Der C Präprozessor

von Christian Peter

31. März 2013

Inhaltsverzeichnis

1 Definition **3**

2 Funktionsweisen **3**

3 Benutzung des C-Präprozessors **4**
3.1 Der # Präfix 4
3.2 Ersetzung von Trigraphen 4
3.3 #include / #import 4
3.4 Makros / #define 5
3.4.1 Makros 5
3.4.2 Strings expandieren 6
3.4.3 Argumente als Konstrukt verknüpfen 6
3.4.4 #undef 7
3.5 #if, #elif , #ifdef & #else 7
3.5.1 #if 7
3.5.2 #elif 8
3.5.3 #else 8
3.5.4 #ifdef / #ifndef 9
3.5.5 Operationen der bedingten Ausführung 9
3.5.6 #ifdef / #ifndef als Schutz vor doppelter Einbindung (#include) 10

4 Vordefinierte Konstanten **12**
4.1 Prüfung der Systemarchitektur durch den Präprozessor 12
4.2 #error 13
4.3 #line 13

5 #pragma **13**
5.1 #pragma „gcc -D“ 14

6 Gefahren / Risiko **14**

7 Effizienz / Performance **15**

8 Bewertung **17**

1 Definition

„EinăPräprozessor ist einăComputerprogramm, welches Eingabedaten vorbereitet und zur weiteren Bearbeitung an ein anderes Programm weitergibt. Der Präprozessor wird häufig vonăCompilernăoderăInterpretern dazu verwendet einen Eingabetext zu konvertieren und das Ergebnis im eigentlichen Programm weiter zu verarbeiten.“ (Quelle: Wikipedia)

Da wir in dieser Ausarbeitung auf den C-Präprozessor eingehen, bedeutet das, dass dieser als Eingabedaten einen Quelltext erhält, den er nach bestimmten Syntaktischen Regeln bearbeitet und an den C-Compiler weitergibt. Wir werden dafür den Gnu C Compiler 'gcc' verwenden. Welche Regeln diesem Bearbeitungsprozess zu Grunde liegen und welche Möglichkeiten bzw. Risiken sich dadurch für den Programmierer ergeben, werden neben des Themas der Effizienz ebenfalls behandelt. Die Funktionen des Präprozessors sind jedoch den meisten C-Programmierern unbekannt und da diese in direkter Weise darauf abzielen eine vorallem zeitlich effiziente Programmierung zu gewährleisten, wird im Folgenden versucht einen möglichst umfangreichen Überblick über den Funktionsumfang des C-Präprozessors zu geben. Zum Verständnis wird ein Grundwissen in Programmiersprachen vorausgesetzt.

2 Funktionsweisen

Präprozessoren ersetzen im wesentlichen Text, bevor der Compiler den Quelltext übersetzt. Dies kann dem Compiler arbeit ersparen, z.B. indem unbenötigte Codeblöcke vor dem Compilen bereits aus dem Quelltext entfernt werden. Außerdem können sie dem Programmierer viel Arbeit ersparen, indem z.B. häufig benutzte Codeabschnitte in den Quelltext eingebunden werden, ohne, dass sie der Programmierer jedes mal in den Quelltext kopieren muss. Außerdem kann durch Ersetzung von komplexen Ausdrücken bereits die benötigte Rechenzeit des übersetzten Programmes reduziert werden. Dadurch kann der Präprozessor zur Unterstützung einer effizienten Programmierung und Übersichtlichkeit des Programmcodes genutzt werden. Den Präprozessoren liegt dafür eine von der Programmiersprache unabhängige Syntax zu Grunde. So könnten theoretisch auch simple Texte oder Quelltexte anderer Programmiersprachen als Eingabedaten für den C-Präprozessor dienen.

3 Benutzung des C-Präprozessors

3.1 Der # Präfix

Der C Compiler interpretiert alle Zeilen mit einem vorangehenden ‚#' als Anweisung. Diese Anweisungen, auch Direktiven genannt, müssen nicht mit einem ; abgeschlossen werden, da der Präprozessor den Befehl bis zum Ende der Zeile intepretiert. Soll ein Befehl über eine Zeile hinaus gehen, so kann am Ende der Zeile ein '\' eingefügt werden. Dieses veranlasst den Präprozessor die nächste Zeile so zu behandeln als würde sie direkt an die aktuelle Zeile anschließen. Dies führt sich so lange fort, bis eine Zeile kein '\' am Ende besitzt.

