Die Aztec C-Compiler-Installation

Erik Bartmanns

Amiga 1200 Buch

Maschinensprache, C-Programmierung und Shell

bombini verlag

Aztec C-Compiler

Folgende Themen werden besprochen.

- Der Aztec C-Compiler
- Die Installation des Aztec C-Compilers
- Ein erster Test

Warum ein alter Compiler- Der Aztec C-Compiler

Während moderne Cross-Compiler wie *vbcc* oder *gcc* zweifellos mächtig sind, gibt es triftige Gründe – vor allem nostalgische und systemnahe – die für den Einsatz des *Aztec C Compilers* sprechen. Das mag mir - und das ist mir vollkommen egal - herbe Kritik entgegenbringen. Hier ist ein Entwurf, der diese Brücke schlägt:

Warum Aztec C - Ein Plädoyer für Authentizität und Systemnähe

Wenn wir heute über die Programmierung des Amigas sprechen, stellt sich unweigerlich die Frage nach dem Werkzeug. Warum sollten wir uns in der heutigen Zeit mit dem *Aztec C Compiler* (von Manx Software Systems) auseinandersetzen, wo es doch moderne, hochoptimierende Cross-Compiler wie vbcc gibt, die bequem auf Windows, Mac oder Linux laufen? Die Antwort liegt in der Philosophie dieses Buches: Wir wollen den Amiga nicht nur als Zielplattform benutzen, sondern ihn in seiner Gesamtheit verstehen und erleben.

1. Das "Native" Erlebnis: Programmieren auf der Zielmaschine

Aztec C war in der Blütezeit des Amigas einer der Industriestandards. Wenn du Aztec C nutzt, arbeitest du in derselben Umgebung wie die Pioniere der 80er und 90er Jahre. Du nutzt die Shell, die Editoren und die Linker direkt auf dem Amiga (oder in einer originalgetreuen Emulation). Dieses "native" Gefühl vermittelt ein tieferes Verständnis dafür, wie Software damals unter den Ressourcenbeschränkungen von 7 MHz und wenigen Megabyte RAM entstand.

2. Die Nähe zur Dokumentation

Die klassische Amiga-Literatur – allen voran die berühmten "Rom-Kernal Manuals" von Commodore – und unzählige Listings in Zeitschriften wie das *Amiga Magazin* oder *68000er* beziehen sich oft direkt auf die Syntax und die Bibliotheken von Aztec C. Wer diese historischen Quellen studiert, wird feststellen, dass der Code "out of the box" mit Aztec C funktioniert, während moderne Compiler oft Anpassungen an Headern oder Linker-Optionen erfordern.

3. Kompaktheit und Geschwindigkeit des Workflows

Aztec C wurde für Maschinen mit wenig Speicher optimiert. Auf einem echten Amiga 500 oder 1200 ist der Workflow mit Aztec oft flüssiger als der Versuch, ein modernes, speicherhungriges *Monster-Toolchain* zu bändigen. Aztec C ist flink, die Fehlermeldungen sind prägnant, und man lernt zwangsläufig, wie das Zusammenspiel zwischen Compiler, Assembler und Linker im Detail funktioniert.

4. Einzigartige Features und der 68000-Fokus

Während moderne Compiler versuchen, den Code für alle möglichen 68k-Varianten (020, 040, 060) zu generalisieren, ist Aztec C ein Kind der 68000-Ära. Er erzeugt Code, der sich "ehrlich" anfühlt. Zudem bietet er exzellente Möglichkeiten, Inline-Assembler zu nutzen – ein unschätzbarer Vorteil für dich, da du bereits über grundlegendes Wissen in Maschinensprache verfügst.

Fazit: Das Beste aus zwei Welten

Den Aztec C Compiler zu nutzen bedeutet, das Handwerk von der Pike auf zu lernen. Es ist der Unterschied zwischen dem Fahren eines modernen Autos mit Automatikgetriebe und einem Klassiker mit Handschaltung: Man spürt die Maschine einfach besser. In diesem Buch verwende ich also Aztec C als ein weiteres Werkzeug, um die Seele des Amiga-Betriebssystems einzufangen.

Die Installation des Aztec C-Compilers

Die Installation von Aztec C soll nun besprochen werden. Die näheren Hinweise für die Nutzung oder den Download sind natürlich im Internet zu finden. Ein geeigneter Einstiegspunkt könnte die folgende Adresse sein.

https://www.aztecmuseum.ca/

Nach dem Herunterladen und Entpacken sind bei mir die folgenden vier Disketten im adf-Format vorhanden.

Name	Änderungsdatum	Typ	Größe
AztecC52a_D1.adf	08.12.2025 19:48	ADF-Datei	880 KB
AztecC52a_D2.adf	08.12.2025 19:48	ADF-Datei	880 KB
AztecC52a_D3.adf	08.12.2025 19:48	ADF-Datei	880 KB
AztecC52a_D4.adf	08.12.2025 19:48	ADF-Datei	880 KB

Abbildung 1 - Der Aztec-C-Compiler mit vier adf-Dateien

Ich lege diese allesamt in die Diskettenlaufwerke ein, was dann im Endeffekt in der Workbench so ausschaut.

