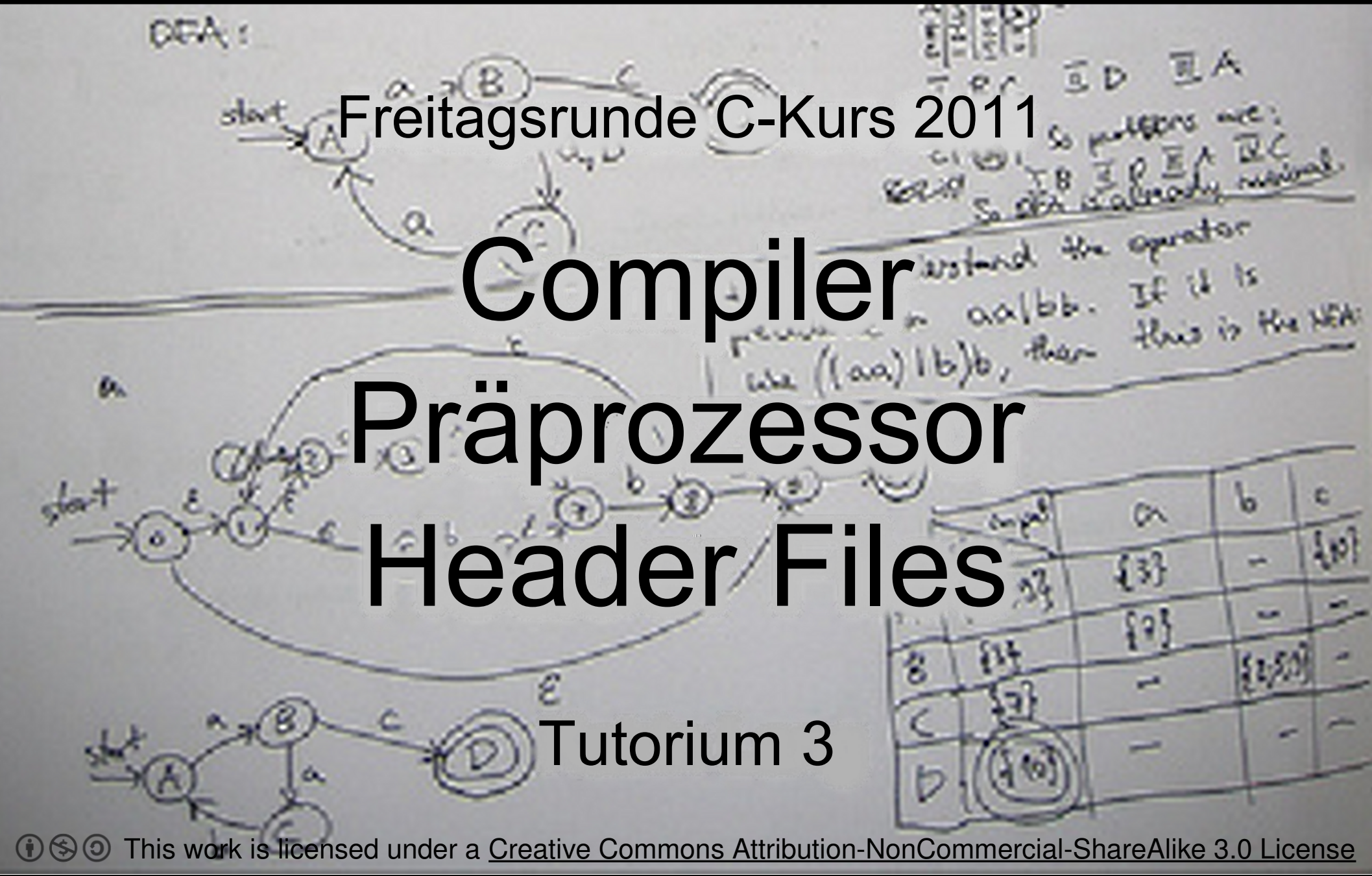
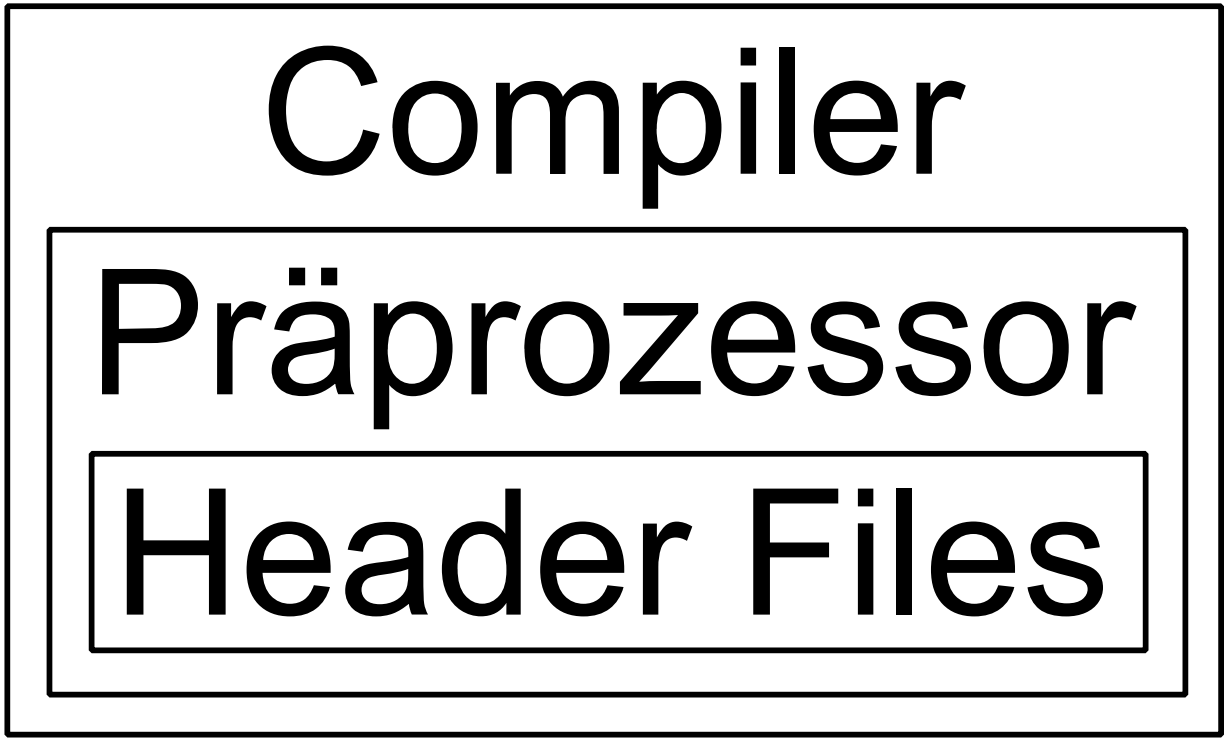


Freitagsrunde C-Kurs 2011

Compiler Präprozessor Header Files

Tutorium 3

Gliederung



Hello World Revisited

Datei hello.c

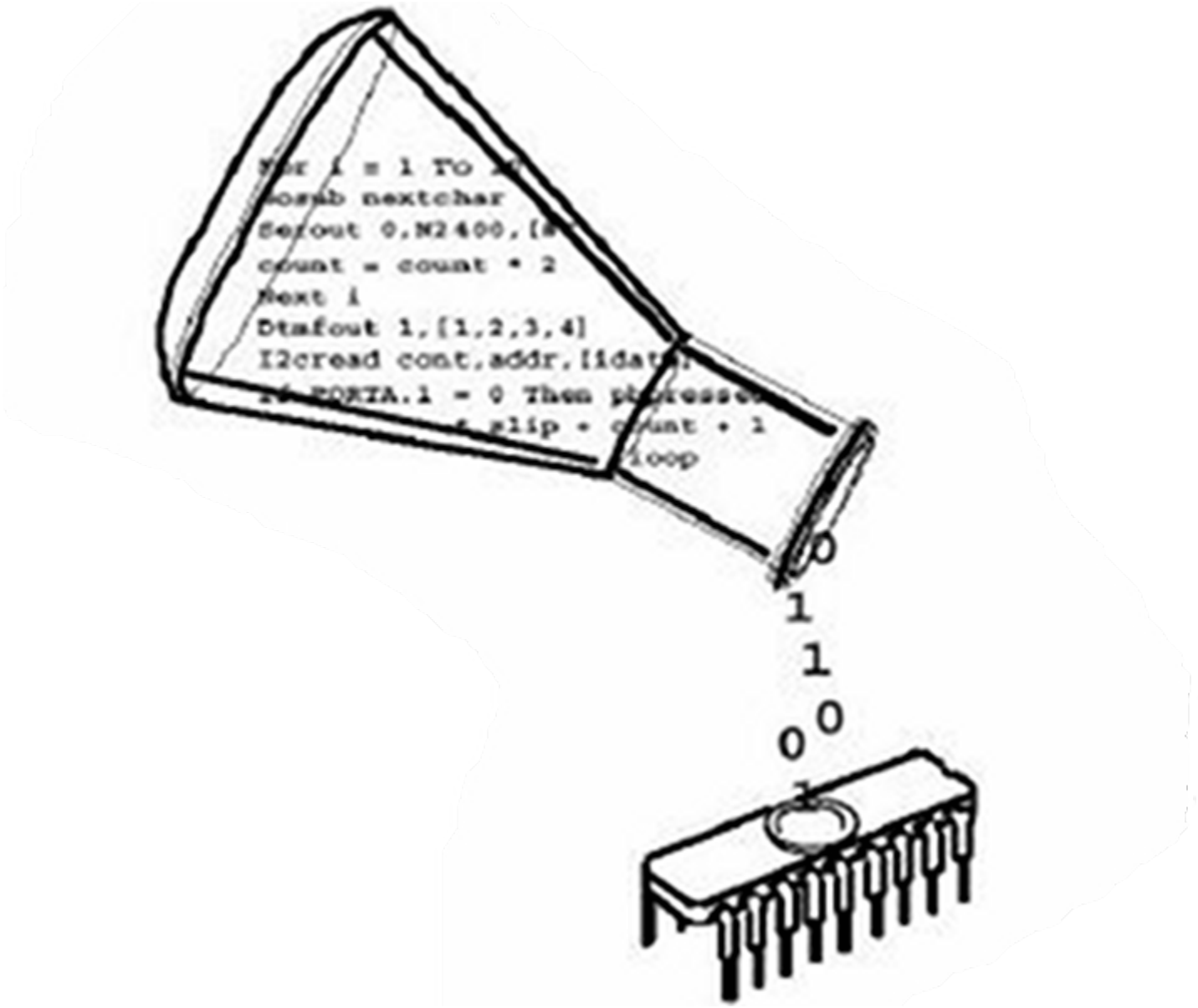
```
#include <stdio.h>

int main(){

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

Kommandozeile

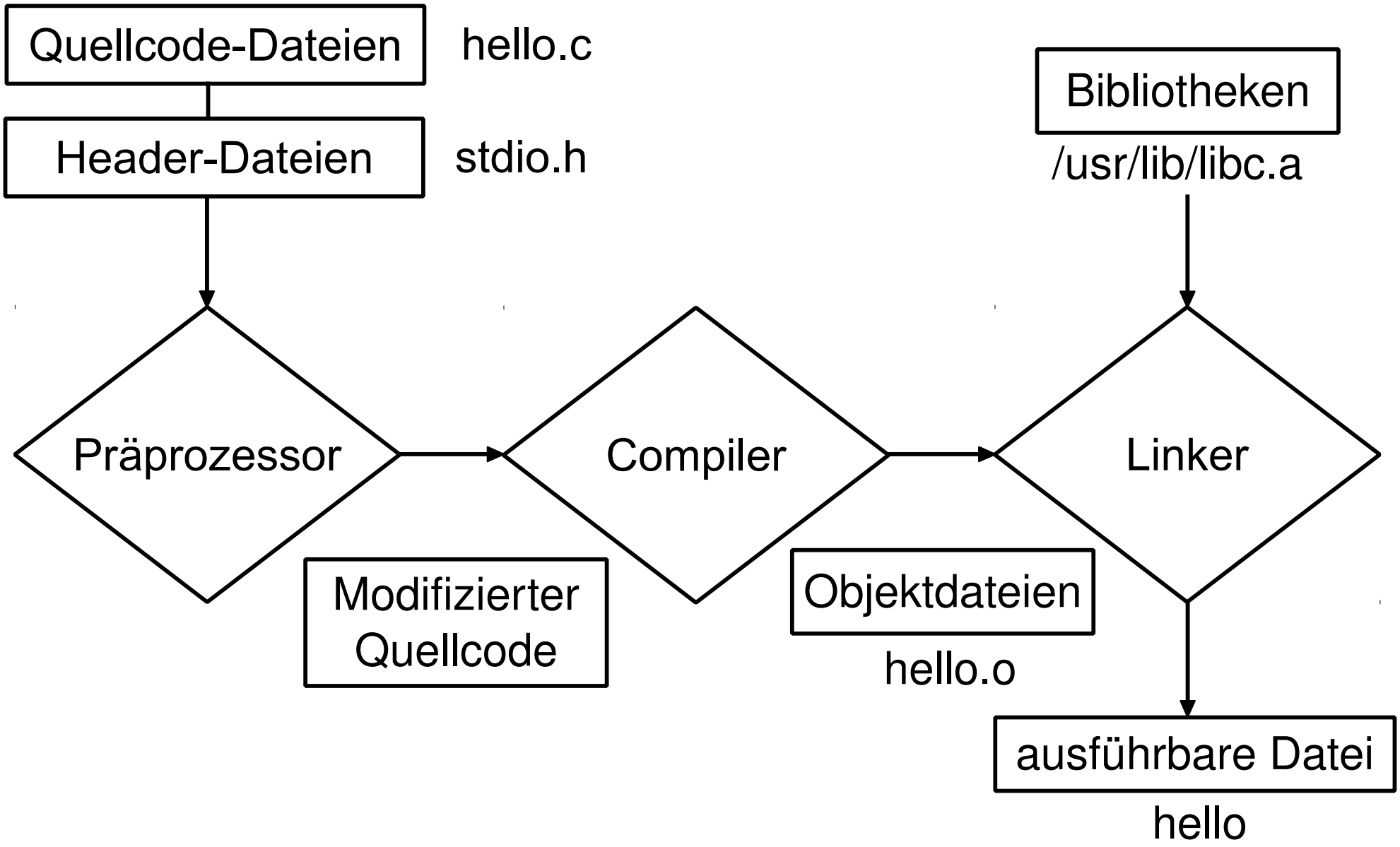
```
$ gcc -o hello hello.c
$ ./hello
Hello, World!
$
```

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!
$
```

