8. Die Umgebung von C-Programmen

8.1 Vom Quellcode zur Programmausführung

8.2 Der Präprozessor

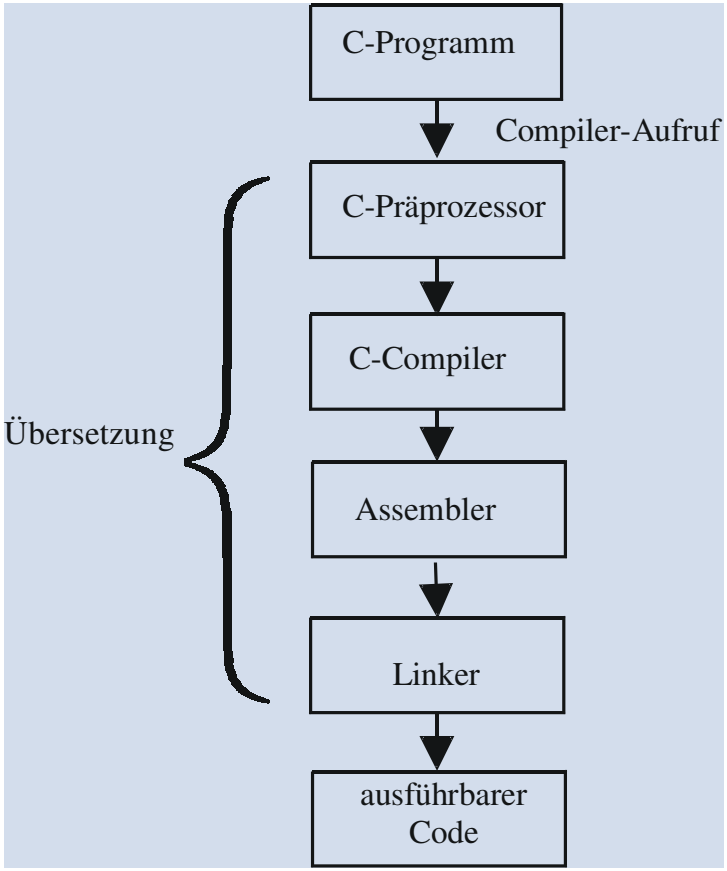
8.3 Compiler, Assembler, Binder

8.4 Das Make-Utility

8.5 Datenübergabe vom / zum Betriebssystem

8.1 Vom Quellcode zur ausführbaren Programm

Wie wird ein C-Programm ausführbar?



Dem C-Compiler können verschiedene Parameter übergeben werden, um die Übersetzung an den verschiedenen Stufen zu unterbrechen und dem Präprozessor/Compiler/Linker bestimmte Hinweise zu übergeben.

8.2 Der Präprozessor

Der Präprozessor "filtert" den Quellcode, er

- eliminiert Kommentare und
- löst Präprozessor-Anweisungen auf Programmzeilen mit "#" in der ersten Spalte).

Wichtige Präprozessoranweisungen

#include

#define

Die **"#include"** -Direktive dient dem Einfügen einer Textdatei. In der Regel werden nur Header-Files (Definitionsdateien) damit eingebunden.
Definitionsdateien enthalten:

- #define - Anweisungen
- #include - Anweisungen
- Typdeklarationen (typedef)
- extern-Deklarationen
- Funktionsprototypen
- Makros.

Sie sollten aber normalerweise keinen Funktionscode enthalten.

Der Dateibezeichner für Definitionsdateien endet üblicherweise mit ".h".

Die #include-Anweisung

Dient zum Einfügen („Hereinziehen“) von externen Dateien vor der Kompilation.

#include "filename.h“ /* mit Gänsefüßchen */

- sucht die Datei zunächst im aktuellen Verzeichnis
- wird in der Regel für projektbezogene Headerfiles verwendet.

#include <filename.h> /* ohne Gänsefüßchen */

- sucht die Datei zunächst in speziellen include-Verzeichnissen des Betriebssystems. Der Pfad dafür ist dem Betriebssystem bekannt.
- wird für Definitionsdateien der *Standardbibliotheken* verwendet.

Beispiel

#include <stdio.h>

Anmerkungen zu #include-Dateien (1)

Die Standardbibliotheken gehören zum C-System. Sie enthalten verschiedene Funktionen, die ein C- Programmierer oft benötigt oder die systemabhängig sind (wie zum Beispiel die Ein- und Ausgabe). Ihr Quellcode ist für den Programmierer meist nicht verfügbar. Er kann jedoch über die Definitionsdateien auf die Funktionsprototypen zugreifen. Diese dienen ihm als Schnittstelle.

