6.9 Extras/Präprozessor

6.9.1 Was ist ein Präprozessor?

Wenn man einen Quelltext zu einem lauffähigen Programm übersetzt, dann hat dieser Vorgang bei der Programmiersprache C vier Stufen (Abbildung 1):

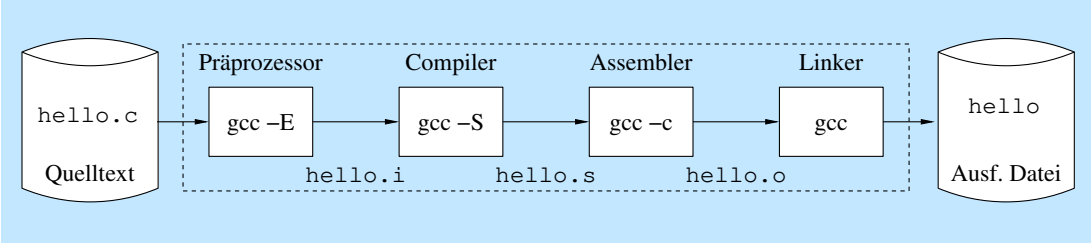

Abbildung 1: Stufen der Übersetzung

a) Der Präprozessor ist eine Art Textverarbeitung, die sich auf das Einbinden von Text (Headerdateien) und auf Suchen-und-Ersetzen (Makros) versteht.

b) Die eigentliche Compiler-Stufe übersetzt von C in die Assembler-Sprache. Diese Sprache besteht bereits aus den CPU-Befehlen, allerdings in für Menschen lesbarer Form (so genannte Mnemonics, Dateiname: `hello.S`).

c) Der Assembler-Stufe übersetzt jeden CPU-Befehl aus der für Menschen lesbaren Form in das für die CPU lesbare Bitmuster. Es entsteht der Objectcode (Dateiname: `hello.o`).

d) Der Linker bindet die Objectcodes der einzelnen Module, die Standard-C-Bibliothek und den Startcode zu einem lauffähigen Programm (Dateiname: `hello`, `hello.exe` oder `a.out`).

Hier soll es nun um die erste Stufe gehen, den Präprozessor. Er liest den Quelltext zeilenweise ein, verändert eventuell an der Zeile etwas und gibt die veränderte Zeile aus. Einige Zeilen sind besonders. Sie beginnen mit dem Zeichen # und werden nicht ausgegeben. Stattdessen sagen sie dem Präprozessor, wie die ab jetzt folgenden Zeilen verändert werden sollen. Diese Zeilen sind Befehle für den Präprozessor (nicht für das Programm selbst). Man nennt diese Befehle *Präprozessor-Direktiven*. Abbildung 2 zeigt das geschilderte Vorgehen als Struktogramm.

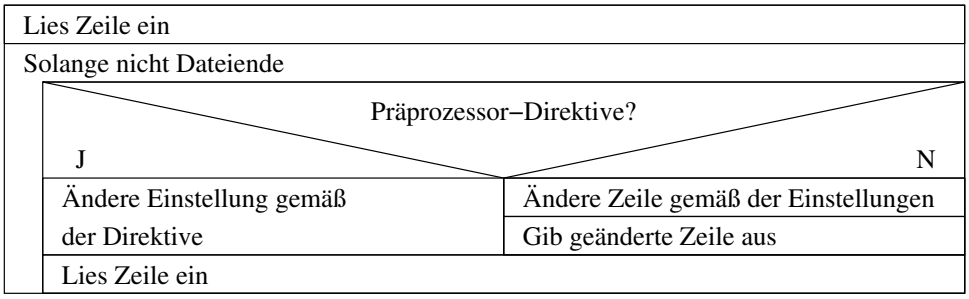

Abbildung 2: Arbeit des C-Präprozessors

6.9.2 C-Präprozessor: Übersicht

Der in C eingebaute Präprozessor hat folgende wichtige Aufgabenfelder:

a) Einlesen von Quelltext (meistens Headerdateien) mit der `#include`-Direktive

b) Ersetzen von Text durch anderen Text mit der `#define`-Direktive

c) Ausblenden bestimmter Textzeilen in Abhängigkeit Präprozessor-interner Variablen mit den Direktiven `#if`, `#ifdef`, `#ifndef` und `#endif`

Daneben gibt es noch systemabhängige Spezialitäten, die mit der Direktive `#pragma` gesteuert werden können.