Datei Makefile

```
all: main

main: main.o hello.o bye.o
→ $(CC) -o hello main.o hello.o bye.o

hello.o: hello.c
→ $(CC) -c hello.c

bye.o: bye.c
→ $(CC) -c bye.c

main.o: main.c
→ $(CC) -c main.c

clean:
→ rm -f main main.o hello.o bye.o
```

Übungsaufgabe

Kopiert euch die Beispieldateien und das Makefile in einen Ordner.

➜ Kompiliert die Dateien zunächst per Hand separat und linkt sie dann zusammen! Führt jedesmal **ls** aus!

➜ Löscht jetzt alle Objekt- und ausführbaren Dateien!

➜ Führt **make** aus und seht euch die Ausgabe an!

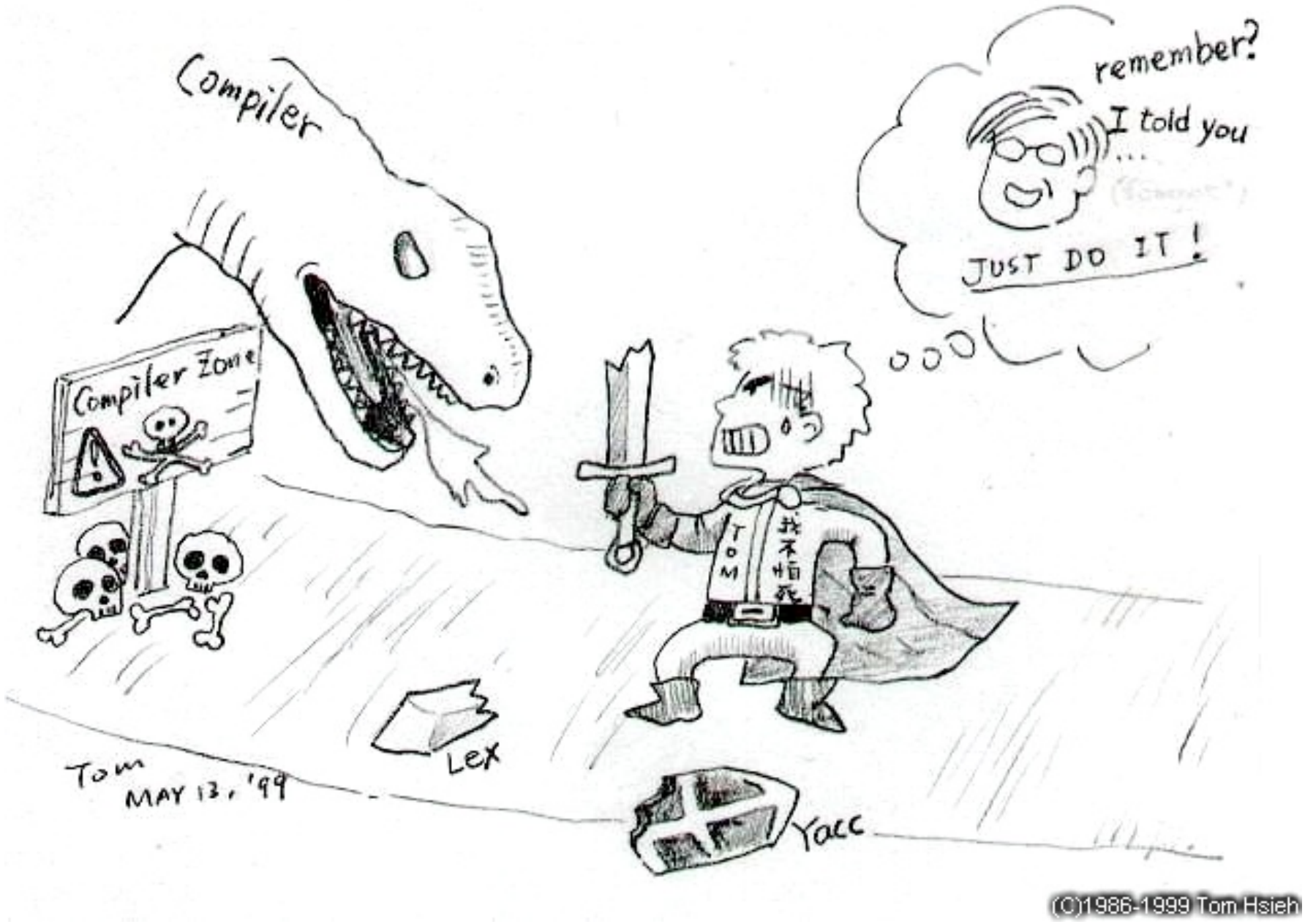
➜ Löscht die Datei **bye.o**!

➜ Führt erneut **make** aus und seht euch die Ausgabe an!

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 #

- Einfügen von Dateien:

  ➜ #include
- Ersetzen von Text (Makros):

  ➜ #define
- Bedingte Kompilierung:

  ➜ #if, #ifdef, #ifndef, #else, #elif, #endif

Präprozessoranweisungen

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

- Einfügen von Dateien:

  ➜ #include

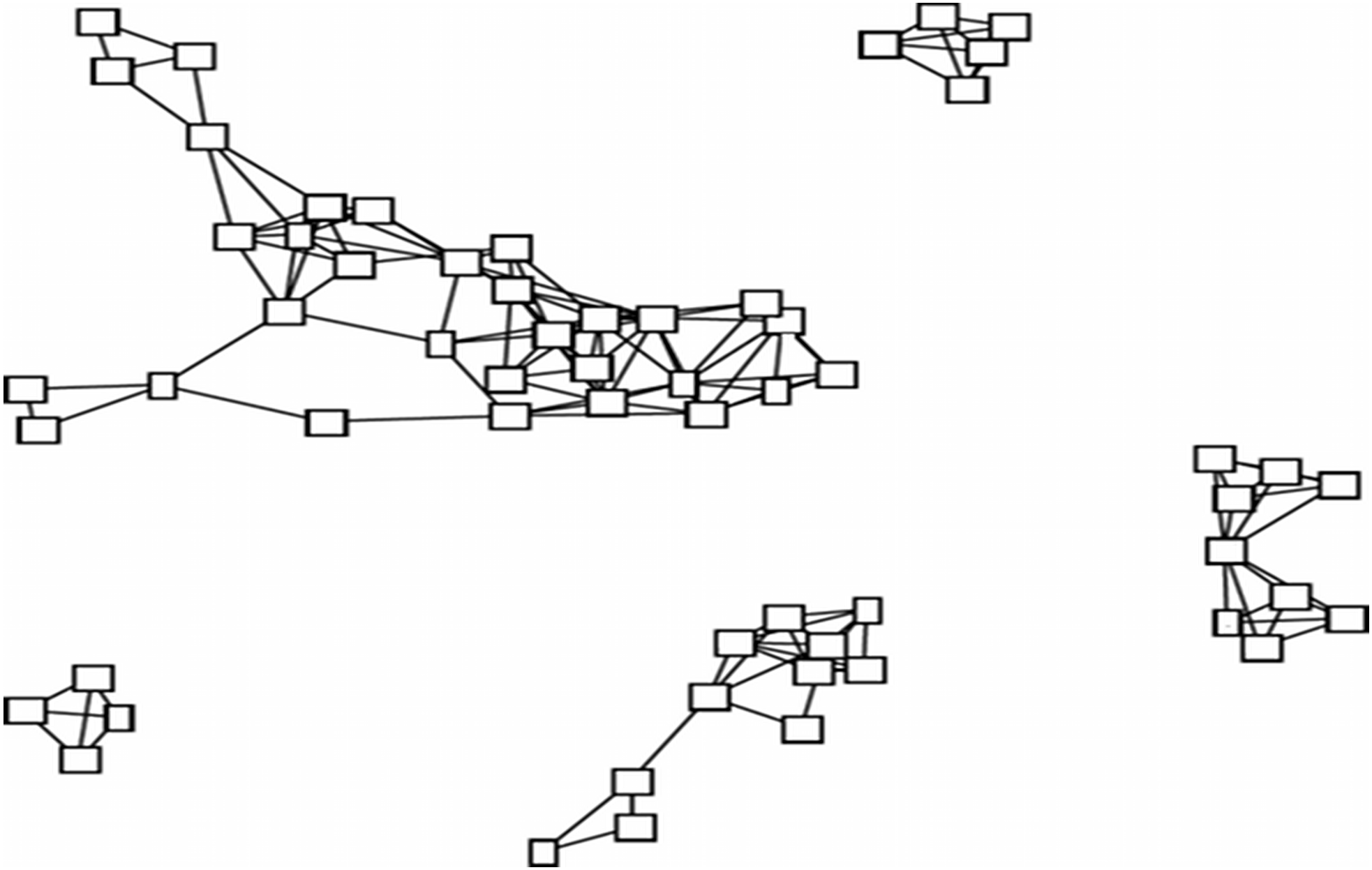
- Ersetzen von Text (Makros):

