



## Ähnlichkeit mit Unterschieden

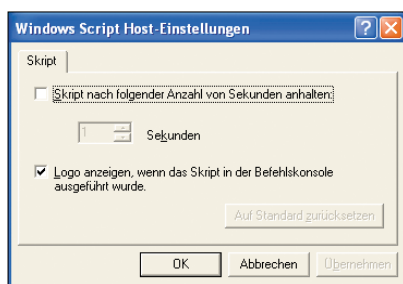
# Erst C – dann C++

C++ ist der **Nachfolger** der Sprache "C" und an vielen **Stellen** sehr ähnlich – an anderen aber wiederum **völlig anders**. Dennoch kann man guten Gewissens sagen, dass "C" mehr oder minder ein **Subset** von C++ darstellt.

THOMAS WÖLFER

**P**raktisch jedes C-Programm kann mit minimalen Änderungen auch als C++ kompiliert werden. Bei den Gemeinsamkeiten gibt es einige wichtige Sprachelemente, die Sie zuerst verstehen sollten. Die folgenden Beispielpprogramme verwenden nur ein kleines Subset von C++: Objektorientierte Eigenschaften werden nur zum Teil verwendet. Hauptsächlich werden Sie "C"-ähnliche Beispiele finden. Das hat aber einen guten Grund, denn schließlich handelt es sich bei diesem Sonderheft um eine Ausgabe, die das Programmieren mit C++ vermitteln will, und dazu muss man eben am Anfang beginnen.

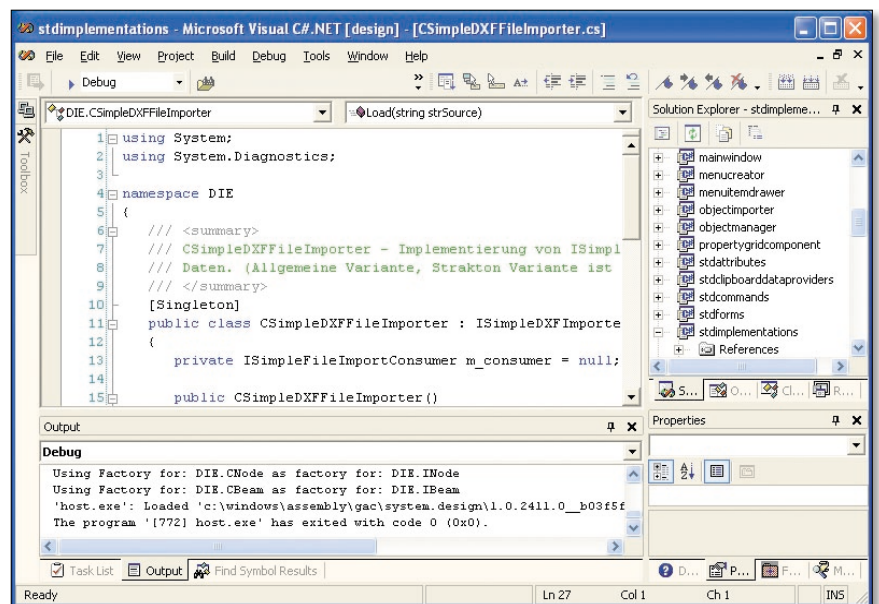
Das wichtigste Element von C und einer der wichtigsten Bausteine von C++ sind Zeiger. Verwendet man die nicht,



**DER WINDOWS SCRIPTING HOST** stellt interpretierte Sprachen zur Verfügung.

könnte man genauso gut in einer anderen Sprache programmieren. Mit Zeigern lassen sich sehr effiziente Datenstrukturen zusammenbauen, und Zeiger helfen – beim richtigen Einsatz – auch dabei, ein Programm übersichtlicher zu gestalten.

Die Verwendung von Zeigern muss man einfach beherrschen und zwar am besten im Schlaf. Aus diesem Grund gehen die folgenden Beispiele im Sonder-



**SPRACHEN WIE C, C++ ODER C#** werden kompiliert.

heft sehr stark auf Zeiger ein. Am Ende der Beispiele sind Sie in der Lage, dieses Sprachelement so zu verwenden, dass robuste Programme dabei herauskommen.

