2.9 Präprozessor-Direktiven

Wenn in der Sprache C ein Programm übersetzt (kompiliert und gelinkt) werden soll, dann wird, bevor der Compiler den Quelltext verarbeitet, von einem besonderen Teil des Compilers – dem Präprozessor – ein zusätzlicher Übersetzungslauf durchgeführt.

Bei Präprozessor-Direktiven steht immer das Zeichen # am Anfang der Zeile.

2.9 Präprozessor-Direktiven

Die folgenden Arbeiten fallen für den Präprozessor neben der Quelltextersetzung ebenfalls an:

- Stringliterale werden zusammengefasst (konkateniert).
- Zeilenumbrüche mit einem Backslash am Anfang werden entfernt.
- Kommentare werden entfernt und durch Leerzeichen ersetzt.
- Whitespace-Zeichen zwischen Tokens werden gelöscht.

Des Weiteren gibt es Aufgaben für den Präprozessor, die vom Programmierer gesteuert werden können:

- Header- und Quelldateien in den Quelltext kopieren (#include)
- symbolische Konstanten einbinden (#define)
- bedingte Kompilierung (#ifdef, #elseif, ...)

2.9.1 Einkopieren von Dateien #include

Die Direktive #include kopiert andere, benannte (Include-)Dateien in das Programm ein. Meistens handelt es sich dabei um Headerdateien mit der Extension *.h oder *.hpp.

Syntax der Präprozessor-Direktive include:

```
#include <header >
#include "header“
```

Der Präprozessor entfernt die include-Zeile und ersetzt diese durch den Quelltext der include-Datei. Der Compiler erhält anschließend einen modifizierten Text zur Übersetzung.

2.9.1 Einkopieren von Dateien #include

Steht die Headerdatei hingegen zwischen eckigen Klammern (wie dies bei Standardbibliotheken meistens der Fall ist), also so:

```
#include <datei.h>
```

so wird die Headerdatei datei.h im implementierungsdefinierten Pfad gesucht. Dieser Pfad befindet sich in dem Pfad, in dem sich die Headerdateien Ihres Compilers befinden.

Steht die Headerdatei zwischen zwei Hochkommata, also so:

```
#include "datei.h“
```

so wird diese im aktuellen Arbeitsverzeichnis oder in dem Verzeichnis gesucht, das mit dem Compiler-Aufruf `-I` angegeben wurde – vorausgesetzt, Sie übersetzen das Programm in der Kommandozeile. Sollte diese Suche erfolglos sein, so wird in denselben Pfaden gesucht, als wäre `#include <datei.h>` angegeben.

2.9.1 Wichtige #include Dateien

stdio.h	Input/Output Functions
conio.h	console input/output
assert.h	Diagnostics Functions
ctype.h	Character Handling Functions
cocale.h	Localization Functions
math.h	Mathematics Functions
setjmp.h	Nonlocal Jump Functions
signal.h	Signal Handling Functions
stdarg.h	Variable Argument List Functions
stdlib.h	General Utility Functions
string.h	String Functions
time.h	Date and Time Functions

complex.h	A set of function for manipulating complex numbers
stdalign.h	For querying and specifying the alignment of objects
errno.h	For testing error codes
locale.h	Defines localization functions
stdatomic.h	For atomic operations on data shared between threads
stdnoreturn.h	For specifying non-returning functions
uchar.h	Types and functions for manipulating Unicode characters
fenv.h	A set of functions for controlling the floating-point environment
wchar.h	Defines wide string handling functions
tgmath.h	Type-generic mathematical functions
stdarg.h	Accessing a varying number of arguments passed to functions
stdbool.h	Defines a boolean data type

2.9.2 Bedingte Kompilierung

Diese Direktiven werden eingesetzt, um zu überprüfen, ob ein Symbol zuvor schon mit `#define` definiert wurde. Ist symbol definiert, liefern diese Direktiven 1 zurück, ansonsten 0. Abgeschlossen wird eine bedingte Übersetzung mit der Direktive `#endif`.

```
#ifdef symbol1
// Do something if symbol1
#elif symbol2
//do something if symbol2
#else
//else
#endif
```

2.9.2 Bedingte Kompilierung

Der `#ifndef-Wrapper` ist der traditionelle und C-konforme Ansatz, dass Problem der Mehrfacheinbindung zu lösen.

```
#ifndef A_H
#define A_H
    struct A
    { /* ... */
    }a;
#endif /* A_H */
```

Die obenstehenden Präprozessor-Befehle bewirken, dass beim erstmaligen Einbinden von `A.h` das Makro `A_H` noch nicht definiert ist und der Präprozessor die Definitionen durchläuft. Beim zweiten Einbinden (aus `B.h`) ist das Makro bereits definiert und der Präprozessor überspringt den Block `#ifndef … #endif`.