3.2 Ersetzung von Trigraphen

Eine mittlerweile kaum mehr benötigte Funktion des Prärpozessors ist die Ersetzung sogenannter Trigraphen (= Drei Symbole). Diese wurden früher benutzt, um für die Programmierung benötigte Sonderzeichen (siehe Tabelle) in allen Zeichensätzen darstellen zu können. Da heutzutage aber Ascii bzw. Unicode diese Sonderzeichen bereits darstellen können und alle modernen Tastaturen diese ebenfalls enthalten, spielt die Ersetzung von Trigraphen keine bedeutende Rolle mehr.

Die verwendeten Trigraphen(links) und ihre Entsprechenden Sonderzeichen (rechts):

??\	\
?¿	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~
??=	#

3.3 #include / #import

'#include <Dateiname>' Weist den Präprozessor an, den gesamten Inhalt der Datei an dieser Stelle in den Quelltext zu kopieren. Hierdurch wird gewährleistet, dass vorallem die häufig gebrauchten standardisierten Funktionen nicht in dem vom Programmierer aktuell bearbeiteten Quelltextes stehen, wodurch dieser wesentlich übersichtlicher gehalten wird.

'#include <stdio.h>' lädt somit den kompletten Inhalt der Datei stdio.h aus dem standard Includeverzeichnis '/usr/local/include' oder '/usr/include' und fügt ihn dem Quelltext hinzu. Werden Anführungszeichen verwendet, so sucht der Präprozessor stattdessen im aktuellen Arbeitsverzeichnis nach der Datei. Auch die inkludierte Datei kann nun wiederum Präprozessordirektiven enthalten. z.B. könnte ebenfalls ein #include Befehl in dieser Datei verwendet werden.

Dadurch ergibt sich das Problem, dass Dateien mehrfach an verschiedenen Stellen im Quelltext durch einen #include Befehl kopiert werden. Dies bedeutet einerseits eine unnötig Große ausführbare Datei am Ende, als auch mehr Arbeit für den Compiler, da mehr Zeilen zum Compilen entstehen. Außerdem kann es darüber hinaus zu Fehlern kommen, wenn bereits definierte Funktionen ein zweites mal definiert werden.

Um dies zu verhindern, gibt es die auf #include aufbauende Direktive '#import'. Diese hat die gleiche Funktionsweise wie '#include', nur jedoch mit dem Zusatz, dass mit '#import' geladene Dateien im gesamten Quelltext kein zweites mal geladen werden können. Die gcc Dokumentation warnt jedoch davor sich darauf 100% zu verlassen. Hier wird als alternative die Direktive '#pragma once' empfohlen. Liest der Präprozessor diese Direktive in einer durch #include oder #import inkludierten Datei, wird er diese Datei unter keinen Umständen ein zweites Mal lesen. '#pragma' direktiven sind jedoch nicht portabel und funktionieren nicht mit jedem Präprozessor, daher sollte es nur verwendet werden, wenn ausschließlich die gcc bzw. ein kompatibler Präprozessor verwendet wird.

Eine weitere Möglichkeit besteht in der Verwendung von '#ifdef'. Siehe Abschnitt '#if, #elif , #ifdev & #else'.

3.4 Makros / #define

Mit #define lassen sich Symbole und Konstanten definieren. '#define testsymbol' definiert z.B. das Symbol mit dem Namen testsymbol. Diesem Symbol lässt sich ein Wert zuweisen. Darüber hinaus kann überprüft werden, ob dieses Symbol existiert und welcher Wert ihm zugewiesen wurde. (siehe #ifdef)

Soll ein Symbol PI mit dem Wert 3.14159265 definiert werden, so lautet die Anweisung: '#define PI 3.14159265'.