Bevor das aber alles passiert, müssen Sie eine ganze Menge an Grundwissen über das Programmieren im Allgemeinen und das Programmieren mit C/C++ im Besonderen sammeln. Dieses Grundwissen erhalten Sie im Zuge dieses Beitrages.

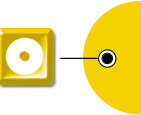
### ■ Kompilieren oder nicht

Zunächst einmal ist es wichtig, zwischen zwei unterschiedlichen Konzepten bei Programmiersprachen zu unterscheiden: Es gibt interpretierte Sprachen und kompilierte Sprachen. Bei einer interpretierten Sprache wird der Quellcode eines Programms von einem anderen Programm geladen, Schritt für Schritt interpretiert und dabei ausgeführt.

Dieses andere Programm nennt man Interpreter. Beispiele für eine solche Sprache sind das bei vielen Windows-Versionen mitgelieferte QBasic und Skripte, die für den Windows Scripting Host geschrieben werden: Sie werden interpretiert.

Interpretierte Sprachen haben Vor- aber auch Nachteile. Ein Vorteil ist zum Beispiel, dass ein interpretiertes Programm auf jedem System ausgeführt werden kann, auf dem auch ein Interpreter für die entsprechende Sprache vorliegt. Zumindest geht das in der Theorie.

Ein Beispiel wäre ein Perl-Skript. Perl-Interpreter liegen für praktisch jedes verfügbare Betriebssystem vor. Das bedeutet, Sie können ein Perl-Programm zum Beispiel unter Windows programmieren und dort mit dem Perl-Interpreter für Windows testen – dann den Quellcode auf einen Linux-Rechner transportieren und dort unter dem



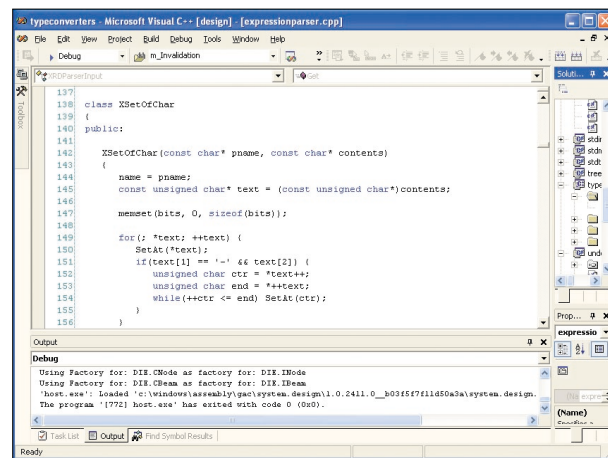
Perl-Interpreter für Linux ausführen. Wirklich funktionieren wird das natürlich nur dann, wenn Sie im Programm keine Funktionen verwenden, die plattformabhängig sind.

Das Problem dabei ist, dass die Instruktionen im Quellcode jedes Mal neu interpretiert werden müssen, wenn Sie ausgeführt werden. Das kostet natürlich Zeit und dadurch werden solche Programme tendenziell langsamer, als sie sein müssten. Ferner ist es dabei so, dass Sie den Quellcode des Programms an die Personen weitergeben müssten, die es benutzen wollen: Das ist aber nicht immer wünschenswert.

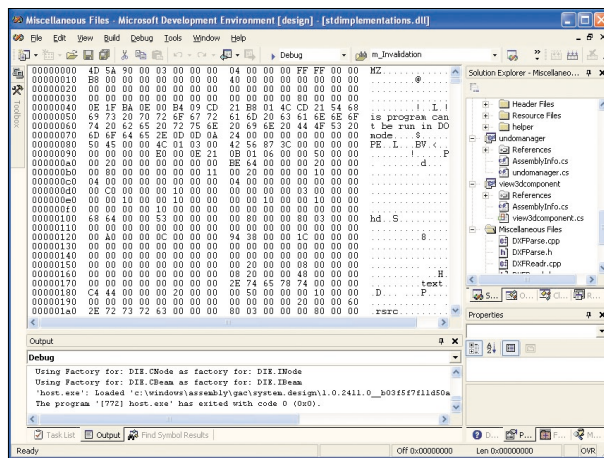
Bei kompilierten Programmen sieht die Sache anders aus. Auch hier gibt es ein Programm, das den Quellcode des eigentlichen Programms liest – das passiert aber nur einmal. Dieses Programm nennt man Compiler. Der Compiler liest den Quellcode des Programms und übersetzt diesen in Maschineninstruktionen, die der Rechner direkt ausführen kann. Diese übersetzten Anweisungen werden in einer anderen Datei abgelegt – meist als .EXE, für (EXE)cutable, also "ausführbar").