2.9.2 Bedingte Kompilierung

Durch die bedingte Kompilierung besteht z.B. die Möglichkeit, Programme einfacher auf andere Systeme zu portieren.

Hier wurden mehrere main()-Funktionen verwendet. Auf dem System, für das die bedingte Kompilierung gilt, wird die entsprechende main-Funktion auch ausgeführt.

```c
/* t_system.c */
#include <stdio.h>
#include <stdlib.h>

#ifdef __MSDOS__
int main(void) {
   printf("Programm läuft unter MSDOS \n");
   return EXIT_SUCCESS;
}

#elif __WIN32__ || _MSC_VER
int main(void) {
   printf("Programm läuft unter Win32\n");
   return EXIT_SUCCESS;
}

#elif __unix__ || __linux__
int main(void) {
   printf("Programm läuft unter UNIX/LINUX\n");
   return EXIT_SUCCESS;
}

#else
int main(void) {
   printf("Unbekanntes Betriebssystem!!\n");
   return EXIT_SUCCESS;
}
#endif
```

2.10 Dynamische Speicherverwaltung

Bevor gezeigt wird, wie Speicher dynamisch reserviert werden kann, folgt ein Exkurs über das Speicherkonzept von laufenden Programmen. Ein Programm besteht aus den vier Speicherbereichen.

Speicherbereich	Verwendung
Code	Maschinencode des Programms
Daten	statische und globale Variablen
Stack	Funktionsaufrufe und lokale Variablen
Heap	dynamisch reservierter Speicher

2.10 Dynamische Speicherverwaltung

Code-Speicher

Der Code-Speicher wird in den Arbeitsspeicher geladen, und von dort aus werden die Maschinenbefehle der Reihe nach in den Prozessor (genauer gesagt in die Prozessor-Register) geschoben und ausgeführt.

Daten-Speicher

Im Daten-Speicher befinden sich alle statischen Daten, die bis zum Programmende verfügbar sind (globale und statische Variablen).

Stack-Speicher

Im Stack-Speicher werden die Funktionsaufrufe mit ihren lokalen Variablen verwaltet.

Heap-Speicher

Über ihn wird die dynamische Speicherreservierung mit Funktionen wie malloc() erst realisiert. Der Heap funktioniert ähnlich wie der Stack. Bei einer Speicheranforderung erhöht sich der Heap-Speicher, und bei einer Freigabe wird er wieder verringert. Wenn ein Speicher angefordert wurde, sieht das Betriebssystem nach, ob sich im Heap noch genügend zusammenhängender freier Speicher dieser Größe befindet. Bei Erfolg wird die Anfangsadresse des passenden Speicherblocks zurückgegeben.

2.10.1 Speicherallokation mit malloc()

Mit der Funktion malloc() kann Speicher dynamisch reserviert werden. Es wird dabei auch von einer Speicherallokation (allocate, dt. zuweisen) gesprochen. Die Syntax dieser Funktion sieht so aus:

```
void *malloc(size_t size);
```

bei erfolgreichem Aufruf liefert die Funktion malloc() die Anfangsadresse mit der Größe size Bytes vom Heap zurück. Da die Funktion einen void-Zeiger zurückliefert, hängt diese nicht von einem Datentyp ab.

```
int main(void) {
   int *p;

   p = malloc(sizeof(int));
   if(p != NULL) {
      *p=99;
      printf("Allokationerfolgreich ... \n");
   }
   else {
      printf("Kein virtueller RAM mehr verfügbar ...\n");
      return EXIT_FAILURE;
   }
   return EXIT_SUCCESS;
}
```

2.10.2 Speicher wieder freigeben free()

Wenn Sie Speicher vom Heap angefordert haben, sollten Sie diesen auch wieder zurückgeben. Der allozierte Speicher wird mit folgender Funktion freigegeben:

```
void free (void *p)
```

Der Speicher wird übrigens auch ohne einen Aufruf von free() freigegeben, wenn sich das Programm beendet.

```
int main(void) {
   int *p = malloc(sizeof(int));

   if(p != NULL) {
      *p=99;
      printf("Allokation erfolgreich ... \n");
      }
   else {
      printf("Kein virtueller RAM mehr verfügbar ... \n");
      return EXIT_FAILURE;
   }
   if(p != NULL)
      free(p);
   return EXIT_SUCCESS;
}
```

