



Praxis des Programmierens

Clemens Gröpl

Wintersemester 2009

6. Präprozessor, Linker, Make

Bioinformatik

Universität Greifswald

An dieser Stelle ein herzlicher Dank an Oliver Kohlbacher
für die Überlassung der Präsentation! - Clemens Gröpl

Informatik II

Oliver Kohlbacher

Sommersemester 2006

6. Präprozessor, Linker, Make

Abt. Simulation biologischer Systeme
WSI/ZBIT, Eberhard Karls Universität Tübingen

Gliederung

- Übersetzung von C-Programmen
 - Präprozessordirektiven
 - Makros
 - Definition und Deklaration
 - Header- und Sourcedateien
- Linken
 - Rolle des Linkers
 - Linken mit dem GCC
 - Bibliotheken
- Makefile
 - Grundlegende Syntax
 - Bedingtes Bauen größerer Projekte
 - Beispiele

Präprozessormakros

- C-Präprozessor definiert eine einfache Sprache mit der Ersetzungen im Quellcode vorgenommen werden können
- Am wichtigsten ist **#include**, das eine Datei einfügt
- Daneben können Makros aber auf **Funktionen** oder **Konstanten** definieren (**#define**)
- **Bedingungen** (**#ifdef/#endif**) werden häufig zur bedingten Übersetzung eingesetzt (bei plattformabhängigem Code)
- Vorsicht mit Makros:
 - Code wird schnell unleserlich
 - Debuggen schwierig, da man die Makrodefinitionen nicht leicht findet
- **Beispiel:**
Das böartige Makro

#define while if

ersetzt alle While-Schleifen durch If-Anweisungen!

Das beschleunigt manches Programm, aber nicht das Debuggen!

#include

- **#include** fügt an der Stelle der Anweisung eine Datei ein
- Dateiname in Doppelhochkommata wird im aktuellen Verzeichnis gesucht
- Dateiname in spitzen Klammern wird in Standardsuchpfaden für Header gesucht
- Standardsuchpfad dafür kann dem Compiler mit „-I“ mitgegeben werden
- Mit **#include** eingefügte Dateien können ihrerseits **#include**-Anweisungen enthalten, sie werden ebenfalls vom Präprozessor behandelt
- **Beispiel:**

```
#include "header.h"
```

sucht nach header.h im aktuellen Verzeichnis

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

sucht nach stdio.h in den Standardpfaden

```
#include <GL/gl.h>
```

sucht nach gl.h in einem Verzeichnis GL in den Standardpfaden

#define/#undef

- Mit `#define` wird einem Text ein Wert zugewiesen, der an jedem Vorkommen ersetzt wird
- `#define SIZE 5`
ersetzt z.B. an jeder nachfolgenden(!) Stelle des Codes den Text `SIZE` durch `5`
- `#undef SIZE` macht diese Definition wieder rückgängig
- Ersetzung erfolgt jedoch halbwegs intelligent:
 - Nicht in Stringkonstanten
 - Nicht in zusammengesetzten Ausdrücken (`SIZE` wird nicht in `MYSIZE` ersetzt)

- **Beispiel:**

```
#define SIZE 6
#include <stdio.h>
int main()
{
    printf("SIZE = %d\n", SIZE);
}
```

Ausgabe:
SIZE = 6

Definitionen durch den Compiler

- Äquivalent zu einer `#define`-Direktive ist die **Kommandozeilenoption** **"-D"** bei den meisten Compilern
- `#define <SYMBOLNAME>` entspricht dabei `-D<SYMBOLNAME>` auf der Kommandozeile
- `#define <SYMBOLNAME> <WERT>` entspricht analog `-D<SYMBOLNAME>=<WERT>`
- Damit kann man Präprozessordefinitionen zur **Compile-Zeit** aktivieren, ohne Änderungen am Sourcecode durchführen zu müssen
- So kann man z.B. **Debug-Code** "wegcompilieren"
 - Ist die Definition "eingeschaltet" (z.B. mit **-DDEBUG**), so wird der Debugcode incompiliert (Entwicklungsphase der Software)
 - Ohne diese Definition wird der Debugcode nicht mitcompiliert und führt somit auch nicht zu schlechterer Performance (Produktionsphase der Software)
- Diese Art von Definition wird darüber hinaus oft zur Aktivierung **plattformspezifischer Codeteile** verwendet (**-DWINDOWS** o.ä)

