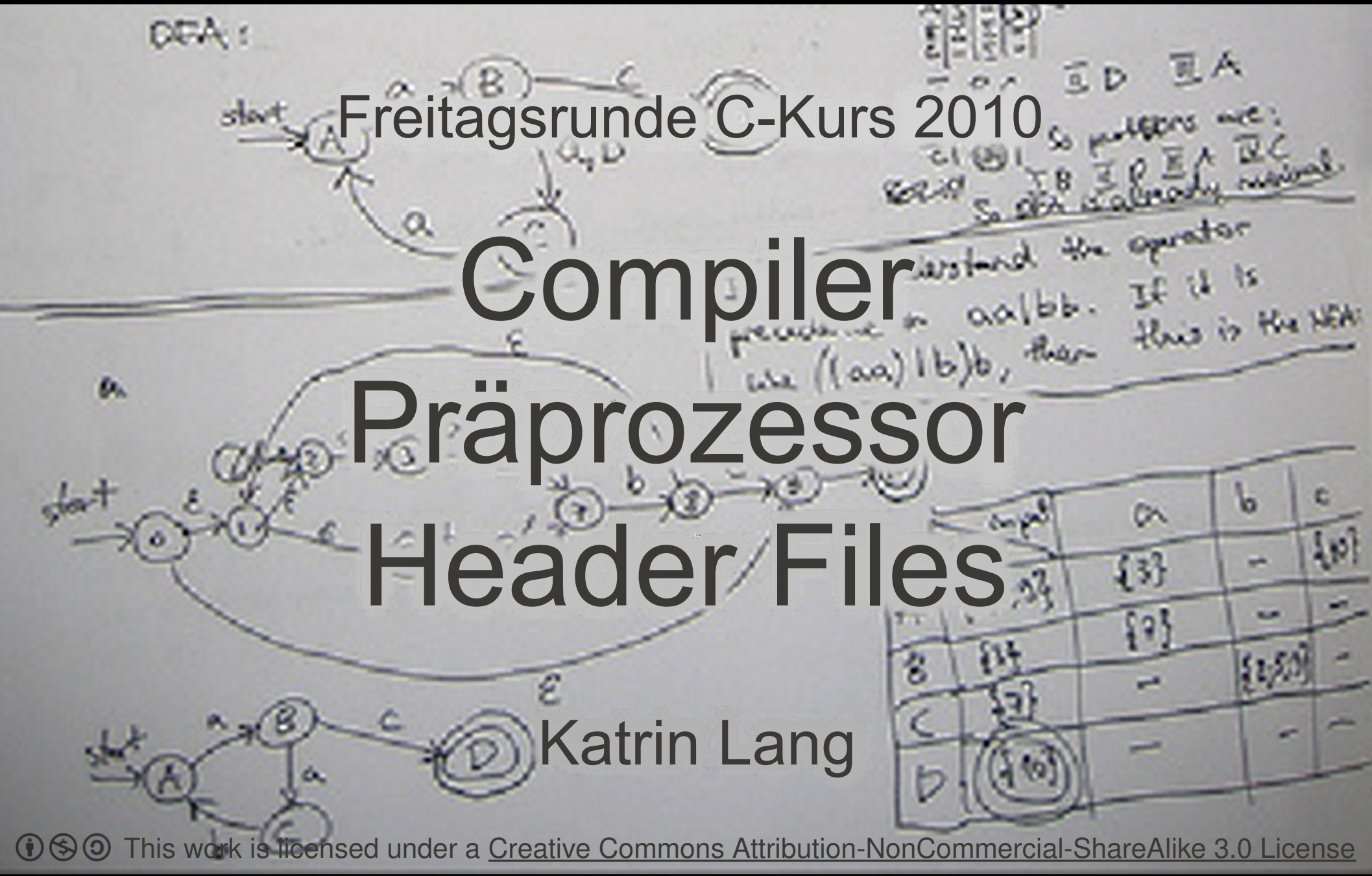
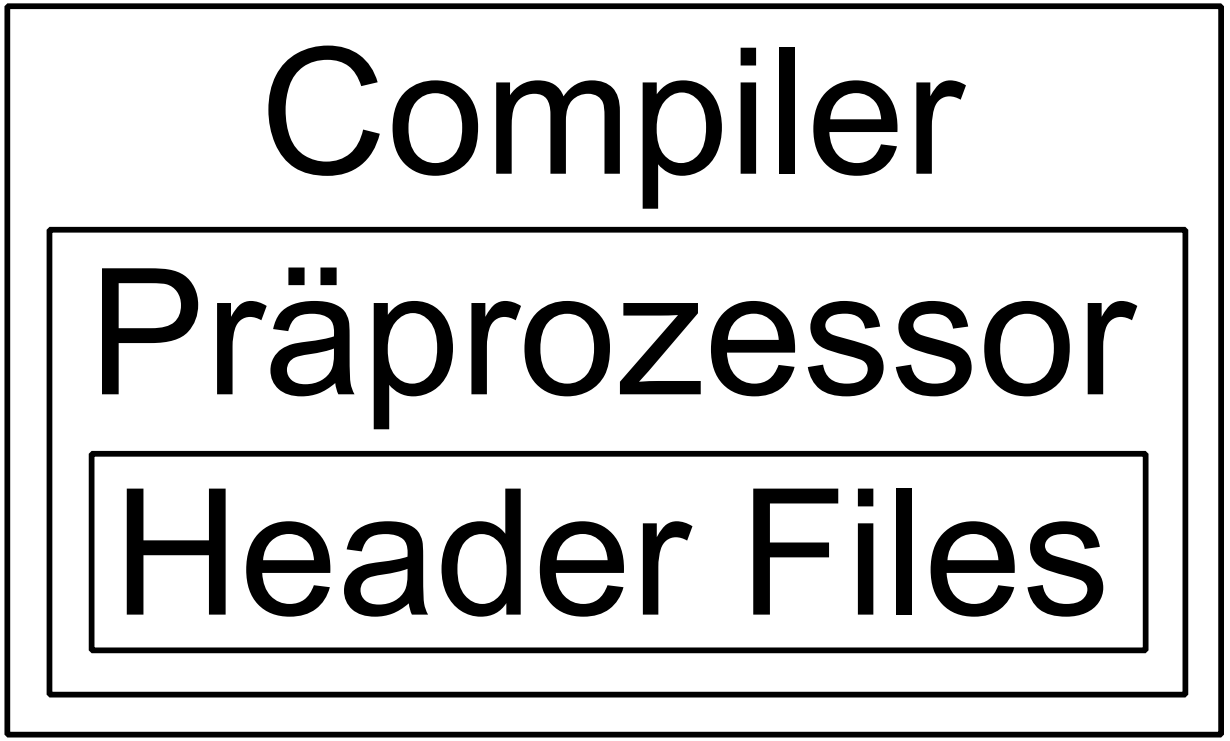


