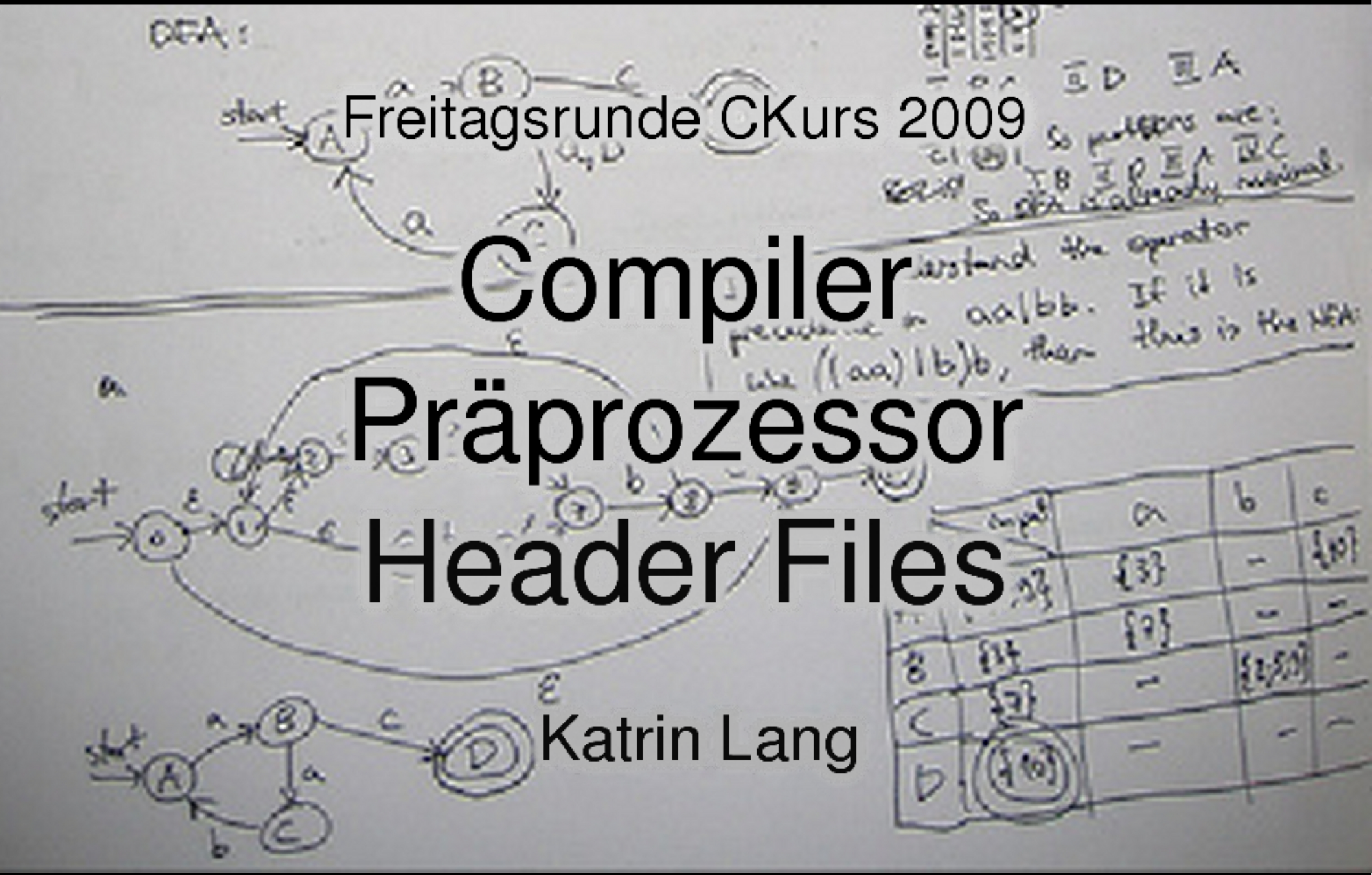


Freitagsrunde CKurs 2009

Compiler Präprozessor Header Files

Katrin Lang

Hello World Revisited

Datei hello.c

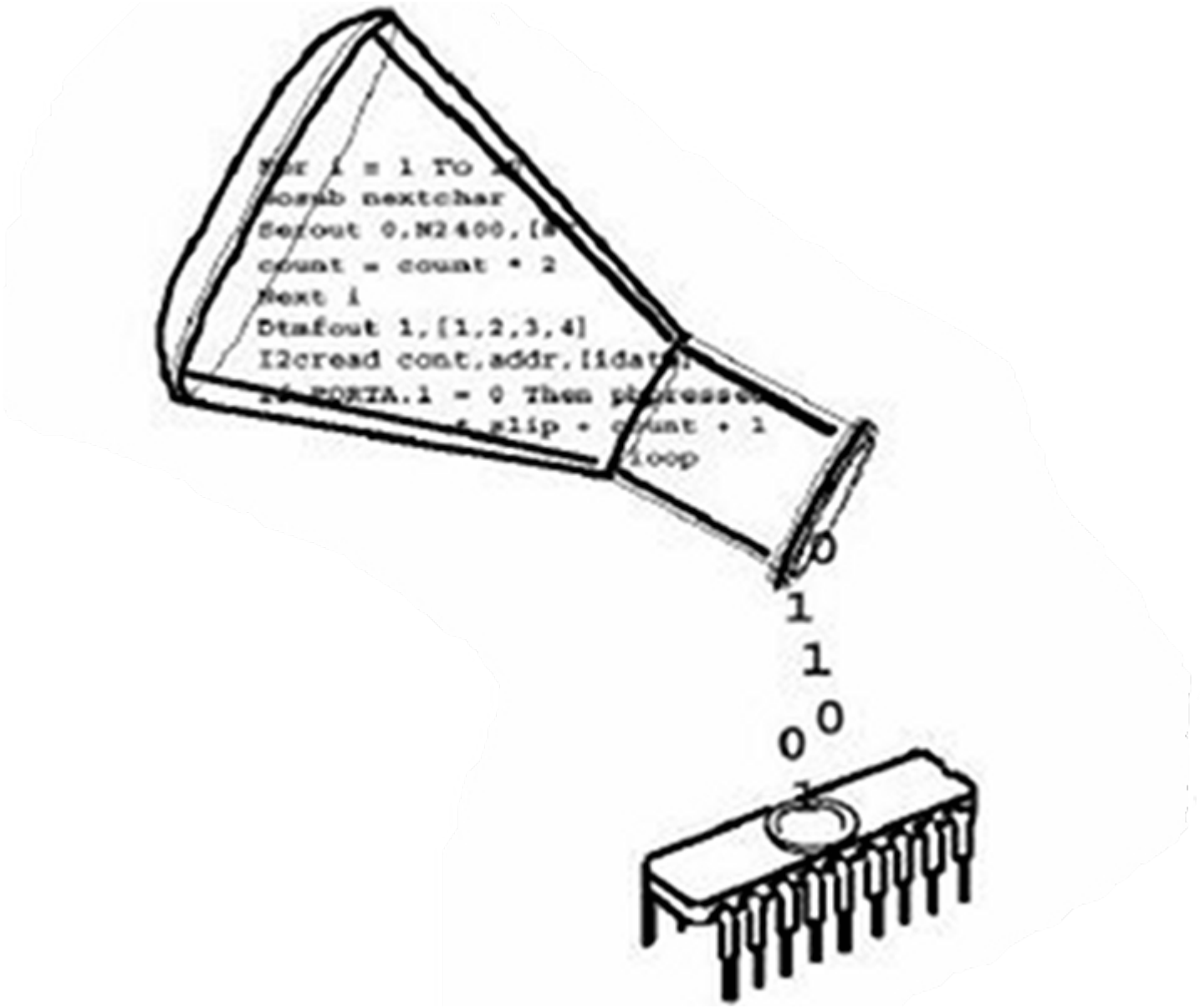
```
#include <stdio.h>

int main(){

    printf("Hello, World!\n");
    return 0;
}
```

Kommandozeile

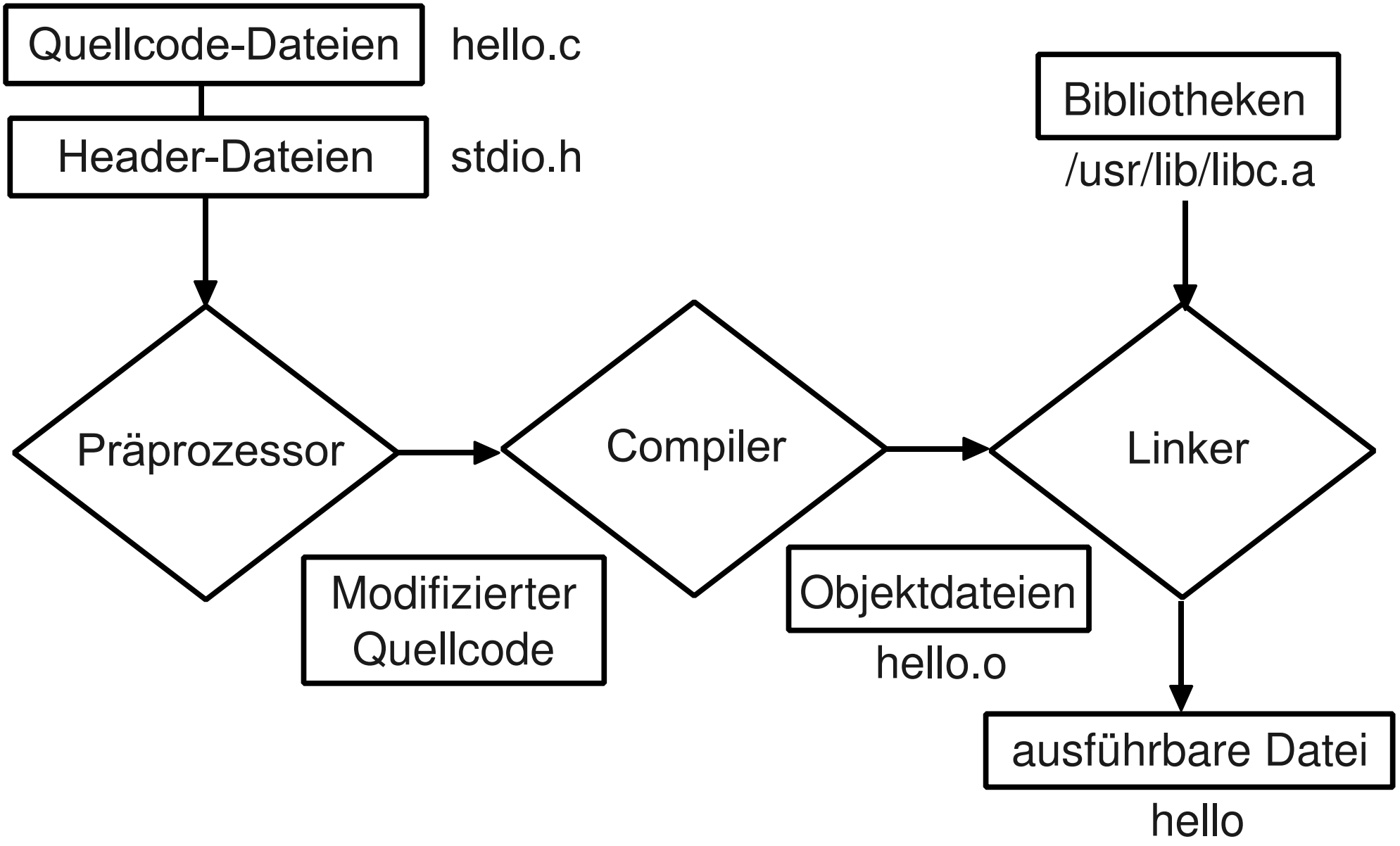
```
$ gcc -o hello hello.c
$ ./hello
Hello, World!
$
```

hello kann nach der Kompilierung wie jedes andere Unix-Kommando ausführt werden

Der C-Compiler



Kompilierung als mehrstufiger Prozess



Hello World bestehend aus mehreren Dateien

Datei main.c

```
extern hello(char* who);
extern bye(char* who);

int main(){

    hello("World");
    bye("World");
    return 0;
}
```

