4.1 Von C nach C++/Einführung in die Objektorientierung

4.1.1 Warum eine neue Programmiersprache?

Hier geht es darum, von der Programmiersprache C zu einer Erweiterung, der Programmiersprache C++ zu kommen. Aber warum?

Tatsächlich ist es nicht nötig, neben C eine weitere, ähnliche Sprache zu lernen, es sei denn, man braucht sie für ein aktuelles Projekt. C hat alles, was eine Programmiersprache braucht [1].

Eine neue Programmiersprache zu lernen ist nur dann sinnvoll, wenn die neue Sprache auch neue Möglichkeiten bietet. Und genau das ist bei C++ der Fall. C++ ist nämlich eine *objektorientierte Sprache*.

Objektorientierung bietet für größere Projekte zusätzliche Möglichkeiten. Durch eine andere Herangehensweise (objektorientiertes Design) werden größere Projekte besser handhabbar; zusätzlich fallen als Nebeneffekt häufig wiederverwendbare Module an.

Wenn es um größere Programmierprojekte geht, so werden viele davon heute in C++ begonnen. Das liegt daran, dass C++ einerseits objektorientiertes Programmieren erlaubt, andererseits aber vollständig abwärtskompatibel zu C ist. Jedes C-Programm ist auch mit einem C++-Compiler übersetzbar[2].

Die fast schon regelmäßige Erweiterung der Sprache im drei-Jahres-Rhythmus sichert C++ eine gleichbleibende Aufmerksamkeit und führt dazu, dass modische Trends (das gibt es wirklich im Programmier-Bereich!) zügig aufgegriffen werden.

Genau darin liegt aber auch ein Nachteil von C++: Wenn man alle Möglichkeiten von C++ benutzt, dann ist die Sprache ziemlich umfangreich. Das muss aber niemand: Man kann (siehe oben) sogar C-Programme als C++ verkaufen.

Und man hat den Vorteil, dass man beim Umstieg auf ähnliche Sprachen (Objective-C, Java oder C#) wenig hinzulernen muss, abgesehen von den riesigen Standard-Bibliotheken, die bei Java- und C#-Programmen mitgeschleppt werden müssen.

4.1.2 Ausgangsbasis: Prozedurale Programmierung

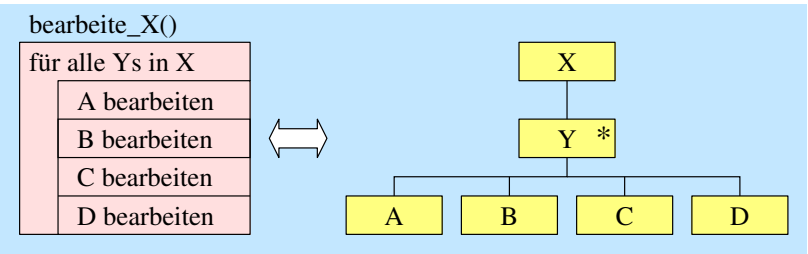

Abbildung 1: Programm- und Datenstrukturen in der prozeduralen Programmierung

Die prozedurale Programmierung ist Standard bei C und vielen anderen Programmiersprachen. Programmstrukturen (=Funktionen, Sequenzen, Schleifen und Verzweigungen) und Datenstrukturen (=Records+Arrays+...) liegen nebeneinander vor. Mit Funktionen greift man auf die Daten zu (Abbildung 1).

Es fällt auf, dass die Programm- und die Datenstrukturen oft aufeinander bezogen sind. So greift man auf die Elemente eines Records nacheinander, also in einer Sequenz zu. Auf ein Array greift man in einer Zählschleife zu. Deshalb sind die Jackson-Diagramme für Daten- und für Programmstruktur auch manchmal ähnlich.

Die Programmierung größerer Projekte erfolgt mit Hilfe der Top-Down-Methode.

[1] Lediglich im Browser braucht man Javascript, das liegt aber nur an den Vorlieben der Browser-Produzenten in den 90er-Jahren.

[2] Es sei denn, man hat einen Variablennamen benutzt, der bei C++ ein Schlüsselwort ist. Solche oder ähnliche Missgeschicke kann man mit wenig Aufwand ändern.