  ➜ #define

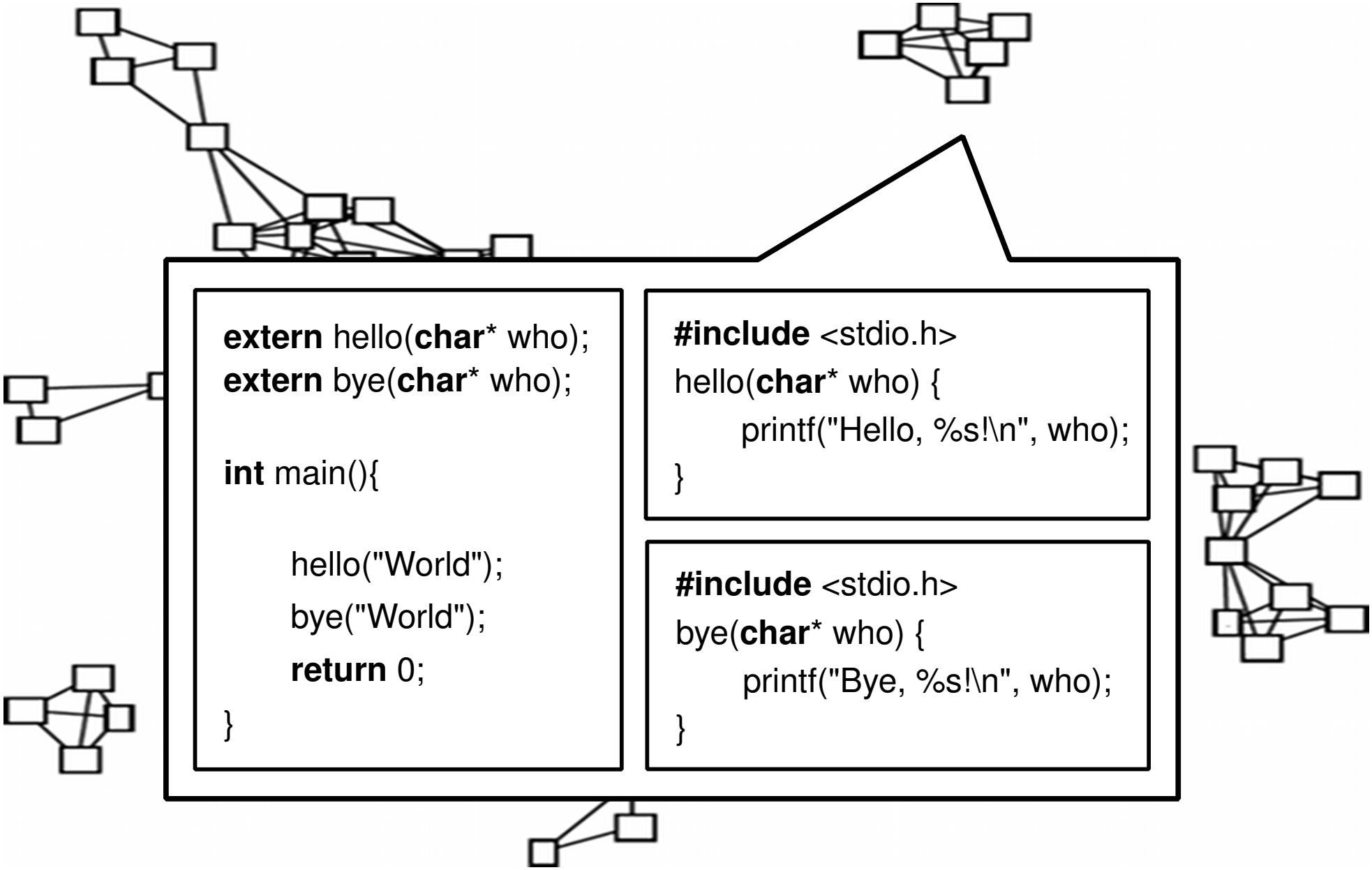
- Bedingte Kompilierung:

  ➜ #if, #ifdef, #ifndef, #else, #elif, #endif

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 selbstgeschriebene 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.

-E-Compileroption (gcc)

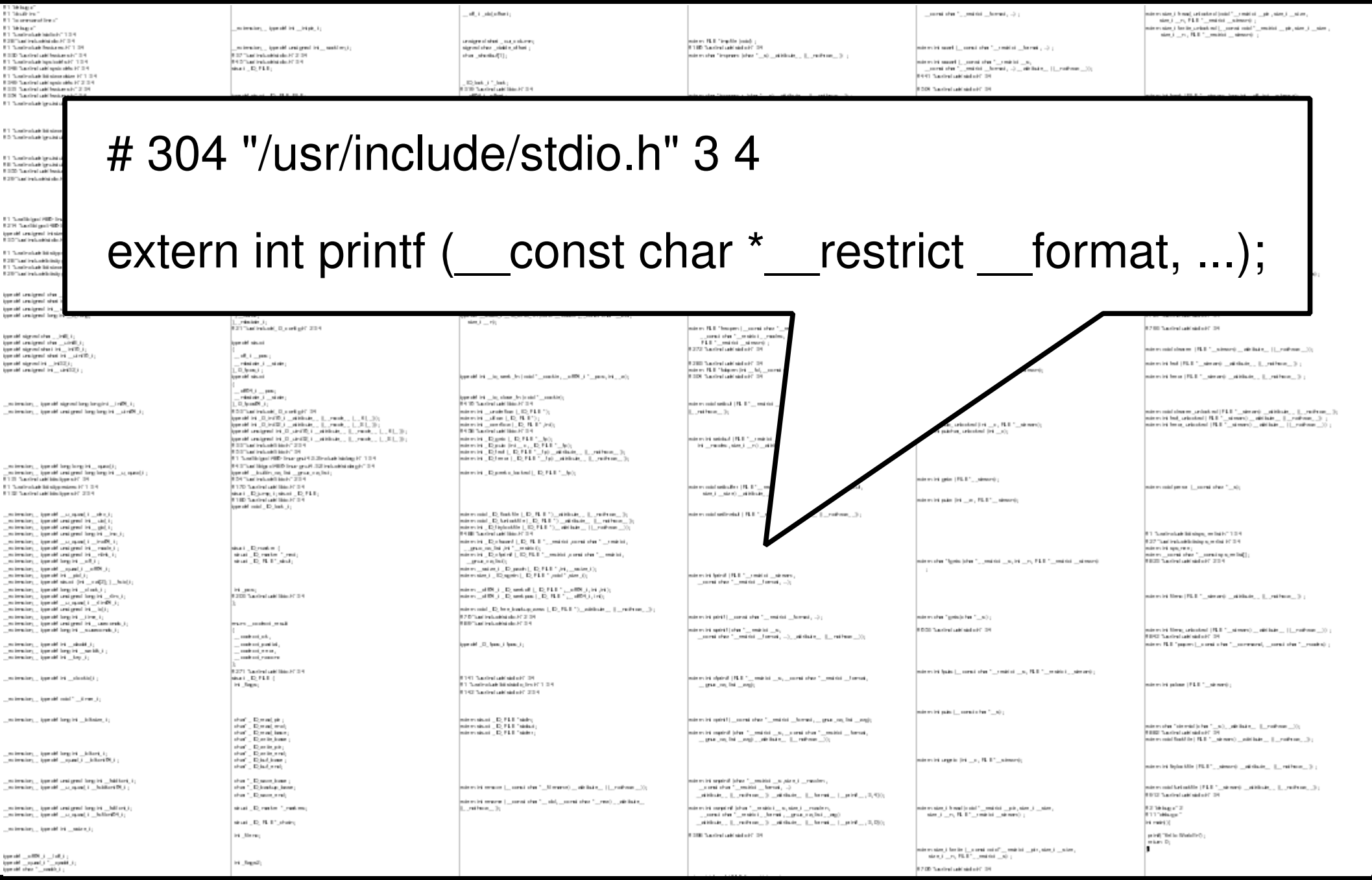
Was geschieht nun aber eigentlich beim Inkludieren eines Files?