In Wirklichkeit ist dieser Prozess aber noch ein wenig komplizierter – doch dazu später mehr. Dieses kompilierte Programm kann nun direkt, also ohne die Verwendung eines Interpreters, ausgeführt werden (der Interpreter ist in diesem Fall die Hardware-Maschine selbst). Das macht die Sache schneller, und außerdem muss der Quellcode des Programms nicht an Dritte weitergegeben werden. Dafür läuft das Programm aber nur auf Plattformen, für die es übersetzt wurde: Sie können also ein Programm, das Sie für einen X86-Rechner übersetzt haben, nicht auf anderen CPUs laufen lassen – und natürlich auch nicht unter anderen Betriebssystemen.

C/C++ ist eine kompilierte Sprache, und darum werden interpretierte Sprachen im Rest des Beitrags im Großen und Ganzen ignoriert.



**DER QUELLCODE** eines C++-Programms wird zunächst vom Präprozessor und erst dann vom Compiler weiterverarbeitet.



**BEIM KOMPIlierEN** wird eine Binär-Datei angelegt, die Sie mit einem Binär-Editor ansehen können.

## ■ Programme bauen, übersetzen und linken

Wie bereits erwähnt ist der Vorgang, ein Programm in die ausführbare Form zu bringen, etwas komplizierter als bisher beschrieben. Tatsächlich kommen noch eine ganze Reihe weiterer Tools als der Compiler zum Zuge bis das fertige ausführbare Programm vorliegt – und dann auch tatsächlich ausgeführt werden kann.

Wenn Sie sich an das "Hallo Welt"-Beispiel aus dem vorhergehenden Beitrag erinnern, wissen Sie noch, dass dieses Programm einige bestimmte Eigenschaften hatte.

Zum einen wurden oben im Quellcode Zeilen mit "#include" hineingeschrieben, dann gab es eine *main*-Funktion und schließlich gab es Zeilen mit Befehlen, wie *getch* und *printf* und *return*.

Hier kommen nun verschiedene Konzepte zum Zuge. Im Wesentlichen werden beim "Bauen" – man spricht auch

vom "build"-Prozess – eines C/C++-Programms drei Werkzeuge eingesetzt. Dabei handelt es sich um den Präprozessor, den Compiler und den Linker. (Eigentlich gibt es noch mehr Werkzeuge, aber auch dazu später mehr.) Innerhalb der Entwicklungsumgebung ist das aber heute transparent – zwar geben die entsprechenden Tools Meldungen von sich, aber sie müssen nicht wie

früher einzeln angestoßen werden.

Zunächst kommt der Präprozessor zum Zuge. Dieses Werkzeug liest Ihren Quellcode und sucht dabei nach Präprozessor-Statements. Diese Statements erkennen Sie an der vorangestellten Raute (#). Diese und nur diese Statements werden vom Präprozessor beachtet. Dabei tut der Präprozessor nicht sonderlich viel: Im Wesentlichen handelt es sich dabei um ein Programm zur Textersetzung. Der Ausdruck

```
#include <stdio.h>
```

bedeutet zum Beispiel, dass die Datei mit dem Namen an "stdio.h" an der Stelle in den Quellcode eingefügt (inkludiert) werden soll, an der der Ausdruck steht. Dabei verändert der Präprozessor natürlich nicht den von Ihnen geschriebenen Code,



sondern erzeugt eine temporäre Datei, die Ihren Code und die vom Präprozessor ersetzten Textstellen enthält.

Das `#include`-Statement ist dabei zunächst einmal das wichtigste Statement und das liegt daran, dass Sie ohne dieses keine vordefinierten Funktionen verwenden können. Bevor Sie in C/C++ eine Funktion aufrufen, müssen Sie diese Funktion dem Compiler bekannt machen. Das passiert, indem der Quellcode Ihres Programms einen Prototypen der Funktion enthält. Für die Funktionen, die beim Compiler mitgeliefert werden, sind diese Prototypen in einer ganzen Reihe von Header (\*.h)-Dateien enthalten.