Außerdem filtert der Präprozessor Kommentare aus und fügt zusammenliegende Zeichenketten zusammen:

```
1  printf("Hel"
2         "lo, World!\n"); // Toll!
```

Wird zu:

```
1  printf("Hello, World!\n");
```

6.9.3 Einlesen von Quelltext

Beim Auftreten der Direktive `#include` wird sofort der Inhalt der genannten Datei eingebunden. Falls der Pfadname der Datei von Anführungszeichen umgeben ist, gilt er als relativer oder absoluter Pfadname im Dateisystem.

Falls er (wie meistens bisher) von spitzen Klammern umgeben ist, sucht der Präprozessor die Datei in einem Systemverzeichnis. Unter Linux sucht er dann im Verzeichnis `/usr/include`.

Mit der Option `-I xyz` kann man ein Verzeichnis `xyz` hinzufügen, in dem der Präprozessor nach Include-Dateien suchen soll.

6.9.4 Ersetzen von Text

Mit der Direktive `#define` wird bestimmt, dass ab sofort das direkt nach `#define` folgende Wort durch die bis zum Ende der Zeile folgende Zeichenkette ersetzt wird. Das zu ersetzende Wort nennt man dabei *Makro*. Hier ein Beispiel:

```
1 #define PI 3.14
```

Hier ist `PI` das Makro. Im weiteren Verlauf des Quelltextes wird jedes Wort `PI` durch den Text `3.14` ersetzt, und zwar unabhängig vo irgendwelchen Sichtbarkeitsbereichen.

Mit der Direktive `#undef` erreicht man, dass ab dieser Stelle das betreffende Makro vergessen wird:

```
1 #undef PI
```

Man kann den Präprozessor nicht nur für Quelltexte benutzen, sondern für beliebige Textdateien:

entchen.txt

```
1 #define LIED schwimmen auf dem See
2 Alle meine Entchen LIED, LIED, ...
```

Diese Datei kann man nun durch den Präprozessor bearbeiten lassen:

Terminal

```
schueler@debian964:~$ cat entchen.txt | gcc -E -
# 1 "<stdin>"
# 1 "<built-in>"
...
Alle meine Entchen schwimmen auf dem See, schwimmen auf dem See, ...
```

Die mit # beginnenden Zeilen dienen dem Compiler als Zusatzinformation und werden hier nicht benötigt:

Terminal
```
schueler@debian964:~$ cat entchen.txt | gcc -E -| grep -v '^#'
Alle meine Entchen schwimmen auf dem See, schwimmen auf dem See, ...
```

Der Ersatztext kann im Prinzip *alles* enthalten: Zahlen, Zeichenketten, Variablennamen, Sonderzeichen und ganze Programmteile:

```
1 #define PI 3.14
2 #define AUSGABETEXT "Bitte warten!"
3 #define EINGABEVAR x
4 #define BEGIN {
5 #define END }
6 #define FUNKTION void bla(void) {printf("blubb");}
```

Falls ein Ersatztext zu lang ist, kann man ihn auf mehrere Zeilen verteilen. Dabei müssen aber alle Zeilen außer der letzten mit dem Backslash enden:

```
1 #define AUSGABE(zahl)      \
2     printf("========\n");   \
3     printf("%d\n", zahl);  \
4     printf("========\n");   \
5     getchar();
```

Es hat sich eingebürgert, alle Makros in Großbuchstaben zu schreiben, um eine Verwechselung mit Variablen- und Funktionsnamen zu vermeiden. Hier ein Beispiel dafür, was sonst passieren kann:

```
1 #define pi 3.14
2 ... // 100 Zeilen spaeter:
3 const int pi=3.14159;
```

Hier wird also versehentlich eine Variable definiert, die genauso heißt wie ein Makro.