Dateien hello.c und bye.c

```
#include <stdio.h>
hello(char* who) {
    printf("Hello, %s!\n", who);
}
```

```
#include <stdio.h>
bye(char* who) {
    printf("Bye, %s!\n", who);
}
```

Separate Kompilierung

```
$ ls
bye.c       hello.c         main.c
$ gcc -c hello.c
$ gcc -c main.c
$ gcc -c bye.c
$ ls
bye.c       hello.c         main.c
bye.o       hello.o         main.o
$
```

Linken

```
$ gcc -o hello main.o hello.o bye.o
$ ./hello
Hello, World!
Bye, World!
$
```

Typen von Fehlern

- Präprozessor-Fehler, z.B
  - falsch geschriebene Präprozessoranweisung
  - undefinierte symbolische Konstante
- Compiler-Fehler, z.B
  - Syntaxfehler
  - Typfehler
- Linker-Fehler, z.B

  undefined reference to `foo'
  collect2: ld returned 1 exit status
- Laufzeitfehler, z.B.
  - divide by zero
  - Speicherzugriffsfehler: segmentation fault / bus error

Der Präprozessor

Präprozessoranweisungen

- am Zeichen # zu Beginn der Anweisung zu erkennen
- der Präprozessor bearbeitet nur Zeilen beginnend mit #

- Einfügen von Dateien:
  - #include
- Ersetzen von Text (Makros):
  - #define
- Bedingte Kompilierung:
  - #if, #ifdef, #ifndef, #else, #elif, #endif

Parameterlose Makros

- Syntax: **#define** name [replacement]

- Anwendung: Definition von symbolischen Konstanten

  #define PI 3.141529

- Präprozessor ersetzt vor der Kompilierung jedes Vorkommen von PI mit 3.141529

- Erhöht die Lesbarkeit und Wartbarkeit des Programms

Makros mit Parametern

- Syntax: #define name(dummy1[,dummy2][,...]) tokenstring
- Expression Makros
  - übersetzt in einen Ausdruck
  - ähnlich einer Funktion, die einen Wert zurückgibt

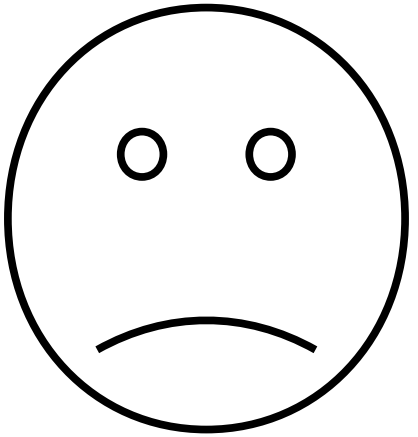
```
#define RADTODEG(x) ((x) * 57.29578)
```

- Statement Makros
  - übersetzt in ein oder mehrere volle C statements
  - ähnlich einer Funktion, die void zurückgibt

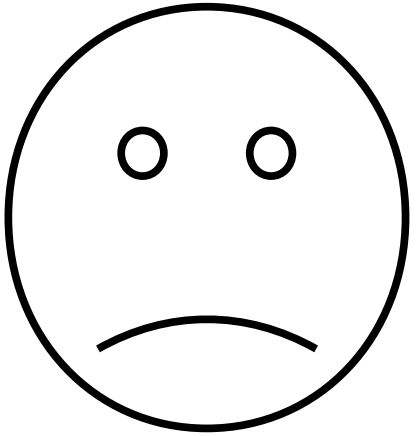
```
#define SWAP(x, y) \
  do{  tmp = x;   x = y;   y = tmp;  } while(0)
```

Warum Klammern?

```
#define RADTODEG(x) (x * 57.29578)

RADTODEG(a + b)

(a + b * 57.29578)
```



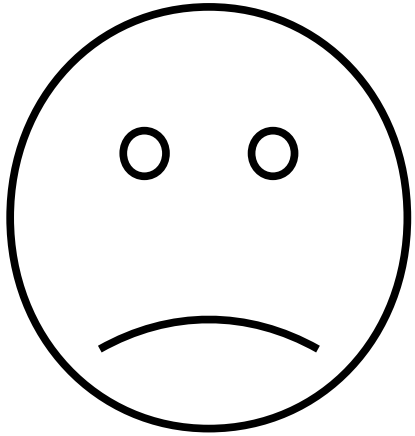
```
#define RADTODEG(x) (x) * 57.29578

1 / RADTODEG(a)

1 / (a) * 57.29578
```



Warum kein Semikolon am Ende eines Makros?

Semikolon beim Aufruf sieht natürlicher aus

-> Programmierer könnten Semikolons doppelt setzen
und damit den Control Flow eines Programms ändern



```
#define RADTODEG(x) ((x) * 57.29578);
#define DEGTORAD(x) ((x) * 0.017453);
```

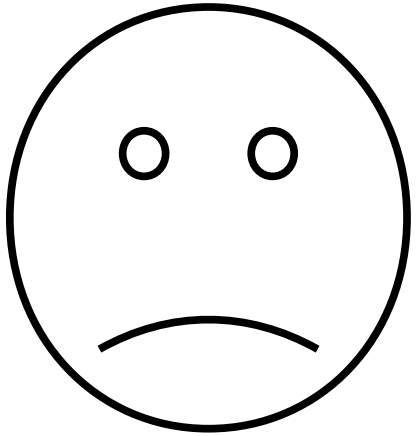
```
if(to_degree)
    y= RADTODEG(x);
else
    y= DEGTORAD(x);
```

```
if(to_degree)
    y= ((x) * 57.29578);;
else
    y= ((x) * 0.017453);;
```

Warum kein Semikolon am Ende eines Makros?

zusätzliches Problem bei Expression Makros

#define RADTODEG(x) ((x) * 57.29578);



if(RADTODEG(x)>180) ⟶ **if**(((x) * 57.29578);>180)

y= RADTODEG(x)+180; ⟶ y= ((x) * 57.29578);+180;

Mehrzeilige Statement Makros

```
/*
 * Swaps two values.
 * Requires tmp variable to be defined.
 */

#define SWAP(x, y) \
    do{ \
        tmp = x; \
        x = y; \
        y = tmp; \
    } \
    while(0)
```

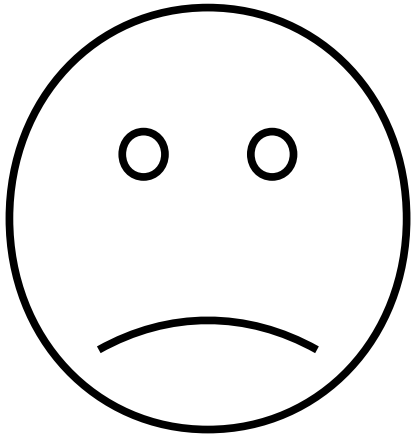
```
/*
 * Swaps two values.
 * Requires type passed as parameter
 */

#define SWAP(x, y, type) \
    do{ \
        type tmp = x; \
        x = y; \
        y = tmp; \
    } \
    while(0)
```

Elegante Lösung (nur gcc)

```
/*
 * Swaps two values.
 * Requires gcc typeof extension
 */

#define SWAP(x, y) \
    do{ \
        typeof(x) tmp = x; \
        x = y; \
        y = tmp; \
    } \
    while(0)
```

Warum do{...} while(0) ?

```
#define SWAP(x, y) \
    tmp = x; \
    x = y; \
    y = tmp
```

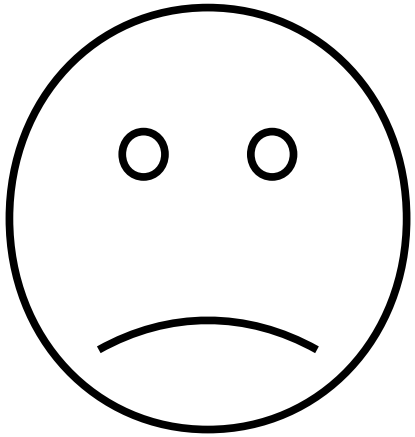


```
int x, y, tmp;
if (x > y)
    SWAP(x, y);
```

```
int x, y, tmp;
if(x > y)
    tmp = x;
x = y;
y = tmp;
```

Warum do{...} while(0) ?

```
#define SWAP(x, y){ \
    tmp = x; \
    x = y; \
    y = tmp; \
}
```



```
int x, y, tmp;
if (x > y)
    SWAP(x, y);
else
    ...
```

```
int x, y, tmp;
if (x > y) {
    tmp = x;
    x = y;
    y = tmp;
};
else
    ...
```

Zusammenfassung

- Wenn ein Dummy-Argument für einen Wert (oder einen Pointer zu einem Wert) steht, alle Vorkommen im Tokenstring klammern
- Den gesamten Tokenstring von Expression Makros klammern
- Keine Semikolons am Ende eines Makros
- Mehrzeilige Statement Makros mit \ trennen
- Tokenstring von mehrzeiligen Statement Makros mit do{..} while(0) umschließen

Weitere Probleme

- das Symbol erscheint beim Testen des Programms mittels Debugger nicht mehr
- Makros können sich ungewollt überschreiben
- Makros werden erst expandiert und daher vom Compiler überprüft, wenn sie tatsächlich aufgerufen werden
- Mehrfache Seiteneffekte

```
#define MIN(a, b) ((a)>(b)?(b):(a))
x = y = 1;
MIN(++x, ++y);
```

Makro Styleguide

- Namenskonvention Großbuchstaben
- Wenn möglich, Funktionen statt Makros verwenden
- Alternativen:
  - Parameterlose Makros – const (C90)
  - Makros mit Parametern – inline (C99)
- Vorsicht bei Makros zur Optimierung
- Makros kurz halten
- keine Hacks!