Mit **-E** könnt ihr euch die Ausgabe des Präprozessors ansehen. Probiert's aus:

Kommandozeile:

```
$ gcc -E hello.c
```

Präprozessor-Output von Hello World



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“
...
```

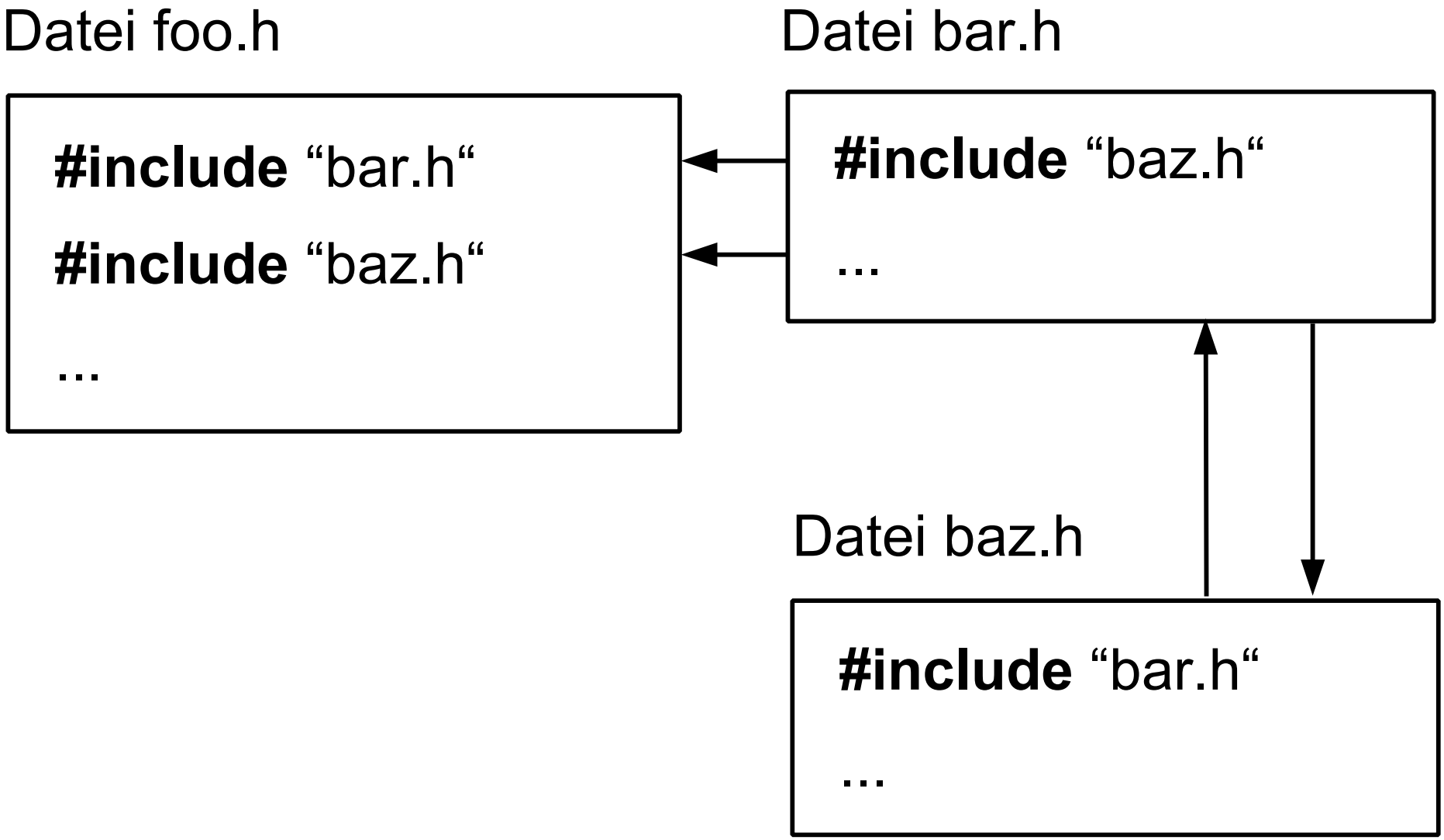
Datei bar.h

```
#include “baz.h“
...
```

Datei baz.h

```
#include “bar.h“
...
```

Problem: Mehrfachinklusion

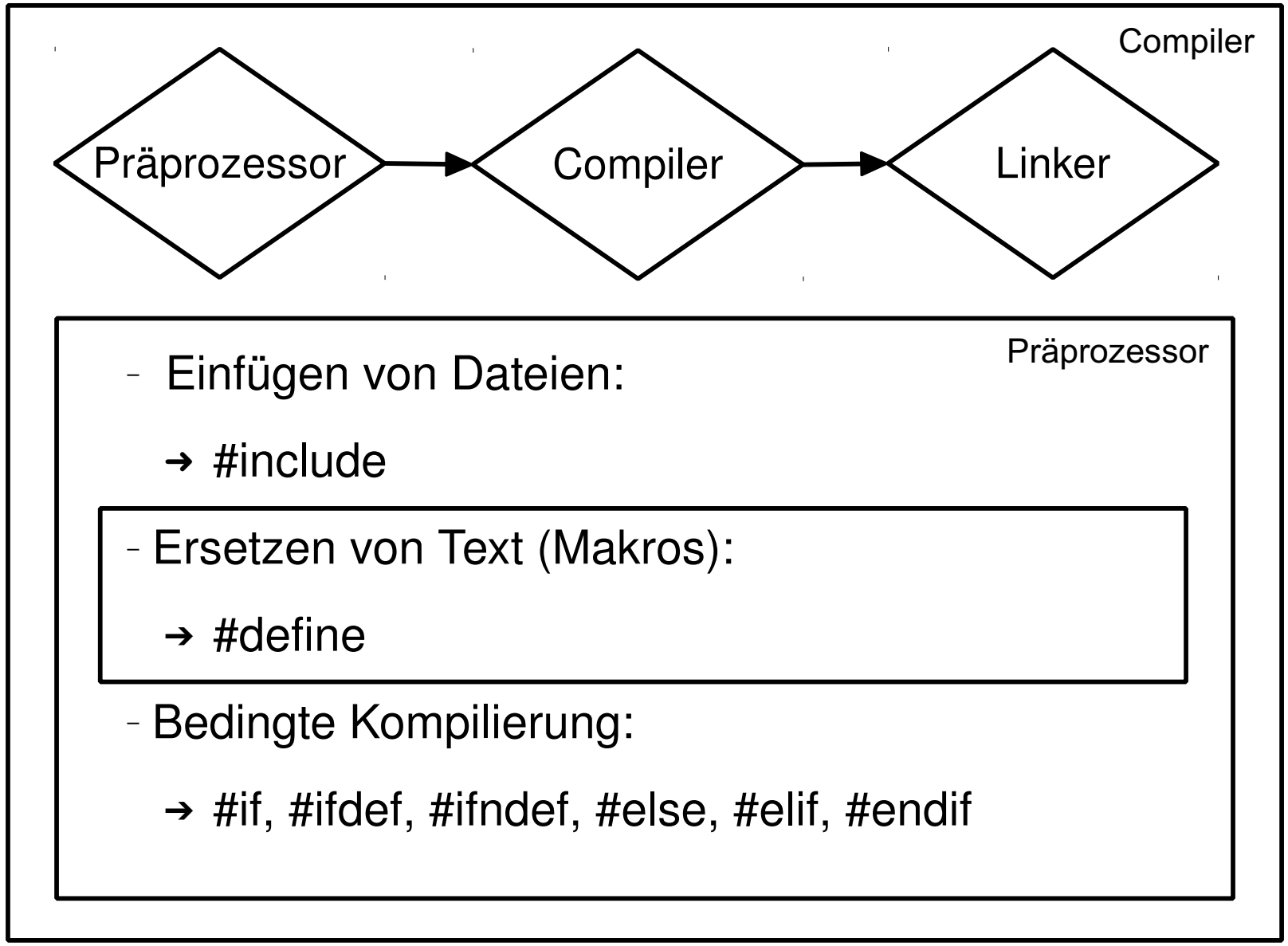


Vermeidung von Mehrfachinklusion

Datei foo.h

```
#ifndef FOO_H
   #define FOO_H


   extern int foo(int x, int y);


#endif
```

Die nötigen Sprachmittel, um diesen Code verstehen zu können, werden wir uns im Folgenden aneignen


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

Parameterlose Makros

- Syntax: **#define** NAME [replacement]
- Präprozessor ersetzt vor der Kompilierung jedes Vorkommen von **NAME** mit dem Ersetzungstext
- Fehlt das replacement, so ist der **NAME** dem System im Folgenden bekannt, anstatt undefiniert zu sein
- Um den semantischen Unterschied klarzumachen, sollten Makros immer GROSS geschrieben werden!
- Ein ausführliches Beispiel werden wir später betrachten

Syntax:

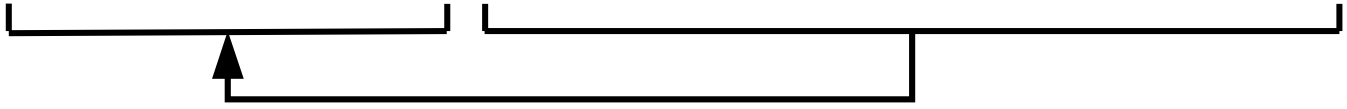
kein Leerzeichen Leerzeichen

#define NAME(dummy1 [[,dummy2], ...]) ... dummy1 ...

Parameterliste Tokenstring

Beispiel:

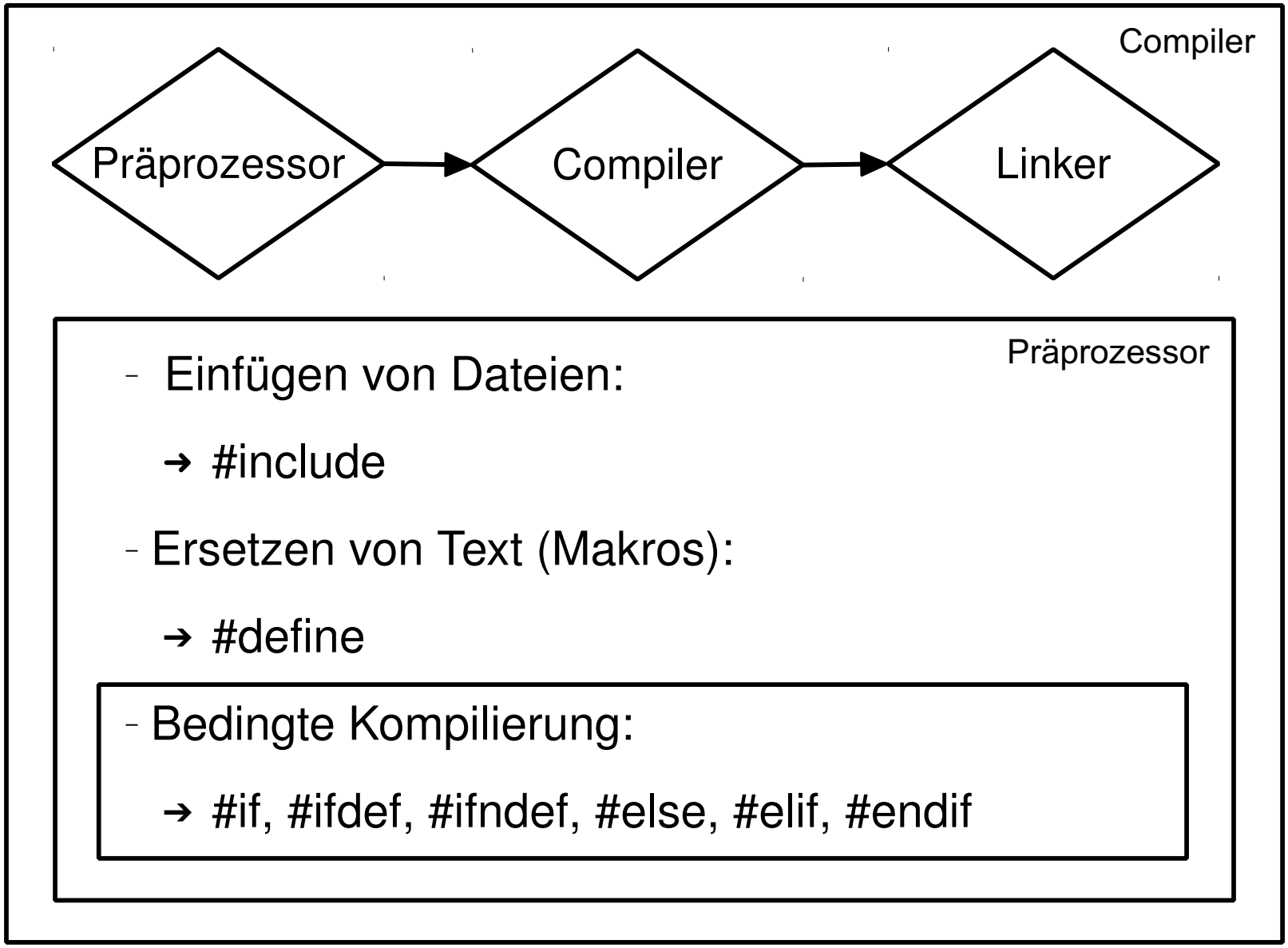
#define SIZEOF(array) sizeof(array)/sizeof(array[0])



Ein Wort zur Warnung

Da man sich mit Makros sehr, sehr leicht ins Knie schiessen kann, empfehle ich euch für den Fall, dass ihr tiefer ins Thema einsteigen wollt, dringend, euch die Folien vom Vorjahr anzugucken, bevor ihr anfangt, eigene Makros zu schreiben!

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
```

