrachten Sie dazu die folgende globale Funktion:

```
void CreditInterest(IInterestEarning* i)
{
   i->IncreaseBalanceByInterest();
}
```

Dieser Code weiß nichts über Bankkonten, Sparbücher, `CSavingAccount` oder `CStatementSavingAccount`. Alles, was er kennt, ist die Schnittstelle `IInterestEarning`. Und dennoch können Sie der Funktion einen Zeiger auf eine der beiden Bankkonten-Klassen übergeben:

```
CreditInterest(save);
CreditInterest(statement);
```

In gleicher Weise könnten Sie in irgendeiner beliebigen dynamischen Datenstruktur alle möglichen Konten, die von `IInterestEarning` abgeleitet sind, zusammenfassen und nacheinander an die Funktion übergeben. Da `IInterestEarning` Basisklasse zu allen Konto-Klassen ist, kann der Compiler sicherstellen, dass die Zeiger auf die Klasseninstanzen korrekte Parameter für die Funktion sind. Analog könnten Sie alle Arten von Konten, die `IStatement` erben, an eine Funktion übergeben, die einmal im Monat ausgeführt wird und die Kontoauszüge ausdruckt. Auf diese Weise erleichtert die Schnittstellenprogrammierung dem Entwickler die Arbeit und verbessert die Wartbarkeit der Programme.

20.6 Eigenschaften

Wer mit verwalteten Klassen arbeitet, der möchte womöglich auch ein besonderes Element dieser Klassen nutzen: die Eigenschaften. Sicherlich haben Sie schon Klassen wie die aus Listing 20.8 geschrieben:

Listing 20.8: Eine Klasse mit einfachen `Get`- und `Set`-Methoden

```
class Employee
{
private:
   int salary; // in Euro
```

```
   // weitere Membervariablen
public:
   int getSalary() {return salary;}
   void setSalary(int s)
      {salary = s;}
   // weitere public-Methoden
};
```

Der Code der Funktion `setSalary()` könnte komplexer sein und mit einer adäquaten Fehlerbehandlung ausgestattet sein, doch ändert dies nichts an dem zugrunde liegende Konzept: eine private Variable mit öffentlichen (`public`) `get`- und `set`-Funktionen. Nicht jede Variable wird in dieser Weise behandelt. So sollte die Variable für den Kontostand aus dem obigen Beispiel nicht von außen gesetzt werden können, sondern nur von verschiedenen Geschäftsoperationen wie Einzahlungen und Abhebungen. Trotzdem gibt es viele Klassen, die das hier vorgestellte »get und set«-Paradigma verwenden.

Der große Vorteil dieses Paradigmas ist die Kapselung des bearbeitenden Codes. Ich kann den Code der `get`- oder der `set`-Methode ohne Probleme noch nach der teilweisen Fertigstellung des Systems oder in einer noch späteren Phase des Entwicklungsprozesses um eine passende Fehlerbehandlung oder anderen Code erweitern. Ich kann sogar den Typ der Variable verändern und den Code an diese Änderung anpassen. Kapselung ist eine wunderbare Sache, auf die kein guter Programmierer leichtfertig verzichten wird.

Die Verwendung von `get`- und `set`-Funktionen ist aber auch mit einem Nachteil verbunden: der eher unschöne Code. Wenn `e` ein Zeiger auf eine Instanz der Klasse `Employee` ist und Sie das Gehalt des zugehörigen Angestellten um 5000 Euro erhöhen möchten, sieht der resultierende Code beispielsweise wie folgt aus:

```
e->setSalary(e->getSalary() + 5000);
```

Der Code ist korrekt, jedoch nicht sehr schön. Im .NET Framework hingegen können Sie die Rose ohne Dornen genießen. Sie müssen lediglich die Definition von `Employee` ein wenig abwandeln:

Listing 20.9: Eigenschaften in einer verwalteten Klasse

```
__gc class Employee
{
private:
   int salary; // in Euro
   // weitere Membervariablen
public:
   __property int get_Salary()
      {return salary;}
```

```
   __property void set_Salary(int s)
      {salary = s;}
   // weitere public-Methoden
};
```

Jetzt verfügt die Klasse über eine Eigenschaft namens `Salary`. Diese steht in direkter Verbindung zur Membervariablen `salary`. Dies muss allerdings nicht so sein. Tatsächlich dürfen der Name der Eigenschaft und der Name der Membervariable nicht identisch sein, weshalb in der `Employee`-Klasse der Eigenschaftsname (der in den Funktionsnamen auf die Präfixe `get_` und `set_` folgt) `Salary` lautet, während die Membervariable den Namen `salary` trägt.