Freitagsrunde C-Kurs 2010

Compiler Präprozessor Header Files

Katrin Lang

Gliederung



Hello World Revisited

Datei hello.c

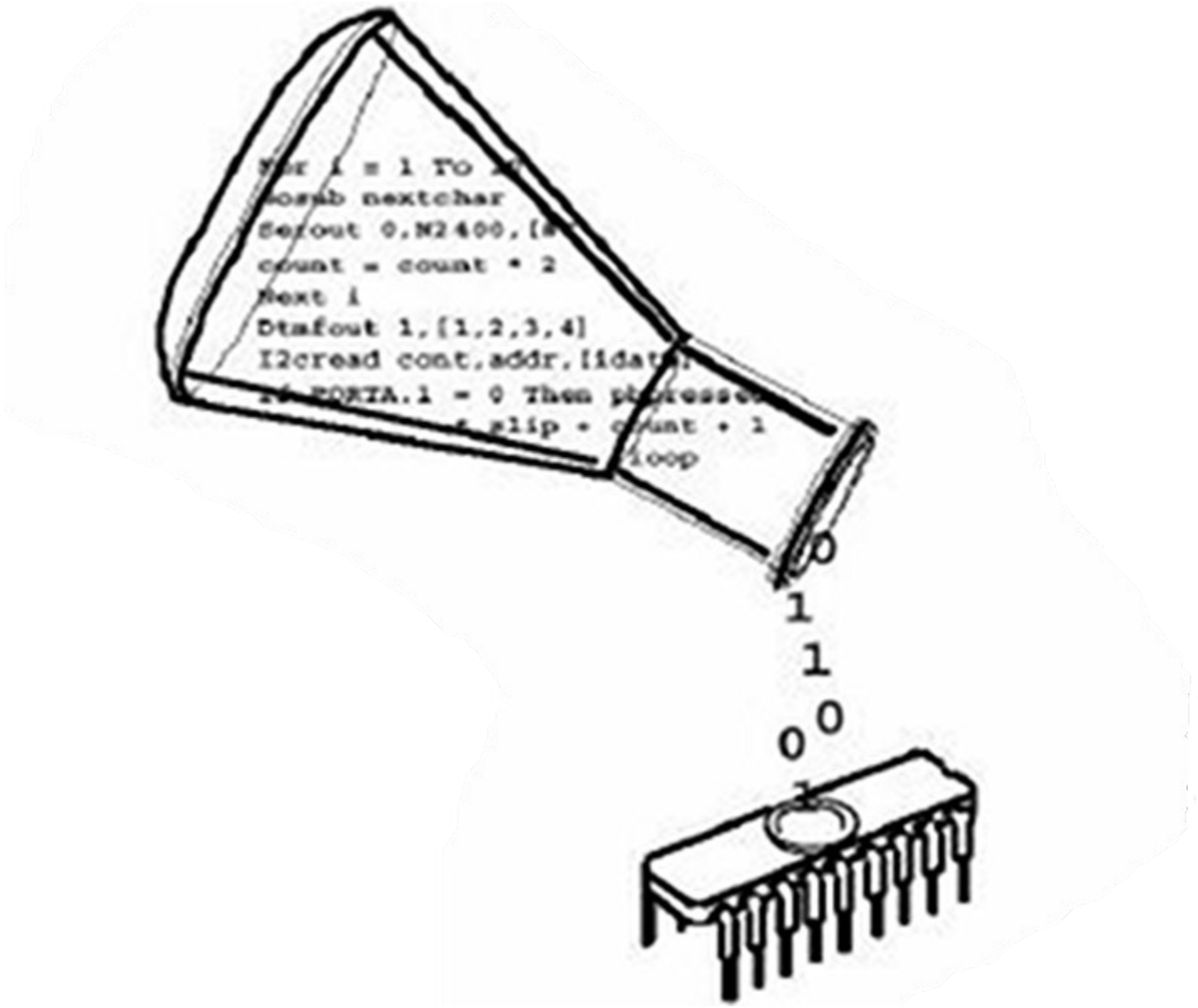
```
#include <stdio.h>

int main(){

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

Kommandozeile

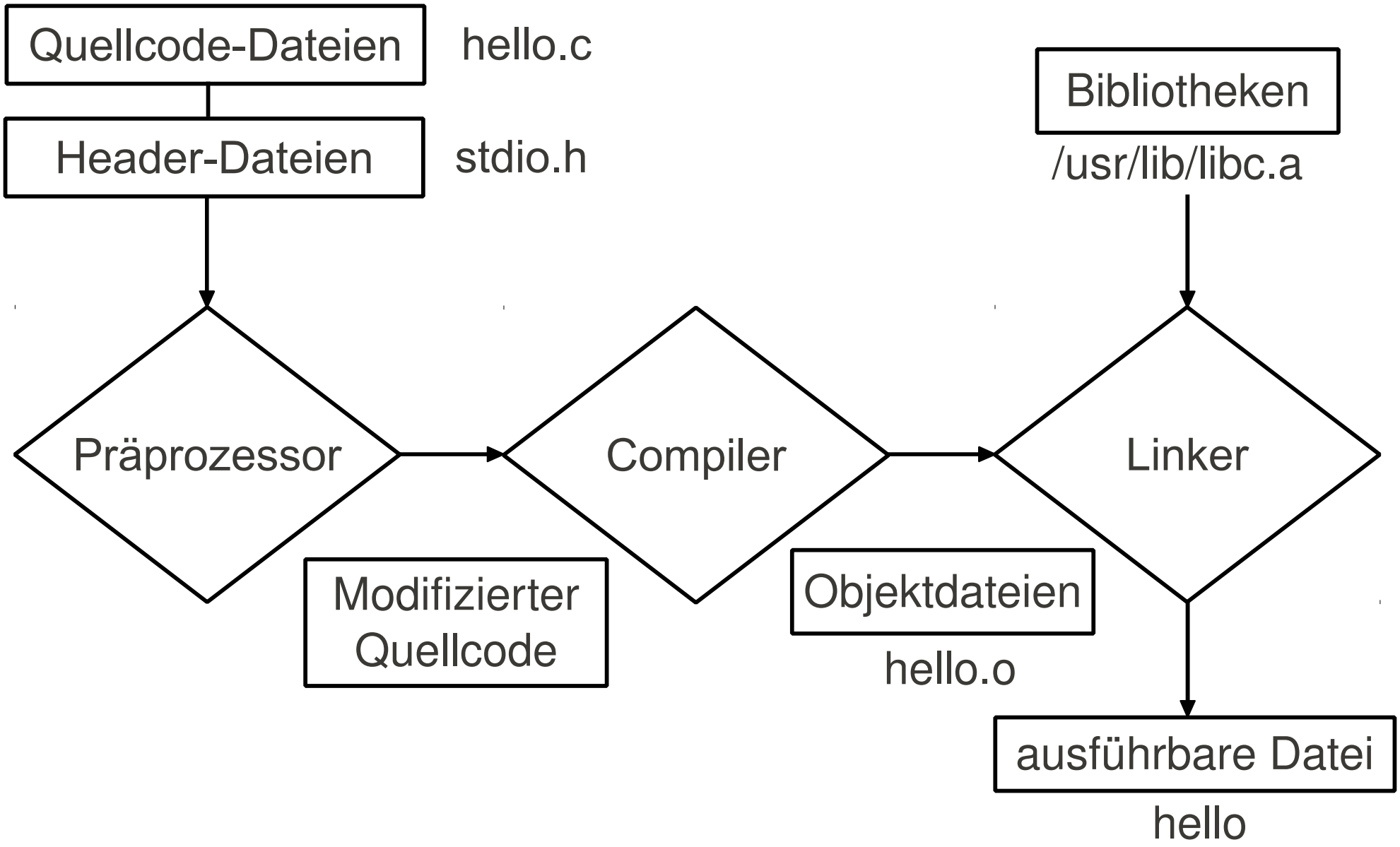
```
$ gcc -o hello hello.c
$ ./hello
Hello, World!
$
```

hello kann nach der Kompilierung wie jedes andere Unix-Kommando ausgeführt 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 main.c
$ gcc -c hello.c
$ gcc -c bye.c
$ ls
bye.c       hello.c         main.c
bye.o       hello.o         main.o
$
```

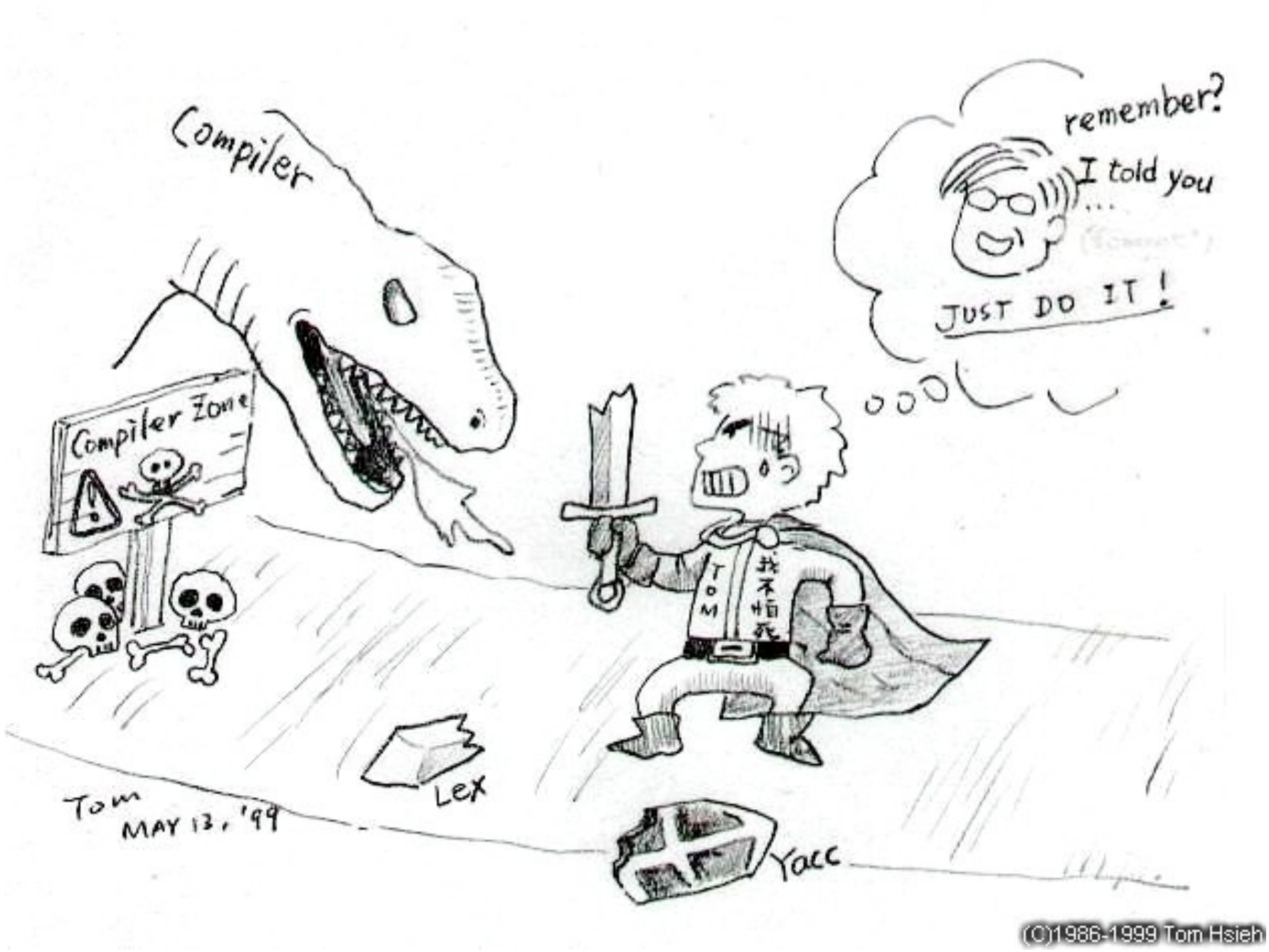
Linken

```
$ 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 `hello'
    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 erkennt nur Zeilen beginnend mit #

Präprozessoranweisungen

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

- Ersetzen von Text (Makros):

  ➔ #define
- Bedingte Kompilierung:

  ➔ #if, #ifdef, #ifndef, #else, #elif, #endif
- Einfügen von Dateien:

  ➔ #include

Parameterlose Makros

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

- Anwendung: Definition von symbolischen Konstanten

```
#define PI 3.141529
```

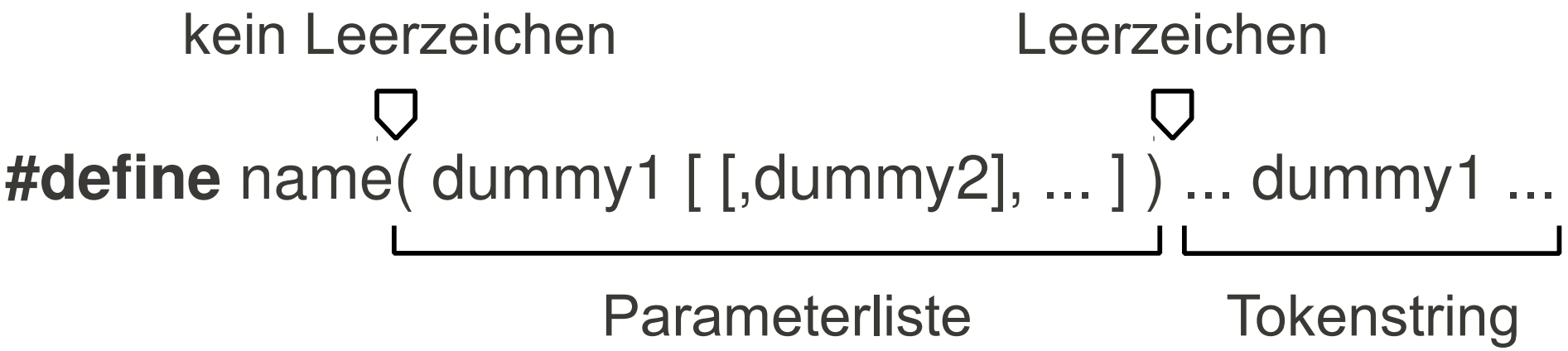
```
#define HALF_PI  PI / 2
```

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

- Erhöht die Lesbarkeit und Wartbarkeit des Programms

Makros mit Parametern

Syntax:



Makros mit Parametern

Expression Makros

- übersetzt in einen Ausdruck
- ähnlich einer Funktion, die einen Wert zurückgibt

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

Statement Makros

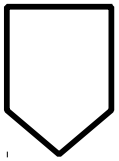
- übersetzt in ein oder mehrere volle C statements
- ähnlich einer Funktion, die void zurückgibt

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

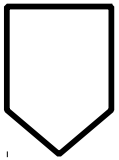
Quiz

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