#ifdef/#ifndef/#endif

- Die Direktiven `#ifdef`, `#if`, `#else`, `#elseif` und `#endif` können für bedingte Übersetzung benutzt werden
- Damit lässt sich z.B. plattformabhängiger Code schreiben oder Debug-Code nur bedingt eincompilieren
- `#ifdef` überprüft ob ein bestimmtes Makro definiert ist (unabhängig vom Wert)
- `#if` erlaubt das Überprüfen einfacher arithmetischer und logischer Bedingungen (`<`, `>`, `=`, `&&`, `||`, ...), die Konstanten und Präprozessormakros enthalten können

```
#ifdef OS_WINDOWS
    os_name = "Windows";
#elif OS_DARWIN
    os_name = "MacOS";
#elif OS_LINUX
    os_name = "Linux";
#else
    os_name = "Unknown";
#endif

...

#if QT_VERSION_MAJOR >= 4
    #define VOLUME_RENDERING
#endif
```

Definition durch den Compiler

```
/* 06_debug.c */
#include <stdio.h>
long fib(long n)
{
    #ifdef DEBUG
        printf("fib(%d)\n", n);
    #endif
    if (n < 3) return 1;
    return fib(n-2) + fib(n-1);
}

int main()
{
    long n = 5;
    printf("fib(%d) = %d\n",
           n, fib(n));
}
```

```
>gcc -o test -DDEBUG 06_debug.c
>./test
fib(5)
fib(3)
fib(1)
fib(2)
fib(4)
fib(2)
fib(3)
fib(1)
fib(2)
fib(5) = 5
>gcc -o test 06_debug.c
>./test
fib(5) = 5
>
```

#ifdef-Wrapper für Header

- In komplexeren Softwaresystemen ist ein häufiger Fehler, dass Header mehrfach per `#include` eingebunden werden (von unterschiedlichen Headern aus)
- Der Compiler weist jedoch mehrfache Deklarationen als Fehler zurück
- Um dies zu verhindern verwendet man folgende Direktiven, die sicherstellen, dass ein Header nur einmal eingebunden wird:

```
#ifndef HEADER_NAME_H
#define HEADER_NAME_H
...
[eigentlicher Header]
...
#endif
```

- Beim ersten Einbinden des Headers wird `HEADER_NAME_H` definiert und bei nachfolgendem Einbinden wird der eigentliche Header nicht mehr an den Compiler weitergereicht
- Der verwendete Name muss natürlich für jeden Header anders sein, meist verwendet man den Dateinamen

Präprozessormakros

- Mit `#define` können auch Makros definiert werden, die Parameter besitzen
- Ein beliebtes Makro war z.B. `max()`, das einen einfachen C-Ausdruck für das Maximum zweier Zahlen einfügt:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

- Nach dieser Definition können Sie im Quellcode `max()` wie eine Funktion verwenden
- Vollständige Klammerung verhindert Probleme beim Einsetzen beliebiger Ausdrücke
- Der Präprozessor expandiert alle Vorkommen von `max` und reicht obigen Code an den Compiler weiter
- **Vermeiden Sie Präprozessormakros wo es nur geht!**
- Praktisch alles was Sie damit machen können, können Sie auch mit "richtigen" Funktionen machen (es gibt eine Reihe nützlicher Ausnahmen in denen explizit die textuelle Ersetzung notwendig ist, diese sind jedoch überwiegend üble Hacks!)