Die Ausgabe des Präprozessors ergibt:

```
1 const int 3.14=3.14159;
```

Solche Fehler sind schwer zu finden, denn der Compiler weiß ja gar nicht, dass es ein Makro namens `pi` gibt.

Zur Fehlersuche bietet es sich in solchen Fällen an, den Quelltext erst einmal nur durch den Präprozessor laufen zu lassen:

Terminal
```
schueler@debian964:~$ gcc -E -o hello.i hello.c
```

6.9.5 Vordefinierte Makros

Ein paar Makros sind bereits fertig definiert und helfen eventuell bei der Fehlersuche:

a) `__LINE__`: Zeilennummer als Zahl

b) `__FILE__`: Dateiname als String, z. B. `"hello.c"`

c) `__DATE__`: Aktuelles Datum als String

d) `__TIME__`: Aktuelle Zeit als String

e) `__STDC__`: Wird ersetzt zu 1, wenn der Compiler den ANSI-C-Standard erfüllt.

Die aktuelle Zeit (und Datum) ist die Zeit, wenn der Quelltext durch den Präprozessor läuft, also meistens die Zeit, zu der der Quelltext übersetzt wird.

Damit kann man Fehlermeldungen so erzeugen, dass der Programmierer die fehlerhafte Zeile sofort findet:

```
1    if(dateimerker==NULL)
2    {
3        printf("Fehler in Zeile %d, %s\n", __LINE__, __FILE__);
4        return 0;
5    }
```

6.9.6 Makro mit Parameter(n)

Wie Funktionen dürfen auch Makros Parameter haben (auch mehrere). Die Parameter werden beim Ersetzen durch die aktuelle Zeichenkette ersetzt:

```
1 #define XQ(zahl) zahl*zahl
2     printf("%d\n", XQ(5));
3     printf("%f\n", XQ(3.5));
```

Wird zu:

```
1     printf("%d\n", 5*5);
2     printf("%f\n", 3.5*3.5);
```

Positiv ist: So ein Makro ist typ-unabhängig. Negativ ist: Hier gibt es eine Stelle, an der man aufpassen muss:

```
1 #define XQ(zahl) zahl*zahl
2     ergebnis = XQ(5+1);
```

Wird zu:

```
1     ergebnis = 5+1*5+1;
```

Hier wird 11 statt 36 ausgegeben. Abhilfe schaffen Klammern um jeden Operanden:

```
1 #define XQ(zahl) (zahl)*(zahl)
2     ergebnis =  XQ(5+1);
```

Wird zu:

```
1     ergebnis = (5+1)*(5+1);
```

Nun wird 36 ausgegeben.

Aber nicht nur jeder Operand muss geklammert werden, auch der ganze Ausdruck:

```
1 #define VERDOPPEL(zahl) (zahl)+(zahl)
2     ergebnis = 8*VERDOPPEL(5);
```

Wird zu:

```
1     ergebnis = 8*(5)+(5);
```

Hier wird 45 statt 80 ausgegeben. Abhilfe schaffen Klammern um den ganzen Ausdruck:

```
1 #define VERDOPPEL(zahl) ((zahl)+(zahl))
2     ergebnis = 8*VERDOPPEL(5);
```

Wird jetzt zu:

```
1    ergebnis = 8*((5)+(5));
```

Jetzt wird 80 ausgegeben.

Fazit: Bei Makros mit Parametern müssen jeder Operand und der gesamte Ausdruck geklammert werden.