```
float x= RADTODEG(PI)
```



```
float x= RADTODEG(3.141529)
```



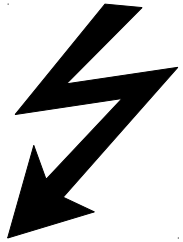
```
float x= ((3.141529) * 57.29578)
```

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

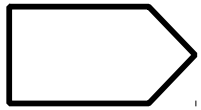
```
#define SWAP(x, y) \
    do{ \
        tmp= x; \
        x= y; \
        y= tmp; \
    } \
    while(0)
```

Warum Klammern?

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

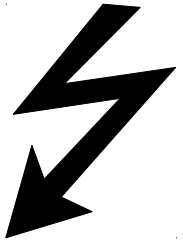


```
RADTODEG(a + b)
```

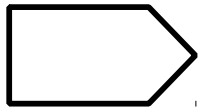


```
(a + b * 57.29578)
```

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



```
1 / RADTODEG(a)
```



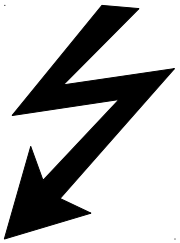
```
1 / (a) * 57.29578
```

Warum kein Semikolon am Ende eines Makros?

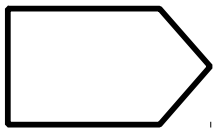
Semikolon beim Aufruf sieht natürlicher aus

➔ Programmierer könnten Semikolons doppelt setzen und damit den Control Flow ändern

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



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

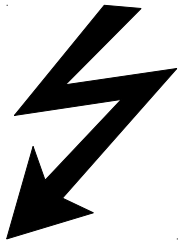


```
if(to_degree)
    y= ((x) * 57.29578);;
else
    y= ((x) * 0.017453);;
```

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 before call.
 */

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

Alternatives SWAP

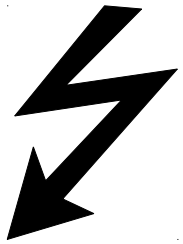
```
/*
 * Swaps two values.
 * Requires type passed as parameter
 */

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

Elegante Lösung (nur gcc)

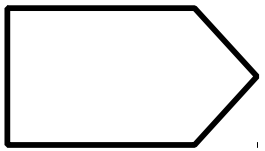
```
/*
 * Swaps two values.
 * Requires gcc typeof extension
 */

#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
```



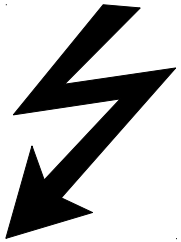
```
int x, y, tmp;
if (x > y)
    SWAP(x, y);
```



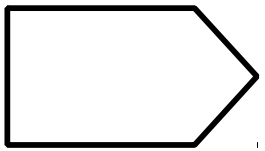
```
int x, y, tmp;
if(x > y)
    tmp= x;
x= y;
y= tmp;
```

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
    ...
```

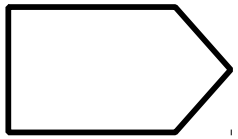
Zusammenfassung

- 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

Mehrfache Seiteneffekte

```
#define MIN(a, b) ((a)<(b)?(a):(b))
```

```
int i= 1;
int j= 2;
i= MIN(++i, ++j);
```



```
int i= 1;
int j= 1;
i= ((++i)<(++j)?(++i):(++j));
```

Mehrfache Seiteneffekte vermeiden (nur c99)

```
#define MIN(a, b) \
    do{
        typeof(a) tmp_a= a;
        typeof(b) tmp_b= b;
        ((tmp_a)>(tmp_b)?(tmp_b):(tmp_a))
    while(0)
```

Weitere Probleme

- das Symbol erscheint nicht mehr im Debugger
- Makros können sich ungewollt überschreiben
- Makros werden erst expandiert und daher vom Compiler überprüft, wenn sie tatsächlich aufgerufen werden

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!

Funktionen statt Makros

```
const double PI= 3.141529;

inline double rad_to_deg(double x){
    return x*57.29578;
}

int main(){
    printf("PI %f", rad_to_deg(PI));
    return 0;
}
```

```
$ gcc  -std=c99 -o angles angles.c
```

Lokaler Scope

```
static const double PI= 3.141529;

static double rad_to_deg(double x){
    return x*57.29578;
}

int main(){
    printf("PI %f", rad_to_deg(PI));
    return 0;
}
```

```
$ gcc  -std=c90 -o angles angles.c
```

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=2 hello.c

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;
}
```

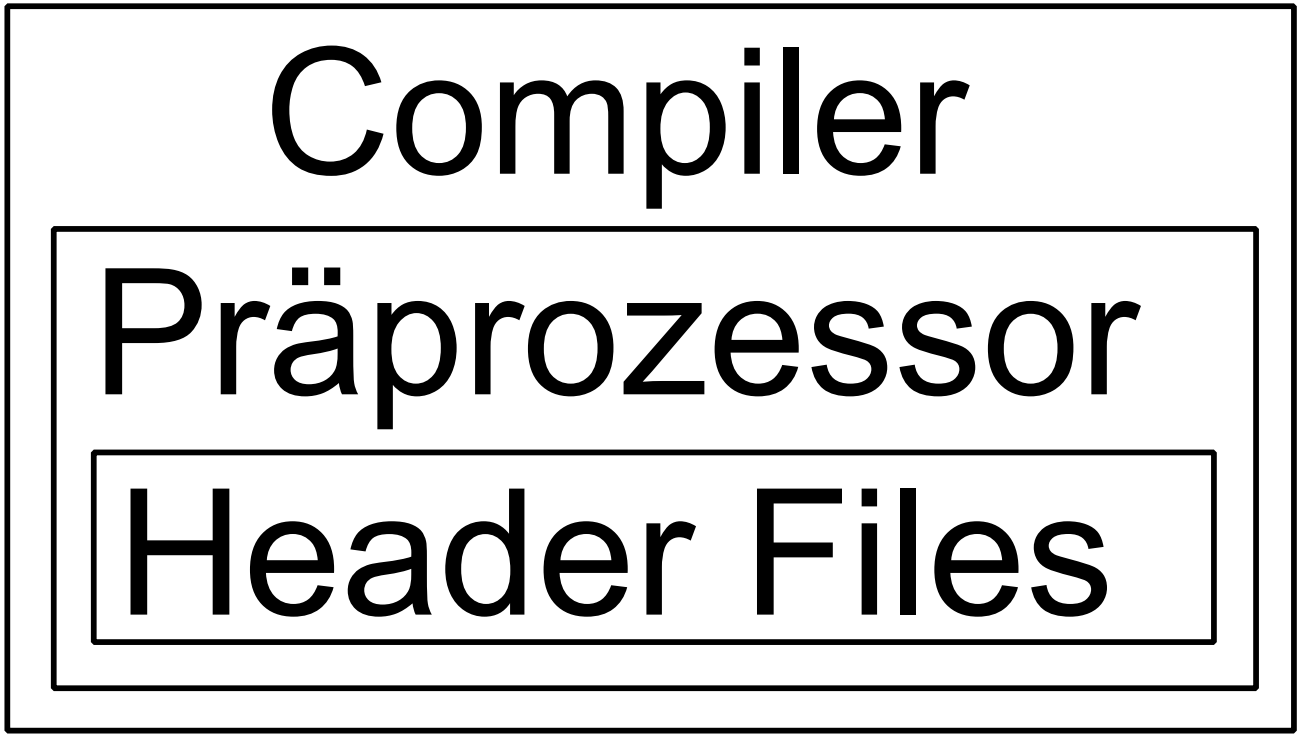
Debug Level als Argument (C99)

Datei debug.h

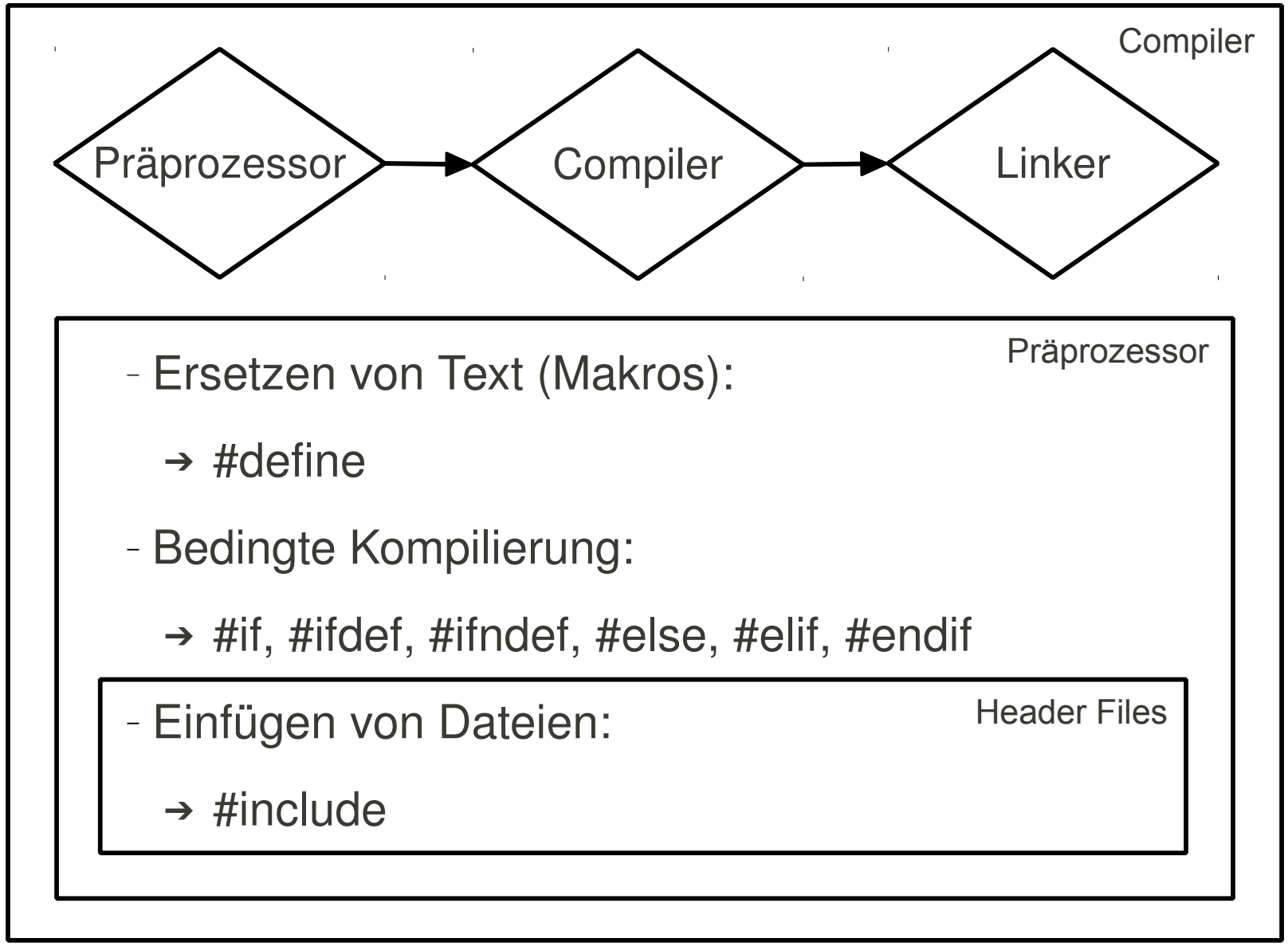
```
#include <stdio.h>

#ifdef DEBUG
#define LOG(level, ...) \
    if(level <= VERBOSE) \
        fprintf(stderr, __VA_ARGS__)
#else
#define LOG(level, ...)
#endif
```