Definition und Deklaration

- In C können Funktionen und Variablen unabhängig voneinander deklariert und definiert werden
- **Deklaration** beschreibt dabei nur die Funktion (Rückgabewert, Argumente, Name) und erfolgt in Headerdatei
- **Definition** enthält Implementierung („{}“), erfolgt in Sourcedatei

```
/* 06_Fak.h */  
...  
long fak(long i);
```

```
/* 06_main.c */  
#include <stdio.h>  
#include "06_Fak.h"  
int main()  
{  
    printf("7!=%d\n", fak(7));  
}
```

```
/* 06_Fak.c */  
#include "06_Fak.h"  
long fak(long i)  
{  
    if (i > 2)  
    {  
        return i * fak(i-1);  
    }  
    return i;  
}
```

- Um die Funktion zu verwenden, genügt ein `#include` des Headers

Headerdateien

- Headerdateien enthalten Deklarationen, in der Regel jedoch keine Definitionen
- Eine Deklaration veranlasst den Compiler nicht dazu Code zu erzeugen
- Headerfiles definieren somit das Interface zu den Implementierungen in den Sourcefiles
- **Beispiel:**

```
/* 06_Fak.h */  
#ifndef FAK_H  
#define FAK_H  
  
long fak(long i);  
  
#endif
```

Diese Headerdatei deklariert die Funktion *fak* und enthält darüber hinaus einen `#ifndef`-Wrapper, um mehrfaches Einbinden des Headers zu vermeiden.

Sourcdateien

- *Per se* existiert für den Compiler keinerlei Unterschied zwischen Headern und Sourcefiles
- Vereinbarungsgemäß finden sich in den Sourcefiles die Definitionen
- Sourcefiles werden einzeln übersetzt, man spricht auch von einzelnen *Compilation Units*
- **Beispiel:**

```
/* 06_test.c */
#include "06_Fak.h"
long fak(long i)
{
    if (i > 2) { return i * fak(i-1); }
    return i;
}
```

Funktion *fak* wird im Sourcefile definiert, Compiler erzeugt dafür Code

- Gewöhnlich wird der Header inkludiert, wodurch sich Inkonsistenzen zwischen Header- und Sourcdatei erkennen lassen

Objektdateien

- Enthält eine Sourcedatei Definitionen, so erzeugt der Compiler Maschinencode in Form einer Objektdatei (.o)
- Maschinencode ist über **symbolische Namen** (*symbols*) erreichbar
- Mit dem Unix-Kommando **nm** lässt sich das "Inhaltsverzeichnis" einer Objektdatei anschauen

- **Beispiel:**

```
>gcc -c 06_Fak.c
>nm -o 06_Fak.o
06_Fak.o:00000000 b .bss
06_Fak.o:00000000 d .data
06_Fak.o:00000000 t .text
06_Fak.o:00000000 T _fak
```

- In der ersten Spalte steht dabei "T" für "Text" (Symbole im Codesegment), "D" für ein Symbol im Datensegment, "U" stünde für undefinierte Symbole (mehr dazu in [4])

Objektdateien

Hinweis:

Das Unix-Kommando **objdump** kann den Inhalt von Objektdateien in vielen unterschiedlichen Formaten ausgeben. Mehr dazu in der man-Page.

```
>objdump -d 06_Fak.o
```

```
06_Fak.o:      file format pe-i386
```

```
Disassembly of section .text:
```

```
00000000 <_fak>:
```

```
  0:    55                push   %ebp
  1:    89 e5             mov    %esp,%ebp
  3:    83 ec 08          sub   $0x8,%esp
  6:    83 7d 08 02       cml   $0x2,0x8(%ebp)
  a:    7e 15             jle   21 <_fak+0x21>
  c:    8b 45 08          mov   0x8(%ebp),%eax
  f:    48               dec   %eax
 10:   89 04 24          mov   %eax,(%esp)
 13:   e8 e8 ff ff ff   call  0 <_fak>
 18:   0f af 45 08       imul 0x8(%ebp),%eax
 1c:   89 45 fc          mov   %eax,0xffffffffc(%ebp)
 1f:   eb 06             jmp   27 <_fak+0x27>
 21:   8b 45 08          mov   0x8(%ebp),%eax
 24:   89 45 fc          mov   %eax,0xffffffffc(%ebp)
 27:   8b 45 fc          mov   0xffffffffc(%ebp),%eax
 2a:   c9               leave
 2b:   c3               ret
```