'#define Konstante testkonstante' definiert äquivalent dazu eine Konstante mit dem Namen testkonstante. Diese Direktive ist mit dem Befehl 'const' in c vergleichbar. Zugewiesene Werte von Konstanten lassen sich nach der einmaligen Zuweisung nicht mehr ändern.

Symbole und Konstanten lassen sich im Gesamten Quelltext verwenden und gelten über Codeblöcke hinaus. Der Compiler bekommt am Ende lediglich den ersetzten Wert zur Weiterverarbeitung.

3.4.1 Makros

Es gibt die Möglichkeit sogenannte Makros zu erzeugen. Diese haben eine ähnliche Anwendung wie simple Funktionen bei C.

z.B. lässt sich durch folgendes Beispiel eine Summenfunktion realisieren:

```
#define summe(a,b) (a+b)
int a = 5:
int b = 1;
int result = summe(a,b);
```

Der Compiler würde folgende Zeile vom Präprozessor übergeben bekommen:

```
int result = (5+1);
```

Damit wäre das Ergebnis 6.

Diese Makros sind jedoch mit Vorsicht zu benutzen (siehe. 8. Gefahren / Risiko)

Es lassen sich sogar etwas komplexere Makros definieren die z.B. mit dem Compiler zusammen das Maximum berechnen:

#define maximum(a,b) (a>b ? a : b)
int a = 5;
int b = 1;
int result = maximum(a,b)

Der Compiler würde folgende Zeile erhalten:
int result = (5>1 ? 5 : 1)

Aus dieser Operation würde der Compiler nun das Maximum bestimmen, indem er prüft ob 5 größer als 1 ist und wenn ja, dann den ersten Wert nach dem Fragezeichen, ansonsten den nach dem Doppelpunkt verwendet.

3.4.2 Strings expandieren

Es ist auch möglich Argumente als String zu expandieren. So ist es möglich für Makros Argumente entgegenzunehmen und in sich selber einzusetzen.

folgendes Beispiel:

#define print(a) printf(#a "ist %d \n",a)
print(2+3);

ergibt nach Durchlauf des Präprozessors:

printf("2 + 3" "ist %d\n",2+3);

und nach Durchlauf des Compilers:

"2 + 3 ist 5"

3.4.3 Argumente als Konstrukt verknüpfen

#define var(a) konstrukt ## a

bewirkt, dass der Wert von a an die Zeichenkette "konstrukt" gehängt wird.
z.B. ergibt nun:
int konstruktA = 5;
printf("die Zahl ist %d \n",var(A));

die Präprozessorausgabe:
printf("Die Zahl ist %d \n", konstruktA);

und daraufhin nach Durchlauf des Compilers:

"Die Zahl ist 5"

3.4.4 #undef

Mit der Direktive #undef lassen sich nun Symbole, Konstanten und Makros wieder löschen. Einmal gelöscht, werden diese Symbole im nachfolgenden Quelltext vom Präprozessor ignoriert und an den Compiler weitergegeben, als wären sie nie definiert worden.

```
int a = 10;
#define a 1
printf("die Zahl ist %d \n", a);
#undef a
printf("die Zahl ist %d \n", a);
```

Würde folgende Ausgabe erzeugen:

"die Zahl ist 1"
"die Zahl ist 10"

Im ersten printf würde das Symbol a durch den Präprozessor durch eine 1 ersetzt werden. Im zweiten printf bekommt der Compiler den Buchstaben a geliefert, da das Symbol a davor gelöscht wurde. Dadurch benutzt der Compiler den zugewiesenen Wert 10 für die Variable a und gibt daher die 10 aus.

3.5 #if, #elif , #ifdef & #else

Der C-Präprozessor bietet außerdem die Möglichkeit zur Prüfung von Bedingung und damit zur bedingten Abarbeitung. So ist es z.B. möglich nur bestimmte Zeilen/Blöcke des ursprünglichen Quelltextes an den Compiler zu übergeben, oder verschiedene Präprozessoranweisungen auszuführen. Dadurch lässt sich eine bedingte Quelltextübersetzung gewährleisten. Z.B. ist es dadurch möglich, dass je nach Systemarchitektur oder Programmversion nur die jeweils benötigten Abschnitte des Quelltextes übersetzt werden.