Abbildung 2 - Die vier Disketten des Aztec-C-Compilers

Öffne die erste Diskette mit der Bezeichnung Aztec1 und gehe wieder über das Kontextmenü *Fenster > Inhalt anzeigen > alle Dateien*, damit du auch wirklich alle Icons zu sehen bekommst. Im nächsten Schritt musst du die markierte Datei *AztecInstall* ausführen.

Abbildung 3 - Die Aztec Installationsdatei

Nach dem ausgeführten Doppelklick erscheint zuerst der einleitende Dialog, der dann den eigentlichen Installationsprozess initiiert.

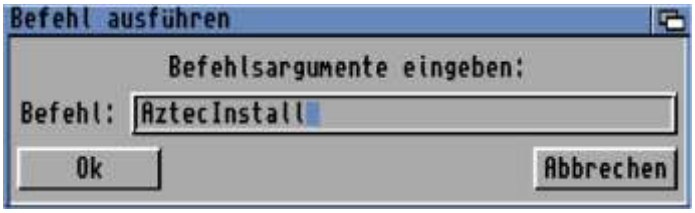

Abbildung 4 - Der Installationsprozess wird hierüber gestartet

Nach dem Klicken auf die *Ok*-Schaltfläche beginnt die eigentliche Installation des Aztec-C-Compilers. Es blitzt kürz der Splash-Screen auf.

Abbildung 5 - Der Splash-Screen des Aztec-C-Compilers

Nach einer Sekunde ist dann das Menü zu sehen, in dem du auswählen kannst wo und was installiert werden soll.

Abbildung 6 - Der Installationsdialog mit den Detailangaben für die Installation

Ich habe auf der linken Seite alles so belassen, wie es der Dialog vorgesehen hat. Also die Installation auf die Systemplatte *DH0:* und lediglich auf der rechten Seite habe ich alle Häkchen unterhalb des Schriftzuges Copy gesetzt, damit ein paar zusätzliche Quellen installiert werden. Das kannst du natürlich alles selbst entscheiden. nach einem Mausklick auf die breite Schaltfläche mit der Bezeichnung BEGIN INSTALLTION erfolgt nochmals eine kleine Abfrage, ob die gesetzten Parameter für die anstehende Installation in Ordnung sind.

Abbildung 7 - Abfrage zur Einleitung der Installation

Wenn du denkst, dass alles jetzt ok ist und es losgehen kann, dann hast du dich getäuscht. Eine weitere Abfrage wartet auf dich.

Abbildung 8 - Wo ist die Diskette 2 denn hin

Klicke auf die *OK*-Schaltfläche, denn diese Diskette befindet sich eigentlich schon im Laufwerk. Es erfolgen weitere Abfragen für die restlichen Disketten, die du alle mit *OK* bestätigen musst. Nach dem Abschluss kannst du das Fenster schließen und im Hintergrund siehst du dann das Ausgabefenster der Installation, das mit der letzten Meldung bestätigt, dass die Operation erfolgreich durchgeführt wurde.

Abbildung 9 - Die Installation wurde erfolgreich durchgeführt

Schließe nun auch dieses Fenster. Im nächsten Schritt gehe über die Funktionstaste

und entferne alle vier Disketten über das Anklicken der *Eject*-Schaltflächen bei DF0 bis DF3. Wenn ich mir jetzt eine Shell aufmache und nachsehe, dann erhalte ich folgende Anzeige.

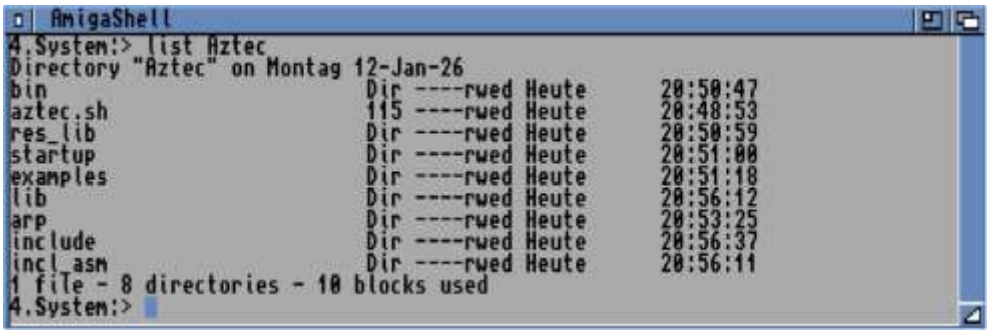

Abbildung 10 - Die Unterverzeichnisse der Aztec-Installation

Es sind also die wichtigsten Unterordner

- ***bin*** - Hier liegen die ausführbaren Programme wie cc (Compiler), as (Assembler) und ln (Linker).
- ***include*** - Die "Baupläne" für C – hier liegen die .h-Dateien, die dem Compiler sagen, wie die Amiga-Systemfunktionen aussehen.
- ***lib*** - Die fertigen Bausteine (Bibliotheken), die der Linker zu Ihrem Programm hinzufügt.

enthalten. Werfen wir noch einen Blick in die Datei *aztec.sh*, die wichtige Informationen für die anstehende Nutzung bereithält. Wechsle dazu in das Aztec-Verzeichnis mit

```
cd Aztec
```

und gib

```
ed aztec.sh
```

ein. Die Anzeige gestaltet sich dann.