Ausführbare Programme

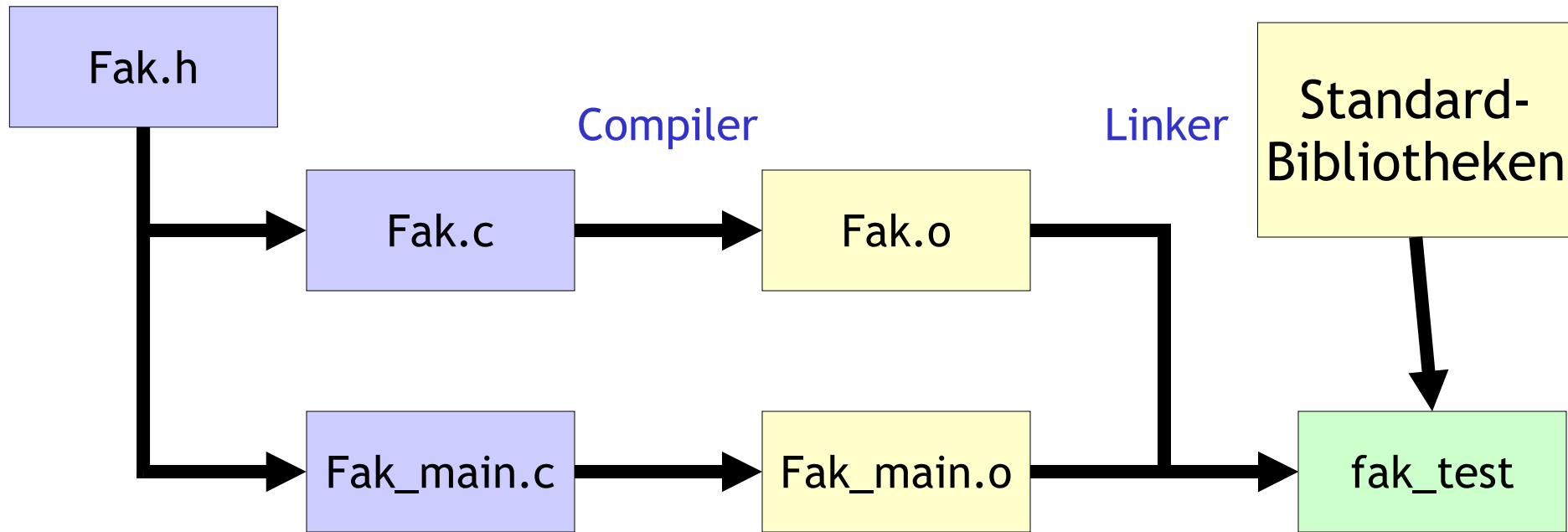
- Der Linker erzeugt aus einem oder mehreren Objektdateien (und möglicherweise einigen Bibliotheken) ein ausführbares Programm
- Dazu muss genau eine Objektdatei eine Funktion "main" enthalten
- **Beispiel:**

```
>gcc -c 06_Fak_main.c
>nm 06_Fak_main.o
00000000 b  .bss
00000000 d  .data
00000000 r  .rdata
00000000 t  .text
                U  __main
                U  __alloca
                U  _fak
00000000 T  _main
                U  _printf
```

```
/* 06_main.c */
#include <stdio.h>
#include "06_Fak.h"
int main()
{
    printf("7!=
%d\n", fak(7));
}
```

- Um aus 06_test_main eine ausführbare Datei zu erzeugen, muss der Linker alle **unaufgelösten Symbole** ("U") aus weiteren Objektdateien hinzufügen (*foo* aus **06_test.o**, *printf* usw. aus externen Bibliotheken)

Linker



- Quelldateien werden getrennt vom Compiler zu Objektdateien übersetzt
- Header sorgen für Sichtbarkeit der Deklaration in Quelldateien und werden von beiden .c-Dateien eingebunden
- Linker fügt Objektdateien und externe Bibliotheken zu ausführbarer Datei zusammen und löst dabei Querbezüge auf