3.5.1 #if

Durch die Direktive #if lassen sich Bedingungen prüfen. Geöffnet durch ein #if, muss ein Block immer durch ein #endif wieder geschlossen werden. Zwischen diesen beiden Zeilen, wird jede Zeile vom Präprozessor beachtet. Weitere Direktiven werden innerhalb dieses Blockes ebenfalls ausgeführt. Durch '#if defined symbol" lässt sich prüfen, ob ein Symbol definiert wurde.

Beispiel:

```
#define test
#if defined test
        "test ist definiert"
#endif
#if defined test2
        "test 2 ist definiert"
#endif
```

Hier wird unabhängig voneinander geprüft, ob das Symbol test bzw. das Symbol test2 definiert wurde.

Da nur das Symbol test definiert wurde, wird der Block von test2 nicht vom Präprozessor beachtet.

Die Ausgabe lautet daher: "test ist definiert"

3.5.2 #elif

Wie aus Programmiersprachen bekannt, können nach #if Bedingungen #else oder #elif (else if) Direktiven verwendet werden. Diese werden vom Präprozessor in dem Fall ausgeführt, in dem die Bedingung der #if Anweisung nicht zutrifft. Die Direktive #else verknüpft dabei keine neue Bedingung und wird daher in dem Fall bedingungslos ausgeführt, sollte die #if Bedingung falsch sein.

#define test3
#if defined test
"test ist definiert, über test2 und test3 können wir keine Aussage treffen"
#elif defined test2
"test2 ist definiert, test1 ist nicht definiert und über test3 können wir keine Aussage treffen"
#elif defined test3
"test3 ist definiert, test1 und test2 sind nicht definiert."
#endif

Das Ergebnis hierbei ist:
"test3 ist definiert, test1 und test2 sind nicht definiert."

Da #elif nur ausgeführt wird, sofern die vorherigen #elif/#if Bedingungen falsch sind, können wir sicher sein, dass test1 und test2 nicht definiert sind.

3.5.3 #else

Erweitert man das Beispiel nun um eine #else Direktive und verändert den Namen des Symbols, welches durch #define definiert wird :

#define test4
#if defined test
"test ist definiert, über test2 und test3 können wir keine Aussage treffen"
#elif defined test2
"test2 ist definiert, test1 ist nicht definiert und über test3 können wir keine Aussage treffen"
#elif defined test3
"test3 ist definiert, test1 und test2 sind nicht definiert."
#else
"weder test1, noch test2, noch test3 sind definiert."
#endif

dann ist die Ausgabe hierbei:
"weder test1, noch test2, noch test3 sind definiert."

3.5.4 #ifdef / #ifndef

Durch #ifdef lässt sich "#if defined" zusammenfassen. (#elifdef gibt es jedoch nicht)

Außerdem lässt sich durch #ifndef die negierte Bedingung prüfen.

#define test
#ifdef test
    "test ist definiert."
#ifdef test2
    "test2 ist definiert."
#ifndef test3
    "test3 ist nicht definiert."
#endif

erzeugt die Ausgabe:
    "test ist definiert.test3 ist nicht definiert"

'#ifdef test' ist genau wie '#ifndef test3' positiv. Darum springt der Präprozessor in beide Blöcke.

3.5.5 Operationen der bedingten Ausführung

Außerdem gibt es ebenfalls für die kombinierte Verwendung mehrerer Bedingungen in einer Direktive folgende Operationen:

A || B	A oder B
A && B	A und B
A==B	A gleich B
A>=B	A größer gleich B
A<=B	A kleiner gleich B

Anwenden lässt sich dies folgendermaßen
#define a 1
#define b 2
#define c 1
#if a == b
    "a ist gleich b"
#else
    "a ist ungleich b"
#endif

Die Ausgabe hierbei ist:
    "a ist ungleich b

Da das Symbol a den Wert 1 hat und das Symbol b den Wert 2, trifft die Bedingung 'a == b' nicht zu, wodurch der Präprozessor in den 'else' Block springt.

