Zurück zu den Wurzeln

Objektorientiertes Programmieren in Pascal

Dank einiger geschickt gewählter Fachausdrücke ist das Objektorientierte Programmieren (OOP) in die höchsten Höhen der theoretischen Informatik entschwebt. Wir holen OOP wieder auf die Erde zurück.

Hinter Ehrfurcht gebietenden Fachausdrücken steckt oft nur etwas Banales. Diese ketzerische Aussage trifft für viele Begriffe zu, die sich um das objektorientierte Programmieren ranken. Im Nebel der modernen Programmier-Mysterien erscheint selbst der Begriff Objekt verschwommen. Bei den Pascal-Dialekten, die um objektorientierte Funktionen erweitert worden sind, lüftet sich der Nebel schnell. Dort ist ein Objekt nichts Geheimnisvolles, sondern schlicht eine Variable eines bestimmten Typs. So ist in Quick Pascal und in Turbo Pascal ein Objekt eine Sonderform des Records. Wie man in Quick Pascal zum Beispiel das Objekt „Figur“ deklariert, sehen Sie in *Listing 1*.

Dieser Record (Objekt) enthält neben den üblichen Daten auch Prozeduren und/oder Funktionen, die letztlich aber auch nur Daten sind, nämlich die Adressen dieser Prozeduren. Procedure oder Function schreibt man zwar, es sind auch solche, aber das sagt der feine objektorientierte Programmierer nicht. Er nennt das ganze vornehm Methode.

Wie auch immer Sie den Datentyp Figur des obigen Beispiels nennen: Nachdem Sie den Datentyp Figur deklariert haben, legen Sie mit einem schlichten „var f1: Figur;“ eine statische Variable an. Sie können aber auch eine dynamische Variable definieren, indem Sie einen Zeiger auf die Variable deklarieren und ihr Speicherplatz zuteilen. Die üblichen Programmierregeln für statische und dynamische Variablen gelten nach wie vor. Auch beim objektorientierten Programmieren kann somit der Platz für die Daten eines Objekts im Daten- oder Stacksegment oder im Heap liegen. In diesem Sinne spricht man von statischen oder dynamischen Variablen oder Objekten.

Dynamisch kontra statisch

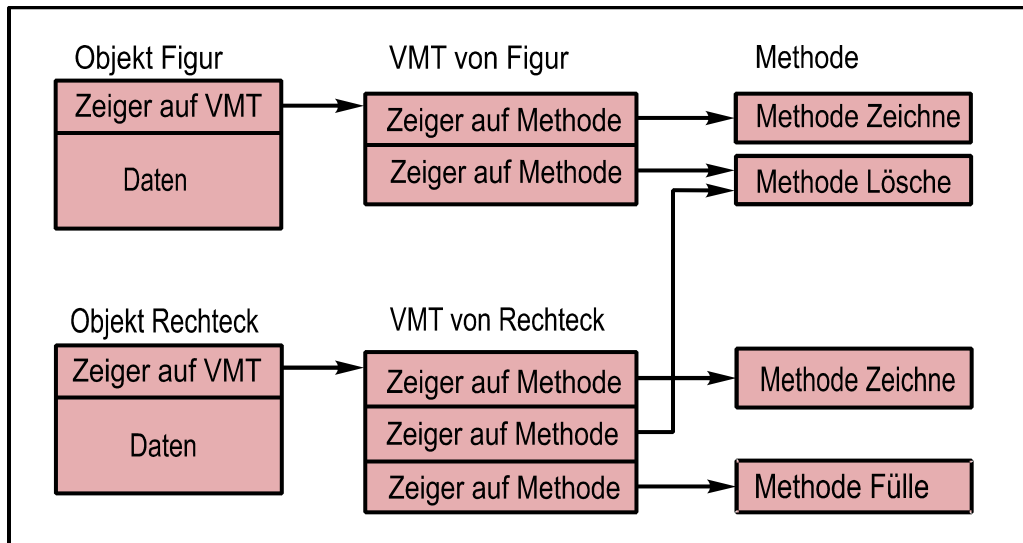
Unter den Objektorientierten laufen oft heiße Diskussionen, ob dynamische oder statische Objekte besser seien und ob ein Compiler unbedingt statische Objekte unterstützen muß. Der Vorteil dynamischer Objekte besteht ganz klar darin, daß man ihren Speicherplatz je nach Bedarf im Heap belegen und wieder freigeben kann. Dagegen bleibt der Speicherplatz für global deklarierte statische Objekte während des ganzen Programmlaufs gebunden. Wenn Sie in einem Programm sehr viele Objekte anlegen, sind daher dynamische Objekte die richtige Wahl. Beim PC mit seinem 80x86-Prozessor kommt hinzu, daß man mit dynamischen Objekten das eh immer zu kleine Daten- oder Stacksegment nicht belastet. Der Nachteil dynamischer Objekte – und dynamischer Variablen überhaupt – ist minimal: Sie können nicht direkt, sondern nur über einen Zeiger angesprochen werden. Damit ist der Zugriff um einige hundert Nanosekunden langsamer als bei statischen Objekten/Variablen. Außerdem sind lokal deklarierte statische Objekte, das heißt Objekte, die innerhalb einer Prozedur oder Funktion deklariert worden sind, alles andere als statisch. Beim Eintritt in die Prozedur wird das Objekt auf dem Stack angelegt und beim Verlassen wieder abgebaut. Hinzu kommt, daß ein gut strukturiertes Programm in viele kleine Funktionen zerlegt ist und folglich auch die statischen Objekte als Funktionsargumente übergeben werden. Dies geschieht dann hoffentlich „by reference“, also wie bei den dynamischen Variablen über Zeiger. Lange Rede, kurzer Sinn: Wenn ein OOP-Compiler keine statischen Objekte unterstützt, ist das in Ordnung, kann er es, ignorieren Sie dieses Feature einfach.