Datei debug.h

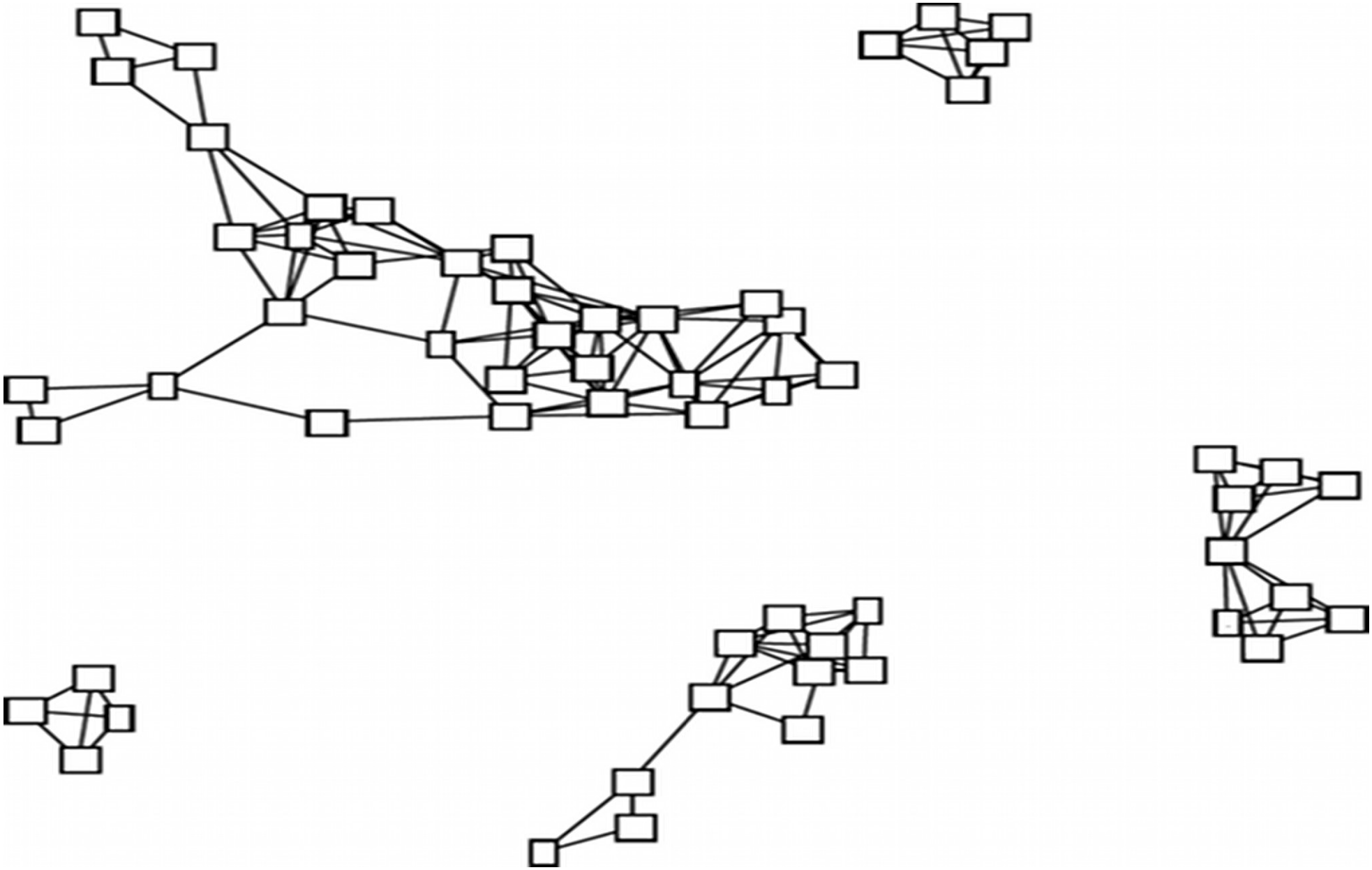
```
#include <stdio.h>

#define WHERESTR  "[file %s, function %s, line %d]: "
#define WHEREARG  __FILE__, __func__, __LINE__

#ifdef DEBUG
#define LOG(level, fmt, ...) \
      if(level <=VERBOSE) \
            fprintf(stderr, WHERESTR fmt, WHEREARG, __VA_ARGS__)
#else
#define LOG(level, fmt, ...)
#endif
```


Compiler
Präprozessor
Header Files

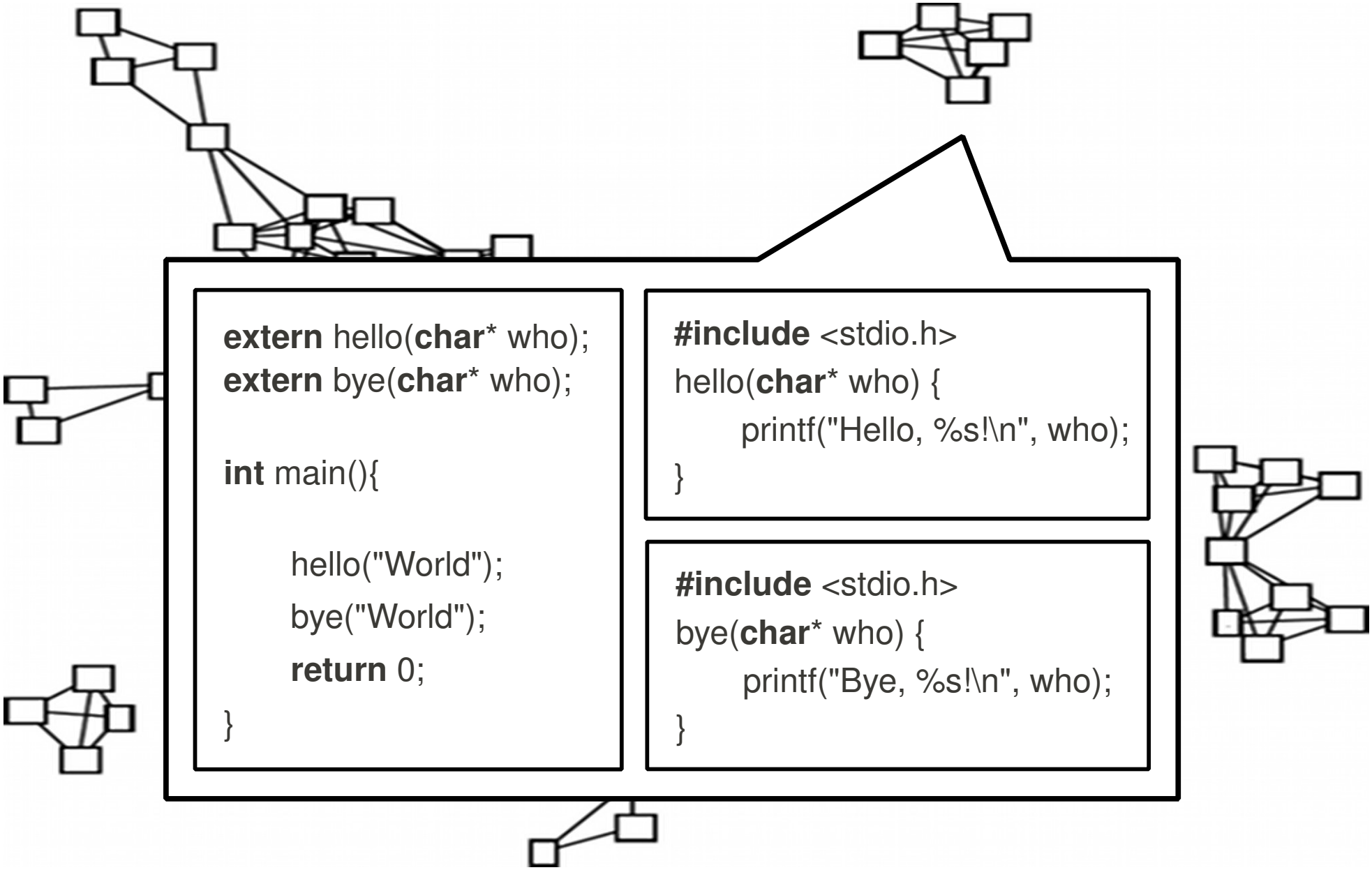
Wiederholung



Header-Dateien



Header-Dateien



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

int main(){

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

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

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

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")