Außerdem lassen sich durch Klammerung mehrere Operationen verwenden:

#define a 1
#define b 2

```
#define c 1
#if (a == b) || (a == c)
        "a ist gleich b oder gleich c"
else
        "a ist ungleich b und ungleich c"
#endif
```

Dies ergibt:

        "a ist gleich b oder gleich c"

Außerdem lassen sich mehrere Bedingungen schachteln:

```
#define a 1
#define b 2
#define c 1
#if (a == b) || (a == c)
        #if (a==b)
                "a ist gleich b"
        #if (a==c)
                "a ist gleich c"
        #endif
#else
        "a ist ungleich b und ungleich c"
#endif
```

Dies führt dazu, dass der Präprozessor in den ersten #if Block springt, da entweder a ==b oder a == c wahr ist, danach die Bedingung a==b prüft, den folgenden Block überspringt, a==c prüft, den Inhalt des Blocks ausführt und dadurch

        "a ist gleich c" ausgibt.

3.5.6 #ifdef / #ifndef als Schutz vor doppelter Einbindung (#include)

Schreibt ein Programmierer einmal eine eigene Bibliothek, um seine Funktionen eventuell auch anderen Programmiereren zur Verfügung zu stellen oder zumindest seine Funktionen in anderen Programmen wiederverwenden zu können, so fängt er dabei nicht bei null an, sondern baut seine Funktionen wiederum auf anderen Bibliotheken auf. Bindet nun ein Programmierer diese Bibliothek und zusätzlich wiederum die Bibliothek auf der diese aufbaut ein, so kommt es zu doppelten Funktionsdefinitionen im Quelltext, welche zu vielfältigen Fehlern von Variablenüberschreibung bis hin zu kritischen Programmabbrüchen führen können. Daher ist es wichtig, dass durch #include eingebundene Dateien nicht ein zweites mal eingebunden werden. Da es aber in manchen seltenen Fällen sinnvoll sein kann, führt es nicht bei jedem Compiler direkt zum Abbruch der Kompilierung. In Abschnitt 3.3. wurde bereits die Direktive #import erwähnt. Jedoch ist #import laut gcc Handbuch nicht zu 100% zuverlässig. Darum ist es generell sinnvoll, wenn der Programmierer beim Einbinden von Bibliotheken

Symbole setzt, welche dann durch #ifdef bzw. #ifndef ausgewertet werden können. Dadurch kann eine doppelte Einbindung theoretisch verhindert werden, sofern der Programmierer alle Bibliotheken und Includebefehle mit genügend solcher Sicherungen versieht.

Beispiel:

```
#ifndef a
        #include <stdio.h>;
        #define a
#endif
...
...
#ifndef a
        #include <stdio.h>;
        #define a
#endif
```

In diesem Beispiel würde lediglich die erste #include Direktive ausgeführt werden, da bei der zweiten Prüfung des Symbols a dieses bereits definiert worden ist. Dies funktioniert auch Dateiübergreifend. z.B. wäre auch folgendes denkbar:

Inhalt Datei "meinQuelltext.c":

```
#ifndef stdio
        #include <stdio.h>
        #define stdio
#endif
#ifndef myLibrary
        #include <myLibrary.h>;
        #define myLibrary
#endif
```

Inhalt Datei "myLibrary.h":

```
#ifndef stdio
        #include <stdio.h>
        #define stdio
#endif
```

Dies hätte zur Folge, dass die Datei stdio.h innerhalb der myLibrary.h nicht mehr geladen wird, da sie aus meinQuelltext.c bereits geladen wurde.

4 Vordefinierte Konstanten

LINE	beinhaltet die aktuelle Zeile
FILE	beinhaltet die aktuelle Datei
DATE	beinhaltet das aktuelle Datum
TIME	beinhaltet die aktuelle Zeit
STDC	ist gesetzt falls "ANSI standard C" benutzt wird
UNIX	gesetzt falls Zielsystem Unix ist
_MSDOS	gesetzt falls Zielsystem MsDos ist
_WINDOWS	falls das Zielsystem Windows ist

Es gibt mehrere Konstanten, die von dem Präprozessor verwendet werden können. _File_ z.B. enthält den aktuellen Dateinamen und kann ausgegeben werden, um die Fehlersuche zu erleichtern. _DATE_ und _TIME_ können verwendet werden, wenn der Präprozessor die Zeit der Übersetzung des Programmes festhalten soll. Sinnvoll für z.B. eine Zeile im fertigen Programm die folgendes Enthält: "Das Programm wurde am 31.03.2013 um 23:59 kompiliert".