Virtuelle Methoden

Bisher haben wir immer nur die Objekte als Datensammlung betrachtet, aber die Methoden außer acht gelassen. Wie Sie dem Bild zu Listing 1 entnehmen können, besteht das Objekt „Figur“ und sein Nachkomme „Rechteck“ nur aus einem Zeiger auf die virtuelle Methodentabelle (VMT) und ihren Daten. Im Extremfall – das Objekt hat keine Daten – besteht es nur aus einem Zeiger. Daß der Compiler sich zusätzlich einiges notiert – zum Beispiel die Größe des Objekts – ist eine andere Sache. Auf jeden Fall hat hier jedes Objekt einen Zeiger auf eine eigene VMT. Sie sehen, daß die virtuelle Methodentabelle nichts anderes als eine Sprungtabelle ist. Diese Tabelle besteht nur aus Zeigern,

die schließlich auf die Methoden hinweisen. Warum das so sein muß, zeigt das Objekt „Rechteck". Es hat mit dem Schlüsselwort OVERWRITE verkündet, daß es die Prozedur (Methode) „Zeichne" nicht von seinem Vorgänger übernimmt. Folglich muß dieser Zeiger auf eine andere Methode gestellt werden. Die Prozedur „Lösche" hingegen wurde vererbt, weshalb dieser VMT-Eintrag auf dieselbe Methode zeigt. Die hinzugefügte Methode „Fülle" braucht natürlich auch noch einen Zeiger. Prinzipiell ist das berühmte „Vererben" in der OOP-Technik nur ein Kopieren und Ändern der VMT. Solange nur vererbt wird, also die Objekte einer Klasse sich nicht unterscheiden, ist der Compiler sogar so clever, alle Objekte auf die selbe VMT zeigen zu lassen.

Dahinter steckt aber noch ein anderer Sinn, als nur die Aufgabe, den Code gemeinsam genutzter Methoden nur einmal im Speicher zu halten.



Das ist das ganze OOP-Vererbungsprinzip: Zeiger, die auf dasselbe Unterprogramm hinweisen.

Dazu betrachten Sie einmal *Listing 2*, welches Listing 1 fortsetzt. Wenn Sie nun den Aufruf von „ZeichneNeu" über die VMTs im Bild verfolgen, dann werden Sie feststellen, daß immer die richtige Methode „Zeichne" aufgerufen wird. Wenn diese Methode nicht virtuell wäre, passierte Schlimmes. Woher soll denn der Compiler bei „ZeichneNeu" wissen, daß „Rechteck" eine andere Methode als „Figur" zum Zeichen hat?