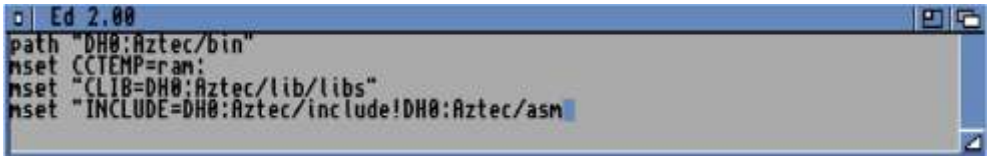

Abbildung 11 - Der Inhalt der aztec.sh-Datei

Wenn du lediglich den Inhalt einer Text basierten Datei anzeigen willst, reicht auch der Befehl

```
type aztec.sh
```

Was ist der Sinn dieser Zeilen? Die Zeile

```
path "DH0:Aztec/bin"
```

fügt *DH0:Aztec/bin* zum Befehlssuchpfad hinzu, so dass Programme ohne eine absolute Pfadangabe in diesem Verzeichnis danach direkt aufgerufen werden können. Also zum Beispiel:

- cc
- as
- ln
- ld

Der Befehl

```
mset CCTEMP=ram:
```

setzt die globale Variable CCTEMP und wird vom Aztec-C-Compiler verwendet. Sie gibt an, wo temporäre Dateien abgelegt werden.

- ram: = RAM-Disk
- Ist schneller als eine Festplatte
- Der Inhalt verschwindet beim Neustart

Die Zeile

```
mset "CLIB=DH0:Aztec/lib/libs"
```

setzt die Variable *CLIB* und teilt Compiler/Linker mit, wo die C-Libraries (.lib) liegen. Und letztendlich die Zeile

```
mset "INCLUDE=DH0:Aztec/include!DH0_Aztec/asm"
```

setzt die Variable INCLUDE und bestimmt darüber, wo der Compiler nach Header-Dateien (.h) suchen muss. Die Suchreihenfolge lautet:

- DH0:Aztec/include
- DH0_Aztec/asm

Das wird zum Beispiel bei *include*-Aufrufen wie

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

genutzt. Kurz zusammengefasst bedeutet diese vier Zeilen:

- Einrichten einer Aztec-C-Entwicklungsumgebung
  - Programme (path)
  - Temp-Dateien (CCTEMP)
  - Libraries (CLIB)
  - Header-Suchpfade (INCLUDE)

Also alles völlig typisch für Amiga-C-Entwicklung der 80er/90er Jahre. Mache doch einmal den folgenden Versuch und gib das Kommando

```
cc
```

in der Shell ein. Dieses Kommando übersetzt den C-Quellcode in ein ausführbares Amiga-Programm. Du bekommst eine Fehlermeldung, dass das Programm bzw. der Befehl nicht bekannt ist. Danach führst du das gezeigte Shell-Skript *Aztec.sh* aus und erneut den *cc*-Befehl. Was stellst du fest?

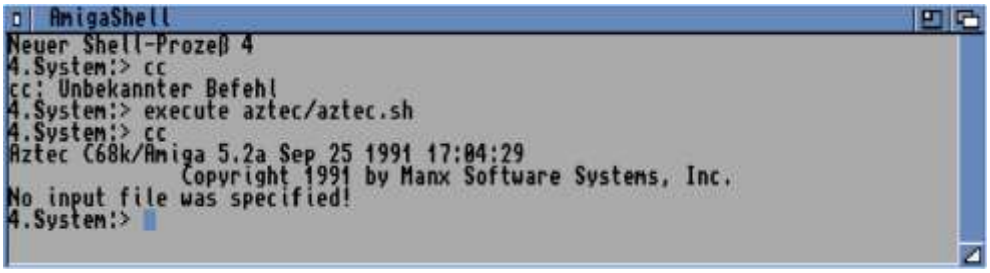

Abbildung 12 - Der cc-Befehl wird ausgeführt

Nun ist der *cc*-Befehl in den Suchpfad aufgenommen und wurde erkannt. Es fehlt zwar noch ein Input-File in Form einer Quelldatei, was jedoch erst einmal zweitrangig ist. Was ist aber, wenn der Amiga gebootet wird? Sind dann die durchgeführten Setzungen immer noch präsent? Das ist nicht der Fall. Nun kannst du immer wieder das Shell-Skript ausführen, was auf die Dauer jedoch sehr lästig wird. Es gibt dazu einen Mechanismus, der es einem Nutzer ermöglicht, dass etwas beim Hochfahren des Systems automatisiert ausgeführt wird. Natürlich existieren diesbezüglich - wie sollte es anders sein - mehrere Ansätze. Du kennst bestimmt noch das *S:*-Verzeichnis. Es ist beim Amiga eines der zentralsten Systemverzeichnisse und enthält Skripte, die den Systemstart und die Systemkonfiguration steuern. *S:* ist ein logisches Device (Assign) und kein echtes Laufwerk. Beim Boot-Prozess, also beim Systemstart ist die wichtigste Datei die *s:startup-sequence*. Diese Datei:

- wird automatisch beim Booten ausgeführt
- startet das gesamte AmigaOS
- lädt Treiber, Patches, Workbench usw.