Woher weiß der Compiler, wo er die Header-Dateien findet?

- Betriebssystem-Pfad z. B. bei <stdio.h>
- Angabe beim Kompilieren als Parameter

Anmerkungen zu #include-Dateien (2)

Woher weiß der Binder (linkage editor), welche Bibliothek er dazu binden muss?

Ortsbestimmung:

- Betriebssystem-Pfad .../lib
- Angabe beim Kompilieren als Parameter

Namensbestimmung:

- Zuordnung ist ihm bekannt, z. B. bei <stdio.h>
- Angabe beim Kompilieren als Parameter

Die #define-Anweisung

1. Textersatz

Syntax

`#define identifier tokenstring`$_{OPT}$

Bedeutung: `#define Suchtext Ersatztext`

- dient der Definition von **Konstanten** (symbolischen Namen)
- bewirkt, dass im Quellcode bei jedem Auftreten von "Suchtext" diese Zeichenkette durch "Ersatztext" ersetzt wird.

Beispiele

```
#define  PI  3.141592
#define max_length 100
```

Vorteile

+ bessere Lesbarkeit

+ leichtere Änderbarkeit

#define-Anweisungen für Makros

2. Makros

#define identifier(identifier, ..., identifier) TokenstringOPT

- dient der Definition von Makros
- die Bezeichnungen in Klammern wirken wie eine formale Parameterliste, die im Ersatztext durch aktuelle Werte ersetzt werden.

Beispiel

#define SQR(x) ((x) * (x)) bewirkt SQR(a + b) → ((a + b) * (a + b))

Man beachte dabei die Klammerung!

Kurze Funktionen werden manchmal als Makros implementiert.

Beispiel für einen **Seiteneffekt:**

SQR (n++) -> ((n++) * (n++))

n wird um 2 erhöht und nicht, wie man vermuten könnte, um 1!

#if, #ifdef und #ifndef (1)

Mit #if und #ifdef kann man eine **bedingte Kompilation** durchführen. Diese dient häufig dazu, dieselbe Quelldatei für mehrere Zielrechner oder Betriebssystem-Umgebungen funktionsfähig zu machen.

Die **#if-Anweisung** markiert einen Block, der nur ausgeführt wird, wenn der Ausdruck, der auf das #if folgt, ungleich Null ist.

Syntax:

```
#if constant_integral-expression
```

Die **#ifdef-Anweisung** überprüft, ob ein Name mit der Anweisung #define festgelegt wurde. Ist dies der Fall, so wird der nachfolgende Block ausgeführt, ansonsten wird der nachfolgende Block übersprungen.

Syntax:

```
#ifdef identifier
```

#if, #ifdef und #ifndef (2)

Die **#ifndef-Anweisung** überprüft, ob der nachfolgende Name *nicht* mit der Anweisung #define festgelegt wurde. Ist dies der Fall, so wird der nachfolgende Block ausgeführt, ansonsten wird der nachfolgende Block übersprungen.

Syntax:

```
#ifndef identifier
```

#if, #ifdef und #ifndef (3)

Die **#endif-Anweisung** markiert das Ende eines Blockes der #if-, #ifdef- oder #ifndef-Anweisung.

Die **#else-Anweisung** leitet die Alternative nach eine #if-, #ifdef- oder #ifndef-Anweisung ein. Der mit #else begonnene Block muss immer mit #endif abgeschlossen werden.

Die **#elif-Anweisung** dient zur Programmierung von Mehrfach-Verzweigungen bei der Kompilation (“else if“). Sie folgt auf eine #if-, #ifdef- oder #ifndef-Anweisung. Der mit #elif begonnene Block wird mit einem weiteren #elif, einem #else oder einem #endif abgeschlossen.