Die Vorteile virtueller Methoden sind eindeutig. Zum einen ist die Wiederverwendbarkeit von Code sichergestellt, zum anderen ist das System klar und logisch. Mit nur einer Methode „ZeichneNeu" kann man jedes Objekt neu zeichnen, ohne sich um weitere Details zu kümmern. Wenn Sie eine Objekt-Bibliothek kaufen, achten Sie darauf. Ein Blick in das Handbuch bringt da schnell Klarheit. Wenn es zum Beispiel in einer Grafikbibliothek die Objekte Rect, Ellipse und Poly und nur eine Methode ReDraw gibt, ist das in Ordnung. Existieren hingegen die Methoden ReDrawRect, ReDrawEllipse und ReDrawPoly, dann hat jemand nur die alte Technik auf die Schnelle in nicht virtuelle Methoden umgesetzt.

Der nächste Ausdruck, der zur restlosen Verwirrung einiger Leute geführt hat, ist das späte Binden. Ein oft gehörter Kommentar dazu lautet: „Durch das späte Binden virtueller Methoden wird eine zeitintensive Suche zur Laufzeit erforderlich... deshalb sind statische Objekte vorzuziehen". Das Mißverständnis kommt nur zustande, weil die Compilerbauer zwischen Binden und Linken unterscheiden. So reduziert sich das Binden bei einem Methodenaufruf wie „Zeichne" im Assembler-Code im einfachsten Fall auf ein schlichtes „CALL Adresse_von_Zeichne".

Was da tatsächlich passiert, hängt davon ab, ob die Objekte und die Methoden statisch oder dynamisch sind, und wie sie miteinander kombiniert werden. Schauen wir uns erst einmal am Beispiel eines 80x86-Rechners den Datenteil an.

Statische globale Objekte werden im Datensegment abgelegt. Der Bezeichner, zum Beispiel „Rechteck“, steht für eine Adresse. Der Zugriff läuft wie bei Variablen über ein schlichtes „mov ax, Rechteck“. Ist das Objekt dynamisch, sprich, gibt es nur einen Zeiger auf „Rechteck“, setzt der Compiler diesen Code ein:

```
LES DI, Rechteck
MOV AX, ES:[DI + Datenoffset]
```

In beiden Fällen macht der Compiler die Arbeit, zur Laufzeit des Programms bleibt nichts mehr zu tun.

Bei den Methoden wird es etwas komplizierter. Bei einem statischen Objekt mit virtuellen Methoden sieht der Assemblercode beim Binden etwa wie in *Listing 3* aus. Wenn das Objekt dynamisch ist, kommt einfach noch eine Indirektion *Listing 4* hinzu.

Die Beispiele zeigen, daß unabhängig von der Binde-Strategie der Compiler zwar mehr oder weniger spät, aber immer noch während des Compilierens, die Adressen oder Offsets feststellt und den endgültigen Code erzeugt. Während der Laufzeit des Programms passiert da nichts mehr. Wie Sie aber auch sehen, sind die Unterschiede in der Adressierung auf einem 80x86-PC erheblich, und FAR-Pointer kostet da bekanntlich Zeit. Dennoch sollte man deshalb nicht die Vorteile dynamischer Objekte und virtueller Methoden aufgeben, sondern das richtige OOP-System einsetzen.

Als richtig in diesem Sinne sehe ich C++, Quick Pascal und Turbo Pascal an. Diese Hybrid-Systeme erlauben sowohl die objektorientierte als auch die klassische Programmierung. Den OOP-Teil sollte man für grafische Objekte, komplexe Datenstrukturen, die Benutzerschnittstelle oder allgemein für übergeordnete Funktionen einsetzen. Eventuelle Unterschiede in der Programmausführungszeit werden wenig auffallen. Systemnahe Funktionen sollte man aber in der traditionellen Technik schreiben. Ein objektorientierter BIOS-Aufruf ist Humbug.