Bedingte Kompilierung

```
#ifdef _WIN32
    /* do Windows specific stuff here */
#endif
#ifdef __APPLE__
    /* do Mac specific stuff here */
#endif
#ifdef __linux__
    /* do Linux specific stuff here */
#endif
```

Beispiel: Debugging

```
#define PI 3.141529
#define RADTODEG(x) ((x) * 57.29578)

int debug= 1;

int main(){
    ...
    if(debug)
        printf("PI %f", RADTODEG(PI));
    ...
    return 0;
}
```

Ein einfaches Debugging Makro

Datei debug.h

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

#ifdef DEBUG
#define LOG printf
#else
#define LOG if(0) printf
#endif
```

Unser neues Hello World

Datei hello.c

```
#include “debug.h“

int main(){

    LOG("Hello World!\n");

    return 0;
}
```

Output des Präprozessors (gcc)

```
$gcc -E hello.c
... 748 weitere Zeilen
# 11 "hello.c"
int main(){

    printf("Hello, World!\n");
    return 0;
}
$
```

define per Kommandozeile (gcc)

gcc [-**D**macro[=defn]...] infile

%gcc -DDEBUG hello.c

% gcc -DDEBUG -DVERBOSE=1 hello.c

Hinzufügen von Debug Levels

Datei debug.h

```
#include <stdio.h>
#define VERBOSE 0

#ifdef DEBUG
#define LOG printf
#else
#define LOG if (0) printf
#endif
```

Unser neues Hello World mit Debug Levels

Datei hello.c

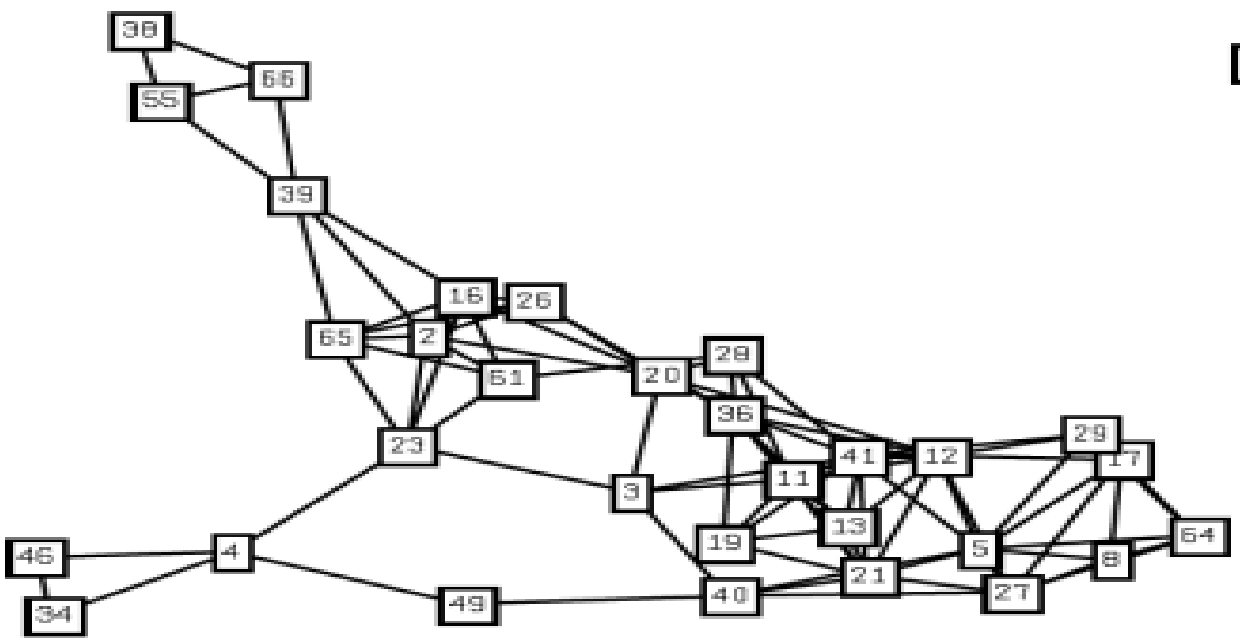
```
#include “debug.h“

int main(){

    if(VERBOSE== 1) LOG("Hello World!\n");

    return 0;
}
```

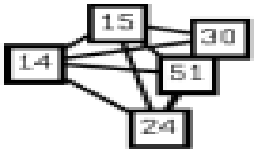
Debug Levels mit bedingter Kompilierung

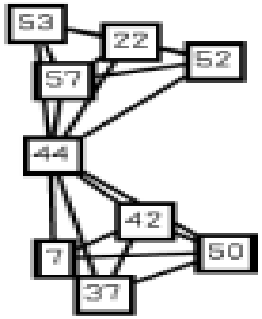
Datei hello.c

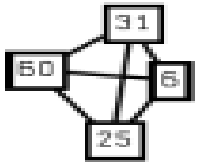
```
#include “debug.h“

int main(){

    #if VERBOSE==1
        LOG("Hello World!\n");
    #endif
    return 0;
}
```

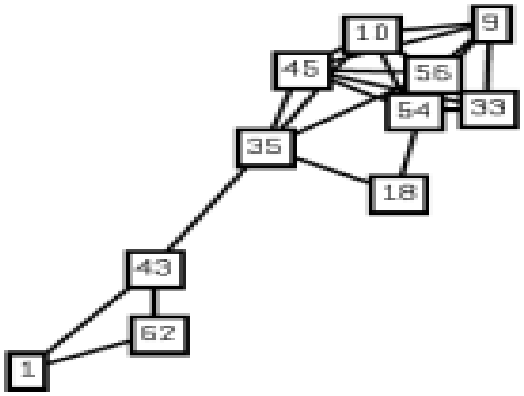
Header-Dateien



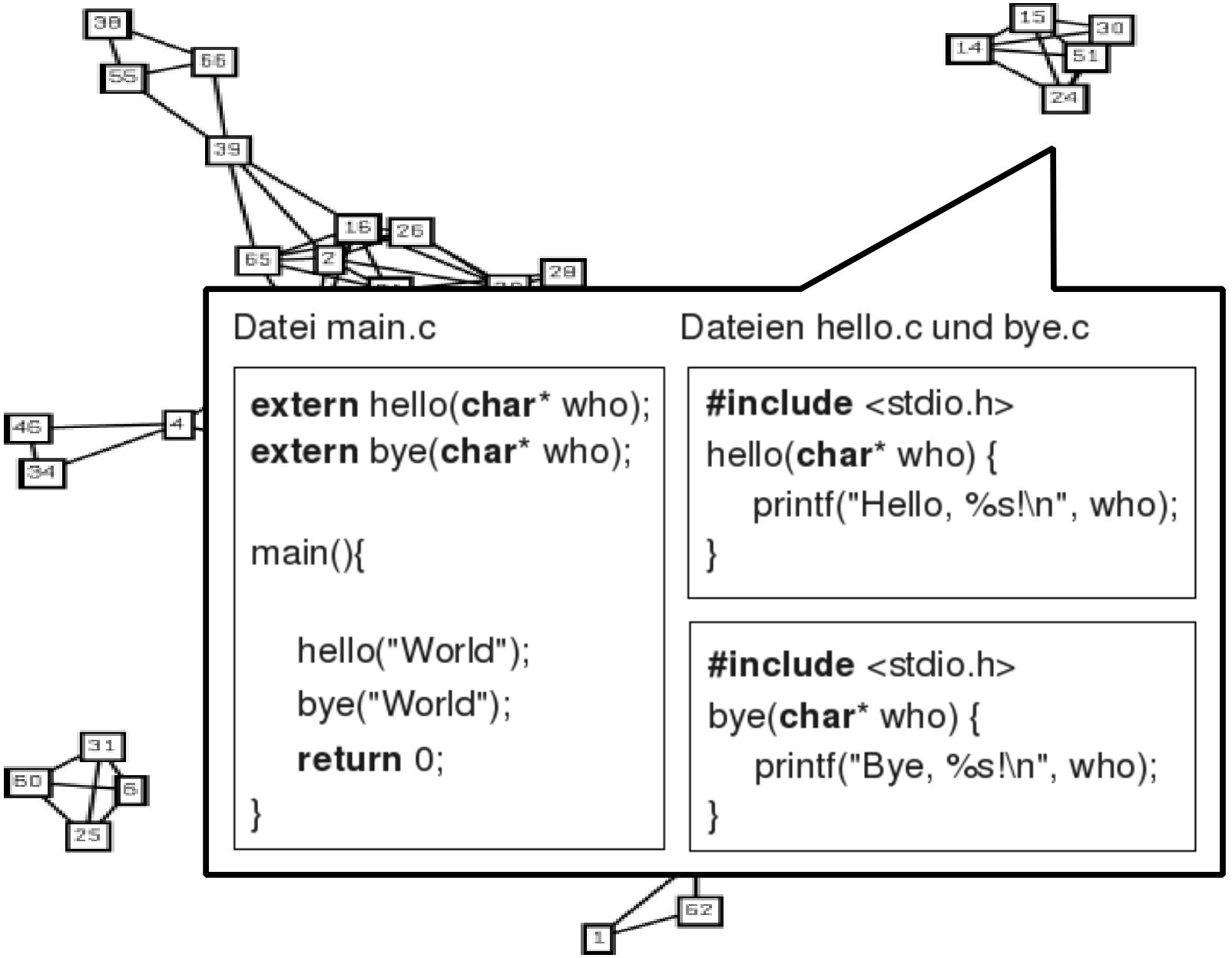








Header-Dateien



Header-Dateien

- Header Dateien erkennt man an der Endung ".h“
- Sie sind Teil von Schnittstellen zwischen Systemen
- Sie enthalten:
  - Funktions-Deklarationen
  - globale Variablen
  - symbolische Konstanten
  - Makros
  - Datentypen (z.B. Strukturen)

Inkludieren von Header-Dateien

#include <name>

- sucht zuerst im Verzeichnis der Systemdateien
- erst dann im Verzeichnis der Quelldatei
- wird normalerweise verwendet, um Headerdateien, die vom System geliefert werden, einzubinden (z.B. #include <stdio.h>)

#include "name"

- sucht zuerst im Verzeichnis der Quelldatei
- erst dann im Verzeichnis der Systemdateien
- wird normalerweise verwendet, um selbst geschriebene Header-Dateien einzubinden (z.B. #include "debug.h")

-I-Compileroption (gcc)

- Erweitert beim Übersetzen eines Programmes die Liste der Verzeichnisse in denen nach einer Datei gesucht wird.

  gcc -Iinclude helloworld.c

- sucht nach stdio.h zuerst als include/stdio.h, und erst dann als /usr/include/stdio.h.

Präprozessor-Output von Hello World

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

extern int printf (__const char *__restrict __format, ...);
```