4.1.3 Modulare Programmierung

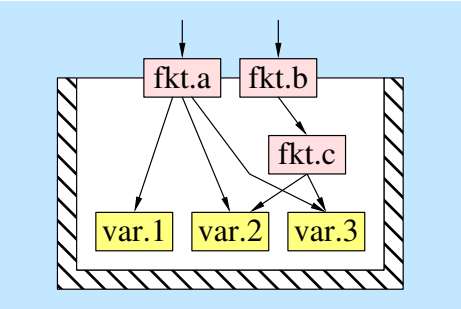

Abbildung 2: Programm- und Datenstrukturen in der modularen Programmierung

Zwei Ideen führen zur modularen Programmierung:

a) Wenn Programm- und Datenstrukturen zusammengehören, dann kann man sie auch gemeinsam organisieren. Diese Idee führt zur Aufteilung eines Programmes in Module (Modularisierung).

b) Wenn man Module bildet, dann kann man auch dafür sorgen, dass von außen der Zugriff auf die Daten nur noch über die passenden Funktionen des Moduls stattfinden kann (Kapselung, siehe Abbildung 2).

Beispiele für modulare Programmiersprachen sind Modula, Modula-2 und ADA. Aber auch C erlaubt es, modular zu programmieren.

Die Programmierung größerer Projekte erfolgt hier so, dass zuerst Module erstellt werden; anschließend werden sie nach der Top-Down-Methode miteinander verbunden.

4.1.4 Datenbankprogrammierung

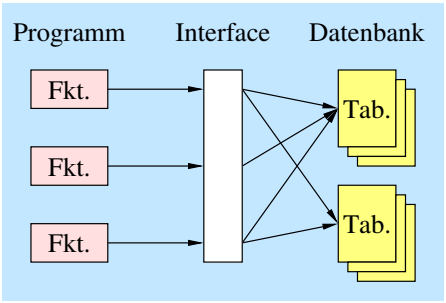

Abbildung 3: Programm- und Datenstrukturen in der Datenbankprogrammierung

Bei modularer Programmierung wird es etwas schwieriger, wenn mehrere Exemplare (*Instanzen*) der Datenstruktur eines Moduls gebraucht werden. Eine Lösungsmöglichkeit besteht darin, vom Programm aus eine relationale (oder eine hierarchische) Datenbank zu benutzen. Eine relationale Datenbank speichert seine Daten in Tabellen, so dass es kein Problem ist, beliebig viele Exemplare irgendwelcher Variablen (z. B. Records) zu speichern.

Fast alle Datenbanksysteme bieten eingebaute Zugriffsfunktionen, die man von einer Programmiersprache aus benutzen kann. So bekommt man vom Programm aus Zugriff auf die Daten in den Tabellen der Datenbank, kann sie lesen und verändern (Abbildung 3). So passiert eine Zweiteilung: Die Datenstrukturen liegen in der Datenbank, die Programmstrukturen im Programm. Beispiele

für diese Art der Programmierung sind immer Paare von Datenbanksystem und Programmiersprache: PERL und mySQL, C und MariaDB, C und Oracle, C und SAP usw.

Die Programmierung großer Projekt funktioniert nun so, dass zuerst die Datenbankstruktur entworfen wird. Danach folgen die Zugriffs-Operationen in der Programmiersprache; zum Schluss wird das Programm z. B. nach der Top-Down-Methode zusammengefügt.

4.1.5 Objektorientierte Programmierung (OOP)

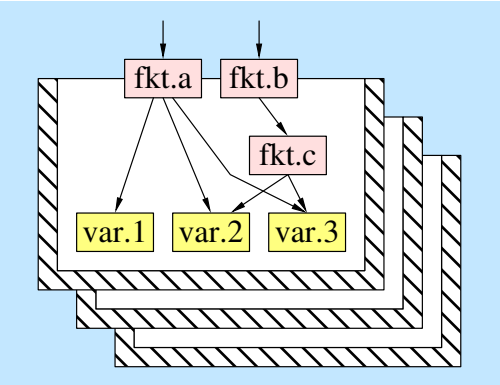

Abbildung 4: Programm- und Datenstrukturen in der objektorientierten Programmierung