Vermeidung von Mehrfachinklusion

Datei foo.h

```
#ifndef FOO_H
   #define FOO_H


   extern int foo(int x, int y);


#endif
```

Was bewirkt nun dieser Code?

Beispiel: Debugging

```
int debug= 1;

int main(){

    if(debug)
        printf("entering main...\n“);
    …
    if(debug)
        printf("exiting main...\n“);

    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 Debugging Makro

Datei debug.h

```
#include <stdio.h>

#ifndef VERBOSE
#define VERBOSE 0
#endif

#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;
}
```

Fancy Debugging

Das Debugging-Makro kann man noch beliebig schöner und aussagekräftiger gestalten (Mit Angabe von Zeilennummer, Funktion, in der man sich gerade befindet, etc). Ihr könnt euch ja mal die Folien von vorigem Jahr angucken!

Übungsaufgabe

Schreibt ein Makefile für unser neues Hello-World-Beispiel!

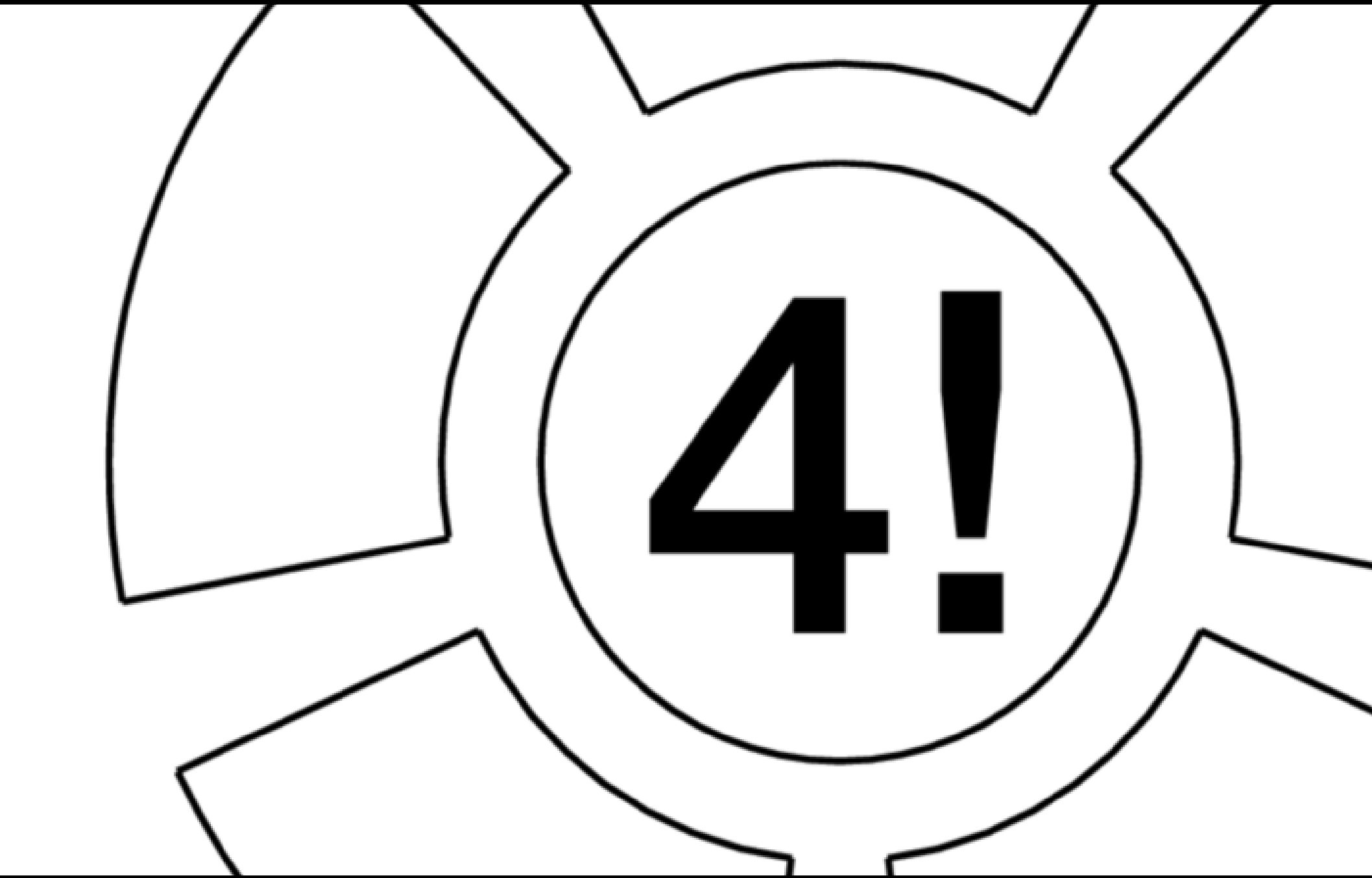
➜ Definiert targets **release, debug0** und **debug1** für Debug-Builds mit verschiedenen verbosity levels!

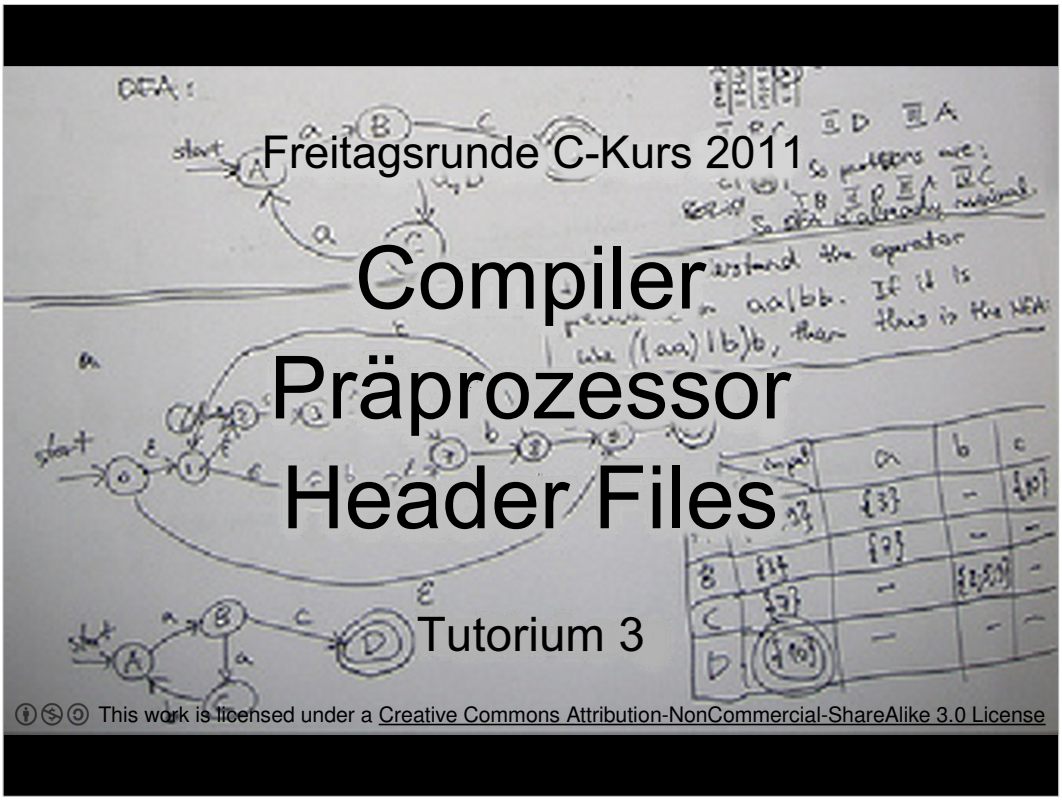
➜ Beachtet, dass auch header-Dateien als Abhängigkeiten angegeben werden müssen!

➜ Kompiliert euren Code für verschiedene Debug-Levels und überzeugt euch anhand des Konsolen- sowie des Präprozessoroutputs, dass er funktioniert

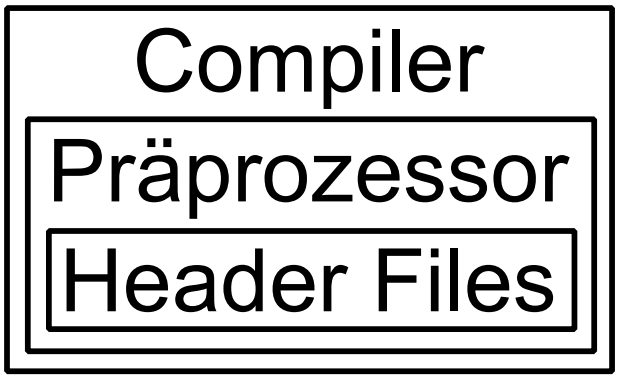
Unknown compiler

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


4!



Gliederung



Hello World Revisited

Datei hello.c

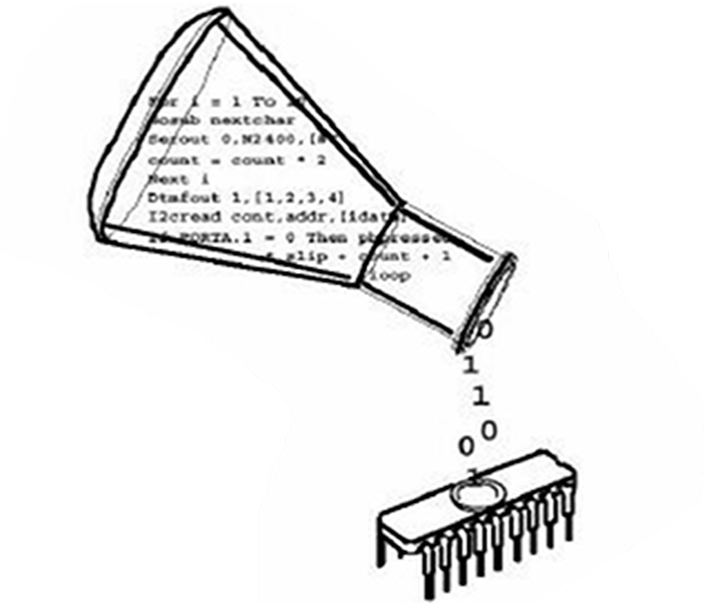
```
#include <stdio.h>