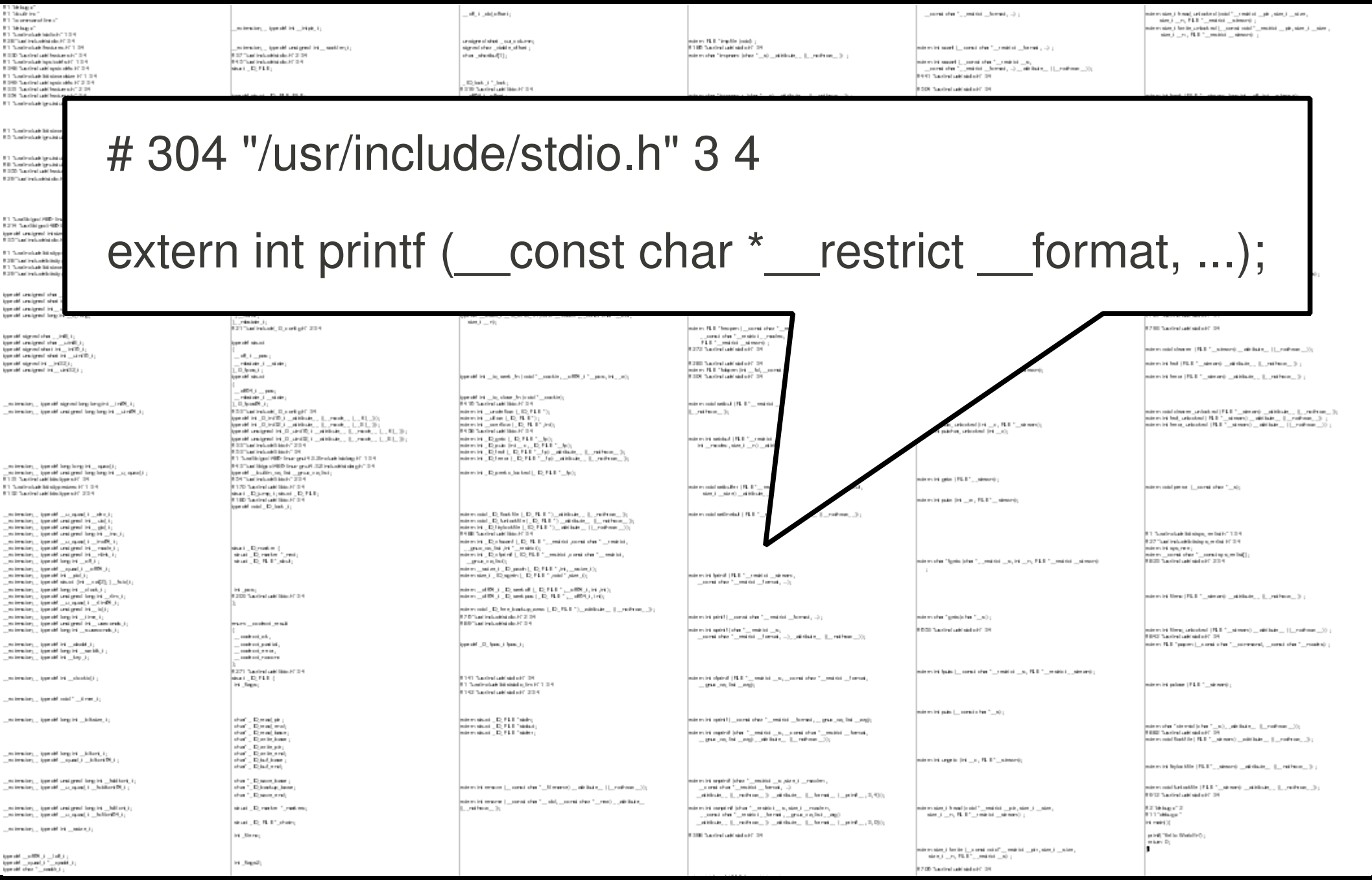
-I-Compileroption (gcc)

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

- gcc -**I**include hello.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

```
...
```

Datei baz.h

```
...
```

Problem: Mehrfachinklusion

Datei foo.h

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

Datei bar.h

```
#include “baz.h“
...
```

Datei baz.h

```
...
```

Problem: Mehrfachinklusion

Datei foo.h

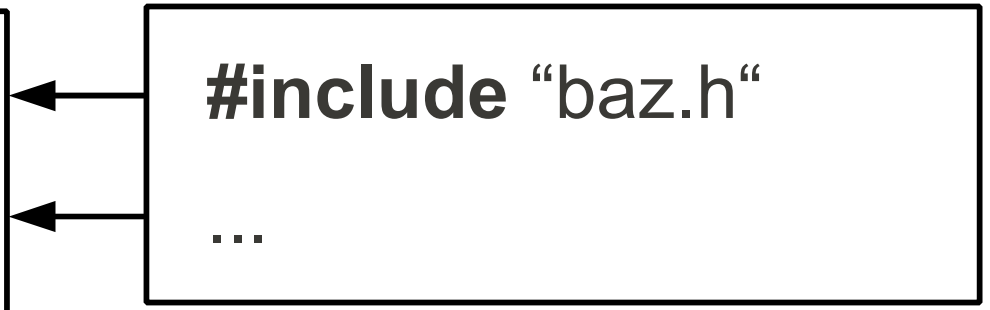
```
#include “bar.h“
#include “baz.h“
...
```

Datei bar.h

```
#include “baz.h“
...
```

Datei baz.h

```
...
```



Problem: Mehrfachinklusion

Datei foo.h

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

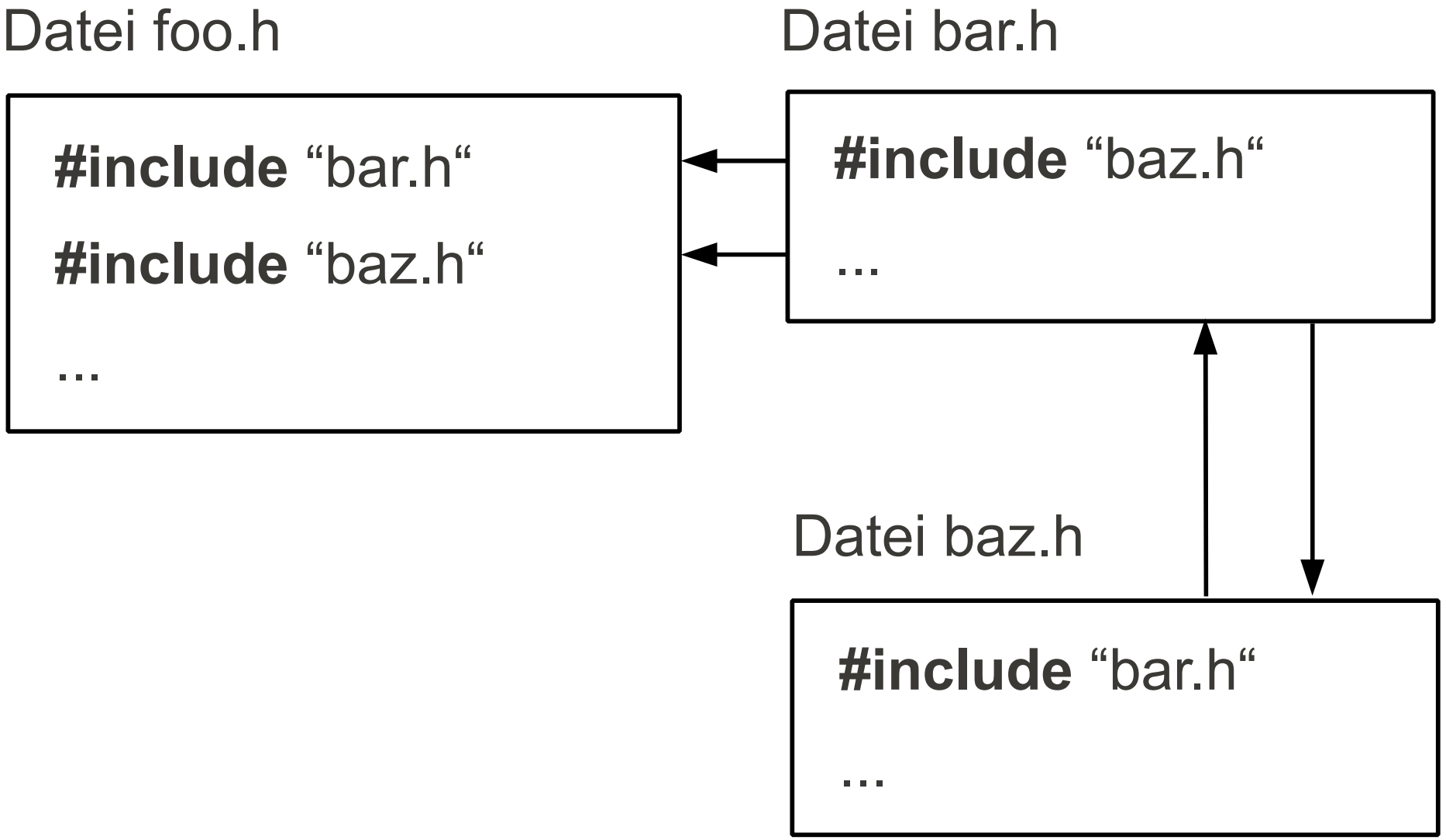
Datei bar.h

```
#include “baz.h“
...
```

Datei baz.h

```
#include “bar.h“
...
```

Problem: Mehrfachinklusion



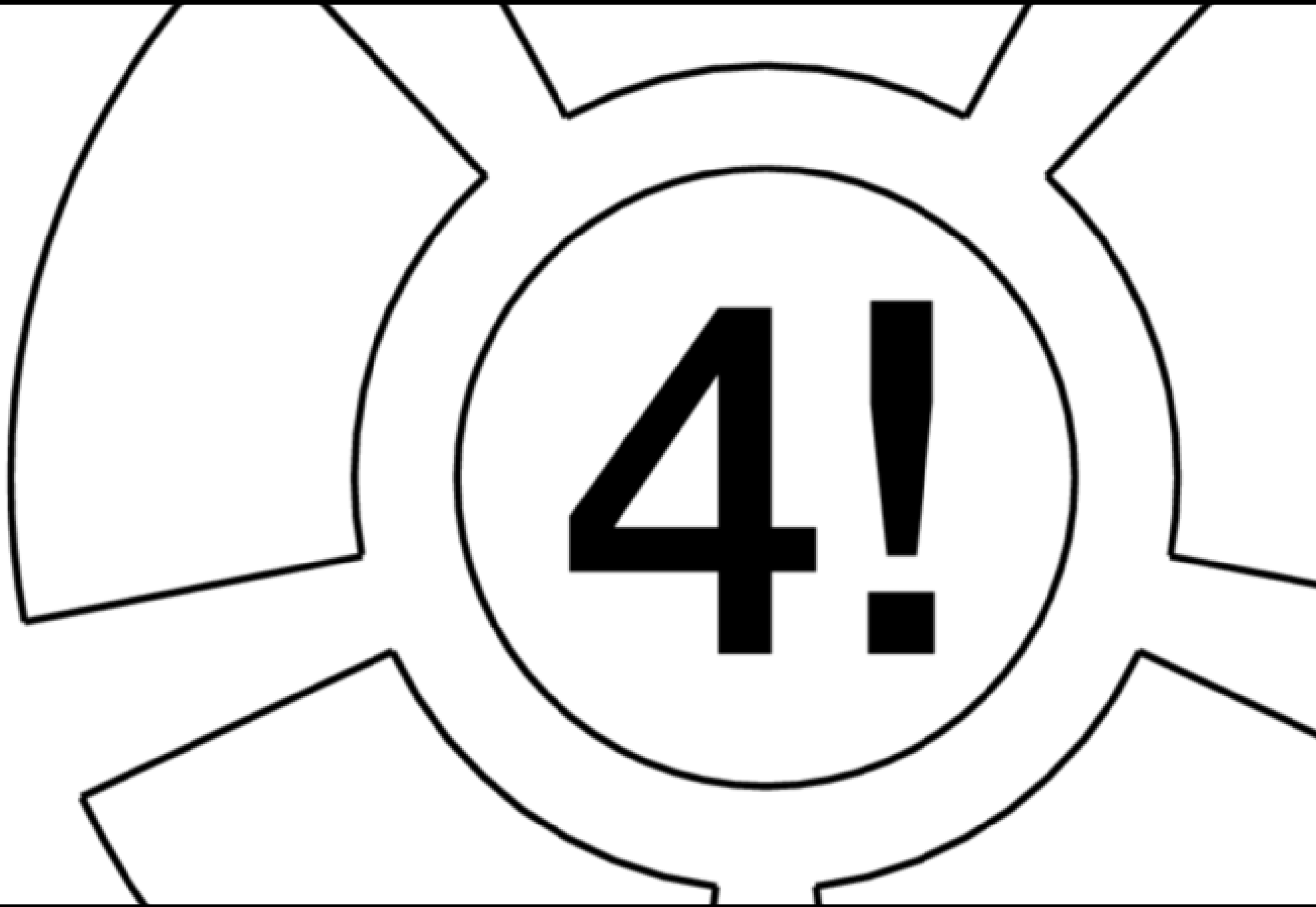
Vermeidung von Mehrfachinklusion

Datei foo.h

```
#ifndef FOO_H
   #define FOO_H


   extern int foo(int x, int y);


#endif
```