2.11 Zeiger auf Zeiger

„Zeiger auf Zeiger“ sind ein recht schwieriges Thema, aber es zu verstehen, lohnt sich. Die Syntax von Zeigern auf Zeiger sieht so aus:

```
datentyp **bezeichner;
```

Was heißt jetzt »Zeiger auf Zeiger« genau?
Sie haben einen Zeiger, der auf einen Zeiger zeigt, der auf eine Variable zeigt, und auf diese Variable zurückgreifen kann. Im Fachjargon wird dabei von einer *mehrfachen Indirektion* gesprochen. Theoretisch ist es auch möglich, Zeiger auf Zeiger auf Zeiger usw. zu verwenden. In der Praxis machen allerdings solche mehrfachen Indirektionen kaum noch Sinn. Meistens verwenden Sie Zeiger auf Zeiger, also zwei Dimensionen.

2.12 Argumentübergabe

Programmaufruf (UNIX)

allg.:*programmname* $parameter_1$ ... $parameter_n$

- Parameter sind Flags (Optionen), die der Auswahl oder der Präzisierung von Programmfunktionen dienen (beginnen mit einem Minuszeichen - erlaubt ist eine beliebige Reihenfolge sowie die Zusammenfassung mehrerer Flags)
- Namen von Objekten mit denen das Programm arbeiten soll (z.B. Filenamen)
- oder (ganz allgemein formuliert) Informationen, die dem Programm bereitgestellt werden

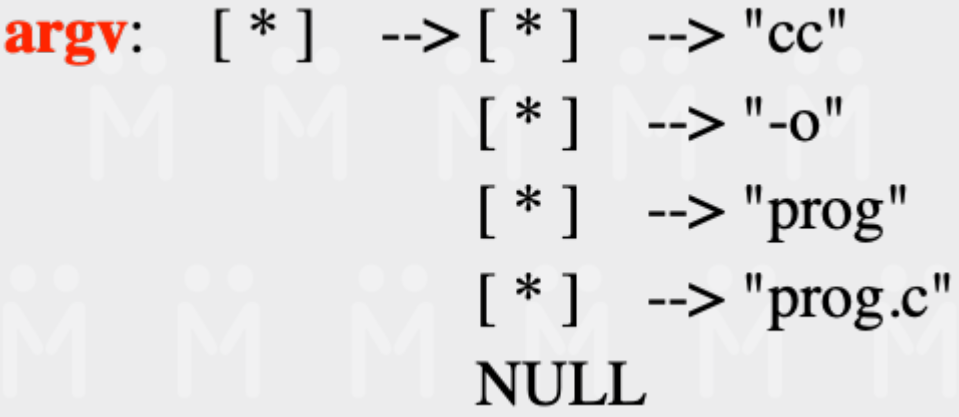
An die **main-Funktion** werden Argumente übergeben. Das erste gibt die Anzahl der Aufrufparameter an (**argc**), das zweite die Adresse eines Feldes (**argv**).

2.12 Argumentübergabe

Bezogen auf die Kommandozeile zum Aufruf des C-Compilers

```
cc -o prog prog.c
```

enthält **argc** den Wert 4, **argv** zeigt auf ein Feld mit **5** Zeigern. Dabei enthalten die einzelnen Feldelemente die Adressen der Zeichenketten **cc**, **-o**, **prog** und **prog.c** . Der Wert des letztes Zeigers ist **NULL** und spezifiziert damit das Ende der Parameterliste.



2.13 Dateien (Streams)

Häufig wendet man elementare Datei-E/A-Funktionen (E/A steht für *Eingabe/Ausgabe*) an, ohne sich Gedanken darüber zu machen, was eine Datei eigentlich ist.

Im Prinzip können Sie sich eine Datei als ein riesengroßes char-Array vorstellen. Das char-Array besteht dabei aus einer Folge von Bits und Bytes – unabhängig davon, ob es sich um eine Textdatei oder eine ausführbare Datei handelt

2.13 Dateien (Streams)

In C besteht die Möglichkeit, von zwei verschiedenen Ebenen aus auf eine Datei zuzugreifen: zum einen von der höheren Ebene (*High-Level*) aus und zum anderen von der niedrigeren Ebene (*Low-Level*).