Beispiele für die bedingte Kompilation

Beispiel 1

```
#ifndef MY_DEFS  /* noch nicht definiert? */
#define MY_DEFS
#define YES 8888 /* meine Konstante für YES */
#define NO 9999  /* meine Konstante für NO  */
#endif
```

Beispiel 2

Ein- und Ausschalten von Ausgaben für Debugging

```
#define DEBUG 1
...
#if DEBUG
   printf("debug: a = %d\n", a);
#endif
```

8.3 Compiler, Assembler, Binder

Der Compiler

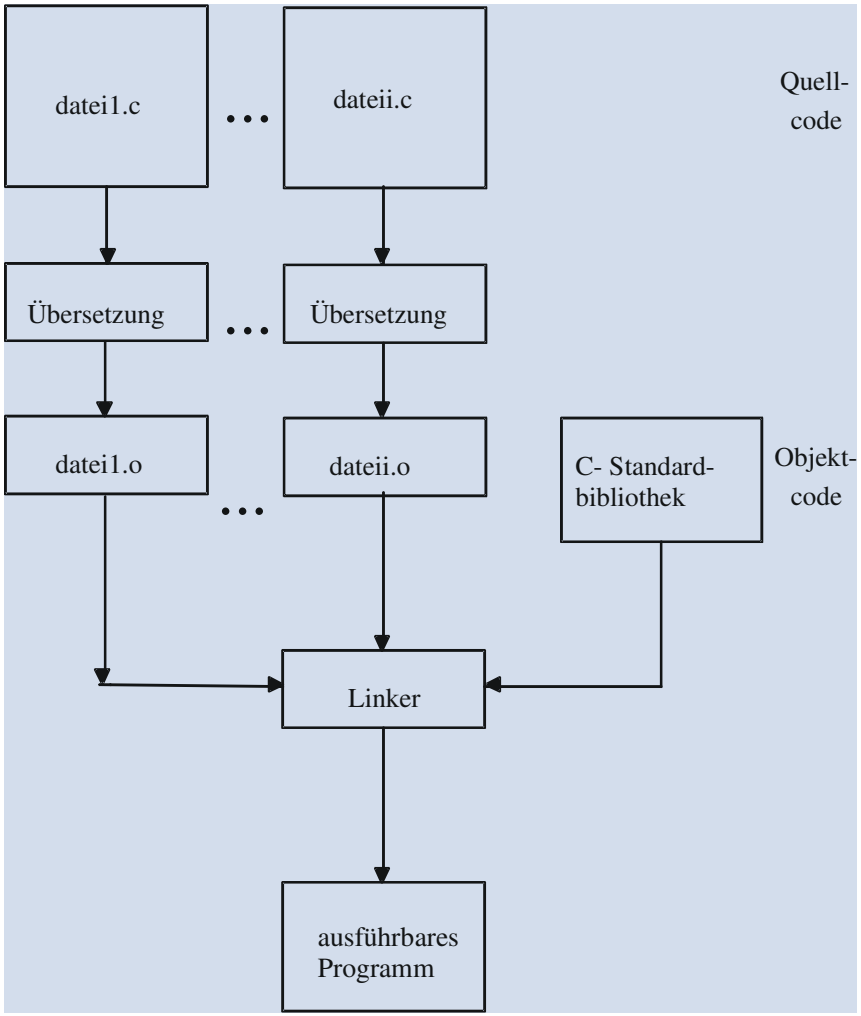
Normalerweise übersetzt der Compiler das vom Präprozessor vorbereitete Programm in Assembler, wobei er die Syntax analysiert und dann, falls korrekt, Code generiert.

Übertragung in Objektcode

Durch den Assembler wird dann der vom Compiler erzeugte Assembler-Code in Objektcode übertragen. Dieser muss noch gebunden („gelinkt“) werden, bevor er ausführbar wird.

Ein letzter Schritt: der Binder (“Linker“)

Der Linker verbindet die Objektcodes der einzelnen Module (inklusive Standard-bibliotheken) zu einem ausführbaren Programm.



Modularisierung

C-Programme werden üblicherweise nicht in einer einzigen Datei erstellt, sondern auf mehreren Dateien verteilt:

- Dateien mit C-Quellcode, die Funktionen enthalten (*.c)
- Dateien, die zu den C-Quellcode-Dateien die Definitionen enthalten (Header-Dateien: *.h)
- Bibliotheksdateien, die vor allem Standard-funktionen zu Verfügung stellen.

Genau eine der C-Quellcode-Dateien muss genau ein Hauptprogramm namens “main“ enthalten.