Ohne sie bootet der Amiga praktisch nicht. Dann gibt es noch die Datei mit dem Namen *S:user-startup*, welche ebenfalls ein AmigaDOS-Skript ist und automatisch beim booten ausgeführt wird. Sie enthält - falls vorhanden - nur Benutzersachen und nicht die Einträge, die für das Betriebssystem selbst zuständig sind. Wenn sie nach einer frischen Installation nicht vorhanden ist, lege sie einfach über den Befehl

```
ed user-startup
```

an, wobei du dich aber in *S* befinden musst. Dort trägst du jetzt den folgenden Pfad ein, der dafür sorgt, dass das gezeigte Skript *aztec.sh* beim Booten ausgeführt wird.

Abbildung 13 - Der Inhalt der user-startup-Datei

Nach der Eingabe kannst du die Taste

drücken. Im Anschluss gibst du hinter dem Stern

ein, was dafür sorgt, dass der Editor die Datei speichert und geschlossen wird. Natürlich besitzt der Editor auch ein Kontextmenü, das du nutzen kannst. Natürlich werde ich die Funktionsweise des Compilers und Linkers noch in einem separaten Kapitel meines Buches weiter unten erläutern. Falls du die Skript-Datei *aztec.sh* nicht mit in das Startup einbindest, musst du für das Kompilieren die folgende Befehlszeile nutzen, wobei ich jetzt die Quelldatei *hello.c* angebe, in dem eine *include*-Direktive vorhanden ist.

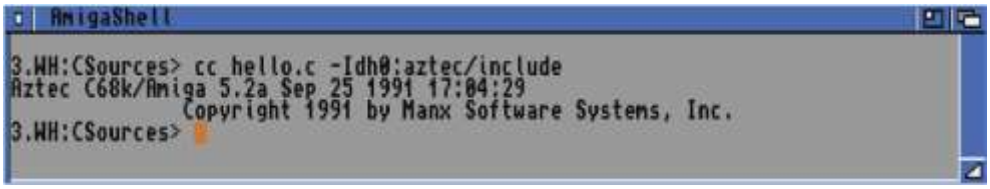

Abbildung 14 - Die Angabe des Include-Pfades beim Kompilieren

Über den Zusatz

```
-Idh0:aztec/include
```

wird über den Schalter ***-I*** der Include-Pfad der Aztec-Installation dem Compiler mitgeteilt. Wird dieser Zusatz weggelassen, kommt es zu einer Fehlermeldung.

Abbildung 15 - Die Fehlermeldung für den fehlenden Include-Pfad

Nicht anders schaut es für den Linker aus. Du musst dazu die folgende Befehlszeile mit der Angabe der Objekt-Datei *hello.o* eingeben.

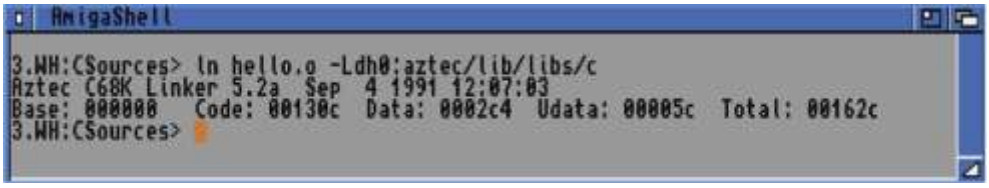

Abbildung 16 - Die Angabe des Library-Pfades beim Linken

Über den Zusatz

```
-Idh0:aztec/lib/libs/c
```

wird über den Schalter ***-I*** der Library-Pfad der Aztec-Installation und der *c.lib*-Datei dem Linker mitgeteilt. Wird dieser Zusatz weggelassen, kommt es ebenfalls zu einer Fehlermeldung.

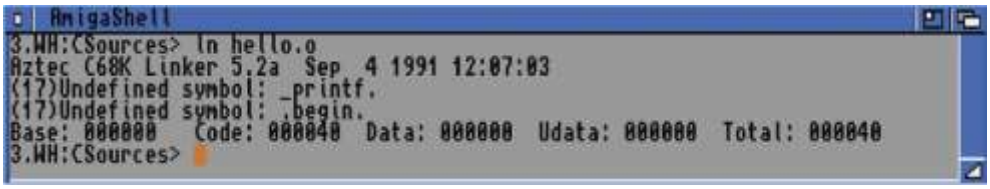

Abbildung 17 - Die Fehlermeldung für den fehlenden Library-Pfad

Ein erster Test des Aztec C-Compilers

Nun sollte alles so weit vorbereitet sein, dass du einen Quellcode in C eintippen kannst. Falls du noch nie mit C gearbeitet hast, macht das nichts, denn ich werde gleich die grundlegenden Dinge mit dir besprechen. Gib in einem Verzeichnis deiner Wahl folgendes Kommando ein.

```
ed hello.c
```

was den Texteditor unter dem Namen *hello.c* öffnet. Natürlich besitzt diese Datei noch einen Inhalt, was wir aber jetzt ändern wollen. Gib dann den folgenden Inhalt ein und kümmere dich erst einmal nicht um die Details. Der Text, der nach rechts eingerückt ist, wurde über die *Tab*-Taste

positioniert. Du wirst sicherlich auf ein Problem stoßen, wenn es um die Eingabe bestimmter Zeichen geht, wie zum Beispiel das geschweifte Klammerpaar { und } oder den sogenannten Backslash \. Das hat mich wirklich in den Wahnsinn getrieben und es erforderte viel Geduld und Ausprobieren. Das wird bei dir möglicherweise auch der Fall sein. Es steht und fällt mit den getroffenen Einstellungen für die Zeichensätze. Bei mir ist das im Moment

- Prefs > *Input: American*
- Prefs > *Locale: Deutschland*

Dann kann es vielleicht noch wichtig sein, das folgende Häkchen zu setzen, um über die Funktionstaste

den Backslash einzufügen.