6.9.7 Der Fragezeichen-Operator

In Makros mit Parametern wird gerne der Fragezeichen-Operator benutzt, der wie eine kleine Verzweigung funktioniert. Er funktioniert so:

```
1    min_zahl = (a<b)?a:b;
```

Falls der Ausdruck vor dem Fragezeichen ungleich 0, also wahr ist (also `a<b` gilt), dann erhält der gesamte Ausdruck den Wert, der zwischen dem Fragezeichen und dem Doppelpunkt liegt. Andernfalls erhält er den Wert, der hinter dem Doppelpunkt liegt.

Die Anweisung ist also gleichbedeutend zu:

```
1    if(a<b)
2    {
3        min_zahl = a;
4    }
5    else
6    {
7        min_zahl = b;
8    }
```

Normalerweise ist der Fragezeichen-Operator eher unbeliebt, weil Programme mit ihm schwer lesbar sind. In einer Makro-Definition kann er aber sehr nützlich sein, weil er in einer Zeile eine ganze Verzweigung enthält.

Wenn man nun also das Minimum zweier Zahlen braucht, kann man das – statt mit einer Funktion – mit einem Makro realisieren:

minmakro.c

```
1 #define MIN(a,b) (((a)<(b))?(a):(b))
2     min_zahl=MIN(x,y);
```

Mit dem Ergebnis:

Terminal

```
schueler@debian964:~$ gcc -E minmakro.c | grep -v '^#'
   min_zahl=(((x)<(y))?(x):(y))
```

6.9.8 Bedingte Compilierung

6.9.8.1 Portabilität Portabilität heißt die einfache und schnelle Anpassbarkeit einer Software auf andere bzw. neue

a) Hardware

b) Betriebssysteme

c) weitere Software, mit der sie zusammenarbeiten soll

d) Länder, Sprachen und Kulturen (Internationalisierung und Lokalisierung)

6.9.8.2 Internationalisierung und Lokalisierung Internationalisierung[1] und Lokalisierung[2] einer Software sind zusammen ein Aspekt seiner Portabilität.

- Internationalisierung bedeutet, ein Programm so zu planen, zu entwickeln und zu implementieren (evtl. auch ein bestehendes Programm so umzubauen), dass es sich einfach und schnell an viele mögliche Länder, Sprachen und Kulturen anpassen lässt.
- Lokalisierung bedeutet, ein Programm anzupassen an ein bestimmtes Land, eine bestimmte Sprache und eine bestimmte Kultur.

Um Internationalisierung muss man sich für ein Programm einmal ausgiebig kümmern, mit Lokalisierung kann man beim gleichen Programm immer wieder zu tun haben.

6.9.8.3 Herstellen von Portabilität in C Es gibt mehrere Strategien, in C Portabilität zu erreichen. Eine Möglichkeit besteht darin, mehrere Versionen des Programmablaufs zur Verfügung zu stellen, von denen in einer bestimmten (Hardware-, Betriebssystem-, Sprach-) Umgebung nur eine verwendet wird.

Wie groß die unterschiedlichen Zweige sind (wird nur ein Text in einer anderen Sprache ausgegeben oder sind manche Funktionen ganz unterschiedlich in den verschiedenen Versionen?), hängt vom Einzelfall ab, wobei es meist ratsam ist, die Verzweigungen möglichst klein zu halten.

Nun muss noch nach Möglichkeiten gesucht werden, wie man die Versionen unterscheidet. Für kleinere und mittlere Programme sind da vor allem zu nennen:

a) Konstanten (Makros)

b) Bedingte Compilierung

c) Kommandozeilen-Parameter (`argc`, `argv`)

d) Umgebungsvariablen (durch die `getenv()`-Funktion von `<stdlib.h>`), z.B. `${LANG}`

e) Konfigurationsdateien

6.9.8.4 Bedingte Compilierung Mit dem Präprozessor kann man ein Programm in verschiedenen Versionen erstellen. Je nachdem, welche Version aktiviert ist, wird ein anderer Teil des Programmes übersetzt:

gruss.c

```
1 #define VERSION_HH 1
2 #define VERSION_HB 0
3 #if VERSION_HH
4     printf("Liebe␣Hamburger!\n");
5 #elif VERSION_HB
6     printf("Liebe␣Bremer!\n");
7 #else
8     printf("Liebe␣Leute!\n");
9 #endif
```