Eine solche Datei ist zum Beispiel die Datei `"stdio.h"`. Diese enthält alle Prototypen zum Bereich "STandard Input Output" – daher der Name. Der Präprozessor ersetzt also die Zeile mit dem `#include`-Statement durch den Inhalt der angegebenen Datei – und dadurch kennt der Compiler den Prototypen der zugehörigen Funktionen. Mit dem Präprozessor lassen sich auch noch weitere Aufgaben durchführen – Sie können damit alle Aufgaben ausführen, die ein Werkzeug zur Textersetzung erledigen kann.

Im nächsten Schritt kommt der Compiler an die Reihe. Dieser liest nun den vom Präprozessor aufbereiteten Quellcode und übersetzt diesen in ein vom Rechner verständliches Format. Dieses Format nennt man Objekt-Format, und deshalb heißen die erzeugten Dateien auch Objekt-Dateien und haben die Datei-Erweiterung `.obj`. Der Compiler untersucht Ihr Programm auch auf Syntax-Fehler und bemängelt diese. Dabei sollten Sie aber berücksichtigen, dass C bzw. C++ eine sehr "lockere" Sprache ist – man kann so ziemlich alles in ein C-Programm hinschreiben, und der Compiler wird es trotzdem fast immer als gültigen C/C++-Quellcode ansehen. So ist etwa die Zeile

```
'h';
```

durchaus ein gültiges – wenn auch kein sinnvolles – C-Statement.

Diese beiden Schritte – also die Aufbereitung durch den Präprozessor und danach das Übersetzen durch den Compiler geschieht nun für alle am Projekt beteiligten Quellcode-Dateien. Trat nirgendwo ein Fehler auf, liegt am Ende des Arbeitsschrittes für jede beteiligte C/C++-Datei eine `.OBJ`-Datei vor.

Diese Dateien müssen nun noch zu einem ausführbaren Programm zusammengebaut werden, und diese Aufgabe übernimmt der Linker. Beim zuvor geführten Beispiel haben Sie einige Funktionen der C-Laufzeitbibliothek aufgerufen, so zum Beispiel die Funktion `printf()` und die Funktion `getch()`. Beide Funktionen wurden aber nicht von Ihnen programmiert, sondern wurden stattdessen vom Lieferanten des Compilers zur Verfügung gestellt. Beide Funktionen stammen aus der normierten "Standard C"-Bibliothek, die bei jedem ANSI-kompatiblen C-Compiler beige packt ist. Dabei ist es üblich solche Funktionssammlungen nicht in Form

Aufruf der Funktion `printf`. Im Gegensatz zu anderen Instruktionen im Programm, die in tatsächlich ausführbaren Code übersetzt werden, bettet der Compiler bei diesem Funktionsaufruf einfach nur eine Marke in den generierten Code ein.

Diese Marke enthält im Wesentlichen zwei Informationen: Die eine Information besagt, dass an einer bestimmten Stelle eine Funktion aufgerufen werden soll – es wird also eine Position im Programm festgelegt. Die zweite Information legt fest, welche Funktion aufgerufen werden soll: Das geschieht einfach über den Namen der Funktion.

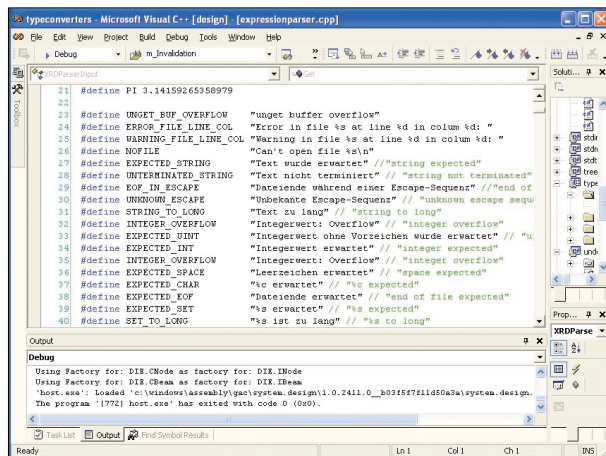
Der Linker untersucht nun die `.OBJ`-Datei und findet darin zunächst die Information "Hier Funktionsaufruf". Dann ermittelt das Programm den Namen der aufzurufenden Funktion und merkt sich sowohl die Position als auch den Namen. Sind nun alle diesbezüglichen Informationen eingesammelt, beginnt der Linker damit, die entsprechenden Funktionen zu suchen.