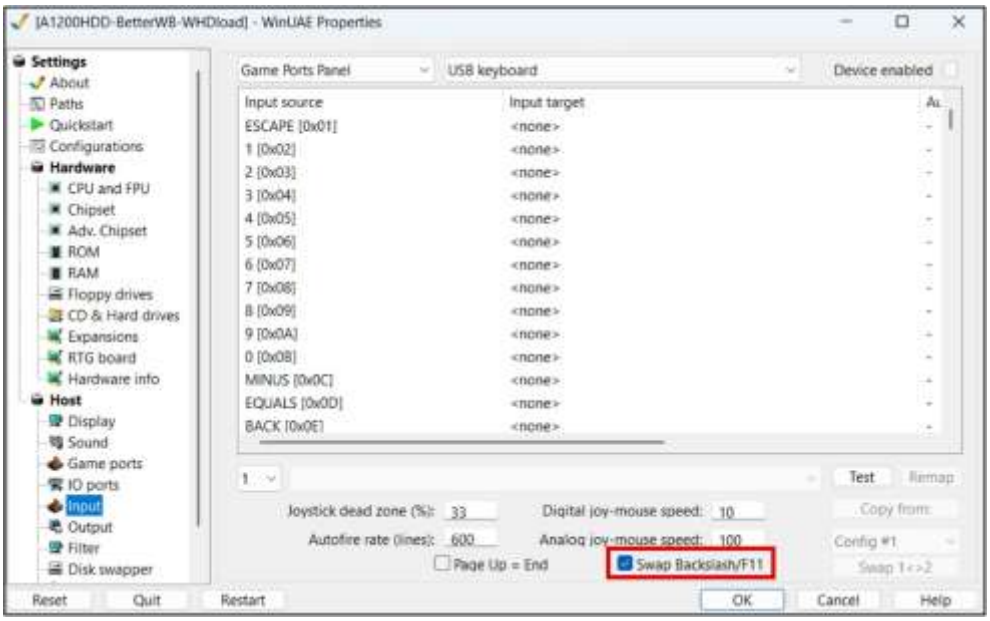

Abbildung 18 - Swap Backslash/F11

Die geschweiften Klammern sind wie folgt einzufügen.

und

Der Lattenzaun # wird über

und die doppelten Anführungszeichen über

erreicht. Hier nun der Quellcode.

Abbildung 19 - Der C-Quellcode für hello.c

Natürlich stehen in einem so einfachen Editor wie dem *ed* keine Zeilennummern oder sogar Syntax-Highlighting zur Verfügung, doch das ist eigentlich egal. Wir konzentrieren uns auf das Essentielle. Klingt wie ein Schönreden, doch ich meine es so, wie ich es sage.

Die Include-Anweisung

Die Befehlszeile

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

bindet Standard-Ein-/Ausgabe ein und bereitet die Nutzung des späteren Befehls *printf()* ein paar Zeilen weiter unten vor. Sie wird auch für viele weitere Befehle der C-Standardbibliothek vorausgesetzt. Die #include-Anweisung ist eine der wichtigsten Grundlagen in C – auch (und gerade) bei Aztec C auf dem Amiga. Sie fügt den Inhalt einer anderen Datei in den Quelltext ein – so, als hätte man ihn dort hineinkopiert.

Die Hauptfunktion

Die Zeile

```
int main()
```

stellt den Startpunkt jedes C-Programms dar. Sie ist quasi der Dreh- und Angelpunkt deines Programms. Man kann sie sich wie den Haupteingang eines Gebäudes vorstellen: Egal wie groß oder komplex das Gebäude ist, hier fängt alles an. Wenn du ein C-Programm startest, sucht das Betriebssystem automatisch nach der Funktion mit dem Namen *main*. Alles, was innerhalb der geschweiften Klammern { ... } dieser Funktion steht, wird nacheinander ausgeführt. Ohne *main* kann der Compiler kein ausführbares Programm erstellen. Das Wörtchen *int* ist der Datentyp des Rückgabewerts (Integer = Ganzzahl). Er sagt dem Betriebssystem, ob das Programm erfolgreich beendet wurde. Die runden Klammern hinter main stehen für die Aufnahme möglicher Parameter, wenn das Programm Argumente von der Kommandozeile entgegennehmen soll. Kommen wir zu den geschweiften Klammern. In C (und vielen anderen Programmiersprachen) definieren die geschweiften Klammern { } einen sogenannten Block oder Gültigkeitsbereich (Scope). Die *main*-Funktion (oder jede andere Funktion) braucht die Klammern, um zu wissen, wo sie anfängt und wo sie aufhört. Alles zwischen { und } gehört direkt zu dieser Funktion.