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

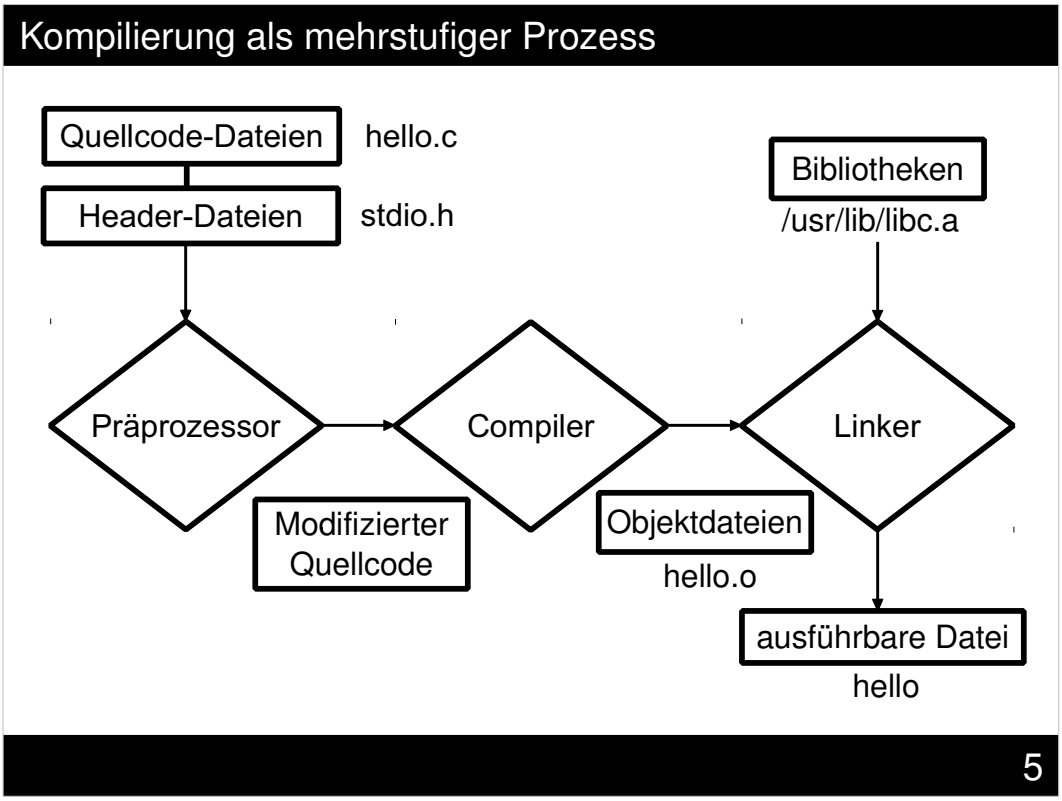


4

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.

Makefiles

Datei Makefile

```
all: main

main: main.o hello.o bye.o
→ $(CC) -o hello main.o hello.o bye.o

hello.o: hello.c
→ $(CC) -c hello.c

bye.o: bye.c
→ $(CC) -c bye.c

main.o: main.c
→ $(CC) -c main.c

clean:
→ rm -f main main.o hello.o bye.o
```

9

Damit man die in den letzten beiden Folien eingeführten Kommandos nicht jedes Mal per Hand eintippen muss, schreiben wir sie in eine Datei. Und zwar handelt es sich hier um eine spezielle Art von Datei – ein Makefile. Diese Datei muss auch Makefile heissen, damit Folgendes funktioniert:

- sobald eine Datei Makefile im selben Ordner wie der Code existiert, kann ich das Kommando make benutzen, um mein Programm vollautomatisch zu bauen.

- bei mehreren sog. Targets (das sind die Bezeichner vor dem Doppelpunkt), wird automatisch das Target all: gebaut, wenn make keine weiteren argumente übergeben werden. In unserem Falle verweist all einfach auf das target main.

- Damit unser Makefile nicht jedesmal den gesamten Code neu übersetzen muss, wenn sich nur eine Datei geändert hat, schreiben wir für jede Objektdatei ein eigenes Target

- Ein Target ist folgendermassen aufgebaut: Erst wird der Name des Targets aufgeführt, gefolgt von einem Doppelpunkt. Nach diesem stehen in der selben Zeile alle Dateien, die für das Target benötigt werden. In der nächsten Zeile, die – wichtig! - mit einem Tab (nicht mit Leerzeichen) beginnen muss, folgt das Kommando, dass zum Bauen des Targets ausgeführt werden soll.

- Zum Abschluss führen wir noch ein spezielles Target auf, das Target clean (das heisst per Konvention immer so). Es bewirkt, dass sämtliche Dateien, die bereits gebaut wurden, gelöscht werden, und der nächste build von scratch erfolgt:

Übungsaufgabe

Kopiert euch die Beispieldateien und das Makefile in einen Ordner.

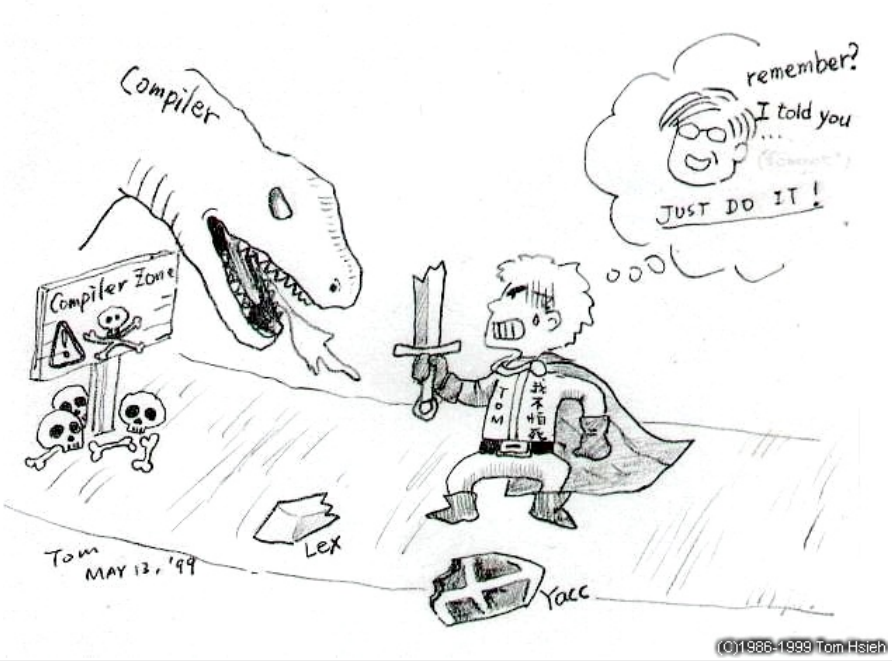
- ➜ Kompiliert die Dateien zunächst per Hand separat und linkt sie dann zusammen! Führt jedesmal **ls** aus!
- ➜ Löscht jetzt alle Objekt- und ausführbaren Dateien!
- ➜ Führt **make** aus und seht euch die Ausgabe an!
- ➜ Löscht die Datei **bye.o**!
- ➜ Führt erneut **make** aus und seht euch die Ausgabe an!

Man beachte: make ist nicht Teil von gcc und muss in der Regel extra installiert werden. Unter Debian-basierten Systemen heißt das ensprechende Paket *build-essential*

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



12

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.

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

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 #

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

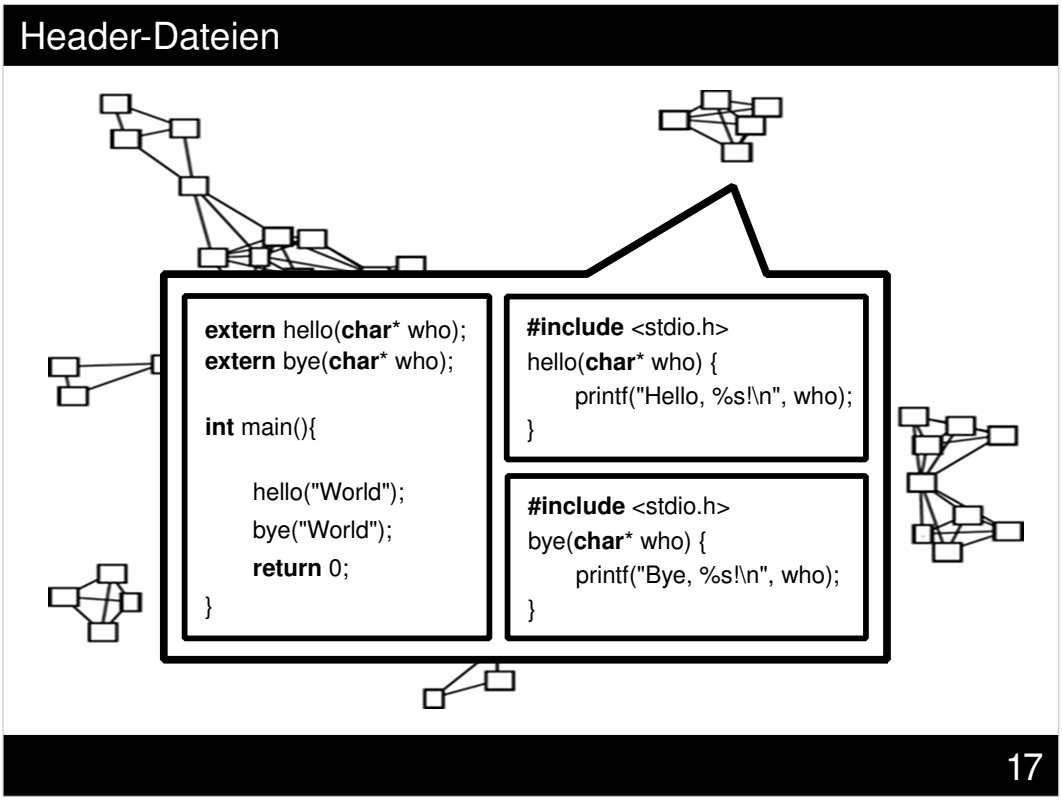
Präprozessoranweisungen

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

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

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)

Frage: Warum ist es keine gute Idee, Definitionen, etwa von Funktionen in header-Dateien zu schreiben? Welchen Fehler erwartet ihr beim Kompilieren?