Mit der höheren Ebene kann wesentlich komfortabler und vor allem portabler programmiert werden. Die Funktionen der höheren Ebene entsprechen dem ANSI-C-Standard. Zu den Vorteilen der höheren Ebene gehören z. B. eine formatierte Ein- und Ausgabe und ein optimal eingestellter Puffer. Der Puffer ist ein Bereich im Arbeitsspeicher, der als Vermittler zwischen Daten und Zielort fungiert.

Der Unterschied zwischen der höheren und der niedrigeren Ebene besteht in der Form, wie die Daten in einem Stream von der Quelle zum Ziel übertragen werden. Bei der höheren Ebene ist der Stream eine formatierte Dateneinheit (wie z. B. mit printf()). Hingegen handelt es sich bei der niedrigeren Ebene um einen unformatierten Byte-Stream.

2.13.1 Standard-Streams

Streams sind einfache Datenströme, mit denen Daten von der Quelle zum Ziel bewegt werden. Es gibt Standard-Streams in C wie

- die Standardeingabe (stdin),
- die Standardausgabe (stdout)
- und die Standardfehlerausgabe (stderr).

Beim Start eines Programms sind die Standard-Streams stdin, stdout und stderr weder byte- noch wide-orientiert.

2.13.2 Ein- Ausgabe <stdio.h>

Ein *Datenstrom (stream)* ist Quelle oder Ziel von Daten und wird mit einer Platte oder einem anderen Peripheriegerät verknüpft. Die Bibliothek unterstützt zwei Arten von Datenströmen, für Text und binäre Information, die allerdings bei manchen Systemen und insbesondere bei UNIX identisch sind. Ein Textstrom ist eine Folge von Zeilen; jede Zeile enthält null oder mehr Zeichen und ist mit **'\n'** abgeschlossen. Eine Umgebung muß möglicherweise zwischen einem Textstrom und einer anderen Repräsentierung umwandeln (also zum Beispiel **'\n'** als Wagenrücklauf und Zeilenvorschub abbilden). Wird ein Binärstrom geschrieben und auf dem gleichen System wieder eingelesen, so entsteht die gleiche Information.

Ein Strom wird durch *Eröffnen (open)* mit einer Datei oder einem Gerät verbunden; die Verbindung wird durch *Abschließen (close)* wieder aufgehoben. Eröffnet man eine Datei, so erhält man einen Zeiger auf ein Objekt vom Typ **FILE**, wo alle Information hinterlegt ist, die zur Kontrolle des Stroms nötig ist. Wenn die Bedeutung eindeutig ist, werden wir die Begriffe **FILE**-Zeiger und Datenstrom gleichberechtigt verwenden.

2.13.2 Ein- Ausgabe <stdio.h>

`FILE *fopen(const char *filename, const char *mode)`

`fopen` eröffnet die angegebene Datei und liefert einen Datenstrom oder NULL bei Mißerfolg. Zu den erlaubten Werten von mode gehören

- "r" Textdatei zum lesen eröffnen
- "w" Textdatei zum Schreiben erzeugen; gegebenenfalls alten Inhalt wegwerfen
- "a" anfügen; Textdatei zum Schreiben am Dateiende eröffnen oder erzeugen
- "r+" Textdatei zum Ändern (lesen/schreiben) eröffnen
- "w+" Textdatei zum Ändern (lesen/schreiben) erzeugen; gegebenenfalls alten Inhalt wegwerfen
- "a+" anfügen; Textdatei zum Ändern (lesen/schreiben am Ende) eröffnen oder erzeugen,

Bewirkt	r	w	a	r+	w+	a+
Datei ist lesbar.	x			x	x	x
Datei ist beschreibbar.		x	x	x	x	x
Datei ist nur am Dateiende beschreibbar.			x			x
Existierender Dateiinhalt geht verloren.		x			x	

2.13.2 Ein- Ausgabe <stdio.h>

```
int fflush(FILE *stream)
```
Bei einem Ausgabestrom sorgt **fflush** dafür, daß gepufferte, aber noch nicht geschriebene Daten geschrieben werden; bei einem Eingabestrom ist der Effekt undefiniert. Die Funktion liefert **EOF** bei einem Schreibfehler und sonst Null. **fflush(NULL)** bezieht sich auf alle offenen Dateien.

```
int fclose(FILE *stream)
```
fclose schreibt noch nicht geschriebene Daten für **stream**, wirft noch nicht gelesene, gepufferte Eingaben weg, gibt automatisch angelegte Puffer frei und schließt den Datenstrom. Die Funktion liefert **EOF** bei Fehlern und sonst Null.

```
int remove(const char *filename)
```
remove entfernt die angegebene Datei, so daß ein anschließender Versuch, sie zu eröffnen, fehlschlagen wird. Die Funktion liefert bei Fehlern einen von Null verschiedenen Wert.

```
int rename(const char *oldname, const char *newname)
```
rename ändert den Namen einer Datei und liefert nicht Null, wenn der Versuch fehlschlägt.