Die Ausgabefunktion

Kommen wir zur Zeile mit dem folgenden Befehl.

```
printf("Hallo, hier spricht dein Amiga!\n");
```

Der Name *printf* (print formatted) steht für eine C-Funktion. Sie gibt Text (und später auch Zahlen, Variablen usw.) aus. Die Ausgabe geht in die aktuelle Shell (CLI), was technisch über das AmigaDOS (stdout=Standard-Out) läuft. In C werden Funktionsaufrufe immer mit Klammern geschrieben und alles darin sind die Argumente für *printf*. Die Anführungszeichen kennzeichnen ein String-Literal, also einen festen Text, der im Programm gespeichert ist. Der Compiler legt ihn im Datensegment ab. Wenn du es kurz und knapp halten möchtest, dann nutze die *PutStr*-Funktion

```
PutStr("Das ist ein einfacher Text!\n");
```

was eine leichtgewichtige Funktion im Gegensatz zu *printf()* darstellt und keine Formatierungsmöglichkeiten bietet und die *dos.library* nutzt. Es erfolgt nach der Ausgabe des Textes kein impliziter Zeilenumbruch und die Ausgabe erfolgt im CLI. Der Zweck dieser

Funktion ist vielfältig für das Schreiben des Textes auf den aktuellen Output-Stream und deckt die folgenden Ziele ab:

- Shell
- umgeleitete Datei
- Pipe
- serielle Ausgabe

Auch die *write*-Funktion

```
Write(Output(), "Test\n", 4);
```

liefert eine mögliche Alternative. Wenn du sie benutzt, begibst du dich auf die Low-Level-Ebene der dos.library. Es ist im Grunde das Amiga-Pendant zum Unix-Systemaufruf *write()*.

1. Der erste Parameter: Output()

- *Output()* ist eine Funktion der dos.library, die dir das Handle über einen BPTR (BCPL-Pointer) für den aktuellen Standard-Ausgabekanal zurückgibt.
- In einer CLI-Umgebung ist das in der Regel das Konsolenfenster.
- *Wichtig*: Ein BPTR ist ein Amiga-spezifischer Zeigertyp und unterscheidet sich von normalen C-Pointern dadurch, dass er auf Langwörter mit 4 Bytes zeigt. Die dos.library nutzt diese fast ausschließlich für Datei-Handles.

2. Der zweite Parameter: "Test" (Der Buffer)

- Hier übergibst du die Speicheradresse der Daten, die geschrieben werden sollen.
- Anders als bei *printf* oder *PutStr* spielt es mit *Write()* keine Rolle, ob der String Null terminiert ist ('\0') oder nicht. Die Funktion liest lediglich so viele Bytes ein, wie du im dritten Parameter angibst.

3. Der dritte Parameter: 4 (Die Länge)

- Du musst über die Länge die exakte Anzahl der zu verarbeitenden Bytes angeben.
- Das macht *Write()* extrem schnell, da die Library nicht erst den String nach dem Terminator ('\0') durchsuchen muss, um die auszugebende Länge zu berechnen.

Das nachfolgende Programm zeigt die Nutzung der *Write*-Funktion. Die Nutzung von Arrays und die Funktionsweise der *sizeof*-Funktion (die Berechnung der Anzahl der Zeichen einer Zeichenkette) werden noch erläutert. Die Ausgabe erfolgt im CLI.

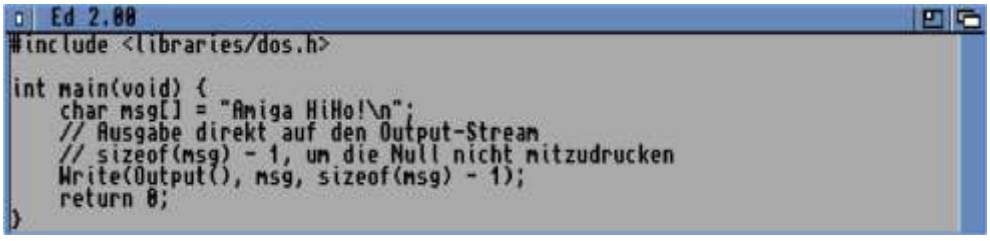

Abbildung 20 - Die Quelldatei von ausgabe1.c

Und sollte es noch nicht genug sein, möchte ich dich mit einer weiteren Ausgabefunktion behelligen. Sie nennt sich *puts* und ist Teil der ANSI-C Standardbibliothek. Die Zeile

```
puts("Hallo Amiga");
```

gibt den gezeigten Text inklusive des Zeilenumbruchs über '\n' im CLI aus. Bedenke, dass alle genannten Ausgabefunktionen

- printf()
- PutStr()
- Write()
- puts()

die Texte im CLI anzeigen und nicht in einem eigenen Fenster. Wenn du das möchtest, musst du dich gedulden, bis ich zum Thema Intuition komme, wo es um das grafische Betriebssystem-Framework des Amigas geht. Es wird sicherlich spannend!

Eine Escape-Sequenz

Du fragst dich möglicherweise, was das *\n* unmittelbar hinter dem Text zu bedeuten hat. Erinnerst du dich vielleicht noch daran? Es handelt sich um ein sogenanntes *Escape*-Zeichen, was durch den Backslash gekennzeichnet und eingeleitet wird. Es bedeutet *Newline* (neue Zeile) und bewirkt nach dem Anzeigen des Textes einen Zeilenvorschub. Der Effekt besteht also darin, dass der Cursor in die nächste Zeile springt. Ohne *\n* würde der Prompt direkt hinter dem ausgegebenen Text erscheinen, was unschön ausschaut. Hier eine kleine Auswahl nützlicher Escape-Sequenzen.

