

# **Präprozessor**

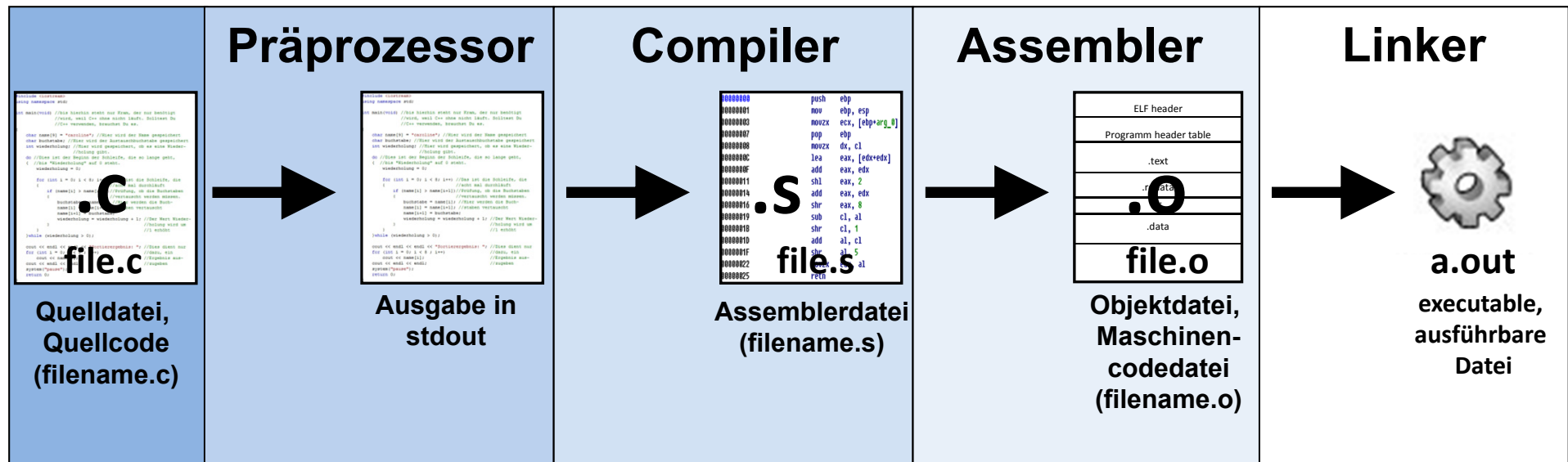
**Einführung in die Programmierung**

**Michael Felderer (QE)**

**Institut für Informatik, Universität Innsbruck**

# Übersetzungsvorgang beim gcc (Wiederholung)

- Wie läuft der Übersetzungsvorgang ab?



Wichtige Suffixe:

file.c	C Quelldatei muss Präprozessorphase durchlaufen
file.i	C Quelldatei darf nicht die Präprozessorphase durchlaufen
file.s	Assemblerdatei
file.o	Objektdatei
a.out	Ausführbare Datei

# Allgemein

- Beim Aufruf eines C-Compilers wird vor den eigentlichen Compiler-Läufen der Präprozessor gestartet.
- Die wichtigsten Aufgaben eines Präprozessors:
  - Textformatierung (Kommentare entfernen, Whitespaces entfernen etc.)
  - Einfügen von Dateien
  - Ersetzen von Codeteilen (`#define`)
  - Bedingte Übersetzung
- Die allgemeine Syntax für einen Präprozessor-Befehl (Präprozessor-Direktive) lautet  
`#Direktive      Text`
- Präprozessor-Befehl endet mit dem Zeilenende und **nicht** mit einem Semikolon!

# Einfügen von Dateien (1)

- Mit Hilfe der include-Direktive

```
#include <filename>
#include "filename"
```
- Präprozessor legt eine temporäre Datei an.
  - Entfernt Direktive und fügt den Quelltext der Datei an dieser Stelle ein.
  - Wenn die Header-Datei auch eine include-Direktive enthält, wird diese auch aufgelöst usw.
  - filename kann auch ein relativer oder absoluter Pfad sein.
  - Auflösung von filename ist implementierungsabhängig.
- 2 Varianten für die Auflösung (typisches Verhalten)
  - <> - Suche startet in den Systemverzeichnissen (z.B. unter Linux /usr/include)
  - " " - Suche startet im aktuellen (oder gegebenen) Arbeitsverzeichnis und wenn nicht erfolgreich, dann in den Systemverzeichnissen.