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 selbstgeschriebene 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.

-E-Compileroption (gcc)

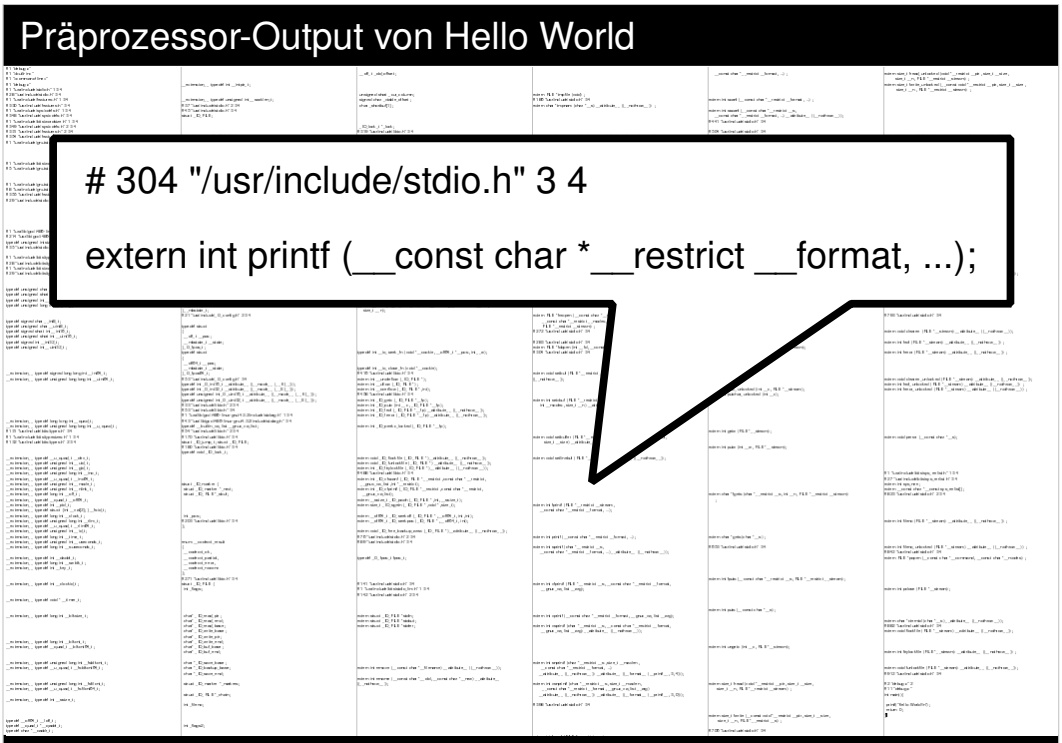
Was geschieht nun aber eigentlich beim Inkludieren eines Files?

Mit **-E** könnt ihr euch die Ausgabe des Präprozessors ansehen. Probiert's aus:

Kommandozeile:

```
$ gcc -E hello.c
```

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“
...
```

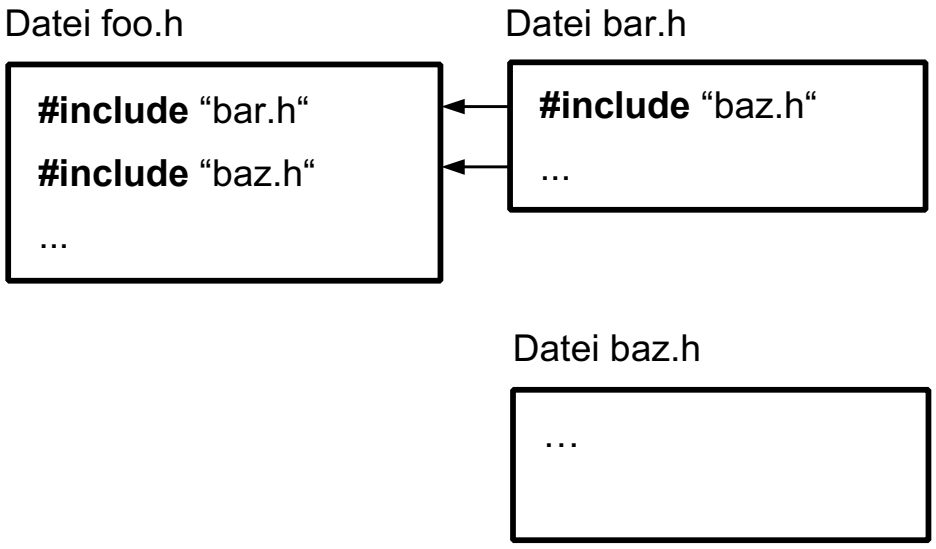
Datei bar.h

```
#include “baz.h“
...
```

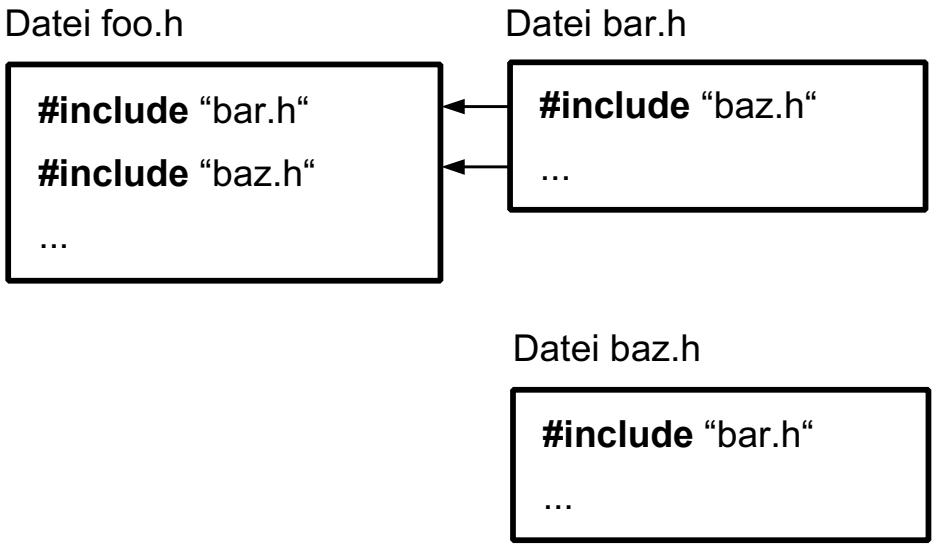
Datei baz.h

```
…
```

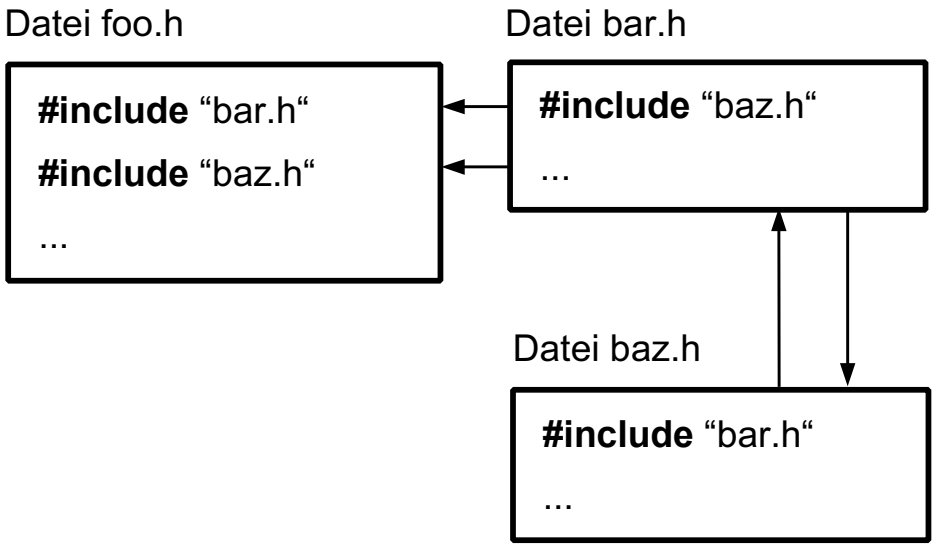
Problem: Mehrfachinklusion



Problem: Mehrfachinklusion



Problem: Mehrfachinklusion



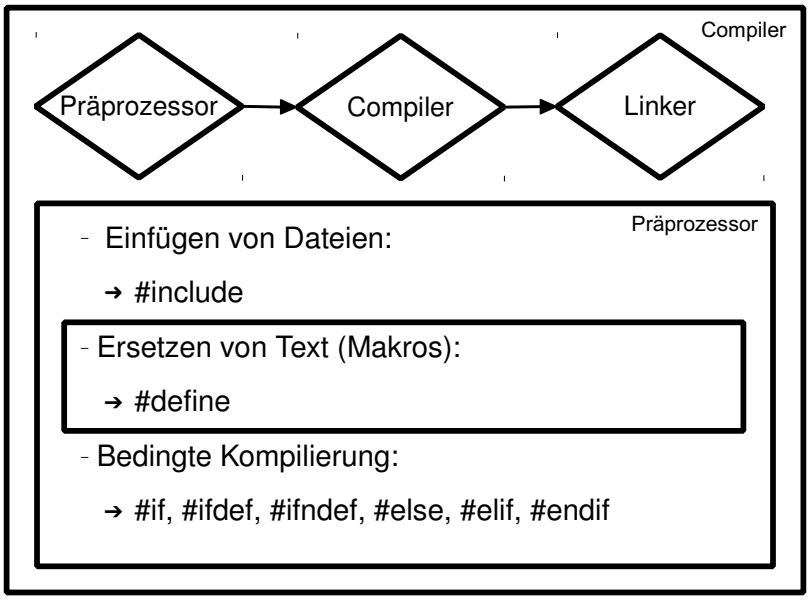
Vermeidung von Mehrfachinklusion

Datei foo.h

```
#ifndef FOO_H
  #define FOO_H

  extern int foo(int x, int y);