Sequenz	Bedeutung	Beschreibung
\n	Newline	Erzeugt einen Zeilenumbruch (geht in die nächste Zeile).
\t	Horizontal Tab	Fügt einen Tabulator-Abstand ein (rückt den Text ein).
\\	Backslash	Gibt einen echten Backslash aus (da der Backslash allein ja das sogenannte Fluchtsymbol ist).
\'	Einfaches Anführungszeichen	Erlaubt die Nutzung von ' innerhalb von Strings, die mit ' begrenzt sind.
\"	Doppeltes Anführungszeichen	Erlaubt die Nutzung von " innerhalb von Strings, die mit " begrenzt sind.
\r	Carriage Return	Setzt den Cursor an den Anfang der aktuellen Zeile zurück. In modernen Systemen oft zusammen mit \n gesehen (\r\n), besonders unter Windows.
\b	Backspace	Löscht das Zeichen links vom Cursor (nützlich für einfache Ladebalken in der Konsole).

Tabelle 1 - Nützliche Escape-Sequenzen

Kommen wir zu einem weiteren und sehr wichtigen Aspekt in der C-Programmierung.

Der Anweisungsabschluss

In C können die einzelnen Befehle theoretisch über mehrere Zeilen geschrieben werden oder viele Befehle in einer einzigen Zeile stehen. Der Compiler ignoriert Leerzeichen und Zeilenumbrüche. Er verlässt sich rein auf das Semikolon, um zu wissen: "Hier ist dieser Befehl fertig."

- Jede Anweisung in C endet mit einem Semikolon

- Es ist das „Punkt“-Zeichen der C-Sprache

Ohne ein Semikolon am Ende eines Befehls gibt es gnadenlos Compilerfehler.

Der Funktionsrückgabewert

Über die Zeile

```
return 0;
```

wird die Hauptfunktion *main* verlassen. Der Rückgabewert 0 besagt, dass es keinen Fehler gegeben hat und ist hier hart kodiert vergeben worden. In den meisten Programmiersprachen wie bei C, C++, Java oder Python und Betriebssystemen, hat ein Rückgabewert von 0 eine ganz bestimmte, standardisierte Bedeutung: *Erfolg*.

In der main-Funktion (Programmende)

Wenn ein Programm beendet wird, gibt es einen "Exit-Status" an das Betriebssystem zurück.

- Rückgabewert 0: Das Programm wurde erfolgreich und ohne Fehler ausgeführt.
- Rückgabewert ungleich 0 (z. B. 1, 2, -1): Es gab einen Fehler. Die spezifische Zahl kann dabei helfen, die Art des Fehlers zu identifizieren (z. B. 1 für "Allgemeiner Fehler", 2 für "Datei nicht gefunden").

Zur Laufzeit

Was passiert zur Laufzeit des Programms nach dem Kompilieren? Beim Ausführen des Programms:

1. *printf* bekommt den Text
2. Der Text wird an *stdout* geschickt
3. AmigaDOS gibt ihn in der Shell aus
4. Die Escape-Sequenz \n erzeugt eine neue Zeile

Doch so weit sind wir im Moment noch nicht. Du hast also den Quellcode unter dem Namen *hello.c* in den Editor eingeben. Die Dateierweiterung *.c* steht natürlich als Kennzeichnung für eine Datei mit dem C-Quellcode. Drücke wieder die Taste

und verlasse den Editor mit

Werfen wir einen Blick ins Dateisystem.

Abbildung 21 - Die Quelldatei von hello.c ist gespeichert worden

Der Compiler

Nun müssen wir natürlich den Quellcode dem Compiler übergeben, was mit der folgenden Befehlszeile erfolgt.

```
cc hello.c
```

Sehen wir uns das im Detail an.

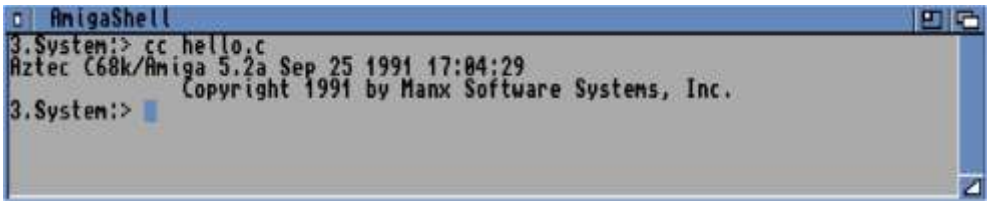

Abbildung 22 - Das Kompilieren des Quellcodes hello.c

Als Antwort auf die Eingabe des Befehls ist lediglich die Anzeige des Textes vom Aztec-Compiler zu sehen, was ein gutes Zeichen ist. Schauen wir erneut ins Dateisystem.