2.13.3 Formatierte Ein-Ausgabe

```
int fprintf(FILE *stream, const char *format, ...)
```

fprintf wandelt Ausgaben um und schreibt sie in **stream** unter Kontrolle von **format**. Der Resultatwert ist die Anzahl der geschriebenen Zeichen; er ist negativ, wenn ein Fehler passiert ist.

```
int sprintf(char *s, const char *format, ...)
```

sprintf funktioniert wie **printf**, nur wird die Ausgabe in den Zeichenvektor **s** geschrieben und mit **'\0'** abgeschlossen. **s** muß groß genug für das Resultat sein. Im Resultatwert wird **'\0'** nicht mitgezählt.

2.13.4 Ein- und Ausgabe von Zeichen

```
int fgetc(FILE *stream)
```
fgetc liefert das nächste Zeichen aus **stream** als **unsigned char** (umgewandelt in **int**) oder **EOF** bei Dateiende oder bei einem Fehler.

```
char *fgets(char *s, int n, FILE *stream)
```
fgets liest höchstens die nächsten **n-1** Zeichen in **s** ein und hört vorher auf, wenn Zeilentrenner gefunden wird. Der Zeilentrenner wird im Vektor abgelegt. Der Vektor wird
mit **'\0'** abgeschlossen. **fgets** liefert **s** oder **NULL** bei Dateiende oder bei einem Fehler.

```
int fputc(int c, FILE *stream)
```
fputc schreibt das Zeichen **c** (umgewandelt in **unsigned char**) in **stream**. Die Funktion liefert das ausgegebene Zeichen oder **EOF** bei Fehler.

```
int fputs(const char *s, FILE *stream)
```
fputs schreibt die Zeichenkette **s** (die **'\n'** nicht zu enthalten braucht) in **stream**. Die Funktion liefert einen nicht-negativen Wert oder **EOF** bei einem Fehler.

2.13.5 Direkte Ein- und Ausgabe

`size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)` **fread** liest aus **stream** in den Vektor **ptr** höchstens **nobj** Objekte der Größe **size** ein. **fread** liefert die Anzahl der eingelesenen Objekte; das kann weniger als die geforderte Zahl sein. Der Zustand des Datenstroms muß mit **feof** und **ferror** untersucht werden.

`size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)`
fwrite schreibt **nobj** Objekte der Größe **size** aus dem Vektor **ptr** in **stream**. Die Funktion liefert die Anzahl der ausgegebenen Objekte; bei Fehler ist das weniger als **nobj**.

2.13.6 Positionieren in Dateien

```
int fseek(FILE *stream, long offset, int origin)
```

fseek setzt die Dateiposition für **stream**; eine nachfolgende Lese- oder Schreiboperation wird auf Daten von der neuen Position ab zugreifen.
Für eine binäre Datei wird die Position auf **offset** Zeichen relativ zu **origin** eingestellt; dabei können die Werte

- **SEEK_SET** (Dateianfang)
- **SEEK_CUR** (aktuelle Position) oder
- **SEEK_END** (Dateiende) angegeben werden.

Für einen Textstrom muß **offset** Null sein oder ein Wert, der von **ftell** stammt (dafür muß dann **origin** den Wert **SEEK_SET** erhalten). **fseek** liefert einen von Null verschiedenen Wert bei Fehler.

2.13.6 Positionieren in Dateien

```
long ftell(FILE *stream)
```
ftell liefert die aktuelle Dateiposition für **stream** oder **-1L** bei Fehler.

```
void rewind(FILE *stream)
```
rewind(fp) ist äquivalent zu **fseek(fp, OL, SEEK_SET);**
clearerr(fp);

```
int fgetpos(FILE *stream, fpos_t *ptr)
```
fgetpos speichert die aktuelle Position für **stream** bei ***ptr**. Der Wert kann später mit **fsetpos** verwendet werden. Der Datentyp **fpos_t** eignet sich zum Speichern von solchen Werten. Bei Fehler liefert **fgetpos** einen von Null verschiedenen Wert.

```
int fsetpos(FILE *stream, const fpos_t *ptr)
```
fsetpos positioniert **stream** auf die Position, die von **fgetpos** in ***ptr** abgelegt wurde. Bei Fehler liefert **fsetpos** einen von Null verschiedenen Wert.