## Einfügen von Dateien (2)

- Was steht in solchen Header Dateien?
  - Konstanten
  - Deklarationen von Typen und Funktionen
  - Makros
  - weitere Include-Direktiven etc.
- Man sollte nicht weitere C-Dateien einbinden!

# Symbolische Konstanten und Makros

- Ersetzt Text  
    `#define Bezeichner Ersatztext`
- Beispiele  
    `#define PI 3.1415`  
    `#define HUND DOG`
- Aufheben  
    `#undef Bezeichner`
  - Wenn ein Makro nicht bis zum Ende einer Datei gültig sein sollte.
- Ersatztext kann auch entfallen.

# Interaktive Aufgabe

- Das folgende Programm gibt nur einmal 0 aus, sollte aber mindestens von 0 bis 9 hochzählen und diese Werte ausgeben. Welcher (logische) Fehler wurde gemacht und wie kann man ihn beheben?

```
#include <stdio.h>
#include <stdlib.h>
#define DEBUG_ERR printf("Fataler Debug-Fehler\n"); \
                    return EXIT_FAILURE
#define MAX 10

int main(void) {

    int i=0;
    do {
        printf("%d\n", i);
        if( ++i >= MAX)
            DEBUG_ERR;
    } while(1);
    return EXIT_SUCCESS;
}
```

# Interaktive Aufgabe

- Wie oft wird die for-Schleife ausgeführt und warum?

```
#include <stdio.h>
#include <stdlib.h>
#define CNT 10

int main(void) {

    int i;
    #undef CNT
    #define CNT 5
    for( i=0; i<CNT; i++) {
    #undef CNT
    #define CNT 20
        printf("%d\n",i);
    }
    return EXIT_SUCCESS;
}
```



# Makros mit Parametern

- Mit der `define`-Direktive kann man parametrisierte Makros schreiben.
  - Ein parametrisiertes Makro erkennt man daran, dass unmittelbar nach dem Makronamen eine Klammer folgt.
- Allgemeine Form  
`#define Bezeichner(Param1, ..., ParamN) Ersatztext`
- Beispiel
  - `#define eval(z) ((z) *= ((z) + 1) / 2)`
  - ...
  - `int a = 5;`
  - `eval(a);` // ersetzt mit `((a) *= ((a) + 1) / 2)`

# Stringersetzung

- Ist in einem Ersetzungstext vor dem Parameter das Zeichen # gesetzt, werden beim Aufruf des Makros das # und der Parameter durch das entsprechende Argument, das dann als String eingesetzt wird, ersetzt.
- Beispiel
  - `#define ausgabe(variable) printf(#variable"=%d\n",variable)`
- Programm
  - `ausgabe(Zahl);`  
wird zu
  - `printf("Zahl" "=%d\n", Zahl);`

# Beispiel (Makros)

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

#define wurzel(zahl)\
    printf(#zahl" von %f = %f\n",zahl,sqrt(zahl))
#define summe(zahl1,zahl2)\
    printf(#zahl1 " + " #zahl2 " = %d\n",zahl1 + zahl2)
#define gibaus(string)\
    printf(#string"\n")
#define wertvon(zahl,format)\
    printf(#zahl" = "format"\n", zahl)

int main(void) {
    float Wurzel;
    int Wert1 = 100, Wert2 = 150, integer = 20;
    char character = 's';
    float floating = 5.550f;
    printf("Zahl eingeben : ");
    scanf("%f", &Wurzel);
    wurzel(Wurzel);
    summe(Wert1, Wert2);
    gibaus(Hallo Welt);
    wertvon(character, "%c");
    wertvon(integer, "%d");
    wertvon(floating, "%f");
    return EXIT_SUCCESS;
}
```

## Ausgabe:

```
Zahl eingeben : 16
Wurzel von 16.000000 = 4.000000
Wert1 + Wert2 = 250
Hallo Welt
character = s
integer = 20
floating = 5.550000
```

# Eigenschaften und Seiteneffekte von Makros

- Falsche Operatoren-Zuordnung
  - Alle Parameter im Ersatztext immer klammern!
- Mehrmalige Auswertung der angegebenen Argumente
- Keine Typüberprüfung
- Makros haben keine Adressen
- Leerzeichen sind bei Makrodefinitionen wichtig
- Eventuell Codeaufblähung durch Makros
- Makros laufen schneller als Funktionen ab

# Beispiel (Makros – Seiteneffekte)

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

#define MINUS(a,b) ((a)-(b))
#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define QUADRAT(x) x * x
#define QUADRAT2(x) ((x) * (x))

int main(void) {
    int a = 5, b = 2;
    printf("%d\n", MINUS(a, b));
    printf("%f\n", MINUS(5.0, 2.5 + 0.5));
    printf("%d\n", MAX(a--, b));
    printf("%d\n", a);
    printf("%d\n", QUADRAT(a));
    printf("%d\n", QUADRAT(a + 1));
    printf("%d\n", QUADRAT2(a));
    printf("%d\n", QUADRAT2(a + 1));
    return EXIT_SUCCESS;
}
```

## Ausgabe:

```
3
2.000000
4
3
9
7
9
16
```

# Interaktive Aufgabe

- Im folgenden Beispiel gibt die Multiplikation den Wert 190 zurück. Korrekt wäre allerdings der Wert 100 ( $10 \cdot (20 - 10)$ ). Wie kann man das Problem beheben?

```
#include <stdio.h>
#include <stdlib.h>
#define MULTI(a,b) (a*b)

int main(void) {

    int val1=10, val2=20;
    printf("Multiplikation = %d\n", MULTI(val1,val2-10));
    return EXIT_SUCCESS;
}
```

# Bedingte Übersetzung

- Für die bedingte Übersetzung gibt es folgende Direktiven
  - `#if Konstanter_Ausdruck`
  - `#elif Konstanter_Ausdruck`
  - `#else`
  - `#endif`
  - `#ifdef Symbol`
  - `#ifndef Symbol`
- Damit kann man festlegen, welcher Teil des Codes übersetzt wird.

```
#ifdef __MINGW32__  
    #define SYS printf("Code für Windows-Rechner\n");  
    ... // weiterer Code nur für Windows-Rechner  
#endif
```

- Damit können auch Testversionen gekennzeichnet werden.

# Beispiel (Bedingte Übersetzung)

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

#ifdef __unix__
    #define clrscr() printf("\x1B[2J")
#elif __BORLANDC__ && __MSDOS__
    #include <conio.h>
#elif __WIN32__ || _MSC_VER
    #define clrscr() system("cls")
#else
    #define clrscr() printf("clrscr() - Fehler!!\n")
#endif

int main(void) {
    /* universelle Routine zum Löschen des Bildschirms */
    clrscr();
    return EXIT_SUCCESS;
}
```



# Bedingte Übersetzung bei Header-Dateien

- Beispiel

```
#ifndef _STDIO_H_
#define _STDIO_H_
...
#endif
```

- Hier überprüft der Präprozessor, ob er die Headerdatei `<stdio.h>` noch nicht eingebunden hat.
  - Erforderlich, wenn mehrere Headerdateien und Module benutzt werden, die `<stdio.h>` benötigen.
  - Somit würden alle Makros in der Headerdatei `<stdio.h>` mehrmals definiert werden, was im schlimmsten Fall sogar einen Fehler auslösen kann.

# Unterstützung beim Debugging

```
#include <stdio.h>
#include <stdlib.h>
#define DEBUG
```

Zur Entwicklungszeit -  
kann danach entfernt werden!

```
int main(void) {
    int ival;
    printf("Bitte Wert eingeben: ");
    scanf("%d", &ival);
#ifdef DEBUG
    printf("DEBUG: %d\n", ival);
#endif
    ival *= ival * 100;
#ifdef DEBUG
    printf("DEBUG: %d\n", ival);
#endif
    ival = ival/100;
    printf("%d\n", ival);
    // ....
    return EXIT_SUCCESS;
}
```

## Ausgabe:

```
Bitte Wert eingeben: 12
DEBUG: 12
DEBUG: 14400
144
```

# defined Operator

- In der Bedingung von `#if` oder `#elif` kann der Operator `defined` benutzt werden.