Dabei untersucht er zunächst einmal, ob – um beim Beispiel zu bleiben – die Funktion `getch()` vielleicht in einer der am Projekt beteiligten `.OBJ`-Dateien enthalten ist. Das wäre dann der Fall, wenn diese Funktion von Ihnen an anderer Stelle programmiert worden wäre. Natürlich findet das Programm die Funktion aber nicht. Im zweiten Schritt durchsucht der Linker die zum Compiler gehörenden Libraries und dort wird er schließlich fündig.

Alle gefundenen Funktionen werden zusammen mit dem Object-Code, der aus Ihrem Quellcode resultiert, in einer `.EXE`-Datei zusammengebunden: Dazu kopiert der Linker den Code der Library-Funktionen einfach in diese Datei. Schließlich ersetzt der Linker alle Marken der Sorte "Hier Funktionsaufruf" durch passenden Code, der zur Laufzeit des Programms dafür sorgt, dass die Kontrolle an die passende Stelle im `.EXE` File übertragen wird.

Der Linker löst also Referenzen auf Funktionen (also Funktionsaufrufe) durch eine Sprunganweisung auf die entsprechende Funktion im fertigen Programm auf und kümmert sich dabei darum, dass Funktionen, die in Libraries vorliegen, ebenfalls ins Programm kopiert werden.

An dieser Stelle stellt sich die Frage, was passiert, wenn eine Funktion nicht



**PRÄPROZESSOR-SYMBOLS HABEN** vielfältige Einsatzgebiete: Sie können sowohl Ausdrücke als auch, wie abgebildet, einfach nur Konstanten enthalten.

von vielen `.OBJ`-Dateien mitzuliefern. Stattdessen werden die Funktionen in Form einer "Library" (Bibliothek) mitgeliefert. Diese Library enthält im Wesentlichen eine Sammlung aus `.OBJ`-Dateien.

Nun ist es zwar ganz nett, dass Sie solche Funktionen mitgeliefert bekommen – damit Sie die aber auch benutzen können und damit das Programm funktioniert, müssen die Funktionen am Ende in irgendeiner Form in Ihrer `.EXE`-Datei enthalten sein. Auch darum kümmert sich der Linker. Das passiert wie folgt: Beim Übersetzen des Quellcodes "sieht" der Compiler beispielsweise den







arbeitet werden, wird dabei durch die C/C++-Schlüsselworte festgelegt. Die wichtigsten davon sind folgende:

## for

Mit dem "for"-Statement programmieren Sie eine Schleife. Das brauchen Sie immer dann, wenn Sie eine bestimmte oder mehrere Befehle mehrfach wiederholt ausführen möchten. Angenommen Sie möchten die bereits bekannte Funktion `printf()` zehnmal hintereinander aufrufen, und dabei jedes Mal die Nummer des aktuellen Durchlaufs anzeigen. In diesem Fall könnten Sie zum Beispiel folgenden Quellcode verwenden:

```
for( int i=0; i<10; i++)
{
    printf("%d", i);
}
```

Der "for"-Ausdruck hat also drei steuernde Elemente: Einen Ausdruck, der einen Startwert bestimmt, eine Abbruchbedingung und einen Schleifenaustrittsdruck. Der Ausdruck zur Festlegung des Startwerts wird dabei immer nur einmal verwendet. Das Beispiel ist also in etwa wie folgt zu lesen:

1. Initialisiere die Integer-Variable "i" mit "1".
2. Überprüfe ob "i" kleiner ist als 10. Ist das nicht der Fall, breche ab.
3. Führe den Quellcode innerhalb der geschweiften Klammern aus.
4. Erhöhe den Wert von i um 1. (i++ ist die vereinfachte Schreibweise von i=i+1)
5. Springe wieder zu 2.

Mit anderen Worten: Der Ausdruck "printf(...)" wird zehnmal ausgeführt, und dann wird die Schleife abgebrochen. Die Variable "i" nimmt dabei die Werte zwischen 0 und 9 an.