Problem: Mehrfachinklusion

Datei foo.h

```
#include “bar.h“
#include “baz.h“
...
```

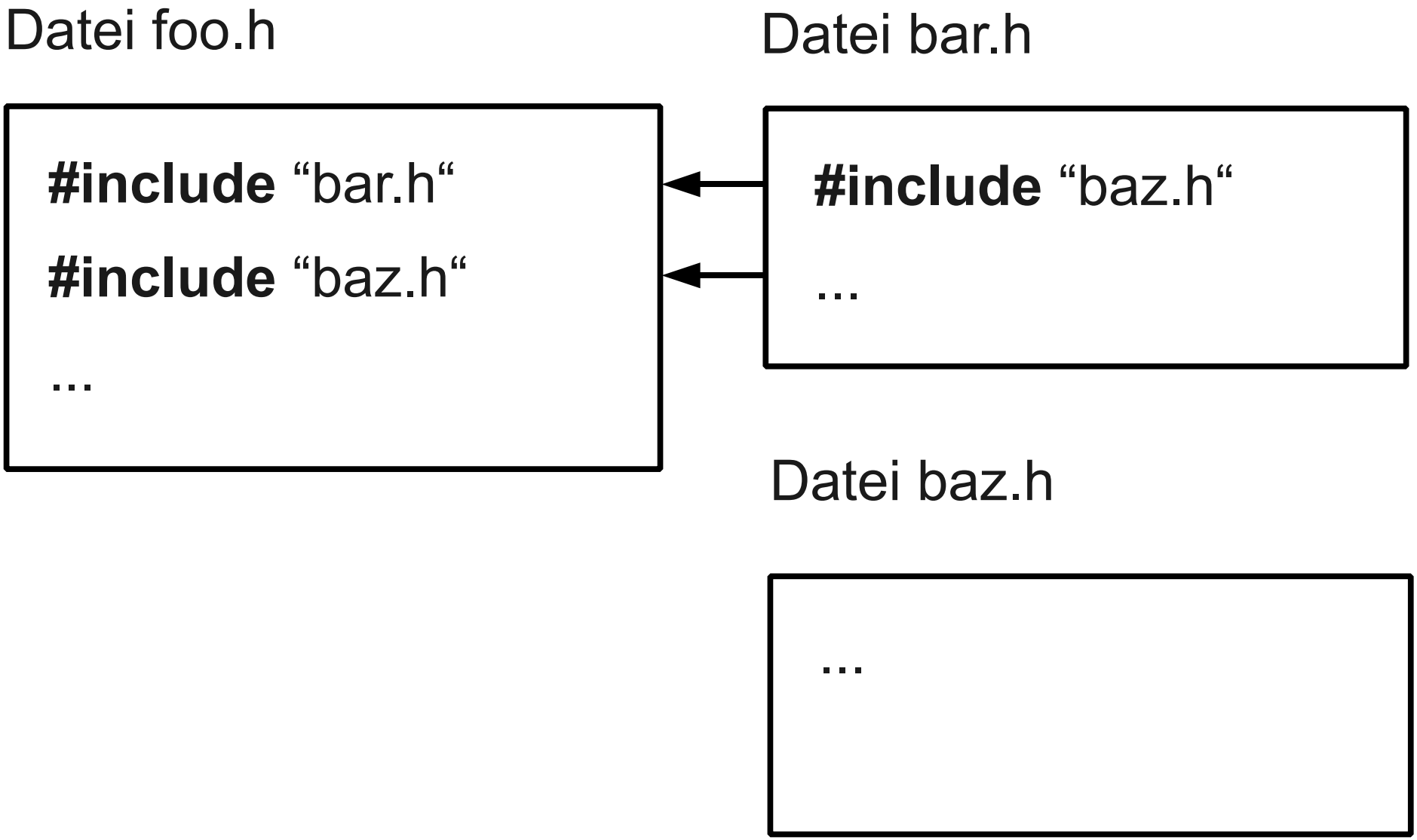
Datei bar.h

```
#include “baz.h“
...
```

Datei baz.h

```
...
```

Problem: Mehrfachinklusion



Problem: Mehrfachinklusion

Datei foo.h

```
#include “bar.h“
#include “baz.h“
...
```

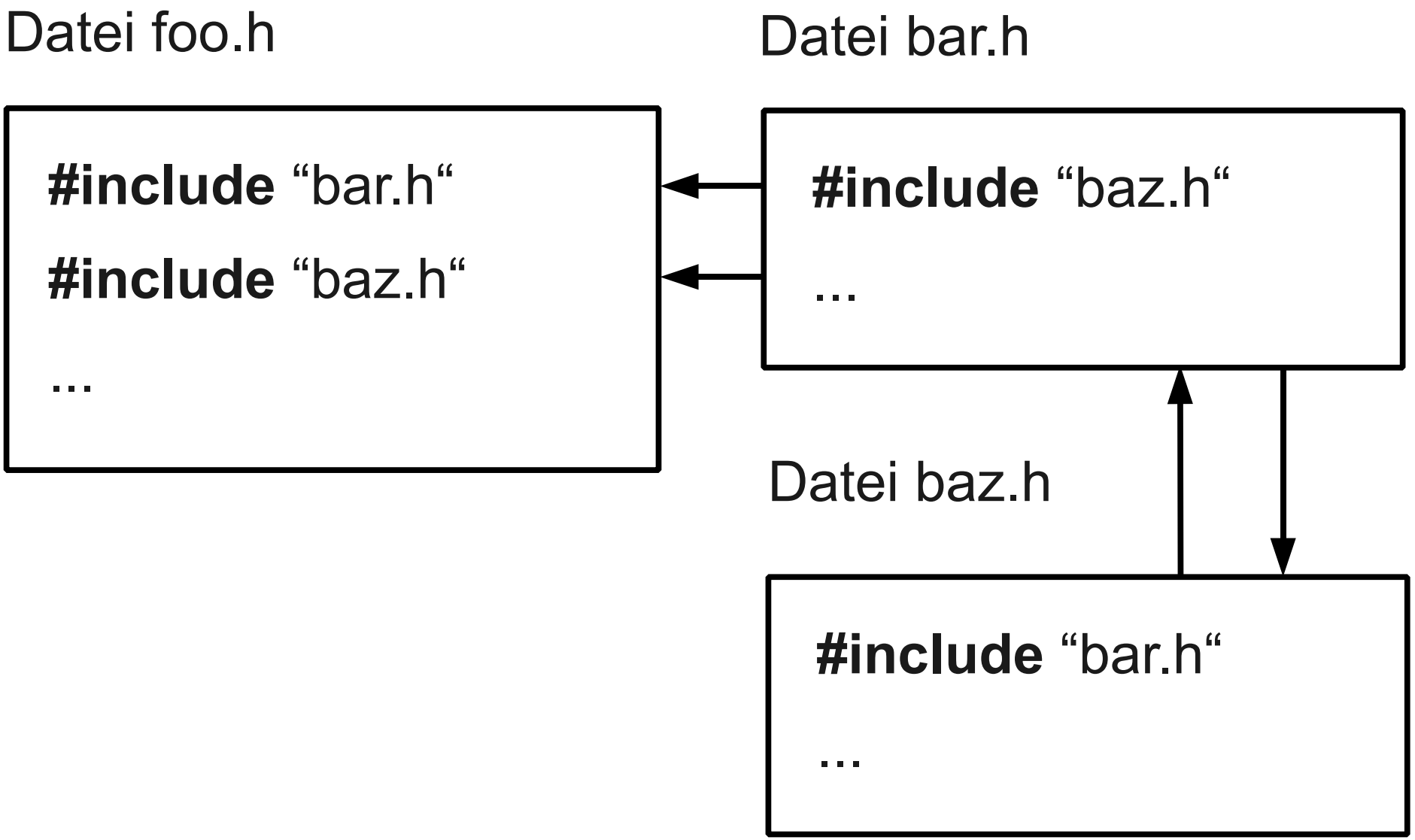
Datei bar.h

```
#include “baz.h“
...
```