Vorteile der Modularisierung

- Möglichkeit zur Gruppenarbeit
- übersichtliche Strukturierung des Codes

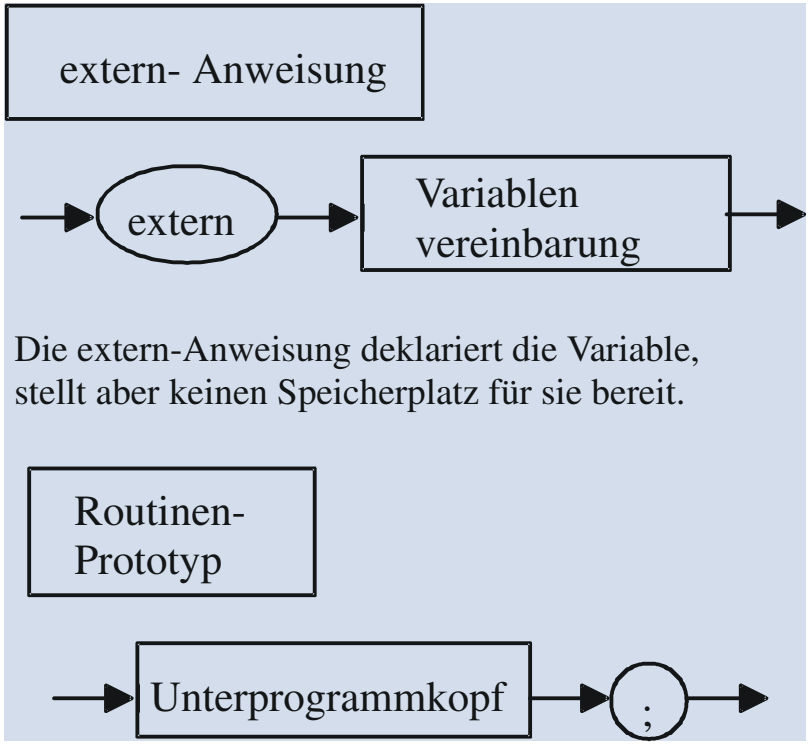
Sichtbarkeit über Dateigrenzen hinweg (1)

Variablen und Funktionen sind nur innerhalb des Blocks sichtbar (gültig), innerhalb dem sie definiert wurden. Eine Datei stellt auch einen Block dar.

Will man Variablen oder Routinen in einer anderen Datei oder vor ihrer Definition verwenden, so ist ein Verweis auf diese Objekte notwendig (ihre Deklaration).

Bei Variablen wird dies mit der **`extern`-Anweisung**, bei Funktionen durch Angabe eines **Prototypen** durchgeführt.

Sichtbarkeit über Dateigrenzen hinweg (2)



Beispiel für eine `extern`-Anweisung

```
extern int a;

void f(char, int);
```

Diese Deklaration sagt dem Compiler, dass er davon ausgehen soll, dass diese Objekte existieren und anderswo definiert werden. Er stellt daraufhin einen Verweis bereit, der beim Binden (“linken“) aufgelöst werden muss.

Merke:

Externe globale Variable sind zu vermeiden - der Datenaustausch über Parameterlisten ist wegen der Vermeidung von Seiteneffekten meist besser.

Routinen in .c und .h

Beispiel

Bei der Erstellung eines kleinen Programmsystems verlangt der Kunde, dass

- alle Routinen (Unterprogramme) in einer Datei gespeichert werden,
- nur das Hauptprogramm extra gespeichert wird.

Erstellung des Systems

routinen.c:
Routinendefinitionen
#include "programm.h"

routinen.h:
Prototypen

programm.c:
main ()
(globale Variablendefinition)

programm.h:
typedefs

#define

(extern)

#include "programm.h"
#include "routinen.h"