Durch die `#define`-Zeile wird das Makro `VERSION_HH` auf den Wert 1 gesetzt. In der `#if`-Zeile wird überprüft, ob das Makro auf einen Wert ungleich null gesetzt ist; wenn ja, wird der `#if`-Zweig an den Compiler weitergereicht, andernfalls der else-Zweig. Solche `#if`-Zeilen können an vielen weiteren Stellen des Quelltextes eingefügt werden.

Was ist der Unterschied zu einer normalen if-else-Konstruktion in C?

[1] abgekürzt auch *i18n* für *internationalisation*
[2] abgekürzt auch *l10n* für *localisation*

- Schon beim Compilieren des Programmes wird eine der beiden printf-Zeilen ausgewählt, die andere wird verworfen und kann vom Programm-Benutzer durch nichts wiedergeholt werden.
- Dadurch wird das Programm schneller, aber unflexibler.
- Dadurch können auf *einem* System (PC) Funktionen eingebunden werden, die auf dem anderen System (Mikrocontroller) fehlen, ohne dass es eine Fehlermeldung gibt.

Mit der Direktive `#ifdef` kann man prüfen, ob ein Makro überhaupt definiert wurde. Mit `#ifndef` prüft man, ob es noch nicht definiert wurde. Und mit `#undef` könnte man die Makro-Definition wieder aufheben.

6.9.8.5 include-Wächter Bei einem Programm mit mehreren Modulen kann es vorkommen, dass eine Headerdatei eine andere Headerdatei einbinden muss, z. B. , weil in der anderen Headerdatei wichtige Datentypen definiert werden:

ma.h

```
1 #include "mb.h"
2 void a(struct btyp);
```

mb.h

```
1 struct btyp{int b1, b2;};
2 void b(struct btyp);
```

Bindet das Hauptprogramm nun beide Headerdateien ein, dann sieht das so aus:

main.c

```
1 #include "ma.h"
2 #include "mb.h"
3 int main(void){}
```

Dies wird zu:

main.i

```
1 struct btyp{int b1, b2;};
2 void b(struct btyp);
3 void a(struct btyp);
4 struct btyp{int b1, b2;};
5 void b(struct btyp);
6 int main(void){}
```

Das gibt einen Fehler vom Compiler:

Terminal

```
schueler@debian964:~$ gcc main.c
In file included from main.c:2:0:
mb.h:1:8: error: redefinition of 'struct btyp'
```

Hier werden also der Datentyp `struct btyp` und die Funktion `b` doppelt definiert, was nach C-Standard nicht erlaubt ist. Abhilfe schafft der so genannte *Include-Wächter*, der durch geschickten Einsatz der bedingten Compilierung dafür sorgt, dass jeder Dateiinhalt maximal einmal eingebunden wird:

ma.h

```
1 #ifndef _MA_H_
2 #define _MA_H_
3 #include "mb.h"
4 void a(struct btyp);
5 #endif
```

mb.h

```
1 #ifndef _MB_H_
2 #define _MB_H_
3 struct btyp{int b1, b2;};
4 void b(struct btyp);
5 #endif
```

Jetzt wird beim Einbinden jeder Headerdatei ein Makro gesetzt, das so ähnlich heißt wie die Datei. Ist das Makro aber schon einmal gesetzt worden, wird der Dateiinhalt nicht noch einmal eingebunden.

6.9.9 Makros setzen im Compileraufruf

Ein Makro kann nicht nur im Quelltext, sondern auch durch den Compileraufruf gesetzt werden. Statt der folgenden Zeile

```
1 #define VERSION_HH 1
```

ist auch der Compileraufruf möglich:

Terminal

```
schueler@debian964:~$ gcc -DVERSION_HH=1 meinprogramm.c
```