Datei baz.h

```
#include “bar.h“
...
```

Problem: Mehrfachinklusion



Vermeidung von Mehrfachinklusion

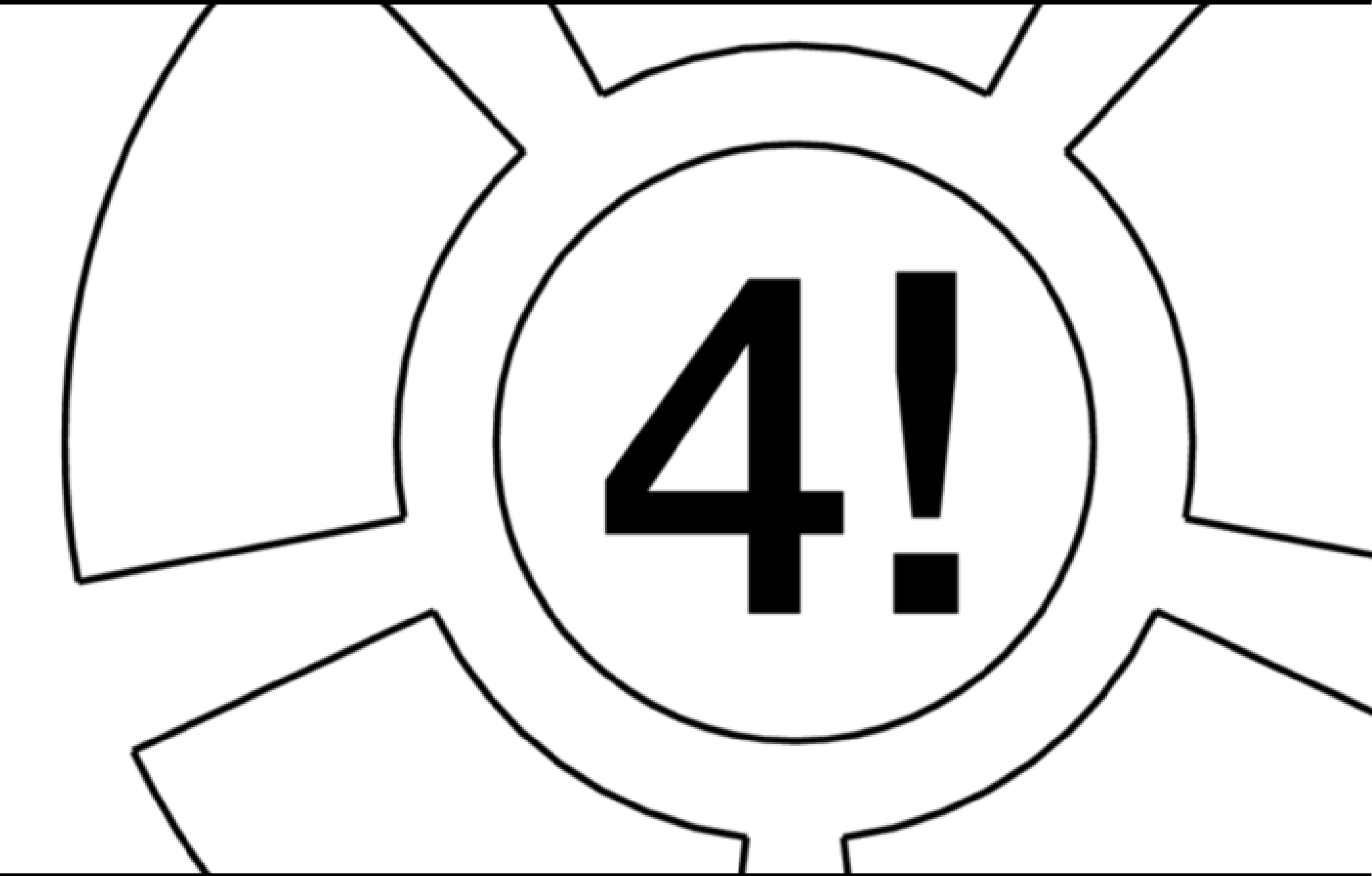
Datei foo.h

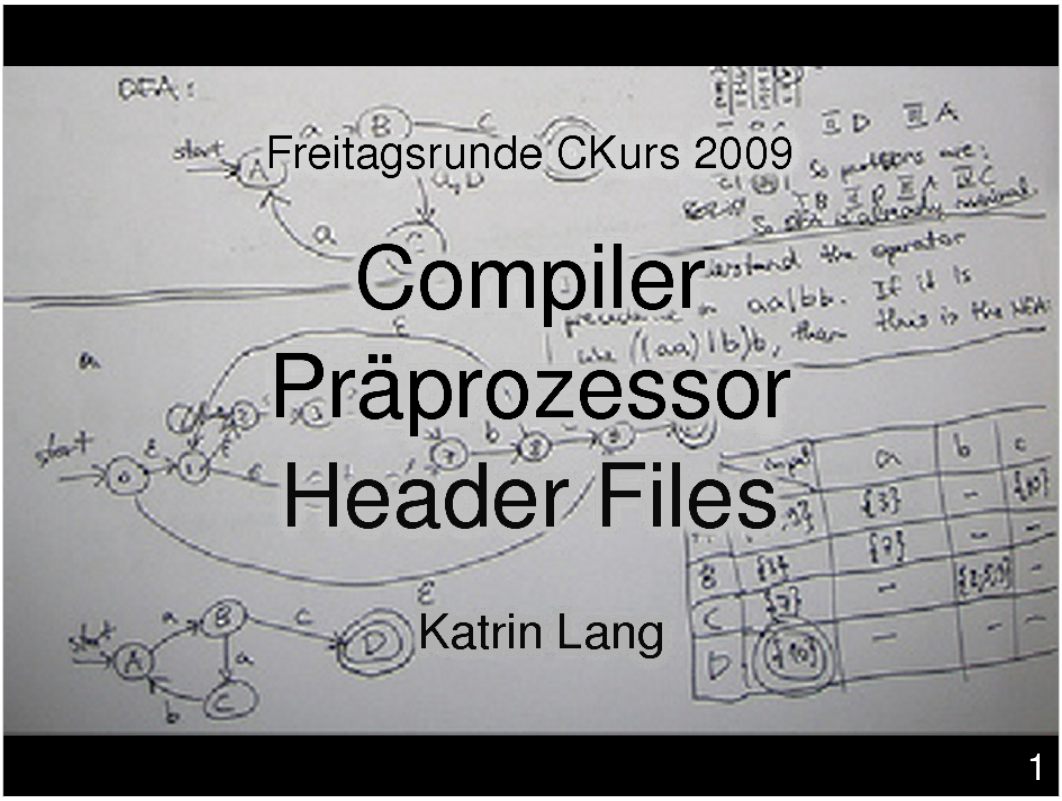
```
#ifndef FOO_H
   #define FOO_H


   extern int foo(int x, int y);


#endif
```

Unknown compiler

Scrapbook
Melbourne, [1860–1916]
Louise Hanson-Dyer Music Library Rare Collections, University of Melbourne

Danke!




DFA:
start
A
B
C
D
a
b
c
ε
Freitagsrunde CKurs 2009
Compiler
Präprozessor
Header Files
Katrin Lang

Hello World Revisited

Datei hello.c

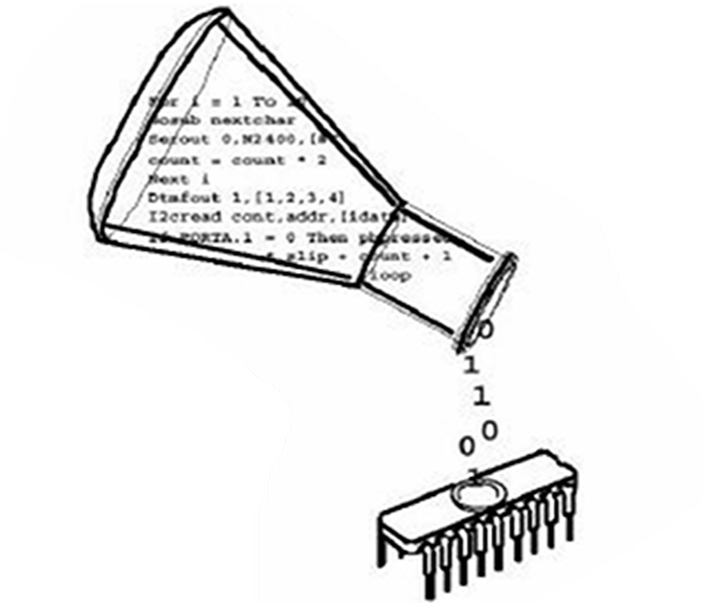
```
#include <stdio.h>

int main(){

    printf("Hello, World!\n");
    return 0;
}
```

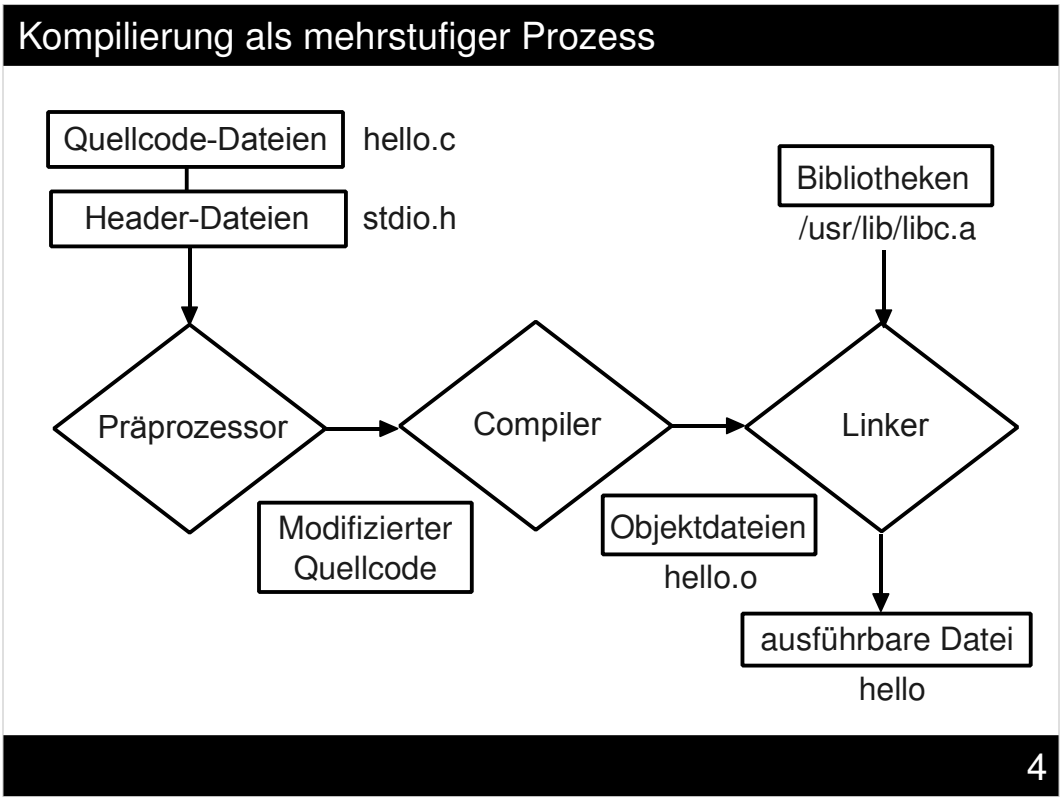
Kommandozeile

```
$ gcc -o hello hello.c
$ ./hello
Hello, World!
$
```

hello kann nach der Kompilierung wie jedes andere Unix-Kommando ausführt werden

Der C-Compiler





Der C-Präprozessor ist zuständig für den ersten Arbeitsgang beim Übersetzen eines Quellprogramms. Er führt vor dem eigentlichen Übersetzen des Programms in Maschinencode eine Art Vorverarbeitung aus.

Der “eigentliche" Compiler erzeugt aus dem Quelltext zusammen mit den durch #include-Anweisungen dazugekommenen Header-Files den Objektcode. Dabei leistet er die Hauptarbeit des Übersetzens:

* Er überprüft die syntaktische und semantische Korrektheit des Programms,

* wandelt die Konstrukte der Sprache C in Instruktionen der Maschinensprache des Systems um,

* optimiert die entstehende Instruktionsfolge. (Dieser letzte Schritt wird oft wieder von besonderen Programmen, den optimizern, erledigt.)

Meistens wird dabei zunächst Assembler Code erzeugt Assemblerdateien haben die Endung.s
Daraus macht der Assembler Instruktionen der Maschinensprache des Systems, also Binärcode.

Das Ergebnis dieser Stufe ist das Object-File, traditionell mit der File-Endung .o, also zum Beispiel hello.o.
hello.o enthält eine Referenz zu der funktion printf. Da wir dieses Symbol nicht definiert haben, ist es eine externe Referenz. Der Maschinencode enthält eine Instruktion, um printf aufzurufen, aber im Objektcode wissen wir noch nicht die tatsächliche Adresse, die notwendig ist, um diese Funktion auszuführen.

Der Linker bekommt als Input Objektdateien und Libraries. Libraries sind Kollektionen von Objektdateien.
Er löst externe Referenzen auf, und trägt die endgültigen Adressen für Funktionen und Variablen ein.

Obwohl man Bibliotheken wie Object-Files verwenden kann, behandelt der Linker sie anders:

* Objektdateien werden vollständig mit zum Programm dazugebunden.

* Bibliotheken werden vom Linker nur nach den Symbolen durchsucht, die im Moment noch nicht definiert sind. (Symbole heißt in unserem Fall ,,globale Funktions- und Variablennamen.") Nur die Object-Files aus dem Archiv, die solche undefinierten Symbole definieren, werden zum Programm dazugebunden.

Wenn alle externen Referenzen vorhanden sind, hat man am Ende eine ausführbare Datei.

Hello World bestehend aus mehreren Dateien

Datei main.c

```
extern hello(char* who);
extern bye(char* who);

int main(){

    hello("World");
    bye("World");
    return 0;
}
```

Dateien hello.c und bye.c

```
#include <stdio.h>
hello(char* who) {
    printf("Hello, %s!\n", who);
}
```

```
#include <stdio.h>
bye(char* who) {
    printf("Bye, %s!\n", who);
}
```


Grosse Programme sollte man schon der Übersichtlichkeit halber in mehrere Source-Files aufteilen, auch weil dann beim Entwicklen der Software nicht nach jeder kleinen Änderung alles neu übersetzt werden muß. Linken geht schnell; das Übersetzen dauert lange.