Konstruktiv und Destruktiv

Bliebe zum Schluß noch die Sache mit den Konstruktoren und Destruktoren zu erklären. Um mit dem Resümee zu beginnen: Constructor und Destructor sind sehr leistungsfähige Bestandteile von C++. Quick Pascal braucht sie nicht, weil der Compiler diese Jobs automatisch erledigt. Turbo Pascal hat sie, doch nur, weil ihm die Quick-Pascal-Automatik fehlt. Die Power von C++ haben die beiden Pascal-Compiler nicht.

Konstruktoren und Destruktoren sind prinzipiell nichts anderes als besondere Methoden zum Initialisieren und Beendigen von Objekten. Ihre typischen Aufgaben sind die Zuteilung und die Freigabe des Speichers für die Objekte. In C++ muß man diese Methode nur deklarieren, die Aufrufe setzt der Compiler automatisch ein. In Turbo-Pascal muß man sie selber aufrufen. Der große Vorteil von C++ ist, daß es die Destruktoren von Objekten auch aufruft, sobald diese den Gültigkeitsbereich von Funktionen verlassen. So eine Speicheraufräum-Automatik ist natürlich sehr praktisch. Turbo Pascal braucht den Konstruktor für virtuelle Objekte, erst sein Aufruf legt die VMT an *(Listing 5)*.

In Quick Pascal deklariert man die Konstrukte einfach nicht und schreibt auch nur „new(fp) oder „dispose(fp)“. Sie erkennen daran sehr schön, daß die Turbo-Lösung etwas umständlich ist. In der VMT steht nämlich, wieviel Bytes zuzuteilen oder freizugeben sind. Der ganze Unterschied ist also, daß in Quick Pascal der Compiler diese Angabe selber findet, während er in Turbo-Pascal vom Programmierer erst mit der Nase darauf gestoßen werden muß.

Natürlich sind die paar Zeichen, die dafür mehr zu tippen sind, kein Argument gegen Turbo-Pascal. Viel wichtiger ist eine andere Frage, nämlich C++ oder OOP-Pascal? Die erste Antwort lautet: Bleiben Sie bei Ihrer Lieblingssprache. Die Antwort für alle, die keine Lieblingssprache haben, lautet: C++ kann mehr als alle objektorientierten Pascal-Varianten. Aber wie in C üblich, lauern auch mehr Gefahren, in ein Programm schwer zu entdeckende Fehler einzubauen.

Peter Wollschlaeger/st

Listing 1. Ein Objekt ist in Quick Pascal eine Sonderform des Records.

```
type
  Figur = OBJECT
    x,y: word;
    procedure Zeichne;
    procedure Lösche;
  end;

  Rechteck = OBJECT(Figur)
    procedure Zeichne; OVERWRITE;
    procedure Fülle;
  end;
```

Listing 2. Die Zeiger in der VMT sorgen auch dafür, daß „ZeichneNeu“ immer zur richtigen Methode führt.

```
var
  f: Figur;
  r: Rechteck;

procedure ZeichneNeu( o: Rechteck );
begin
  o.Lösche;
  o.Zeichne;
end;

begin
  ...
  ZeichneNeu( f );
  ZeichneNeu( r );
  ...
end.
```

Listing 3. Assembler-Code eines statischen Objekts mit virtuellen Methoden.

```
MOV  DI, [Rechteck+0] ; Adresse Sprungtabelle
PUSH DS               ; Zeiger auf sich selbst
LEA  AX,Rechteck
PUSH AX               ; auf den Stack
CALL FAR [DI+Offset_des_Vektors]
```

Listing 4. Assembler-Code eines dynamischen Objekts mit virtuellen Methoden.

```
LES  DI,Rechteck    ; Adresse Sprungtabelle
PUSH ES             ; Zeiger auf sich selbst
PUSH DI
LES  DI, ES:[DI]
CALL FAR ES:[DI+Offset_des_Vektors]
```

Listing 5. Turbo Pascal benötigt zum Anlegen von Objekten einen Konstruktor.

```
TYPE
  Figur = OBJECT
    x,y: word;
    procedure Zeichne; virtual;
    procedure Lösche;
    constructor init;
    destructor  ende;
  END;

var fp : ^Figur;
...

begin
  new(fp, init);
  ...
  dispose(fp, ende);
end.
```