Einen entgegengesetzten Ansatz wählt man bei der objektorientierten Programmierung (OOP). Wie bei der modularen Programmierung werden zusammengehörende Daten- und Programmstrukturen (also Variablen und Funktionen) zu einem Modul zusammengefasst. Statt Modul spricht man hier von *Klasse*. Anders als bei der modularen Programmierung kann man von einer Klasse mehrere Exemplare (so genannte Instanzen) erhalten, genauso wie man in C von einem Datentyp mehrere Variablen haben kann (Abbildung 4). Diese Exemplare nennt man *Objekte*. Beispiele für objektorientierte Programmiersprachen sind Simula-67, Smalltalk, C++, Modula-3, Oberon, Java und C#.

Weitere Aspekte der Objektorientierung sind Vererbung und Polymorphie. Vererbung ist die Anpassung einer vorhandenen Klasse an den eigenen Bedarf. Polymorphie (Gleichförmigkeit) meint, dass man Objekte ähnlicher Klassen gemeinsam auf die gleiche Art behandeln kann, z. B. in einer Programmschleife.

In der OOP ist es sinnvoll, neue und andere Diagrammformen für den Programmentwurf zu benutzen. Weit verbreitet sind die Diagramme der UML (*unified modeling language*, insgesamt 14 Diagrammtypen).

Die Programmierung größerer Projekte in der OOP erfolgt häufig in der Richtung Bottom-Up: Erst werden kleinere, dann größere Klassen erstellt und zum Schluss miteinander verbunden.

4.1.6 Ein erstes Programm in C++

Das Hello-World-Programm in C++ sieht so aus:

```
1 #include <iostream>
2 using namespace std;
3
4 int main(void)
5 {
6     cout << "Hello, World!" << endl;
7     return 0;
8 }
```

Zum Übersetzen des Programms braucht man einen Compiler. Es bietet sich der GNU-C++-Compiler an, der mit dem Befehl `g++` (oder `c++`) aufgerufen werden kann. Dieser Compiler verlangt es übrigens, dass bei einer Quelltextdatei der Dateiname auf `.cpp` endet.

Terminal

```
schueler@debian964:~$ g++ hello.cpp
schueler@debian964:~$ a.out
Hello, World!
```

Man sieht, für den Benutzer ist es gleich, mit welcher Art der Programmierung ein Anwendungsprogramm geschrieben wurde.

Zeile 1 Hier wird die Headerdatei `<iostream>` eingebunden, ein Gegenstück zu `<stdio.h>`. In älteren C++-Versionen wurde `<iostream.h>` eingebunden.

Zeile 2 In C++ können Funktionen und andere Elemente, die man von woanders her einbindet, einen Nachnamen haben. So haben Elemente aus der C++-Standard-Bibliothek den Nachnamen `std`. Ein gemeinsamer Nachname deutet hier also auf eine gemeinsame Herkunft hin. Man spricht von einem gemeinsamen *Namensraum* (auf englisch *namespace*). Das Objekt `cout` in Zeile 6 etwa heißt mit vollem Namen `std::cout`. Wenn man aus Gründen der Bequemlichkeit den Nachnamen weglassen möchte, muss man ihn wie in dieser Zeile eigens angeben. Mit dieser Zeile erreicht man also, dass man den Nachnamen `std` weglassen kann. Man kann hier (wenn nötig) auch weitere derartige Zeilen einfügen, um andere Nachnamen ebenfalls weglassen zu können.

Zeile 4 Das Wort `void` für den Inhalt einer leeren Parameterliste darf man in C++ weglassen (muss man aber nicht).

Zeile 6 `cout` ist das Gegenstück zu `printf`. Der Links-Schiebe-Operator `<<` bedeutet hier, dass die Zeichenkette in das Objekt `cout`, also auf den Bildschirm, geschoben werden soll. Mit dem Objekt `endl` wird das Zeilenende ebenfalls nach `cout` geschoben.