Separate Kompilierung

```
$ ls
bye.c       hello.c       main.c
$ gcc -c hello.c
$ gcc -c main.c
$ gcc -c bye.c
$ ls
bye.c       hello.c       main.c
bye.o       hello.o       main.o
$
```

6

Wenn man nicht (zum Beispiel mit -c) das Gegenteil festlegt, macht ein C-Compiler immer bis zum ausführbaren Programm weiter.
Die .o Dateien werden dann nach dem Kompilierungsvorgang gelöscht

Wenn wir versuchen, die files einzeln ohne -c zu kompilieren, erhalten wir Linker-Fehler

undefined reference to `bye'
undefined reference to `hello'

bye und hello wurden nirgends definiert.

undefined reference to `main'

In c muss jedes Programm genau eine main-Funktion haben, denn das ist der Punkt, an dem die Ausführung des Programms beginnt.

Linken

```
$ gcc -o hello main.o hello.o bye.o
$ ./hello
Hello, World!
Bye, World!
$
```


Um nur die Linker-Stufe des Compilers gcc oder cc zu benutzen, braucht man keine besonderen Optionen anzugeben. Daß eine Datei Objektcode enthält, sieht der Compiler an der Endung: .o für Objektcode.

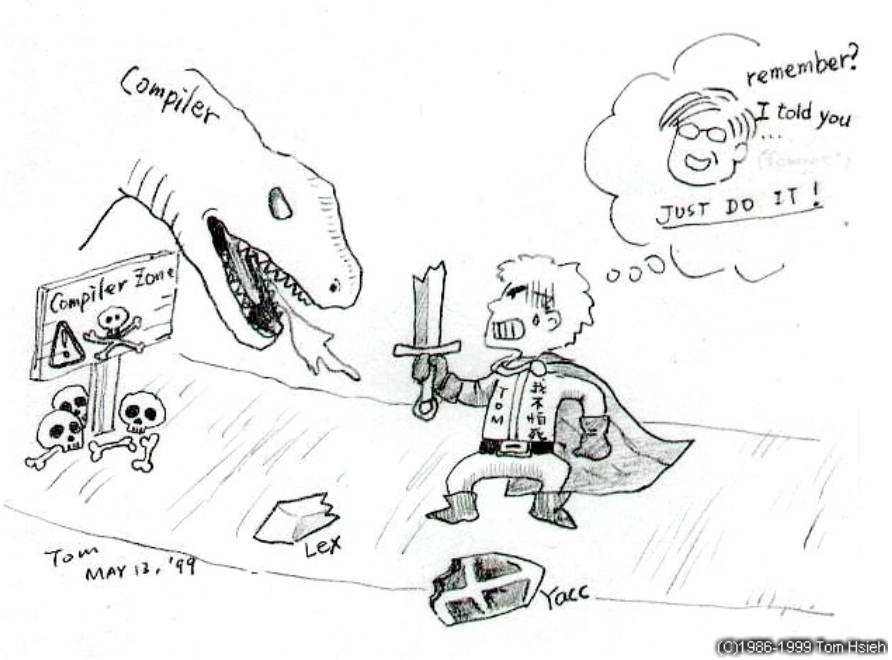
Die -o-Option, die einem in diesem Zusammenhang einfallen könnte, steht für output, nicht für object - das Wort nach ihr sagt, wie das ausführbare Programm heißen soll. Wenn man -o nicht angibt, ist der default a.out; deshalb heißt das File-Format für ausführbare Programme unter Unix auch manchmal ,,a.out-Format.``

Typen von Fehlern

- Präprozessor-Fehler, z.B
  - falsch geschriebene Präprozessoranweisung
  - undefinierte symbolische Konstante
- Compiler-Fehler, z.B
  - Syntaxfehler
  - Typfehler
- Linker-Fehler, z.B

  undefined reference to `foo'
  collect2: ld returned 1 exit status
- Laufzeitfehler, z.B.
  - divide by zero
  - Speicherzugriffsfehler: segmentation fault / bus error

Der Präprozessor



9

Die Tücken des Präprozessors liegen darin, dass er ein reines Textersetzungsprogramm ist.

Das hat zur Folge, dass man mysteriöse Fehlermeldungen bekommt, die sich auf den modifizierten Text beziehen und somit auf Code, den man so nie geschrieben hat.

Präprozessoranweisungen

- am Zeichen # zu Beginn der Anweisung zu erkennen
- der Präprozessor bearbeitet nur Zeilen beginnend mit #

- Einfügen von Dateien:
  - #include
- Ersetzen von Text (Makros):
  - #define
- Bedingte Kompilierung:
  - #if, #ifdef, #ifndef, #else, #elif, #endif

Parameterlose Makros

- Syntax: **#define** name [replacement]
- Anwendung: Definition von symbolischen Konstanten
  #define PI 3.141529
- Präprozessor ersetzt vor der Kompilierung jedes Vorkommen von PI mit 3.141529
- Erhöht die Lesbarkeit und Wartbarkeit des Programms

Makros mit Parametern

- Syntax: #define name(dummy1[,dummy2][,...]) tokenstring
- Expression Makros
  - übersetzt in einen Ausdruck
  - ähnlich einer Funktion, die einen Wert zurückgibt

```
#define RADTODEG(x) ((x) * 57.29578)
```

- Statement Makros
  - übersetzt in ein oder mehrere volle C statements
  - ähnlich einer Funktion, die void zurückgibt

```
#define SWAP(x, y) \
  do{  tmp = x;   x = y;   y = tmp;  } while(0)
```


Die Parameter in der Bezeichnerliste einer Makrodefinition müssen durch Kommas voneinander getrennt werden und innerhalb dieser Liste jeweils eindeutig sein.

Man beachte, dass die Parameterliste nicht durch Leerzeichen vom Namen des Makros getrennt werden darf. Ansonsten wird alles ab der öffnenden Klammer zum Tokenstring gezählt und für den Makronamen substituiert.

Der Tokenstring muss dagegen durch ein oder mehrere "weiße" Leerzeichen (whitespaces) von der Parameterliste getrennt sein, sonst sieht der Präprozessor das Makro sie als Makro ohne Parameter

Beim Aufruf von RADTODEG(PI) passiert folgendes:

Da der Makroaufruf selbst ein Makro enthält, wird dieses zuerst erweitert. Dann ersetzt der Präprozessor jedes Vorkommen von x im Tokenstring mit 3.141529 (nicht PI)

Der Präprozessor substituiert dann den entstandenen Tokenstring für den Makrouafruf.

Dieser Ersatz betrifft nur Terminalsymbole (also eigenständige Vorkommen von bezeichner im Quelltext), nicht aber Zeichenfolgen, die in einem Kommentar erscheinen, Teil eines anderen Bezeichners oder Teil einer Stringkonstanten sind.

Makros können natürlich auch andere Makros aufrufen.

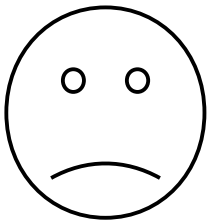
Wenn die Definition eines Makros xyz den Bezeichner xyz im Tokenstring enthält, wird dieser Bezeichner nicht erneut erweitert - auch dann nicht, wenn sich die Zeichenfolge xyz erst durch die Erweiterung eines anderen Makros ergibt. Rekursive Makros sind also nicht möglich.

Warum Klammern?

#define RADTODEG(x) (x * 57.29578)

RADTODEG(a + b)

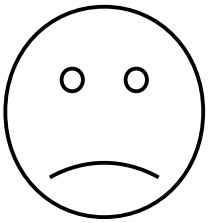
(a + b * 57.29578)



#define RADTODEG(x) (x) * 57.29578

1 / RADTODEG(a)

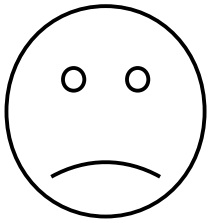
1 / (a) * 57.29578



Warum kein Semikolon am Ende eines Makros?

Semikolon beim Aufruf sieht natürlicher aus

-> Programmierer könnten Semikolons doppelt setzen
und damit den Control Flow eines Programms ändern



```
#define RADTODEG(x) ((x) * 57.29578);
#define DEGTORAD(x) ((x) * 0.017453);
```

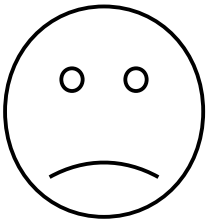
```
if(to_degree)
    y= RADTODEG(x);
else
    y= DEGTORAD(x);
```

```
if(to_degree)
    y= ((x) * 57.29578);;
else
    y= ((x) * 0.017453);;
```

Warum kein Semikolon am Ende eines Makros?

zusätzliches Problem bei Expression Makros

#define RADTODEG(x) ((x) * 57.29578);



if(RADTODEG(x)>180) ⟶ **if**(((x) * 57.29578);>180)

y= RADTODEG(x)+180; ⟶ y= ((x) * 57.29578);+180;

Mehrzeilige Statement Makros

```
/*
 * Swaps two values.
 * Requires tmp variable to be defined.
 */

#define SWAP(x, y) \
    do{ \
        tmp = x; \
        x = y; \
        y = tmp; \
    } \
    while(0)
```


\ unterdrückt den Zeilenumbruch, sonst Compilerfehler

Alternatives SWAP

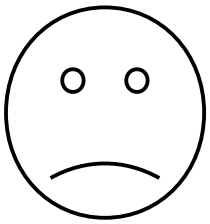
```
/*
 * Swaps two values.
 * Requires type passed as parameter
 */

#define SWAP(x, y, type) \
    do{ \
        type tmp = x; \
        x = y; \
        y = tmp; \
    } \
    while(0)
```

Elegante Lösung (nur gcc)

```
/*
 * Swaps two values.
 * Requires gcc typeof extension
 */

#define SWAP(x, y) \
    do{ \
        typeof(x) tmp = x; \
        x = y; \
        y = tmp; \
    } \
    while(0)
```

Warum do{...} while(0) ?

```
#define SWAP(x, y) \
    tmp = x; \
    x = y; \
    y = tmp
```

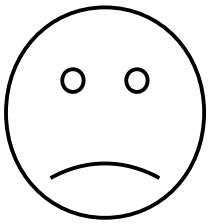


```
int x, y, tmp;
if (x > y)
    SWAP(x, y);
```

```
int x, y, tmp;
if(x > y)
    tmp = x;
x = y;
y = tmp;
```

Warum do{...} while(0) ?

```
#define SWAP(x, y){ \
    tmp = x; \
    x = y; \
    y = tmp; \
}
```



```
int x, y, tmp;
if (x > y)
    SWAP(x, y);
else
    ...
```

```
int x, y, tmp;
if (x > y) {
    tmp = x;
    x = y;
    y = tmp;
};
else
    ...
```


Der Compiler erkennt in der Regel, dass die Schleife nur einmal durchlaufen wird und generiert keinen Code für sie.