4.1 Prüfung der Systemarchitektur durch den Präprozessor

Ein sehr praktisches Anwendungsbeispiel, welches die Größe des übersetzten Programmes erheblich reduzieren kann, ist die Prüfung um welches Betriebssystem oder um welche Architektur es sich handelt. Üblich unter Linuxsystemen ist, dass die Programme auf dem jeweiligen System erst kompiliert werden, bevor sie ausgeführt werden. Da ein Quelltext häufig Routinen, Funktionen u.a. für mehrere Plattformen enthält ist es sinnvoll bereits vor der Kompilierung durch den Präprozessor nicht benötigte Teile des Quelltextes zu entfernen bzw. nur die benötigten Teile zu behalten.

Hierfür stellt der gcc verschiedene Konstanten bereit.

Ein Beispiel dazu wäre:

```
#ifdef Unix
        lade Unix libraries
        setze Umgebungsvariablen für Unix
#else if defined _MSDOS
        lade MSDOS libraries
        setze Umgebungsvariablen für MsDos
#else if defined _WINDOWS
        lade WINDOWS libraries
        setze Umgebungsvariablen für Windows
#endif
```

Dies hat den Vorteil, dass für bestimmte Architekturen und Betriebssysteme spezifische Libraries geladen bzw. Routinen ausgeführt werden können. Danach folgt dann der Programmquelltext, welcher theoretisch auf allen Plattformen lauffähig ist. Falls im späteren Verlauf des Quelltextes noch Betriebssystem-

spezifische Routinen vorkommen, so können diese auf gleiche Weise separiert werden.

4.2 #error

Durch die Direktive #error kann der Kompiliervorgang mit einer Fehlermeldung abgebrochen werden. Dies kann z.B. sinnvoll sein, wenn das Programm für verschiedene Architekturen, Betriebssysteme oder ähnliches nicht geeignet ist oder eventuell Parameter fehlen.

Beispiel:

```
#define importantdefine
#ifndef importantdefine
        #error "importantdefine" ist nicht definiert.
#ifndef veryimportantdefine
        #error "veryimportantdefine" ist nicht definiert.
#endif
```

Dies führt zu der Ausgabe:

```
Datei.c:5:3: Fehler: #error "veryimportantdefine" ist nicht definiert
```

4.3 #line

Ebenfalls sehr hilfreich bei der Fehlersuche kann #line sein. Durch diese Direktive kann man den Compiler dazu bringen ab der Zeile in der #line steht ab dem definierten Wert weiter zu zählen oder auch den aktuellen Wert auszulesen.

Das folgende Beispiel enthält zwei Fehler, welche der C-Compiler bemerkt:

```
a = c;
#line 5992
int a = b;
```

Die Ausgabe des C-Compilers ist:

Datei.c:1:6 Fehler: >>c<< nicht deklariert. (erste Benutzung in dieser Funktion)

Datei.c.5992:11: Fehler: >>b<< nicht deklariert. (erste Benutzung in dieser Funktion)

Obwohl wir also bloß 3 Zeilen haben, gibt der Compiler Zeile 5992 als fehlerhafte Zeile an. Dies kann vorallem dann sinnvoll sein, wenn durch den Präprozessor Zeilen und ganze Blöcke durch Bedingungen wegfallen, da ansonsten der Compiler immer dem ursprünglichen Quelltext fehlerhafte Zeilen ausgeben würde. Der 'gcc' kümmert sich jedoch von allein um die richtige Nummerierung des Zeilen im Quelltext, selbst wenn Blöcke durch den Präprozessor herausgenommen werden.

