# C - PRÄPROZESSOR

Seminar effiziente C Programmierung WS 2012/13

Von Christian Peter

# Themen

- Was sind Präprozessoren?
  - Beispiele für Präprozessoren
  - Funktionsweisen
- Der C - Präprozessor
  - Der # Präfix
    - #include / #import
  - Makros / #define
  - #if, #elif & #else
  - #error
  - #line
  - Vordefinierte Konstanten
  - #pragma
  - „gcc -D“

# Was sind Präprozessoren?

- DEFINITION:

  „Ein **Präprozessor** ist ein Computerprogramm, das Eingabedaten vorbereitet und zur weiteren Bearbeitung an ein anderes Programm weitergibt. Der Präprozessor wird häufig von Compilern oder Interpretern dazu verwendet einen Eingabetext zu konvertieren und das Ergebnis im eigentlichen Programm weiter zu verarbeiten.“ (Quelle: Wikipedia)
- Grenzen sich von „Precompilern“ ab, welche Berechnungen und vor allem Optimierungen auf dem Quellcode durchführen
- Ist klar vom Compiler zu trennen.
  - Wird bei dem GCC („GNU Compiler Collection“) vor dem eigentlichen Compiler ausgeführt
  - Übergibt den modifizierten Quelltext an den Compiler

# Beispiele für Präprozessoren

- PHP als Präprozessor für HTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="de">
 <head>
   <title>Der PHP Präprozessor</title>
 </head>
 <body>
<h1>der aktuelle Tag</h1>
<p><?php echo "Wir haben heute den ".date('d.m.y',time()); ?></p>
 </body>
</html>

----------------------------------------

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="de">
 <head>
   <title>Der PHP Präprozessor</title>
 </head>
 <body>
<h1>der aktuelle Tag</h1>
<p>Wir haben heute den 22.11.2012</p>
 </body>
</html>
```

# Beispiele für Präprozessoren

- CPP als Präprozessor für C

```
#include <stdio.h>
#define CONTENT "Hallo ich bin der content"
main() {


printf(CONTENT);

}

gcc -E preprocessor.c
# 940 "/usr/include/stdio.h" 3 4

# 2 "preprocessor.c" 2

main() {


printf("Hallo ich bin der content");

}
```

# Funktionsweisen

- Präprozessoren ersetzen im wesentlichen Text bevor der Compiler den Quelltext übersetzt.
- Kann dem Compiler Arbeit ersparen
  - Unbenötigte Codezeilen/-Blöcke können vom Präprozessor einfach vor Übergabe an Compiler entfernt werden (#if/#else)
- Der C-Präprozessor hat eine eigene Syntax
  - kann auch für andere Programmiersprachen und sogar Texte verwendet werden.
  - Ist nicht abhängig von der C-Syntax

# DER # PRÄFIX

- Der C Compiler interpretiert alle Zeilen mit einem vorangehenden ‚#‘ als Anweisung
  - Anweisungen werden Direktiven genannt
  - Die Zeilen müssen nicht mit einem ; abgeschlossen werden
  - Pro Zeile darf bloß eine Anweisung stehen
    - Soll ein Befehl sich über mehrere Zeilen erstrecken, so muss am Ende jeder Zeile ein „\“ stehen
    - Die letzte Zeile darf nicht mit einem „\“ enden!
- Beispiel:

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

# Funktionen des C-Präprozessors #include & #import

- #include <filename>
  - Weist den Präprozessor an, den gesamten Inhalt der Datei in den Quelltext zu kopieren
- #import <filename>
  - Gleiche Funktionsweise wie #include
  - Mit dem Zusatz, dass mit #import geladene Dateien im gesamten Quelltext kein zweites mal geladen werden können. (In der Theorie. Die gcc dokumentation warnt davor sich darauf zu verlassen)
- bedingte Ersetzung mit der Präprozessoranweisung #ifdef kann Schutz davor bieten, nach Durchlauf des Präprozessors Funktionsdefinitionen mehrfach im Quelltext zu haben
  - Siehe Kapitel #if, #else, #elif

# Funktionen des C-Präprozessors #include & #import

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

- Lädt den Inhalt der Datei stdio.h aus dem Standard-includeverzeichnis und fügt ihn der Datei hinzu
  - Normalerweise ist das entweder
    - /usr/local/include oder /usr/include
    - Wird statt <filename> „filename“ benutzt, so sucht der Präprozessor im aktuellen Arbeitsverzeichnis nach der Datei
  - Die Datei kann wiederum Präprozessordirektiven enthalten, z.B. einen weiteren #include / #import Befehl

# Funktionen des C-Präprozessors
# Ersetzung von Trigraphen

- Heute kaum noch nötig, da „ASCII“ bzw. „UNICODE“ folgende Sonderzeichen bereits unterstützen
  - Alle modernen Tastaturlayouts stellen die Sonderzeichen daher ebenfalls zur Verfügung

| Trigraph | ASCII |
|---|---|
| ??/ | \ |
| ??‘ | ^ |
| ??( | [ |
| ??) | ] |
| ??! | \| |
| ??< | { |
| ??> | } |
| ??- | ~ |
| ??= | # |

Da C alle Zeichen des ASCII Zeichensatzes benutzt, früher aber nicht jede Tastatur den vollen ASCII Zeichensatz bedienen konnte, hat der C-Präprozessor die Aufgabe übernommen die sogenannten Trigraphen (Tri = 3 Stellen) durch die entsprechenden ASCII Zeichen zu ersetzen

# Funktionen des C-Präprozessors #DEFINE

- #define definiert ein Symbol