Zusammenfassung

- Wenn ein Dummy-Argument für einen Wert (oder einen Pointer zu einem Wert) steht, alle Vorkommen im Tokenstring klammern
- Den gesamten Tokenstring von Expression Makros klammern
- Keine Semikolons am Ende eines Makros
- Mehrzeilige Statement Makros mit \ trennen
- Tokenstring von mehrzeiligen Statement Makros mit do{..} while(0) umschließen

Weitere Probleme

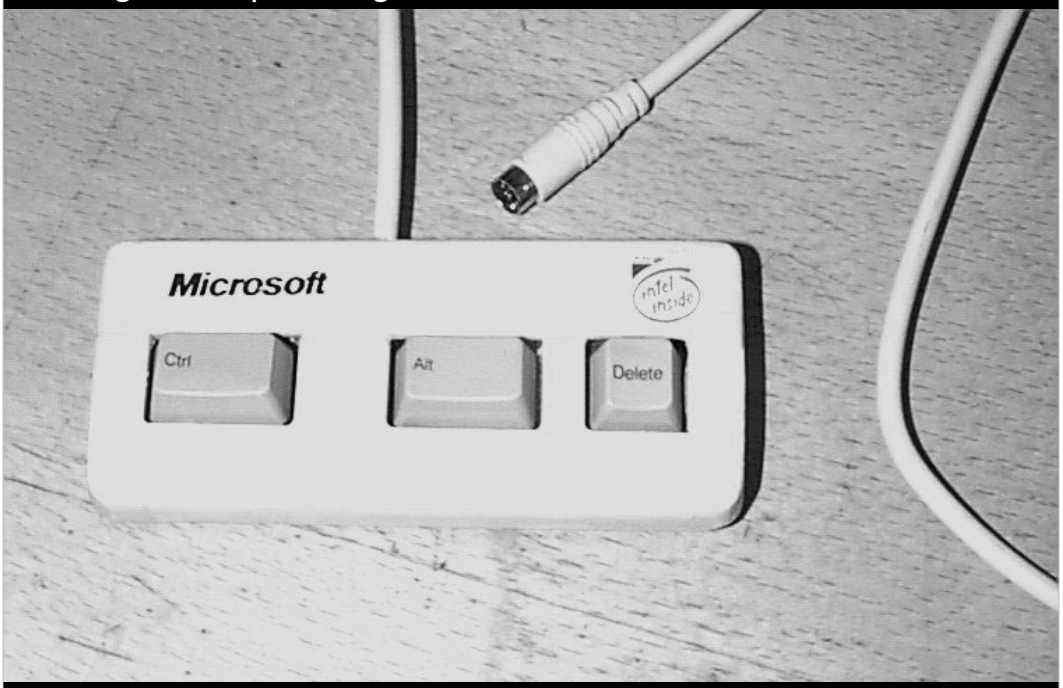
- das Symbol erscheint beim Testen des Programms mittels Debugger nicht mehr
- Makros können sich ungewollt überschreiben
- Makros werden erst expandiert und daher vom Compiler überprüft, wenn sie tatsächlich aufgerufen werden
- Mehrfache Seiteneffekte

```
#define MIN(a, b) ((a)>(b)?(b):(a))
x = y = 1;
MIN(++x, ++y);
```

Makro Styleguide

- Namenskonvention Großbuchstaben
- Wenn möglich, Funktionen statt Makros verwenden
- Alternativen:
  - Parameterlose Makros – const (C90)
  - Makros mit Parametern – inline (C99)
- Vorsicht bei Makros zur Optimierung
- Makros kurz halten
- keine Hacks!

Bedingte Kompilierung



Bedingte Kompilierung

```
#ifdef _WIN32
    /* do Windows specific stuff here */
#endif
#ifdef __APPLE__
    /* do Mac specific stuff here */
#endif
#ifdef __linux__
    /* do Linux specific stuff here */
#endif
```

Beispiel: Debugging

```
#define PI 3.141529
#define RADTODEG(x) ((x) * 57.29578)

int debug= 1;

int main(){
    ...
    if(debug)
        printf("PI %f", RADTODEG(PI));
    ...
    return 0;
}
```

Ein einfaches Debugging Makro

Datei debug.h

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

#ifdef DEBUG
#define LOG printf
#else
#define LOG if(0) printf
#endif
```

Unser neues Hello World

Datei hello.c

```
#include “debug.h“

int main(){

    LOG("Hello World!\n");
    return 0;
}
```

Output des Präprozessors (gcc)

```
$gcc -E hello.c
... 748 weitere Zeilen
# 11 "hello.c"
int main(){

    printf("Hello, World!\n");
    return 0;
}
$
```

define per Kommandozeile (gcc)

gcc [-**D**macro[=defn]...] infile

%gcc -DDEBUG hello.c

% gcc -DDEBUG -DVERBOSE=1 hello.c

Es ist bequem, dieses ,,DEBUG`` von der Aufrufzeile des Compilers aus ein- (Testing) und auszuschalten (Release-Build), ohne daß man jedesmal die Quelldatei selbst editieren muß. (Übersetzen muß man sie trotzdem neu; ohne, daß der Präprozessor läuft, können Präprozessor-#defines nicht wirken.)

Hinzufügen von Debug Levels

Datei debug.h

```
#include <stdio.h>
#define VERBOSE 0

#ifdef DEBUG
#define LOG printf
#else
#define LOG if (0) printf
#endif
```

Unser neues Hello World mit Debug Levels

Datei hello.c

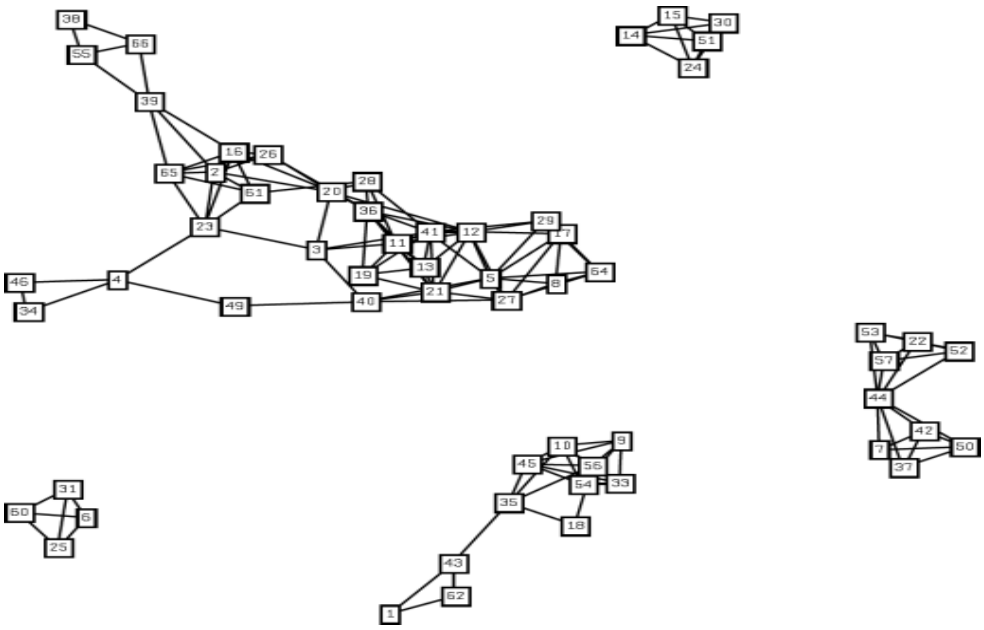
```
#include “debug.h“

int main(){

    if(VERBOSE== 1) LOG("Hello World!\n");
    return 0;
}
```

Debug Levels mit bedingter Kompilierung

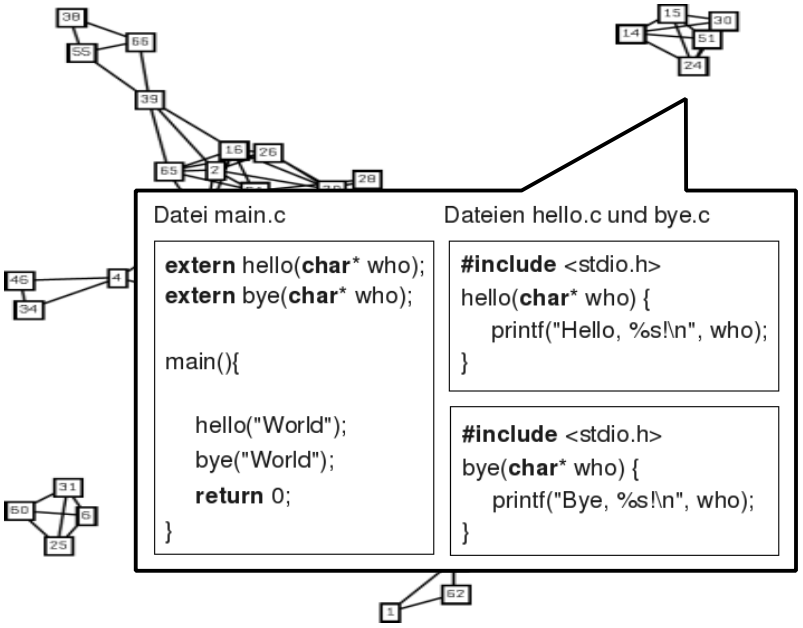
Datei hello.c

```
#include “debug.h“

int main(){

    #if VERBOSE==1
        LOG("Hello World!\n");
    #endif
    return 0;
}
```

Header-Dateien



Header-Dateien



Header-Dateien

- Header Dateien erkennt man an der Endung ".h“
- Sie sind Teil von Schnittstellen zwischen Systemen
- Sie enthalten:
  - Funktions-Deklarationen
  - globale Variablen
  - symbolische Konstanten
  - Makros
  - Datentypen (z.B. Strukturen)

Inkludieren von Header-Dateien

#include <name>

- sucht zuerst im Verzeichnis der Systemdateien
- erst dann im Verzeichnis der Quelldatei
- wird normalerweise verwendet, um Headerdateien, die vom System geliefert werden, einzubinden (z.B. #include <stdio.h>)

#include "name"

- sucht zuerst im Verzeichnis der Quelldatei
- erst dann im Verzeichnis der Systemdateien
- wird normalerweise verwendet, um selbst geschriebene Header-Dateien einzubinden (z.B. #include "debug.h")

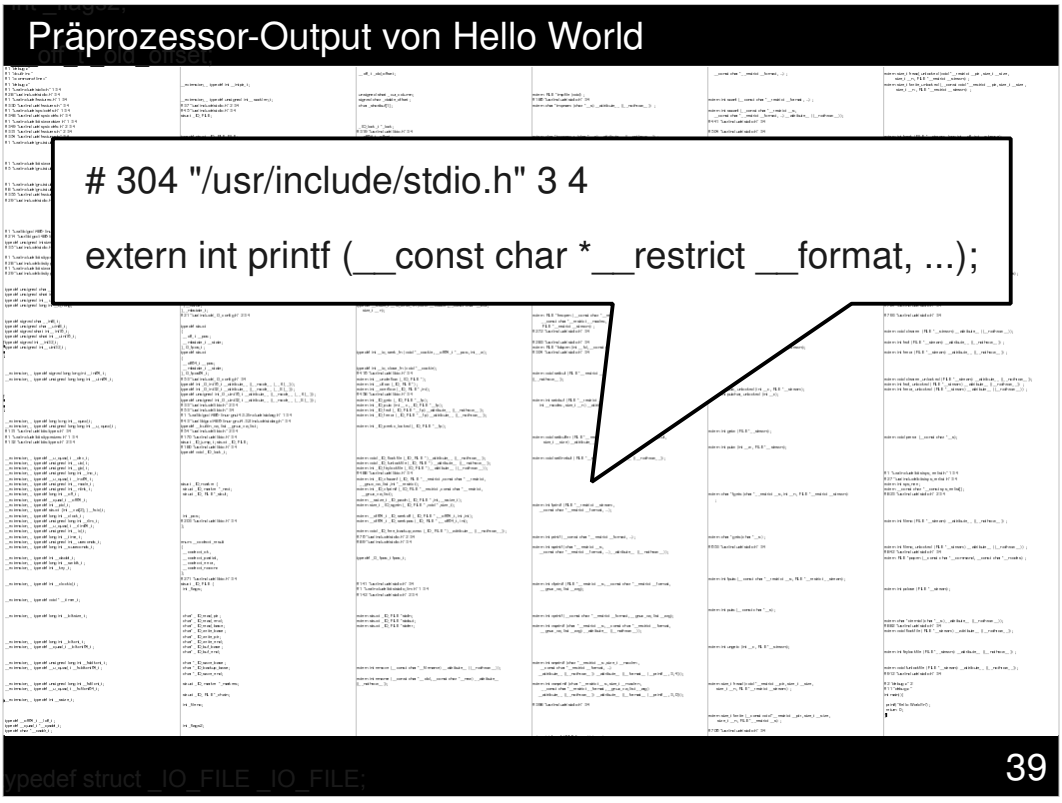
-I-Compileroption (gcc)

- Erweitert beim Übersetzen eines Programmes die Liste der Verzeichnisse in denen nach einer Datei gesucht wird.

  gcc -Iinclude helloworld.c

- sucht nach stdio.h zuerst als include/stdio.h, und erst dann als /usr/include/stdio.h.

Präprozessor-Output von Hello World



per Copy & Paste fügt #include den Inhalt der Datei stdio.h ein

Problem: Mehrfachinklusion

Datei foo.h