5 #pragma

#Pragma direktiven erweitern den Funktionsumfang einiger Präprozessorimplementationen und kann damit Direktiven einleiten, die Anweisungen für spezielle

Compiler beinhalten. gcc z.B. kann mit ‚#pragma once' gewährleisten, dass Dateien nur einmal geladen werden. Wird in einer Datei #pragma once benutzt, so kann diese nur einmal von einem Programm geladen werden. Dies funktioniert in allen Fällen. „No matter what" (Quelle gcc Handbuch)

Führt man gcc z.B. mit dem flag –fopenmp aus, so stehen eine Vielzahl an „#pragma openmp" Direktiven zur Paralellisierung von Programmen zur Verfügung. Diese können - müssen aber nicht - mit anderen Compilern funktionieren.

5.1 #pragma „gcc -D"

äquivalent zu „#define konstante" im Quelltext, lassen sich durch „gcc -D konstante" Konstanten für den Präprozessor des zu kompilierenden Programmes definieren.

„gcc -D konstante=wert" ist damit das Äquivalent zu der Direktive „#define konstante wert". Hierdurch lassen sich sehr effektiv verschiedene Programmversionen erzeugen. Ein sehr häufig gebrauchter Fall wäre z.B. "gcc -D x64", welches dann im Quelltext durch:

#ifdef x64
...
...
...
#endif

ausgewertet werden würde. Hier lässt sich eine Menge an sinnvollen Möglichkeiten finden. So wäre es auch denkbar Programme mit verschiedenem Funktionsumfang, oder Programme für verschiedene Sprachen, Länder, Kontinente zu generieren. Durch die einfache spezifische/bedingte Kompilierung mittels eines Flags macht es die Anwendbarkeit insbesondere für den Benutzer, aber auch für den Programmierer viel komfortabler.

6 Gefahren / Risiko

Bei der Anwendung von Präprozessordirektiven ist es wichtig, dass das Grundsätzliche Prinzip verinnerlicht wird. Der Präprozessor ersetzt in erster Linie Text. So lassen sich viele Fehler von unerfahrenen Programiereren finden, welche Präprozessordirektiven und Compileranweisungen zusammen benutzen und am Ende falsche Ergebnisse bekommen.

Ein Paradebeispiel für eine falsche Anwendung wäre hierbei folgendes Beispiel:

```
#define summe(a,b) a+b
int a = 5;
int b = 3;
result = 5* summe(a,b);
```

Das gewollte Ergebnis hierbei wäre 40, da wir instinktiv annehmen, dass erst die Funktion summe(5,3) zu 8 ausgewertet und dann mit 5 zu 40 multipliziert wird. Hier passiert jedoch etwas anderes. Der Präprozessor ersetzt lediglich den

Term 'summe(a,b)' durch den Term 'a+b'. Dadurch bekommt der Compiler die Zeile

result = 5 * 5 + 3;

Das tatsächliche Ergebnis ist hier also 28. Solche Fehler lassen sich schwer finden, da im Quelltext nicht immer direkt klar ist, ob es sich um eine Funktion der Programmiersprache handelt, oder um ein Symbol, welches vom Präprozessor ersetzt wird. Hierbei ist es ratsam den Quelltext mit 'gcc -E dateiname.c' vom Präprozessor bearbeiten zu lassen. Das angezeigte Ergebnis ist dann das was an den compiler übergeben werden würde. Hier kann man nun leichter sehen, dass die Zeile 'result = 5 * 5 + 3' keinen Sinn macht.

Generell ist es wichtig, dass bei Präprozessorersetzungen um die Symbole Klammern gesetzt werden, da dies der instinktiven Vorstellung einer Funktion entspricht.

Richtig wäre also:

```
#define summe(a,b) (a+b)
int a = 5;
int b = 3;
result = 5* summe(a,b);
```

Hierbei wird erst summe(a,b) ausgewertet, da der Präprozessor summe(a,b) gegen (a+b) ersetzt. Dies führt zu der Zeile: 'result = 5 * (5+3)' und damit zum gewollten Ergebnis.

Es ist z.B. aber nicht möglich eine Endlosschleife zu konstruieren. Daher würde z.B. folgendes funktionieren:

```
#define a (a + b)
#define b (b + a)
#error ab
```

erwarten würde man nun, dass der Präprozessor rekursiv die Symbole ersetzen würde. Für jedes gefundene a würde er (a+b) einsetzen und daraufhin das b wieder durch ein (b+a) ersetzen usw.