Übersetzung des Systems

1.) gcc -ansi -Wall -c routinen.c

=> routinen.o

2.) gcc -ansi -Wall -c programm.c

=> programm.o

3.) gcc programm.o routinen.o \-o programm

=> programm

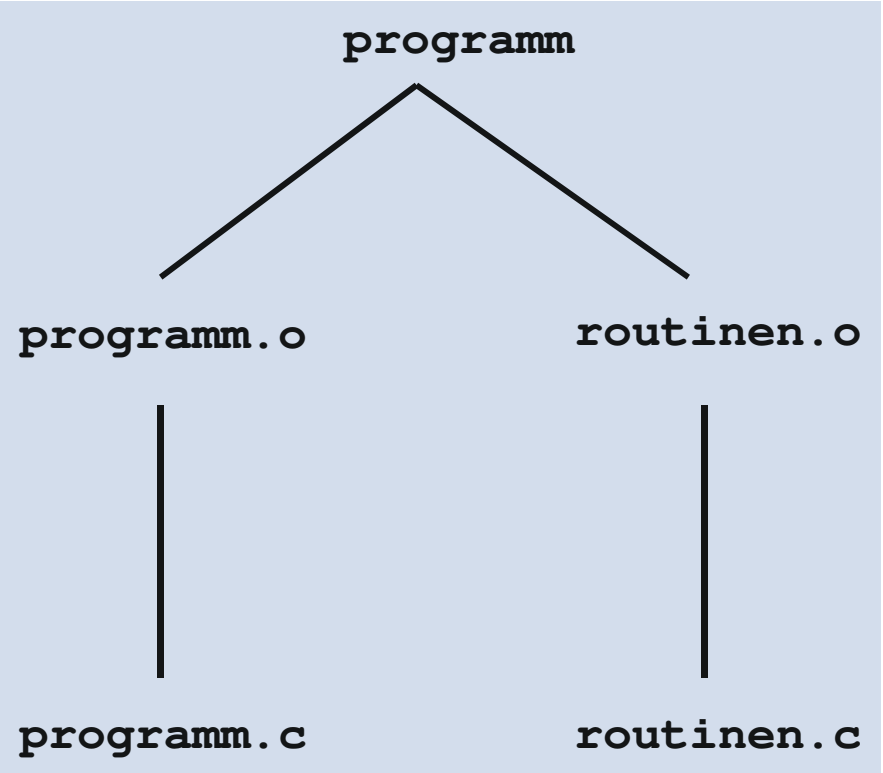
8.4 Das Make-Utility

In Unix gibt es ein nützliches Werkzeug zur Erstellung von ausführbaren Programmen aus verschiedenen Quelldateien: `make`. In einer Datei, dem “description file“, werden die Schritte zur Erstellung des Programms angegeben.

Dabei legt man vor allem fest, wie die Neu-Kompilation und das Neu-Binden von einzelnen Dateien von Änderungen in anderen Dateien abhängen; man sagt also dem Make-Utility, was neu kompiliert und gebunden werden muss, wenn sich in einer bestimmten Datei etwas geändert hat.

Man kann sich diese Abhängigkeiten in Form eines Baumes repräsentiert vorstellen. Dieser Baum wird im “description file“ dann zeilenweise textuell dargestellt.

Abhängigkeitsbaum für Make



Beispiel für Make

```
CC = gcc
CFLAGS = -ansi -Wall -O
#gcc ruft auch den Linker auf
LD = gcc
LDFLAGS =
SRC = programm.c  routinen.c
OBJ = programm.o  routinen.o

routinen.o:     routinen.c
                $(CC)  $(CFLAGS)  -c routinen.c
programm.o:     programm.c
                $(CC)  $(CFLAGS)  -c programm.c
programm:       $(OBJ)
                $(LD)  $(LDFLAGS)  $(OBJ)  -o $@
```

Ein Schritt wird immer dann ausgeführt, wenn die Input-Dateien jünger sind als die Output-Dateien.

Makefile (1)

Aufruf: `make programm`

Die Zuweisungen stellen Textersatz (Makros) dar und die Eintragungen mit ":" **Abhängigkeiten** für die Erstellung des Targets dar.

$@ steht für das aktuelle Ziel.

Abhängigkeiten werden wie folgt definiert:

Ziel1 Ziel2 ... : Abhängigkeit1 Abhängigkeit2 ...

< TAB> Kommandos

- Ziele müssen durch Leerzeichen getrennt werden
- zwischen Zielen und Abhängigkeiten muss ein ":" stehen
- Abhängigkeiten müssen durch Leerzeichen getrennt werden
- Kommandos beginnen an der ersten Tabulatorposition.

Mit dem Aufruf "`make <ziel>`" wird eine solche Abhängigkeit überprüft.

<ziel> muss immer jünger sein als jede Abhängigkeit.

Makefile (2)

Beispiel

Mit dem Aufruf "make programm" testet make, ob programm.c oder routinen.c geändert wurden, d. h. ob "programm" noch den zum Quellcode passenden Objektcode enthält. Falls ja, macht make nichts. Sonst führt make die Kommandos aus, die in der Abhängigkeit spezifiziert werden.