## while

Bei "while" handelt es sich um eine alternative Methode der Programmierung einer Schleife, bei der nur eine Abbruchbedingung angegeben wird. Die gleiche Schleife wie beim "for"-Beispiel würde mit "while" folgendes Aussehen haben:

```
int i=0;
while ( i<10)
{
    printf("%d", i);
    i++;
}
```

## if / else

"if" dient der Überprüfung von Ausdrücken. Der überprüfte Ausdruck

nimmt dabei immer den Wert "wahr" (true) oder "falsch" (false) an. Dabei ist es in C/C++ so, dass jeder Ausdruck, der Null wird, "false" ist, während alles andere "true" ist. Hier ein Beispiel für die Verwendung von if:

```
int a=5;
int b=6
if(b>a)
{
    printf("b ist grösser als a")
}
else
{
    printf("a ist grösser/gleich b");
}
```

Ist die Bedingung wahr ( b ist größer als a), wird der Quellcode im ersten Block ausgeführt. Trifft die Bedingung nicht zu, so kommt der "else"-Block zum Zuge.

## return

Mit dem "return"-Statement verlassen Sie eine Funktion und kehren zur aufrufenden Funktion zurück. Bei einer Funktion, deren Prototyp festlegt, dass die Funktionen einen Rückgabewert liefert, hat "return" einen Parameter der dann als Rückgabewert verwendet wird. Bei Funktionen ohne Rückgabewert hat "return" auch keinen Parameter. Ferner kann bei einer Funktion ohne Rückgabewert auf "return" ganz verzichtet werden. In diesem Fall wird dann zur aufrufenden Funktion zurückgekehrt, wenn das Ende der Funktion erreicht ist. Beispiel für den Aufruf einer Funktion mit Rückgabewert:

```
int bar();

void foo()
{
    int i = bar();
}

int bar()
{
    return 42;
}
```

Dieses Beispiel setzt sich aus drei Teilen zusammen: Zunächst sehen Sie den Prototypen für die Funktion `bar()`. Dieser Prototyp wird benötigt, damit die Funktion in der Funktion `foo()` aufgerufen werden kann. Das passiert auch, und dabei wird der Returnwert von `bar()` in der Variable i gespeichert. Die Funktion `bar()` selbst tut nicht viel, sondern liefert einfach nur einen Integer-Wert zurück.

## switch/case

Mit den Schlüsselworten "switch" und "case" ist es deutlich einfacher, mehrere Be-

dingungen abzufragen. Angenommen Sie haben eine Variable, die drei Werte annehmen kann. Wenn Sie den Zustand dieser Variable mit "if" testen, käme in etwa der folgende Code dabei heraus:

```
if( i == 1)
{
    //hier passiert etwas
}
else if( i == 2)
{
    //hier passiert etwas anderes
}
else if( i == 3)
{
    //und noch etwas anderes
}
```

Man kann sich nun leicht vorstellen, dass so ein Quellcode schnell unübersichtlich wird – besonders dann, wenn mehr als nur drei Fallunterscheidungen benötigt werden. Genau für diesen Fall ist switch/case gedacht. Mit "switch" wird dabei geklärt, welche Variable zu untersuchen ist, während die einzelnen Fälle jeweils von einem "case"-Statement behandelt werden:

```
switch( i)
{
    case 1:
        // hier passiert was
        break;

    case 2:
        // hier passiert was
        break;

    case 3:
        // hier passiert was
        break;
}
```

In diesem Zusammenhang kommt außerdem der Präprozessor noch oft ins Spiel: Ein Vergleich mit Werten wie "1", "2" oder "3" ist nicht sonderlich aussagekräftig. Daher definiert man in der Praxis zunächst einen symbolischen Namen und arbeitet dann mit diesem weiter. Die Definition dieses Symbols erledigt man mit dem Präprozessor:

```
#define ZUSTAND_1 1
#define ZUSTAND_2 2
#define ZUSTAND_3 3

switch( i)
{
    case ZUSTAND_1:
        break;

    // und so weiter...
}
```

Soviel zum ersten Hintergrundwissen. Im Laufe der folgenden Beiträge wird sich das noch mehr vertiefen, doch für den Anfang sollte das jetzige Wissen reichen. Im folgenden Beitrag geht es dann in die Praxis. UR