Wenn Sie im Code auf die Eigenschaft zugreifen, behandeln Sie sie wie eine `public`-Membervariable.

```
e->Salary = 10000;
e->Salary = e->Salary + 5000;
```

Sie können alle C++-Operatoren auf die Eigenschaft anwenden:

```
e->Salary += 5000;
```

Und im Hintergrund werden stets Ihre `get`- und `set`-Funktionen mit dem integrierten Fehlerbehandlungscode aufgerufen. Sie nutzen die Vorteile der Kapselung und Ihr Code ist trotzdem so leicht zu lesen, als wenn alle Ihre Membervariablen `public` wären.

20.7 Gemeinsame Verwendung von verwaltetem und nicht verwaltetem Code

Nachdem Sie sich nun von den Vorteilen der verwalteten Klassen überzeugen konnten, werden Sie diese sicher schon für Ihre Neuentwicklungen nutzen wollen. Wie aber sieht es mit bestehendem Code aus? Genügt es, allen Klassendeklarationen ein `__gc` voranzustellen und dann neu zu kompilieren? Sicher nicht, denn es wird in Ihrem Code bestimmt Klassen geben, die den in diesem Kapitel beschriebenen Anforderungen für verwaltete Klassen nicht genügen. Andererseits werden Sie wohl kaum Lust haben, alle Stellen im Code aufzuspüren, wo Objekte auf dem Stack erzeugt werden, und diese durch `new`-Allokationen zu ersetzen, nur um danach jede einzelne Zeile, in der das Objekt mit '`.`' verwendet wird, so abzuändern, dass der Objektzeiger mit '`->`' dereferenziert wird. Das ist eine Menge Arbeit.

Bedeutet das, dass Sie in Projekten, die viele nicht verwaltete Klassen enthalten, auf die vielfältigen Möglichkeiten der .NET Framework-Ressourcen verzichten müssen? Nein!

20.7.1 Die .NET-Unterstützung aktivieren

Wenn Sie ein neues .NET-Projekt anlegen, schreibt Visual Studio die beiden folgenden Zeilen an den Anfang Ihrer Quelldatei:

```
#using <mscorlib.dll>
using namespace System;
```

Diese beiden Zeilen können Sie ohne Probleme in jede Quelldatei übertragen, an der Sie arbeiten. Doch damit sind Sie noch nicht am Ziel Ihrer Wünsche. Zusätzlich müssen Sie noch die .NET-Compiler-Option `/CLR` setzen, die anzeigt, dass Ihr Projekt verwaltet werden soll. Wenn Sie ein neues .NET-Projekt anlegen, wird diese Einstellung automatisch vorgenommen. Um die Einstellung zu kontrollieren oder für ein nicht verwaltetes Projekt zu setzen, gehen Sie wie folgt vor:

1. Wechseln Sie in die Klassenansicht oder zum Projektmappen-Explorer, indem Sie den Befehl ANSICHT/PROJEKTMAPPEN-EXPLORER beziehungsweise ANSICHT/KLASSENANSICHT aufrufen.
2. Klicken Sie auf den Namen der Projektmappe.
3. Wählen Sie ANSICHT/EIGENSCHAFTENSEITEN. Dies ist der letzte Befehl im Menü. Lassen Sie sich nicht von dem EIGENSCHAFTENFENSTER-Befehl verführen.
4. Das EIGENSCHAFTENSEITEN-Dialogfeld wird angezeigt. Sie sollten sich jetzt bei *Allgemein*, direkt unter dem *Konfigurationseigenschaften*-Ordner, befinden. Falls nicht, expandieren Sie gegebenenfalls die Konfigurationseigenschaften und klicken dann auf *Allgemein*.
5. Setzen Sie die Eigenschaft *Verwaltete Erweiterungen verwenden* auf *Ja*.
6. Klicken Sie auf OK, damit die Einstellungen wirksam werden.