Jedoch läuft der Präprozessor jede Zeile exakt nur einmal durch. Das Ergebnis wäre also eine Fehlermeldung mit dem Inhalt (a+b)(b+(a+b))

7 Effizienz / Performance

Einige Vorteile von Prärpozessordirektiven lassen sich direkt logisch errechnen. Wird die Hälfte des Quelltextes durch den Präprozessor weggeschnitten, benötigt der Compiler lediglich ca. 50% der Rechenzeit und das entstandene Programm benötigt bloß ca. 50% der ursprünglichen Größe. Ein paar andere Performanceunterschiede werden nun im Folgenden aufgezeigt.

Vergleich der c 'const' Funktion im Gegensatz zu #define:

100 mio. Durchläufe einer Multiplikation zweier Werte die einmal durch #define und einmal durch const definiert wurden, ergab folgende Ergebnisse:

#define : 0.193s

const: 0.200s

Wir sehen also einen Laufzeitunterschied von 3.5% beim Durchlauf des Programmes.

Die benötigte Kompilierungszeit (Präprozessoroptimierung bereits vorher ausgeführt) beträgt:

#define: 0.36s

const: 0.38s

Komplipilierungszeit inklusive Präprozessoroptimierung:

#define: 0.42s

const: 0.38s

Bei der Kompilierung ist die Variante mit Const leicht im Vorteil und wird ca. 12% schneller kompiliert.

Als nächstes vergleichen wir den Unterschied zwischen einer Summenfunktion von C und einer bloßen Ersetzung durch ein Präprozessorsymbol.

Quelltext mit Summenfunktion:

```
int summe (a,b) {
        return (a+b);
}
int main () {
        int i; int b;
        for(i=0;i<1000000;i++) {
                b = summe(10,15);
        }
}
```

Kompilierung: 0.40s

Laufzeit: 0.240s

Quelltext mit Ersetzung durch Präprozessor:

```
#define summe(a,b) (a+b)
int main () {
        int i; int b;
        for(i=0;i<100000000;i++) {
                b = summe(10,15);
        }
}
```

Kompilierung (nur Compiler): 0.35s

Kompilierung (kombiniert mit Präprozessor): 0.40s

Laufzeit: 0.192s

Wir sehen, dass bei der Version mit #define die benötigte Kompilierungszeit des Compilers, wenn man die Laufzeit des Präprozessors abzieht, ca. 12,5% kürzer ist.

Die Laufzeit des Programmes am Ende ist sogar 20% kürzer.

8 Bewertung

Es wird ersichtlich, dass sich simple Ersetzungen durch den Präprozessor mit Leistungssteigerung und Laufzeitverkürzung des Programmes bemerkbar machen. Werden Präprozessordirektiven richtig eingesetzt, können sie erheblichen Vorteil bieten. Trotzdem erhöhen sich durch den Einsatz des Präprozessor auch die möglichen Fehlerquellen. Ob der entstehende Quelltext besser lesbar ist, ist am Ende eine Frages des subjektiven Geschmacks des Programmierers. Wichtig ist nur in Erinnerung zu behalten, dass sich der aktive Einsatz des Präprozessors lohnen kann, sofern viele simple Funktionen wie Summenbildung oder Multiplikation im Quelltext bestehen, die nur der Übersichtlichkeit halber existieren und nicht aufgrund ihrer Komplexität. Komplexe Funktionen sind kaum durch simple Ersetzungen ersetzbar.

Literatur

[1] http://de.wikibooks.org/wiki/c-programmierung:praeprozessor.

[2] http://de.wikipedia.org/wiki/c-praeprozessor.

[3] Microsoft. c/c++ preprocessor - http://msdn.microsoft.com/en-us/library/y4skk93w(v=vs.80).aspx.

[4] GNU Project. Gcc preprocessor doc http://gcc.gnu.org/onlinedocs/cpp/.

[5] Wikipedia. http://de.wikipedia.org/wiki/praeprozessor.