Das sieht für Außenstehende merkwürdig aus. Man sieht zwei Eigenschaften von C++:

a) Man braucht keinen Platzhalter mehr! Der Compiler weiß ja, welchen Datentyp `"Hello, World!"` hat und sucht selbst die richtige Funktion aus. Das heißt: Nie wieder falsche Platzhalter einsetzen.

b) Der Links-Schiebe-Operator wird hier auf kreative Weise anders verwendet. Das ist ein Pluspunkt, den C++ anderen Sprachen (wie Java oder C#) voraushat. In C++ kann man fast alle Operatoren neu definieren. So kann man etwa dafür sorgen, dass das Pluszeichen bei Strings tatsächlich zwei Strings aneinanderhängen kann.

4.1.7 Ein zweites Programm mit etwas OO

Hier ist ein anderes Programm, in das schon etwas Objektorientierung eingebaut wurde.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  struct datumtyp
6  {
7     int tag;
8     int monat;
9     int jahr;
10    void print(void)
11    {
12       cout << tag << "." << monat << "." << jahr;
```

```
13      }
14  };
15
16  int main(void)
17  {
18      cout << "Bitte geben Sie einen Text ohne Leerzeichen ein:";
19      string str;
20      cin >> str;                          // getline(cin, str);
21      cout << "Am ";
22      datumtyp neujahr={1,1,2025};
23      neujahr.print();
24      cout << " bin ich der Meinung: " << str << endl;
25      return 0;
26  }
```

Terminal

```
schueler@debian964:~$ g++ neujahr.cpp
schueler@debian964:~$ a.out
Hallo. Hier ist C++. Bitte geben Sie einen Text ohne Leerzeichen ein:
C++_ist_wunderbar!
Am 1.1.2025 bin ich der Meinung: C++_ist_wunderbar!
```

Zeile 2 In C++ gibt es eine Klasse (einen Datentyp) mit dem Namen `string`. Um sie zu benutzen, müssen wir die Headerdatei `<string>` einbinden. Bei dieser Klasse wird automatisch genug Speicherplatz reserviert.

Zeile 5 Ein Record wie in C.

Zeile 10 Dieser Record enthält eine Funktion (genannt *Methode*). Diese Funktion ist nur in dem Record benutzbar. Und sie liegt in einem Bereich, in dem sie die anderen Elemente des Records sehen und benutzen kann. Deshalb braucht sie keine Parameter.

Zeile 19 Nach der ersten Anweisung folgt hier eine Vereinbarung. `str` ist vom Typ `string`.

In C (zumindest vor C99) war es so, dass der Vereinbarungs- und der Anweisungsteil voneinander getrennt waren: Nach der ersten Anweisung durfte keine weitere Vereinbarung mehr folgen.

In C++ ist das anders: Vereinbarungen und Anweisungen dürfen beliebig gemischt werden. Man sollte diese Möglichkeit aber sparsam nutzen.

Zeile 20 Hier wird die Tastatureingabe aus dem Objekt `cin` mithilfe des Rechts-Schiebe-Operators nach `str` geschrieben.

Wie in C bei `scanf("%s",s)` wird auch hier nur ein Wort eingelesen. Will man eine ganze Zeile einlesen, dann lautet der Befehl: `getline(cin, str);`

Zeile 22 Hier wird ein Objekt (=Record-Variable) der Klasse (=des Datentyps) `struct datumtyp` vereinbart und initialisiert. Das Wort `struct` darf bei der Vereinbarung in C++ wegfallen, ebenso die Wörter `union`, `enum` und `class`. Das Wort `class` ist in C++ ein anderes Wort für `struct` (mit nur einem geringen Unterschied, siehe später).

Zeile 23 So wird in C++ eine Methode aufgerufen, also eine Funktion, die in einer Klasse vereinbart wurde. Genauso wie die Komponenten `tag`, `monat` und `jahr` bekommt auch `print()` eine Zuordnung zu einem bestimmten Objekt (=Record-Variable), in diesem Fall dem Objekt `neujahr`. Dadurch gibt `print` das Datum des richtigen Objekts aus.

Zeile 24 Mit `cout` wird auch der String `str` ausgelesen. Anders als bei `fgets` wurde hier kein Zeilenendezeichen im String gespeichert, so dass wir die Ausgabe mit `endl` abschließen.