Nachdem Sie den Compilerschalter gesetzt haben, müssen Sie noch die `#using`-Zeile in Ihre Quelldateien einfügen, die die Common Language Runtime-Bibliothek verfügbar macht. Die zweite Zeile (die den `System`-Namensraum einbindet) müssen Sie nicht unbedingt einfügen – wenn Sie möchten, können Sie die `System`-Klassen auch über ihre vollen Namen aufrufen.

20.7.2 Nicht verwalteten Code identifizieren

Code und verwaltete Typen sind zwei unterschiedliche Dinge. Verwalteter Code wird im Gegensatz zu nicht verwaltetem Code beim Kompilieren in Intermediate Language (MSIL) übersetzt und in der CLR ausgeführt. Die Entscheidung für oder gegen verwalteten Code hat nichts mit der Entscheidung zu tun, ob eine Klasse verwaltet (sei es Speicherverwaltung, Werttyp-

Klasse oder Schnittstelle) sein soll oder nicht. Beispielsweise können Sie verwalteten Code schreiben, der ausschließlich mit nicht verwalteten Typen arbeitet. Sowie Sie den `/CLR`-Compilerschalter aktiviert haben, wird der gesamte Code des Projekts verwaltet.

Um innerhalb verwalteten Codes mit nicht verwalteten Typen arbeiten zu können, gibt es die Möglichkeit, Klassen mit dem Schlüsselwort `__nogc` zu deklarieren und auf diese Weise als nicht verwaltet zu kennzeichnen. Die Memberfunktionen einer solchen Klasse stellen jedoch weiterhin verwalteten Code dar und verursachen zum Beispiel die mit der Speicherbereinigung verbundenen Mehrkosten.

Gängiger ist es, sich für eine Weile ganz aus der Codeverwaltung auszuklinken. Hierfür gibt es spezielle `pragma`-Direktiven.

```
#pragma unmanaged
// hier beginnt der nicht verwaltete Code
#pragma managed
// zurück im verwalteten Code
```

Mit dieser Technik können Sie verwalteten und nicht verwalteten Code in ein und derselben Quelldatei verwenden, was doch recht ungewöhnlich und bemerkenswert ist. Sie werden auf diese Weise viel Übung im Boxing und Fixieren von Zeigern bekommen und höchstwahrscheinlich werden Sie auch ein oder zwei Hüllklassen schreiben müssen, aber keine andere Sprache bietet Ihnen Vergleichbares – und was man braucht, das braucht man.

Für den Fall, dass alter und neuer Code in unterschiedlichen Dateien stehen, gibt es allerdings noch eine bessere Lösung: Sie können Ihre alten Objekte in COM-Komponenten hüllen und den neuen Code als .NET-Objekte implementieren. Beide können hervorragend miteinander kommunizieren, denn der .NET-Code betrachtet COM-Komponenten als .NET-Objekte (siehe Kapitel 23, »COM-Interoperabilität«).

20.8 Leseleitfaden

Für welchen Weg Sie sich auch entscheiden, denken Sie daran, dass Ihnen die Vorzüge von .NET offen stehen, ohne dass Sie dafür Ihren alten Code aufgeben (oder portieren) müssen. Und wenn Ihnen .NET nichts bieten kann, was Sie interessiert, so stellt dies auch kein Problem dar. Sie können mit Visual Studio .NET auch nach wie vor soliden, »klassischen« Win32-Code schreiben (siehe erste Hälfte des Buches). Wenn Sie mehr über .NET lernen wollen und erfahren möchten, welche Vorzüge .NET neben der Speicherbereinigung und der Vermeidung von Speicherlecks zu bieten hat, lesen Sie in dieser Hälfte des Buches weiter. Es gibt noch eine Menge zu entdecken.