Angenommen, die Datei routinen.c sei verändert worden. Dann transformiert

```
gcc -ansi -Wall -c routinen.c
```

routinen.c in routinen.o.

Weiterhin sei programm.c unverändert geblieben. Dann wird programm.o nicht neu erzeugt, aber der Binder wird aufgerufen:

```
gcc programm.o routinen.o -o programm
```

Er bindet die Objektdateien neu zu einer ausführbaren Datei.

Literaturhinweise zu make

Andrew Oram, Steve Talbott: Managing Projects with make. 2nd Edition, 1991, O'Reilly & Associates Inc., ISBN: 0-937175-900

Man-Pages (das Online-Manual von Unix): "man make"

8.5 Datenübergabe vom / zum Betriebssystem

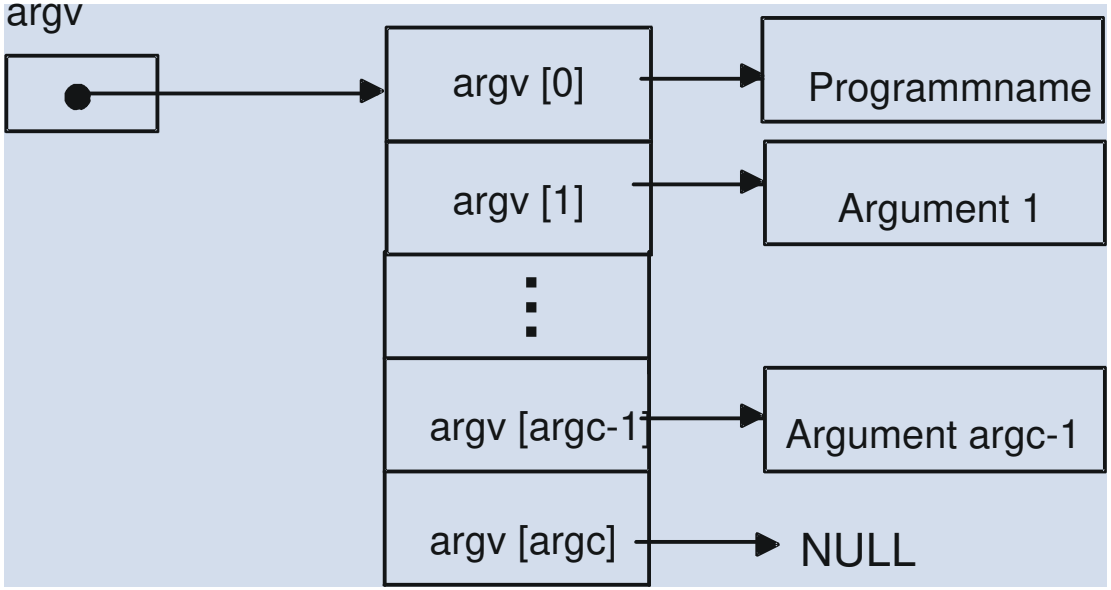
Einem Programm können beim Start in der Kommandozeile Argumente übergeben werden. Dazu muss `main` mit Parametern definiert werden:

```
int main (int argc, char * argv [])
```

Erinnerung

- `char * argv []` entspricht den Argumenten des Programmaufrufs (Vektor mit Zeigern auf eine Zeichenkette)
- `int argc` entspricht der Anzahl der übergebenen Argumente
- `int main`: `main` kann mit `return` einen Resultatwert an die Betriebssystemumgebung zurückliefern

argv


argv
argv [0]
Programmname
argv [1]
Argument 1
argv [argc-1]
Argument argc-1
argv [argc]
NULL

Beispiel: Kommunikation zwischen C-Programm und Unix

```
int main (int argc, char * argv[ ])
{
   int i;
   printf ("Programmname: %s\n", argv[0]);
   for (i = 1; i < argc; i++)
    printf ("Argument %d: %s \n", i, argv[i]);
   return 0;
}
```

- Das Programm gibt zunächst den Programmnamen aus
- dann die einzelnen Argumente, mit denen das Programm aufgerufen wurde
- und gibt zuletzt mit "`return 0`" den Wert 0 an das Betriebssystem zurück.

Üblicherweise gibt in Unix return bei Erfolg 0 zurück, -1 bei Fehler.

`return` in `main` übergibt die Kontrolle zurück an das Betriebssystem, d. h., alle Anweisungen, die nach `return` im Programmtext stehen, werden nicht mehr ausgeführt.