Unknown compiler

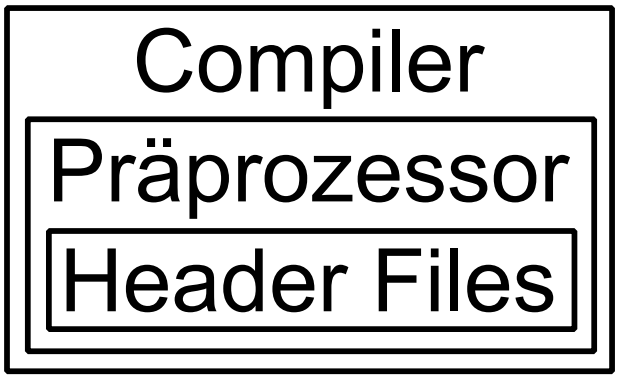
Scrapbook
Melbourne, [1860–1916]
Louise Hanson-Dyer Music Library Rare Collections, University of Melbourne


4!

Freitagsrunde C-Kurs 2010

Compiler
Präprozessor
Header Files

Katrin Lang


Compiler
Präprozessor
Header Files

Hello World Revisited

Datei hello.c

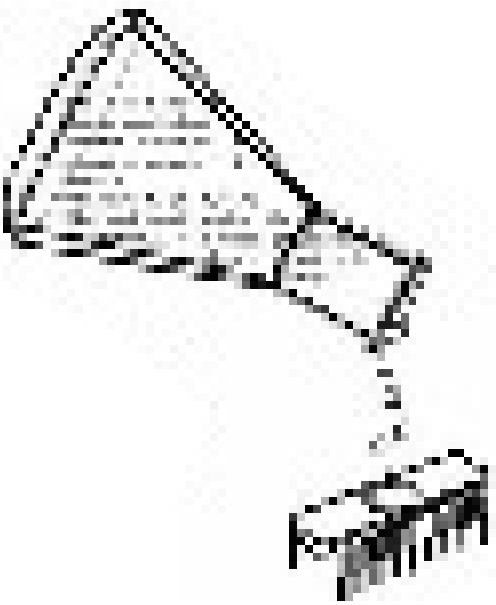
```
#include <stdio.h>

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 ausgeführt werden

Der C-Compiler

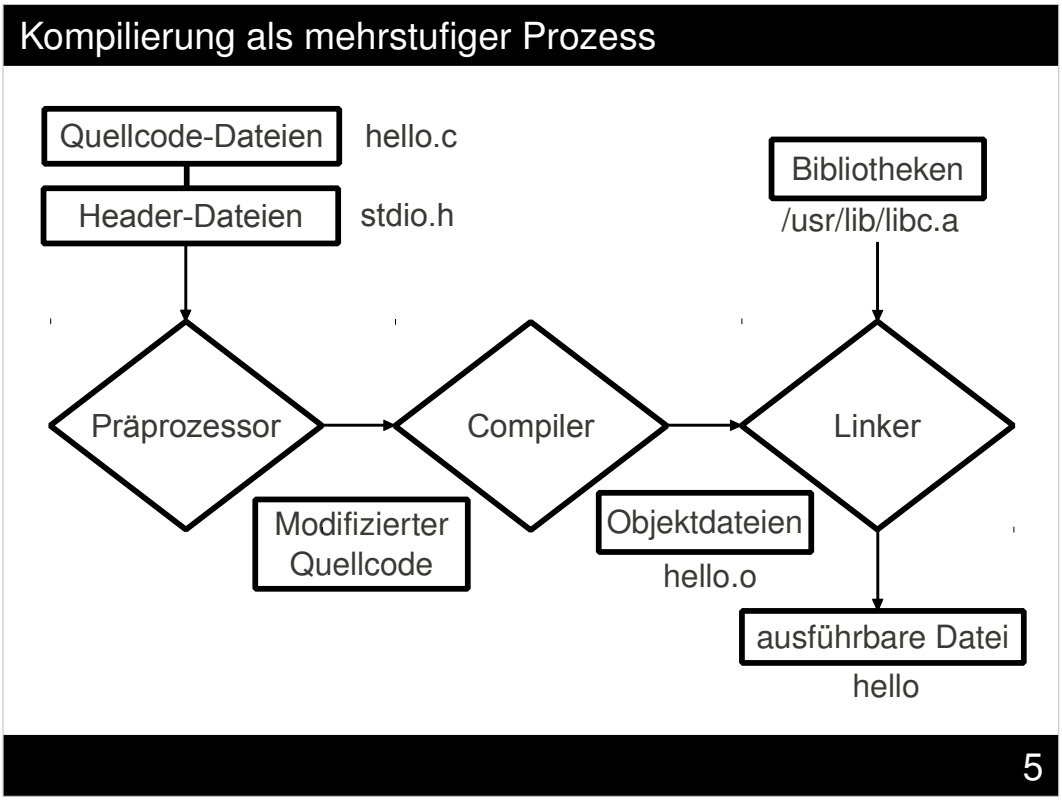


4

Dass man hello nach der Kompilierung wie jedes andere Unix-Kommando ausgeführen kann, liegt daran, dass der C-Compiler im Gegensatz zum Java-Compiler Maschinencode generiert, der direkt auf der Hardware läuft.

In Java musste man ja vor dem Programmaufruf immer java schreiben. Mit diesem Kommando wird die Java Virtual Maschine gestartet, eine Art Software-Prozessor, der den Java-Bytecode ausführt, den der Java-Compiler generiert hat.

Ein Prozessor in Software ist natürlich langsamer als ein richtiger Prozessor. Daher ist C schneller als Java, hat aber den Nachteil, dass man den Code für jede Architektur neu kompilieren muss, während Java-Bytecode überall käuft, wo eine Virtual Machine installiert ist.



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. Mit der -S Option kann man den Compiler dazu bringen, solch ein File zu schreiben und es sich ansehen.

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

7

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“.

Der Name der ausführbaren Datei ist übrigens beliebig. Sie muss nicht genauso heissen, wie die Datei, in der die main-Funktion steht.

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

Der Präprozessor

10

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 erkennt nur Zeilen beginnend mit #

Präprozessoranweisungen

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

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

Parameterlose Makros

- Syntax: **#define** name [replacement]
- Anwendung: Definition von symbolischen Konstanten

```
#define PI 3.141529
```

```
#define HALF_PI  PI / 2
```

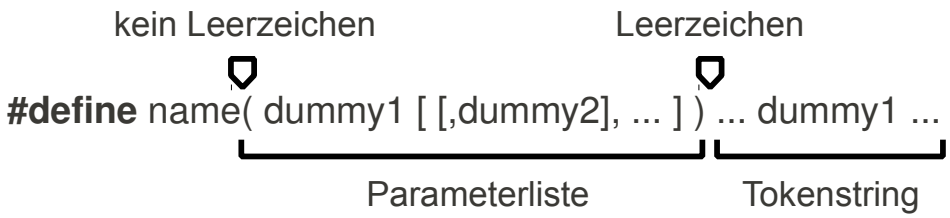
- Präprozessor ersetzt vor der Kompilierung jedes Vorkommen von PI mit 3.141529
- Erhöht die Lesbarkeit und Wartbarkeit des Programms

Symbolische Konstanten erhöhen die Lesbarkeit dadurch, dass nicht mehr im gesamten Quellcode „Magic Numbers“ verstreut sind. Ausserdem ist das Programm leichter wartbar: Wird PI in einer höheren Genauigkeit benötigt, kann diese Änderung zentral an einer Stelle vorgenommen werden.

Der Ersetzungstext kann seinerseits auch wieder Makros enthalten. Dies wird gelöst, indem der Präprozessor jede Zeile mehrmals scannt: Im ersten Pass wird HALF_PI durch PI / 2 ersetzt, im zweiten PI durch 3.141529

Makros mit Parametern

Syntax:



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

Makros mit Parametern

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)
```


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)

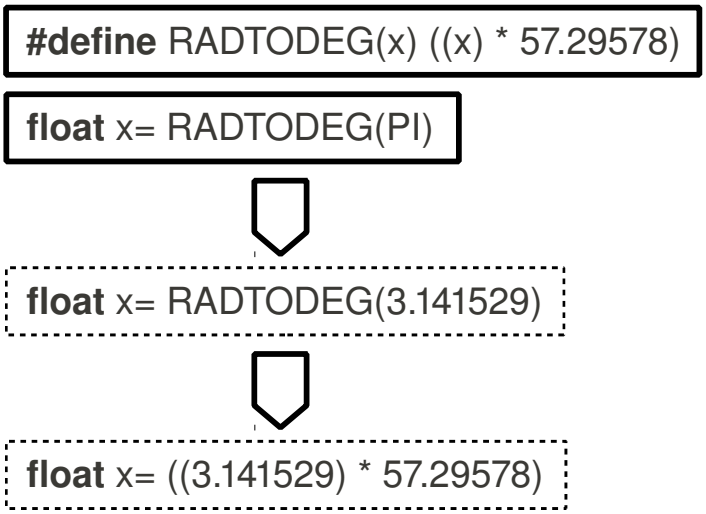
Der Präprozessor substituiert dann den entstandenen Tokenstring für den Makrouafruf.

Dieser Ersatz betrifft nur Terminalsymbole (also eigenständige Vorkommen von Bezeichnern 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. Diese werden dann im zweiten Pass des Präprozessors über die Zeile auf gleiche Weise expandiert.

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.