```
#if !defined (_NO_OLDNAMES)
```

...

- Vorteil gegenüber `#ifdef` und `#ifndef` ist, dass man damit komplexere Ausdrücke formulieren kann.

```
#if (defined (__STDC_VERSION__) && __STDC_VERSION__ >= 199901L) \  
|| !defined __STRICT_ANSI__ || defined __cplusplus
```

...

# Weitere Präprozessor-Direktiven

Direktive	Beschreibung
<code>#error "zeichenkette"</code> <code>(#warning "zeichenkette")</code>	Das Programm lässt sich nicht übersetzen und gibt die Fehlermeldung <code>zeichenkette</code> zurück. Sinnvoll, wenn ein nicht fertiges Programm noch nicht übersetzt werden sollte, oder innerhalb von <code>#if</code> -Konstrukten, die von externen Definitionen abhängen.
<code>#line n</code> <code>#line n "dateiname"</code>	Diese Direktive hat keinen Einfluss auf das Programme selbst. Damit kann man die Zeilennummer festlegen und den Dateinamen auf <code>dateiname</code> setzen.
<code>#pragma</code>	Diese Direktiven sind compilerspezifisch. Wenn ein Compiler diese Direktive nicht kennt, wird diese ignoriert.

# Vorgeschriebene Makros etc. (ANSI-C)

Makroname	Beschreibung
<code>__LINE__</code>	Gibt als Ganzzahl die aktuelle Zeilennummer in der Programmdatei zurück.
<code>__FILE__</code>	Gibt den Namen der Programmdatei als String-Literal zurück.
<code>__DATE__</code>	Gibt das Übersetzungsdatum (Präprozessor) der Programmdatei als String-Literal zurück.
<code>__TIME__</code>	Gibt die Übersetzungszeit der Programmdatei als String-Literal zurück.
<code>__STDC__</code>	Besitzt den Wert 1, wenn es sich beim Compiler um einen ANSI-C-konformen Compiler handelt.
<code>__STDC_HOSTED__</code>	Besitzt den Wert 1, wenn das Programm auf einer gehosteten Umgebung abläuft.
<code>__STDC_VERSION__</code>	Enthält eine ganzzahlige Konstante vom Typ <code>long</code> , wenn der Compiler den C99-Standard unterstützt (nur C99).
<code>__func__</code>	Gibt den Namen der Funktion aus ( <b>kein Makro, in C vordefiniert</b> ).

# Beispiel (Programm tester.c)

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

int main(void) {
    // ...
    printf("%d\n", __LINE__);
    printf("%s\n", __FILE__);
    printf("%s\n", __DATE__);
    printf("%s\n", __TIME__);
    printf("%d\n", __STDC__);
    printf("%s\n", __func__);
    printf("%d\n", __LINE__);
    // ...
    return EXIT_SUCCESS;
}
```

## Ausgabe:

```
6
..\tester.c
Dec  4 2012
10:08:15
1
main
12
```

# gcc – Präprozessor-Optionen

- **-D macro[=defn]**
  - Das Präprozessormakro mit dem Namen `macro` wird definiert.
    - Es können auch komplexere Makros definiert werden; hierbei müssen Buchstaben, welche in der `bash` eine Bedeutung haben, unter Anführungszeichen gesetzt werden (z.B. Leerzeichen).
  - Die Optionen `-D` und `-U` werden in der Reihenfolge auf der Konsole abgearbeitet, in welcher sie auftreten.
- **-E**
  - Stoppt nach dem Aufruf des Präprozessors, und der Compiler wird nicht aufgerufen.
    - Die Ausgabe wird an den Standardausgabekanal (`stdout`) geschickt.
- **-U macro**
  - Hebt vorangegangene Makros auf (sowohl solche die im Programm definiert wurden, als auch solche, die mit der `-D` Option erstellt wurden).

## Viele weitere Punkte ...

- Während der Expansion können zwei Token mit `##` verbunden werden.

```
#define TEXT_A "HELLO"  
#define msg(x) printf("%s", TEXT_ ## x);  
msg(A); // printf("%s", "HELLO");
```

- Makros können auch eine variable Anzahl von Argumenten akzeptieren.
- Makros können sich selbst referenzieren.
  - `#define foo (4 + foo)`
  - Sie sind aber nicht rekursiv!
  - Es gibt nur eine Ersetzung!
- ....