```
#include “bar.h“
#include “baz.h“
...
```

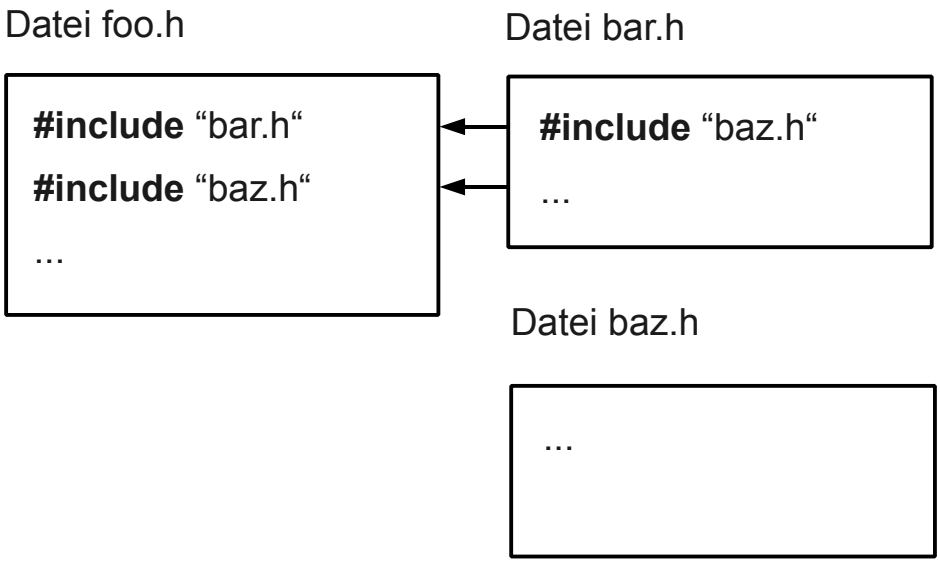
Datei bar.h

```
#include “baz.h“
...
```

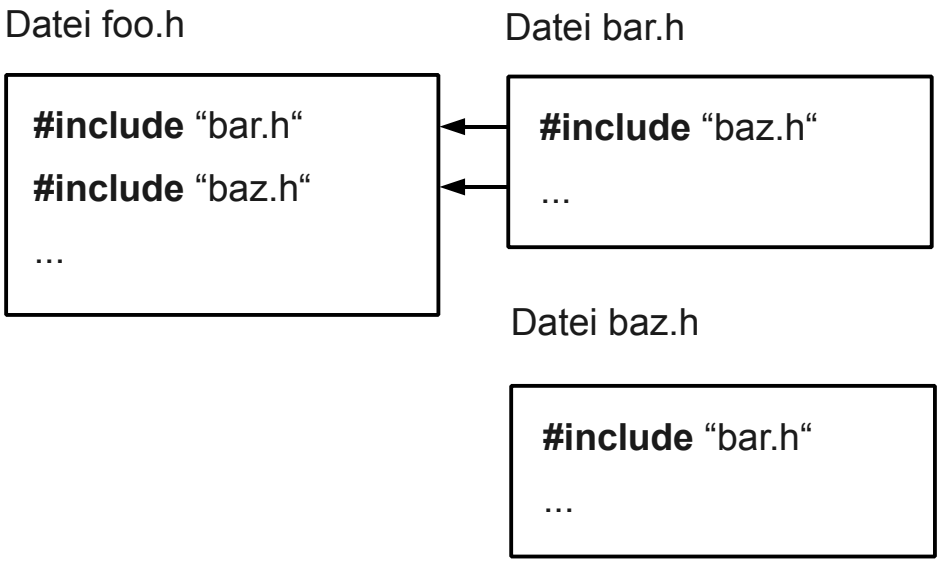
Datei baz.h

```
...
```

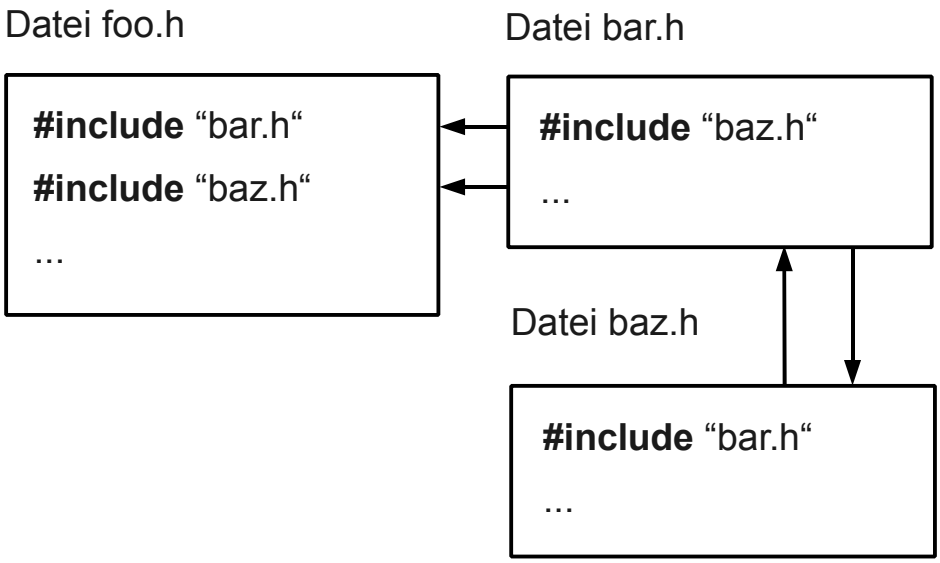
Problem: Mehrfachinklusion



Problem: Mehrfachinklusion



Problem: Mehrfachinklusion

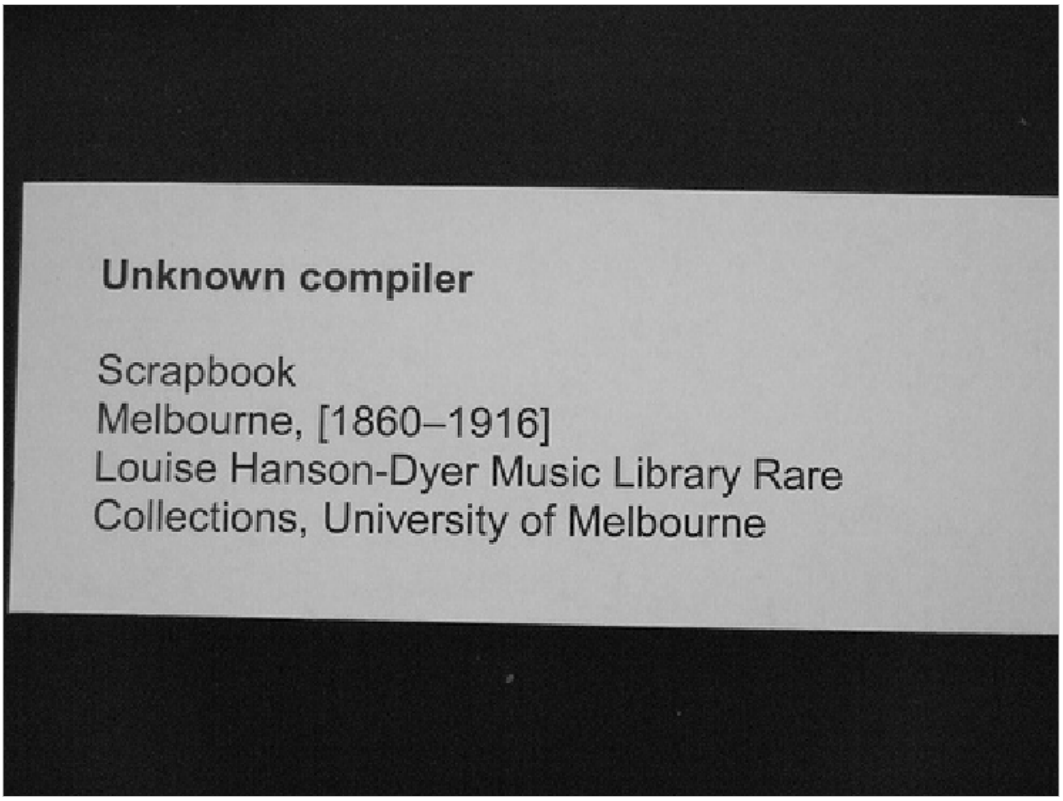


Vermeidung von Mehrfachinklusion

Datei foo.h

```
#ifndef FOO_H
  #define FOO_H

  extern int foo(int x, int y);

#endif
```


Beim ersten Einfügen des Inhalts der Headerdatei in die Quelldatei wird gleichzeitig das Symbol FOO_H definiert. Der Versuch, diese Headerdatei ein zweitesmal einzufügen, scheitert an der bedingten Compilierung, da nun das Symbol FOO_H schon definiert ist. Es ist guter Programmierstil, zur Verhinderung von Mehrfach-Einfügen von Headerdateien oben beschriebenes Muster immer zu verwenden.


Unknown compiler
Scrapbook
Melbourne, [1860–1916]
Louise Hanson-Dyer Music Library Rare
Collections, University of Melbourne

Danke!