#endif
```

Die nötigen Sprachmittel, um diesen Code verstehen zu können, werden wir uns im Folgenden aneignen

Makros


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

Parameterlose Makros

- Syntax: **#define** NAME [replacement]
- Präprozessor ersetzt vor der Kompilierung jedes Vorkommen von **NAME** mit dem Ersetzungstext
- Fehlt das replacement, so ist der **NAME** dem System im Folgenden bekannt, anstatt undefiniert zu sein
- Um den semantischen Unterschied klarzumachen, sollten Makros immer GROSS geschrieben werden!
- Ein ausführliches Beispiel werden wir später betrachten

Makros mit Parametern

Syntax:

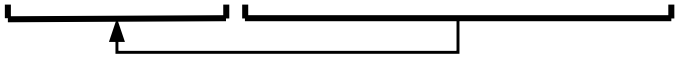
kein Leerzeichen Leerzeichen

#define NAME(dummy1 [[,dummy2], ...]) ... dummy1 ...

Parameterliste Tokenstring

Beispiel:

#define SIZEOF(array) sizeof(array)/sizeof(array[0])



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

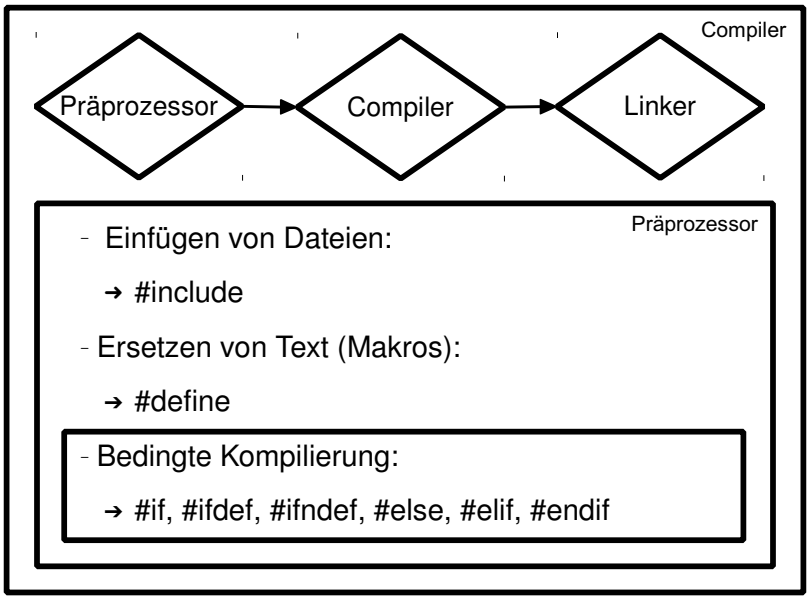
Man beachte, dass die Parameterliste nicht durch Leerzeichen vom Namen des Makros getrennt werden darf. Ansonsten wird alles ab der öffnenden Klammer zum Tokenstring gezählt und für den Makronamen substituiert.

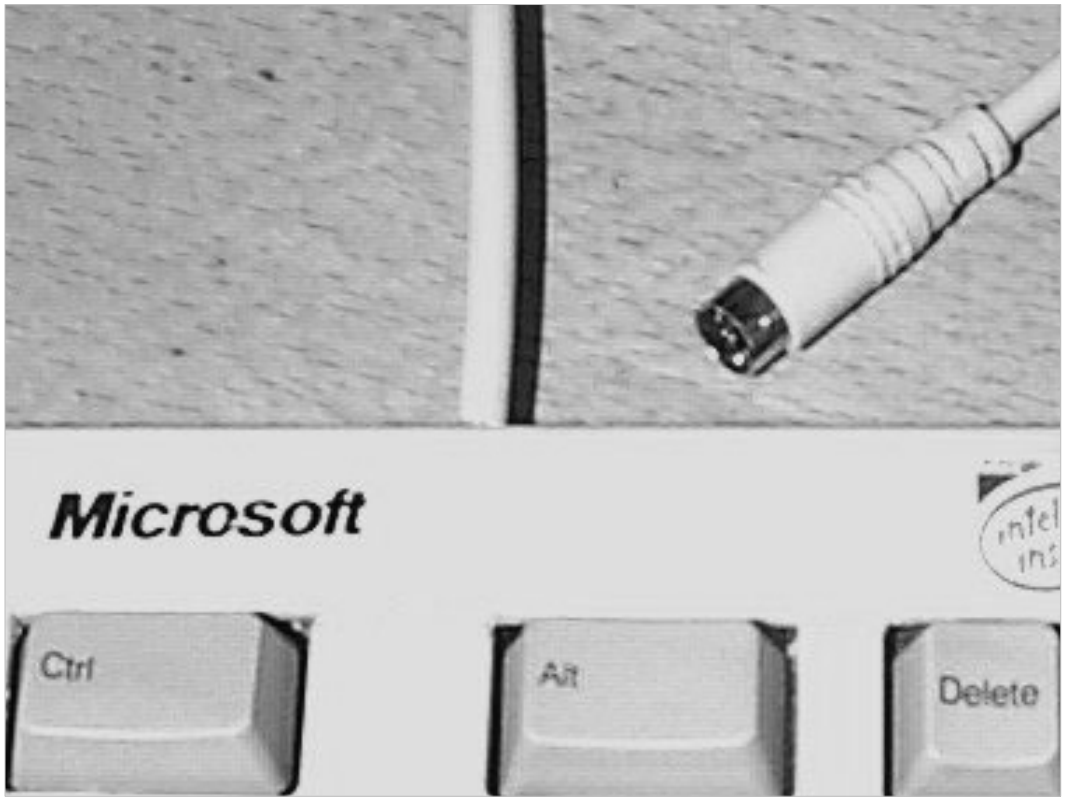
Der Tokenstring muss dagegen durch ein oder mehrere "weiße" Leerzeichen (whitespaces) von der Parameterliste getrennt sein, sonst sieht der Präprozessor das Makro sie als Makro ohne Parameter

Ein Wort zur Warnung

Da man sich mit Makros sehr, sehr leicht ins Knie schiessen kann, empfehle ich euch für den Fall, dass ihr tiefer ins Thema einsteigen wollt, dringend, euch die Folien vom Vorjahr anzugucken, bevor ihr anfangt, eigene Makros zu schreiben!

Bedingte Kompilierung




Microsoft
Ctrl
Alt
Delete

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
```

Vermeidung von Mehrfachinklusion

Datei foo.h

```
#ifndef FOO_H
  #define FOO_H

  extern int foo(int x, int y);

#endif
```

Was bewirkt nun dieser Code?

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.

Beispiel: Debugging

```
int debug= 1;

int main(){

    if(debug)
        printf("entering main...\n“);
    …
    if(debug)
        printf("exiting main...\n“);

    return 0;
}
```


Dieser Code kostet uns kostbare Prozessorzeit, auch wenn wir schon längst nicht mehr debuggen wollen, sondern unser Programm bereits an den Kunden ausgeliefert haben.

Ein einfaches Debugging Makro

Datei debug.h

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

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


Dieser Trick funktioniert folgendermassen:
- Wenn DEBUG definiert wurde, setzt der Präprozessor für LOG einfach printf ein. Das Argument wird ausgegeben
- Wenn DEBUG nicht definiert wurde, optimiert der Compiler – nicht der Präprozessor – die gesamte Zeile einfach weg, und zur Laufzeit findet keine if-Abfrage statt. Denn der Compiler weiss ja, dass die Ausgabe ohnehin nicht stattfinden würde, da das Argument von if immer zu false ausgewertet wird.

Man beachte: Ein einfaches #define LOG im #else Zweig würde nicht ausreichen, denn dann hingen die Argumente zu printf in der Luft, und der Compiler würde einen Fehler melden.

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 Debugging Makro

Datei debug.h

```
#include <stdio.h>

#ifndef VERBOSE
#define VERBOSE 0
#endif

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


Damit unser Vorhaben funktioniert, müssen wir unsere Datei „debug.h“ noch etwas anpassen

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;
}
```


Frage: Welche der beiden Versionen ist nun besser? Anders gefragt: Ist Version 2 zur Laufzeit schneller?
Antwort: Nein, da der Compiler ein if(true) ohnehin wegoptimiert. Ihr könnt also ruihig die kürzere erste Version verwenden.

Fancy Debugging

Das Debugging-Makro kann man noch beliebig schöner und aussagekräftiger gestalten (Mit Angabe von Zeilennummer, Funktion, in der man sich gerade befindet, etc). Ihr könnt euch ja mal die Folien von vorigem Jahr angucken!

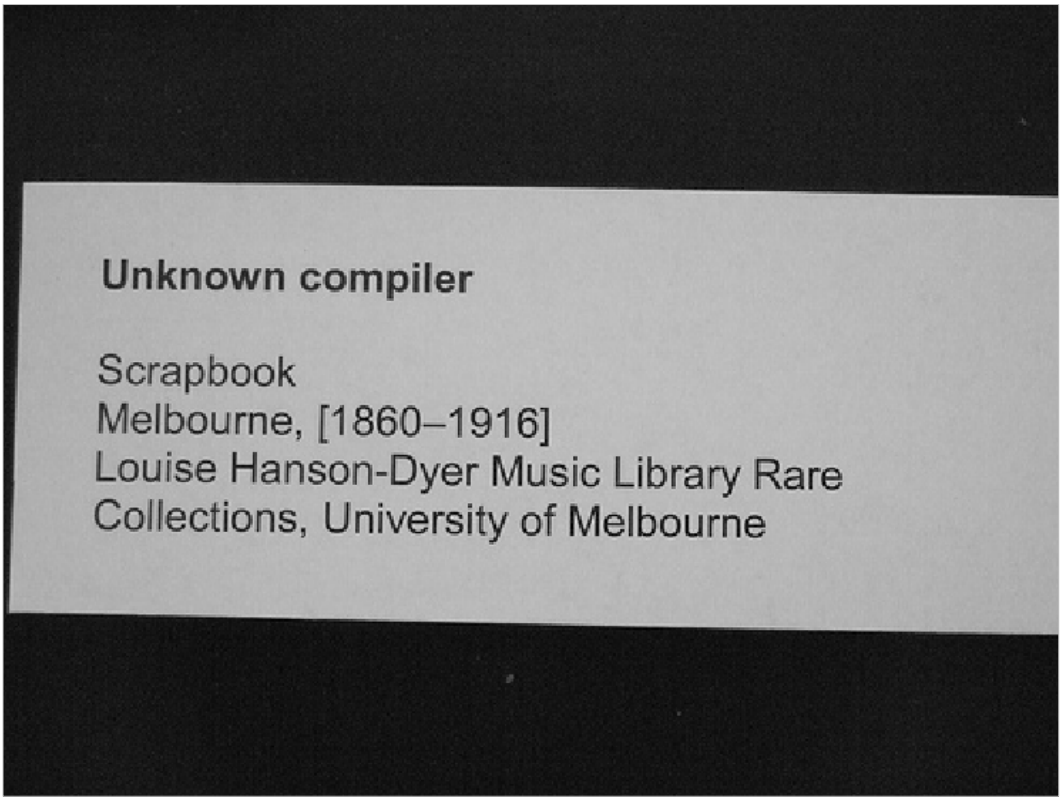
Übungsaufgabe

Schreibt ein Makefile für unser neues Hello-World-Beispiel!

➜ Definiert targets **release, debug0** und **debug1** für Debug-Builds mit verschiedenen verbosity levels!

➜ Beachtet, dass auch header-Dateien als Abhängigkeiten angegeben werden müssen!

➜ Kompiliert euren Code für verschiedene Debug-Levels und überzeugt euch anhand des Konsolen- sowie des Präprozessoroutputs, dass er funktioniert


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

Danke!