```
#define SYMBOL
```

- oder eine Konstante

```
#define KONSTANTE (Wert)
```

- Ähnlicher Effekt wie der Befehl „const“ in C
  - Dieser wird jedoch erst vom Compiler bei der Übersetzung verarbeitet
- Symbole/Konstanten sind im Quelltext beliebig einsetzbar
  - Gelten auch über Blöcke hinaus
- Der Compiler sieht lediglich den ersetzten Wert
- Bei gcc kann man durch gcc -E <file> erreichen, dass der Quelltext nach Durchlauf des Präprozessors angezeigt wird. Dies wird auf den nachfolgenden Folien verwendet.

# Funktionen des C-Präprozessors #DEFINE

```
#include <stdio.h>
#define PI  3.14159265

void main () {
printf(PI);

}
```

Wird zu

```
# 940 "/usr/include/stdio.h" 3 4

# 2 "test.c" 2



void main () {
printf(3.14159265);

}
```

Gleicher Effekt als würde der Compiler im Quelltext direkt die Zahl „3.14159265“ lesen

# Funktionen des C-Präprozessors #DEFINE

- Außerdem gibt es die Möglichkeit ein Makro zu erzeugen

```
#include <stdio.h>
#define PI  3.14159265
#define summe(a,b) (a+b)

void main () {
int a,b,result;
a=5;
b=3;
result = summe(a,b);

}
```

Wird zu

```
# 940 "/usr/include/stdio.h" 3 4

# 2 "test.c" 2



void main () {
int a,b,result;
a=5;
b=3;
result = (a+b);

}
```

# Funktionen des C-Präprozessors #DEFINE

- Auch etwas komplexere Anwendungen sind möglich

```
#include <stdio.h>
#define maximum(a,b) ( a>b ? a : b )

void main () {
int a,b,result;
a=5;
b=3;
result = maximum(a,b);

}
```

Wird zu

```
# 940 "/usr/include/stdio.h" 3 4

# 2 "test.c" 2


void main () {
int a,b,result;
a=5;
b=3;
result = ( a>b ? a : b );

}
```

# Funktionen des C-Präprozessors #DEFINE

Hierdurch lassen sich auch schnell Fehler verursachen:

```
#include <stdio.h>
#define summe(a,b) a+b

void main () {
int a,b,result;
a=5;
b=3;
result = 10*summe(a,b);

}
```

Wäre summe() eine Funktion in C, so würde erst die Summe von a+b errechnet. Der Präprozessor ersetzt aber bloß zu:

```
void main () {
int a,b,result;
a=5;
b=3;
result = 10*a+b;

}
```

Als Resultat erhält man 10a+b und nicht wie ursprünglich erwartet 10*(a+b)

# Funktionen des C-Präprozessors #DEFINE

- Man kann auch Argumente als String expandieren:

```
#define print(a)   printf( #a " ist  %d\n", a )

main()
{
  print( 2+3 );
}
```

Wird zu

```
# 1 "test.c"
# 1 "<Kommandozeile>"
# 1 "test.c"


main()
{
  printf( "2+3" " ist  %d\n", 2+3 );
}
```

# Funktionen des C-Präprozessors #DEFINE

- Außerdem ist es möglich Argumente mit einem Konstrukt zu verknüpfen

```
#define var(a) konstrukt ## a

main()
{
  int
    konstruktA,
    konstruktB;

  konstruktA  = 1;
  konstruktB = 25;
  printf( "%d, %d\n", var(A), var(B) );
}
```

Wird zu

```
main()
{
  int
    konstruktA,
    konstruktB;

  konstruktA = 1;
  konstruktB = 25;
  printf( "%d, %d\n", konstruktA, konstruktB );
}
```

# Funktionen des C-Präprozessors #DEFINE

```
#define var(a) konstrukt ## a

main()
{
  int
    konstruktA,
    konstruktB;

  konstruktA  = 1;
  konstruktB = 25;
#undef var
  printf( "%d, %d\n", var(A), var(B) );
}
```

#undef löscht eine vergebene Definition wieder

Wird zu

```
main()
{
  int
    konstruktA,
    konstruktB;

  konstruktA = 1;
  konstruktB = 25;

  printf( "%d, %d\n", var(A), var(B) );
}
```

# Funktionen des C-Präprozessors #DEFINE

- Es ist sogar möglich Arrays durch Konstanten zu initialisieren

```
#define NUMBERS 1,2,3,4,5

int x[] ={  NUMBERS }
```

Wird zu

```
int x[] ={ 1,2,3,4,5 }
```

# Funktionen des C-Präprozessors #if, #else & #elif

- Der C Präprozessor bietet außerdem die Möglichkeit durch Prüfung von Bedingungen:
  - Zeilen/Blöcke nicht dem Compiler zu übergeben
  - Verschiedene Präprozessoranweisungen auszuführen
  - Feststellen ob Symbole/Konstanten definiert sind
    - Und Fehlermeldungen dazu ausgeben

# Funktionen des C-Präprozessors
# #if, #else & #elif

- Durch #if lassen sich Bedingungen prüfen
  - Ein #if-Block muss immer durch ein #endif geschlossen werden

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

- #if defined ‚symbol‘ prüft, ob das symbol definiert wurde.

Die Ausgabe des Präprozessors lautet:

```
        "test" ist definiert.
```

# Funktionen des C-Präprozessors #if, #else & #elif

- #else und #elif können nach #if-Blöcken benutzt werden:

```
#define test3
#if defined test
        "test" ist als einziges definiert.
#elif defined test2
        "test2" ist als einziges definiert.
#elif defined test3
        "test3" ist als einziges definiert.
#else defined test4
        "test4" ist als einziges definiert.
#endif
```

Die Ausführung ergibt:

```
        "test3" ist als einziges definiert.
```

# Funktionen des C-Präprozessors #if, #else & #elif

- #if defined lässt sich auch zu #ifdev zusammenfassen. Gilt nicht für #elif und #else

```
#define test
#ifdef test
        "test" ist definiert.
#elif test2
        "test2" ist definiert.
#endif
```

Der Präprozessor gibt folgendes aus:

```
        "test" ist definiert.
```

# Funktionen des C-Präprozessors #if, #else & #elif

- Außerdem lässt sich mit #ifndef explizit prüfen ob ein Wert NICHT definiert ist. Gilt ebenfalls nicht für #elif und #else

```
#define test3
#ifndef test
        "test" ist undefiniert.
#endif
```

Dies ergibt als Ausgabe

```
        "test" ist undefiniert.
```

# Funktionen des C-Präprozessors #if, #else & #elif

- Für die bedingte Ausführung sind ebenfalls Operationen verfügbar
  - A || B gilt für A oder B
  - A && B gilt als A und B
  - A==B gilt als A gleich B
  - A>=B gilt als A größer gleich B
  - Je nach Präprozessorversion können unterschiedliche Befehlssätze verwendet werden.

# Funktionen des C-Präprozessors #if, #else & #elif

- 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 dazu ist:

```
        "a ist ungleich b"
```

# Funktionen des C-Präprozessors #if, #else & #elif

- 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"
```