Quiz


#define RADTODEG(x) ((x) * 57.29578)
float x= RADTODEG(PI)
float x= RADTODEG(3.141529)
float x= ((3.141529) * 57.29578)

Quiz

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

```
#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 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 before call.
 */

#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)
```

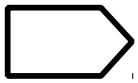
Elegante Lösung (nur gcc)

```
/*
 * 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
    ...
```

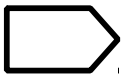
Zusammenfassung

- 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

Mehrfache Seiteneffekte

```
#define MIN(a, b) ((a)<(b)?(a):(b))
```

```
int i= 1;
int j= 2;
i= MIN(++i, ++j);
```



```
int i= 1;
int j= 1;
i= ((++i)<(++j)?(++i):(++j));
```

Mehrfache Seiteneffekte vermeiden (nur c99)

```
#define MIN(a, b) \
    do{
        typeof(a) tmp_a= a;
        typeof(b) tmp_b= b;
        ((tmp_a)>(tmp_b)?(tmp_b):(tmp_a))
    while(0)
```

Weitere Probleme

- das Symbol erscheint nicht mehr im Debugger
- Makros können sich ungewollt überschreiben
- Makros werden erst expandiert und daher vom Compiler überprüft, wenn sie tatsächlich aufgerufen werden

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!

Funktionen statt Makros

```
const double PI= 3.141529;

inline double rad_to_deg(double x){
    return x*57.29578;
}

int main(){
    printf("PI %f", rad_to_deg(PI));
    return 0;
}
```

```
$ gcc  -std=c99 -o angles angles.c
```

31

Nachteile an diesem Code:

PI kann nicht einfach wie ein Makro ersetzt werden, da es eine globale Variable ist (leider ist in c global Scope der default).

Leider sind des weiteren in Konstanten keine Konstanten. Der Speicherbereich kann trotzdem geschrieben werden, nur eben nicht über eine Variable, die als const deklariert ist.

Und da wir Quellcodedateien separat kompilieren können (die Konstante könnte z.B. in einer Winkelberechnungsbibliothek enthalten sein), können wir nicht vorhersehen, wer später noch Zugriff auf unsere Variable braucht. Deswegen muss zur Laufzeit physikalisch Speicher für die Konstante vorhanden sein.

Inlinen kann oft genau den gegenteiligen Effekt haben: Es bläht den Code auf, der dadurch nicht mehr in den schnellen, kleinen Cache-Speicher passt. Dadurch muss das Programm häufiger auf langsameren Speicher zurückgreifen und wird langsamer statt schneller, obwohl es weniger Funktionsaufrufe enthält.

Lokaler Scope

```
static const double PI= 3.141529;

static double rad_to_deg(double x){
    return x*57.29578;
}

int main(){
    printf("PI %f", rad_to_deg(PI));
    return 0;
}
```

```
$ gcc  -std=c90 -o angles angles.c
```


Das Optimieren sollte man generell dem Compiler überlassen. Der kann das besser als ein Mensch, da er genau weiss, wie oft eine Funktion aufgerufen wird, und ob es sich lohnt, sie zu inlinen. Der Compiler inlined sowieso, wenn er das für nötig hält, auch ohne explizites inline.

Das geht aber nur, wenn er weiss, wie oft eine Funktion aufgerufen wird. Bei einer globalen Funktion kann er das nicht sagen (separate Kompilierung, siehe vorangegangene Folie). Deswegen sollte man grundsätzlich vor alle Funktionen, die nicht unbedingt global sein müssen, ststic schreiben.
Das beugt ausserdem Namenskonflikten vor.

Static hat übrigens nichts mit dem Java-static zu tun, es entspricht vielmehr dem private in Java und bedeutet, dass das Symbol nur in der aktuellen Datei sichtbar ist.

Und jetzt kann der Compiler auch das PI wegoptimieren (d.h. Direkt 3.141529 in den Quellcode einsetzen), weil er weiss, dass es niemand ausserhalb dieses File braucht.

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=2 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.)

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;
}
```

Debug Level als Argument (C99)

Datei debug.h

```
#include <stdio.h>

#ifdef DEBUG
#define LOG(level, ...) \
    if(level <= VERBOSE) \
        fprintf(stderr, __VA_ARGS__)
#else
#define LOG(level, ...)
#endif
```


Eigentlich möchte man das Loglevel als Argument übergeben. Das geht aber nur mit c99, da printf beliebig viele Argumente nehmen kann. Dafür steht das … __VA_ARGS__ enthält dann alle diese Argumente

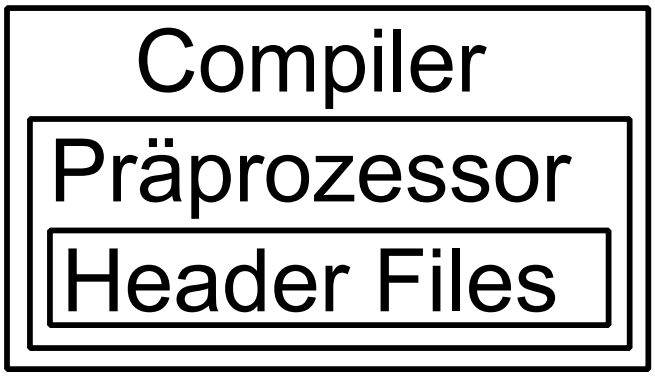
Fancy Debugging Macro (C99)

Datei debug.h

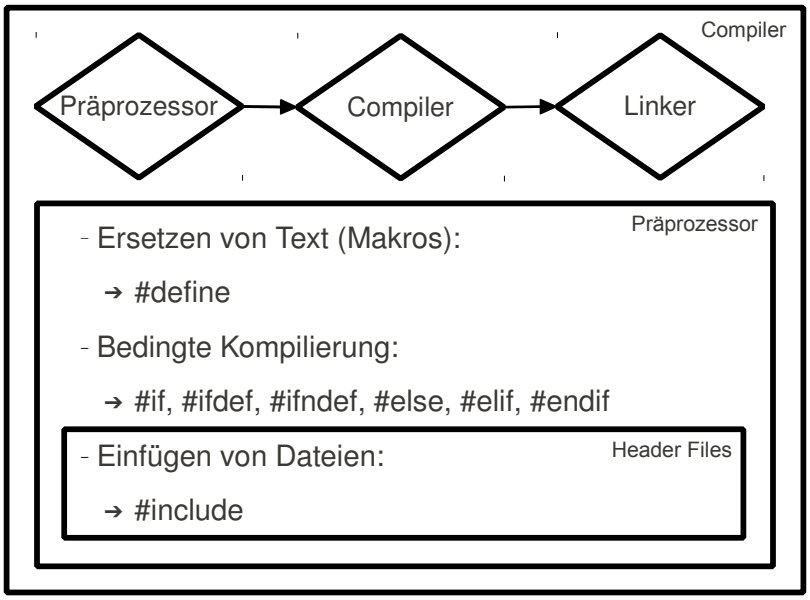
```
#include <stdio.h>

#define WHERESTR  "[file %s, function %s, line %d]: "
#define WHEREARG  __FILE__, __func__, __LINE__

#ifdef DEBUG
#define LOG(level, fmt, ...) \
     if(level <=VERBOSE) \
          fprintf(stderr, WHERESTR fmt, WHEREARG, __VA_ARGS__)
#else
#define LOG(level, fmt, ...)
#endif
```


Das Ganze kann man noch beliebig weitertreiben, und das File und die Zeilenanzahl mit ausgeben. Dafür gibt es die Makros __FILE__ und __LINE__

__func__ gibt es erst ab c99. Das ist aber kein Makro, da der Präprozessor läuft, bevor Funktionen überhaupt definiert wurden. Gcc kannte auch früher schon __function__, aber das ist nicht portabel.

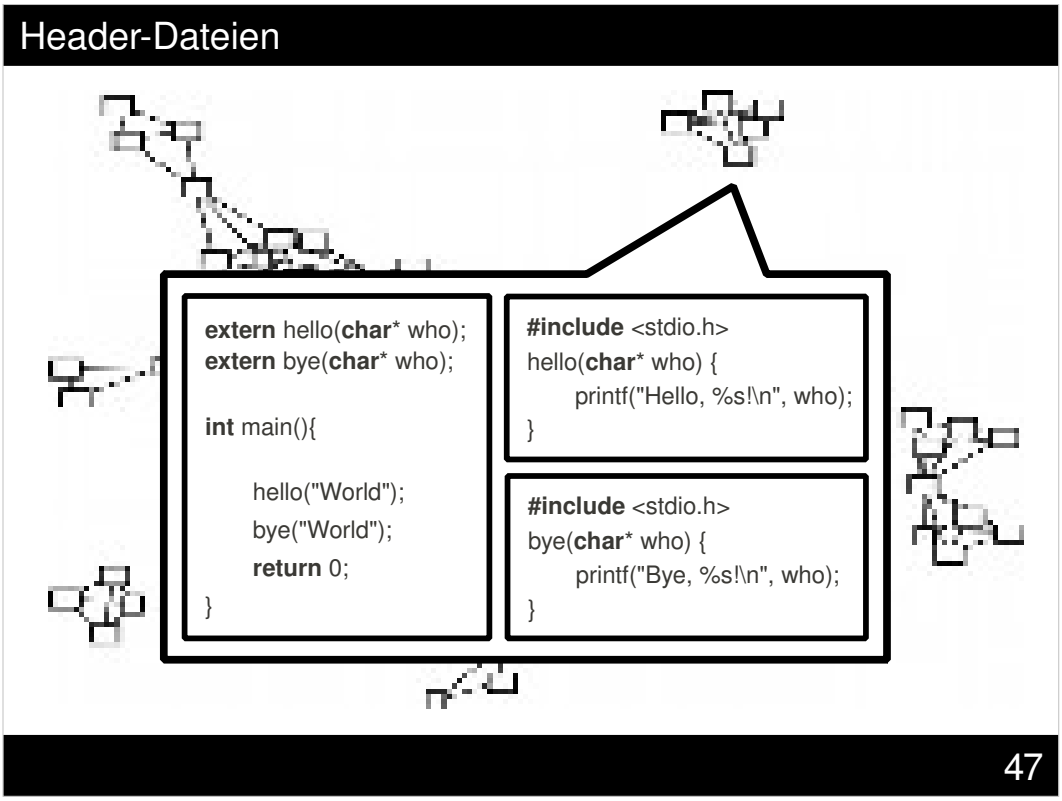

Compiler
Präprozessor
Header Files

Wiederholung



Header-Dateien

Software besteht ist in der Regel komplex und besteht aus vielen Dateien. Diese möchten gerne Funktionen aus anderen Dateien benutzen.



Eines diese Softwaresysteme ist unser Hello World-Beispiel von zu Beginn der Vorlesung. Da hatten wir unsere Funktionen hello und bye in main.c als extern deklariert. Vielleicht finden aber auch andere Programme unsere Funktionen ganz nützlich. Damit nicht jedes File, das die Funktionen benutzt, sie deklarieren muss, packen wir die extern-Deklarationen in eigene Files, die Header Files, und inkludieren sie, wo die Funktionen gebraucht werden.

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")

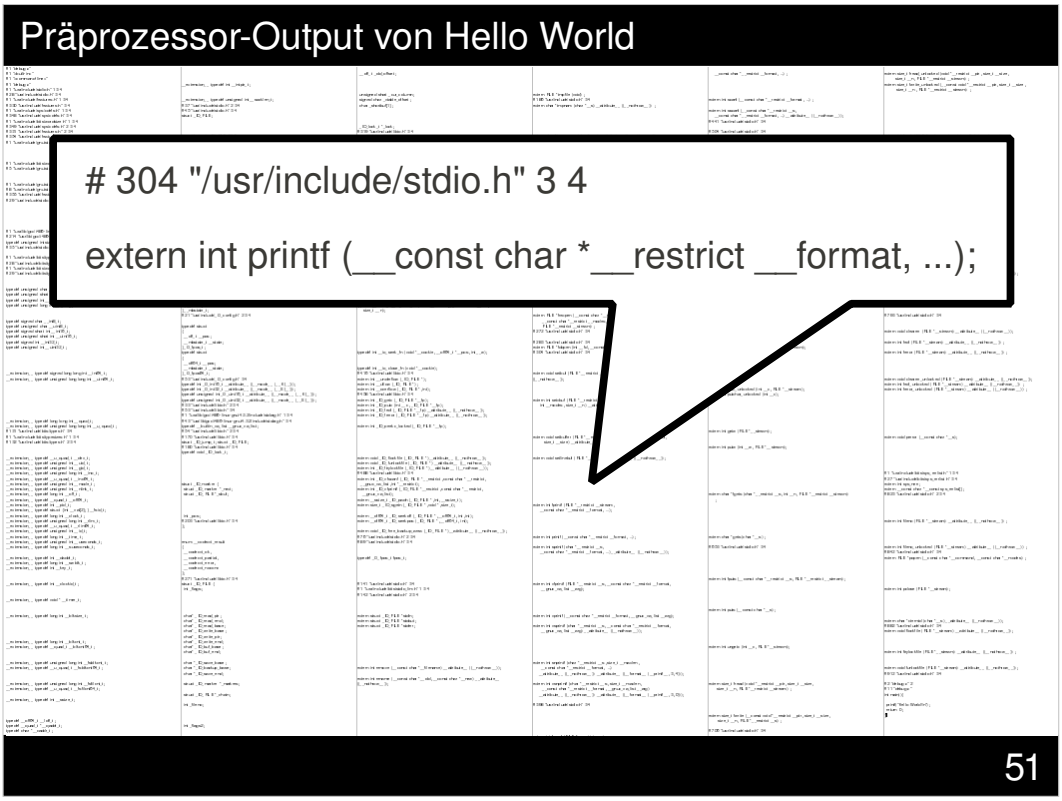
-I-Compileroption (gcc)

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

- gcc -Iinclude hello.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