Abbildung 23 - Die Quelldatei von hello.c in eine Objekt-Datei überführt worden

Es ist eine weitere Datei mit gleichem Namen, jedoch mit der Endung .o zu sehen. Diese Objekt-Datei ist ein Zwischenergebnis beim Übersetzen eines Programms und ist noch kein fertiges ausführbares Programm, sondern ein Baustein daraus. Die Objekt-Datei enthält bereits Maschinencode, aber:

- nicht vollständig verbunden
- nicht alleine lauffähig

Du störst dich vielleicht an der Begrifflichkeit: „nicht vollständig verbunden“. Was bedeutet „verbunden“ in diesem Kontext? Ein fertiges Programm muss alle Verweise kennen:

- Funktionen (z. B. printf)
- Globale Variablen
- Startup-Code (für Amiga-Executables)

Wenn eine Datei nicht vollständig verbunden ist, bedeutet dies:

- die Maschinencode-Befehle existieren bereits
- aber die Adresse von externen Funktionen ist noch unbekannt

Man könnte sagen, dass der Compiler aus dem Quellcode einen Maschinencode generiert, jedoch die Funktion printf intern noch nicht existiert. Die Objekt-Datei enthält nur einen Platzhalter, eine Art „Verweis“ darauf.

Der Linker

Im nächsten Schritt kommt dann der sogenannte *Linker* ins Spiel und greift helfend ein, dass alles zusammengeführt wird, was zusammengehört. Er ersetzt diesen Platzhalter durch die richtige Adresse aus der C-Library (clib.lib). Genau das wird mit der folgenden Zeile erreicht, wobei die Dateiendung .o auch entfallen kann.

```
ln hello.o -lc
```

Die Option für den Linker ist in diesem Fall *-lc* was folgendes bedeutet:

- ***-l*** = „Library einbinden“
- ***c*** = Standard-C-Library (in CLIB, z. B. DH0:Aztec/lib/libs/clib.lib)

Sehen wir uns die Ausgabe an.

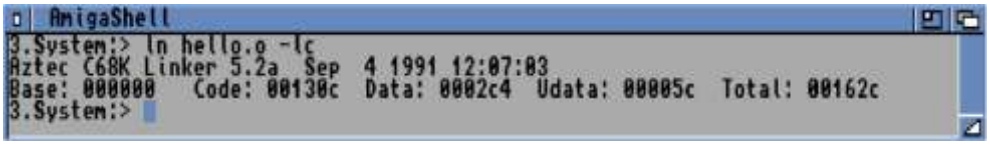

Abbildung 24 - Die Ausgabe des Linkers

Das ist auch ohne eine Fehlermeldung vonstattengegangen. Werfen wir erneut einen Blick ins Dateisystem.

Abbildung 25 - Die ausführbare Datei ist erschienen

Du siehst, dass jetzt eine Datei mit Namen *hello* zu sehen ist, also keine Dateierweiterung vorweist, was auf die ausführbare Datei (Executable) hinweist. Du kannst diese jetzt durch die Eingabe des Namens ausführen und siehst unmittelbar das Ergebnis des Quellcodes.

Abbildung 26 - Das Programm hello wurde ausgeführt

Der Ablauf in der Produktionskette vom Quellcode zum ausführbaren Programm ist nachfolgend zu sehen.

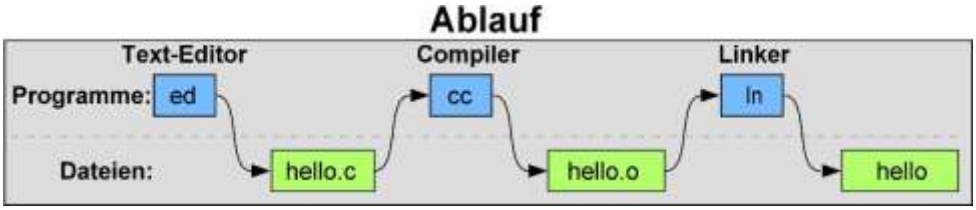

Abbildung 27 - Vom Quellcode zum ausführbaren Programm

Du hast anhand der Kette bestimmt schon gesehen, dass das Erstellen von Programmen eigentlich immer nach einem festen Schema abläuft, was man anhand der folgenden Eckpunkte festmachen kann.

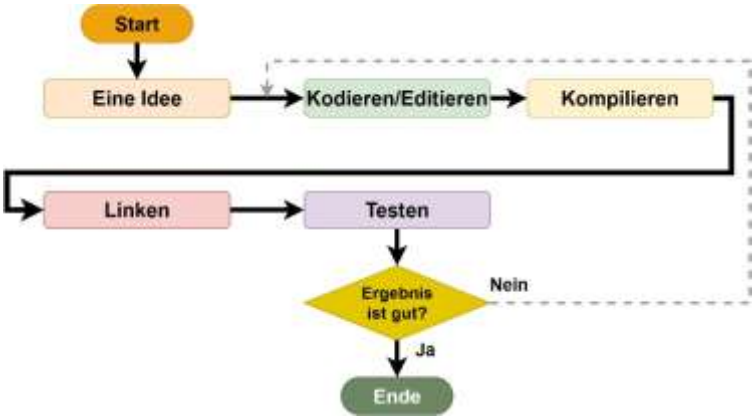

Abbildung 28 - Das Programmieren

Ich denke, dass diese kurze Einführung zur Installation des Aztec C-Compilers mit einem ersten Test genügen sollte. Alles weitere zu C-Programmierung ist in meinem Buch beschrieben.

https://erik-bartmann.de/