# Funktionen des C-Präprozessors #if, #else & #elif

- Es ist ebenfalls möglich mehrere Bedingungen zu Schachteln

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

Ergibt

```
"a ist gleich c"
```

# Funktionen des C-Präprozessors #if, #else & #elif

- Ein sehr praktisches Anwendungsbeispiel für Bedingungen

```
#define Windowsx64 1
#define Windowsx86 2
#define Betriebssystem Windowsx86
#if Betriebssystem == Windowsx86
        Implementation für Windows x86
        ...
        ...
#elif Betriebssystem == Windowsx64
        Implementation für Windows x64
        ...
        ...
#endif
```

Resultiert in:

```
Implementation für Windows x86
...
...
```

Kann auch dafür benutzt werden um je nach Betriebssystem verschiedene Bibliotheken zu laden

# Funktionen des C-Präprozessors #if, #else & #elif

- #ifdev bzw. #ifndef können Schutz davor bieten, dass Dateien doppelt an verschiedenen Stellen im Quelltext eingebunden werden und dadurch Fehler entstehen

```
#ifndef a
        include <stdio.h>
        inhalt der datei
        ...
        #define a
#endif
        Quelltext
        ...
        ...
#ifndef a
        include <stdio.h>
        inhalt der datei
        ...
        #define a
#endif
```

Gibt folgendes aus

```
include <stdio.h>
inhalt der datei
...

Quelltext
...
...
```

# Funktionen des C-Präprozessors #error

- Durch die Direktive #error kann der Kompiliervorgang mit einer Fehlermeldung abgebrochen werden

```
#define importantdefine
main () {
        #ifndef importantdefine
        #error "importantdefine" ist nicht definiert.
        #endif
        #ifndef veryimportantdefine
        #error "veryimportantdefine" ist nicht definiert.
        #endif
}
```

gcc -E gibt hier jetzt bloß folgendes aus

```
main () {

}
```

Jedoch bricht der Kompiliervorgang „gcc test.c“ ab:

```
test.c:7:3: Fehler: #error "veryimportantdefine" ist nicht definiert.
```

# Funktionen des C-Präprozessors #line

- Ebenfalls sehr hilfreich bei der Fehlersuche kann #line sein.
  - Durch #line kann man den Compiler dazu bringen ab der Zeile in der #line steht ab dem definierten Wert weiter zu zählen.

```
main () {
        a = c;
        #line 5992
        int a =  b;

}
```

  - Ergibt beim Compilen den folgenden Fehler

```
test.c:3:6: Fehler: »c« nicht deklariert (erste Benutzung in dieser Funktion)
test.c:5992:11: Fehler: »b« nicht deklariert (erste Benutzung in dieser Funktion
```

  - Obwohl unser Programm nur aus 5 Zeilen besteht, findet der Compiler somit einen Fehler in Zeile 5992
    - Kann hilfreich bei der Fehlersuche sein. Insbesondere wenn durch den Präprozessor Codeblöcke entfernt wurden und der Compiler die ursprünglichen Zeilenangaben nicht mehr verwendet. (bei gcc nicht der Fall, da cpp direkt enthalten)

# Vordefinierte Konstanten

- Es gibt bereits vom Präprozessor 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.

- Außerdem können je nach System noch folgende Konstanten definiert sein:
  - Unix Falls das Zielsystem Unix ist
  - __MSDOS Falls das Zielsystem MsDos ist
  - _WINDOWS Falls das Zielsystem Windows ist

# Funktionen des C-Präprozessors #pragma

- #pragma leitet Direktiven ein, 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)
- Führt man gcc z.B. mit dem flag –fopenmp aus, so stehen eine Vielzah lan „#pragma openmp“ Direktiven zur Paralellisierung von Programmen zur Verfügung.
  - Diese können - müssen aber nicht - mit anderen Compilern funktionieren.

# Funktionen des C-Präprozessors gcc -D

- Durch „gcc -D symbol“ lassen sich Konstanten für den Präprozessor des zu kompilierenden Programmes definieren
  - symbol wird dabei als Konstante definiert
    - Äquivalent zu „#define symbol“ im Quelltext
  - „gcc -D symbol=wert“
    - Äquivalent zu „#define symbol wert“ im Quelltext