```
…
```

Datei baz.h

```
…
```

Problem: Mehrfachinklusion

Datei foo.h

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

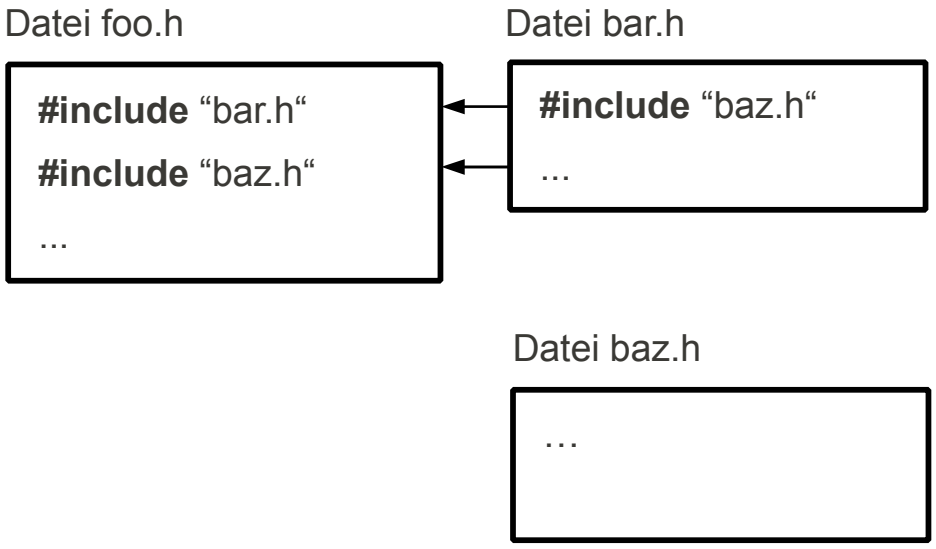
Datei bar.h

```
#include “baz.h“
...
```

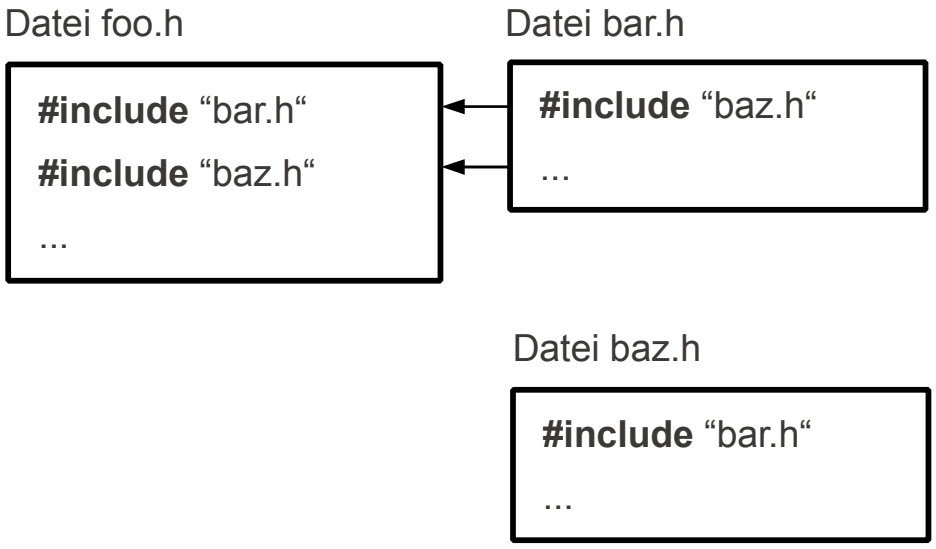
Datei baz.h

```
…
```

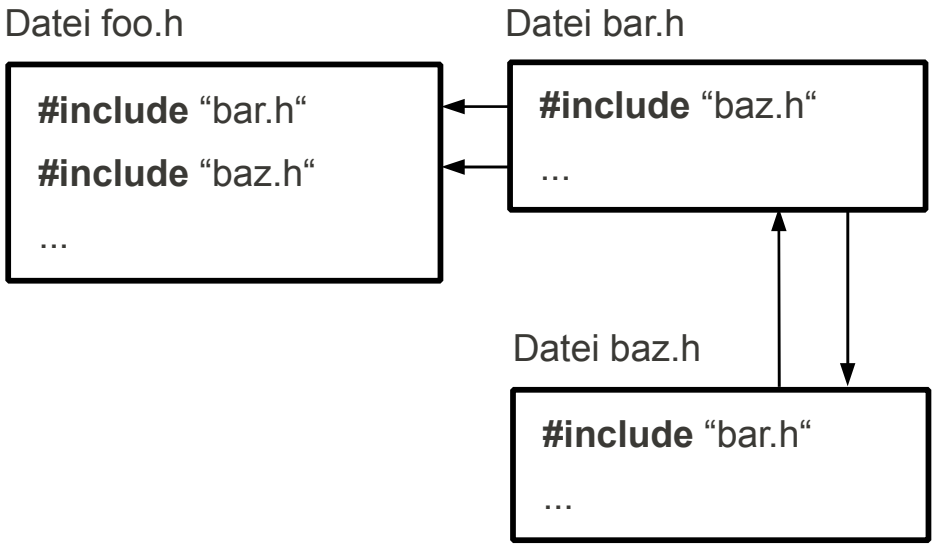
Problem: Mehrfachinklusion



Problem: Mehrfachinklusion



Problem: Mehrfachinklusion

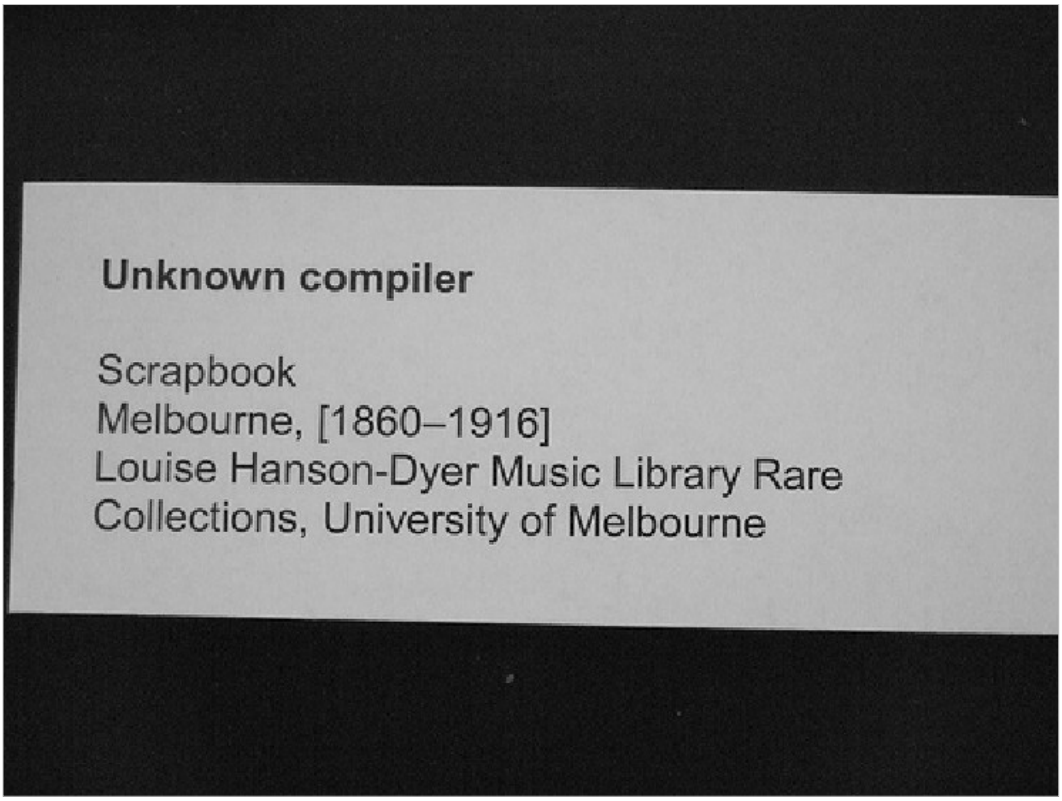


Vermeidung von Mehrfachinklusion

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-Inklusien das beschriebenes Muster immer zu verwenden.


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

Danke!