Linken mit dem GCC

- GCC kann auch nur den Linker ausführen
- Dies macht er automatisch, wenn ihm nur Objektdateien mitgegeben werden
- Option "**-o**" gibt an, wie der Dateiname für das ausführbare Programm sein soll
- Namen für ausführbare Programme sind frei wählbar (unter Unices haben sie üblicherweise keine Endung!)
- Wird kein Dateiname für die ausführbare Datei angegeben, so erzeugt **gcc** eine Datei mit dem Namen **a.out** (Unices, unter Windows: **a.exe**)
- GCC bindet automatisch Standardbibliotheken dazu in denen z.B. der Code für Ein/Ausgabe (*printf* etc.) enthalten ist

Linken mit dem GCC

```
>ls
06_Fak.c      06_Fak.h    06_Fak_main.c
>gcc -c 06_Fak.c
>gcc -c 06_Fak_main.c
>gcc -o fak_test 06_Fak.o 06_Fak_main.o
>ls
06_Fak.c      06_Fak.h    06_Fak_main.c
06_Fak.exe    06_Fak.o    06_Fak_main.o
>./fak_test
7!=5040
>gcc -o fak_test 06_Fak.o 06_Fak.o 06_Fak_main.o
06_Fak.o:06_Fak.c:(.text+0x0): multiple definition of
    _fak'
06_Fak.o:06_Fak.c:(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

Hinweis:

Unter CygWin/Windows werden ausführbare Programme vom Linker automatisch mit der Endung .exe versehen, unter Unix nicht!

- Mehrfach definierte Symbole (Funktion `fak` in zwei Kopien von `06_Fak.o`) führen dazu, dass der Linker nicht weiß, welche Kopie zu verwenden ist!

Bibliotheken

- C/C++-Bibliotheken enthalten (wie auch in Java) ein oder mehrere Objektdateien
- Ist das Interface bekannt (d.h. gibt es eine Header-Datei), so kann der Compiler Verweise auf diese Objektdateien einbauen
- Linker löst diese Verweise gegen die Bibliothek auf
- Im einfachsten Fall sind Bibliotheken Sammlungen von Objektdateien
- Unter Unix können sie mit dem Kommando `ar` (*archiver*) angelegt werden
- `.a`-Dateien sind so genannte statische Bibliotheken und enthalten eine oder mehrere Objektdateien
- Gegen Archive kann man wie gegen Objektdateien linken
- Auch hier dürfen Symbole nicht mehrmals vorkommen um die Eindeutigkeit beim Linken zu garantieren

Erzeugen von Bibliotheken

- Im einfachsten Fall erzeugt man (statische) Bibliotheken mit "ar" (Details zu den Option in [5])

```
>ar r 06_test.a 06_Fak.o 06_test.o  
ar: creating 06_test.a  
>nm 06_test.a
```

```
06_Fak.o:  
00000000 b .bss  
00000000 d .data  
00000000 t .text  
00000000 T _fak
```

```
06_test.o:  
00000000 b .bss  
00000000 d .data  
00000000 r .rdata  
00000000 t .text  
00000010 T _bar  
00000000 T _foo
```

- **nm** auf Bibliotheken angewandt zeigt den Inhalt aller enthaltenen Objektdateien an

Linken gegen statische Bibliotheken

- **gcc** linkt gegen statische Bibliotheken genauso wie gegen gewöhnliche Objektdateien
- Wie bei Objektdateien werden die benötigten Symbole aus der Bibliothek in die ausführbare Datei eingebunden
- In der resultierenden ausführbaren Datei kann man sich alle eingebundenen Symbole ebenfalls mit **nm** ansehen:

```
>gcc -o test 06_Fak_main.o 06_test.a
>nm test.exe
00403020 b  .bss
...
004010a0 T  __fak
00401360 T  __free
00401050 T  __main
00401000 T  __mainCRTStartup
00401370 T  __malloc
00401170 T  __printf
00401350 T  __realloc
004010ed t  done
004040a4 i  fthunk
0040406c i  hname
004010d6 t  probe
```

Make

- Größere Softwareprojekte können schnell mehr als 10^6 Zeilen Code besitzen
- Übersetzen und Binden solcher Projekte ist sehr aufwändig
- Insbesondere möchte man bei Änderungen einer Quelldatei nur die von diesen Änderungen betroffenen Dateien neu bauen
- In Java wird dies in vereinfachter Form vom Java-Compiler geleistet, für C und C++ gibt es das Werkzeug **make**
- **make** baut dabei automatisch den notwendigen Code, löst **Abhängigkeiten** korrekt auf und bestimmt welche Teile nach Änderungen neu gebaut werden müssen
- Mit einem einzigen Kommando (**make**) wird so das gesamte Projekt gebaut

Make

- Make ist per se unabhängig von der Programmiersprache und auch z.B. zum Übersetzen von LaTeX-Dokumenten oder zur Installation von Software einsetzbar
- Ohne Parameter aufgerufen liest **make** die Datei **makefile** im aktuellen Verzeichnis und verarbeitet die darin enthaltenen Regeln
- Mit der Option `-f<Dateiname>` verarbeitet **make** die Regeln aus `<Dateiname>`
- Makefile enthält
 - **Targets** (z.B. eine Liste der ausführbaren Programme oder Bibliotheken die gebaut werden sollen)
 - **Regeln** beschreiben wie diese Targets gebaut werden
 - **Makrodefinitionen** erlauben eine kompakte Beschreibung von Regeln und Targets
 - **Kommentare**
- Als Standard wird das erste Target im Makefile gebaut
- Man kann **make** auch das Target als Option übergeben
make test
würde entsprechende das Target *test* bauen

Explizite Regeln

- Eine **explizite Regel** hat folgende Struktur:

Ziel: Voraussetzungen/Abhängigkeiten

[tab] Befehle zum Erstellen des Ziels...

[tab] ...aus den Voraussetzungen

- Abhängigkeiten werden so aufgelöst, dass **make** einen **Graph von Abhängigkeiten** aufbaut und diese Abhängigkeiten Schritt für Schritt auflöst

- **Beispiel:**

```
test_fak: Fak.o Fak_main.o
```

```
...
```

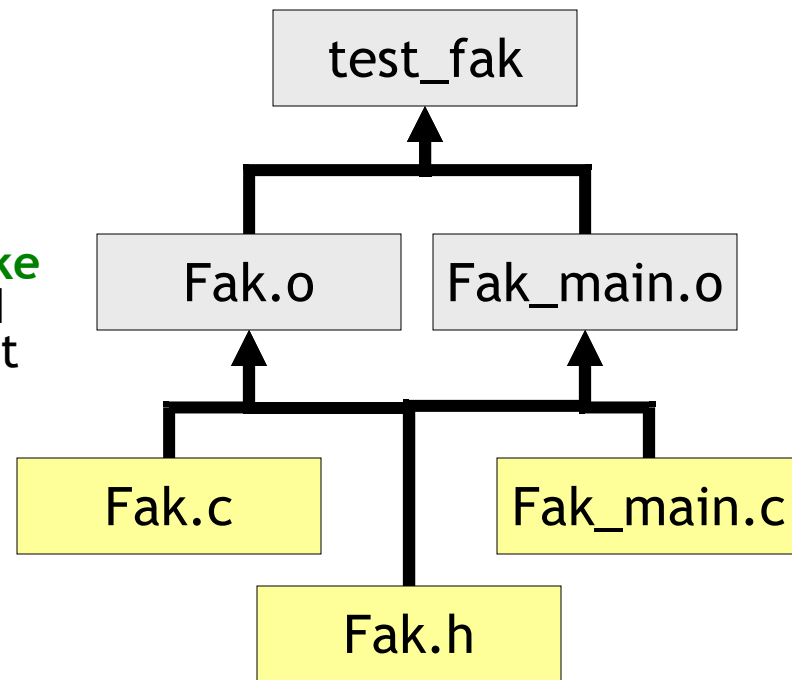
```
Fak.o: Fak.c Fak.h
```

```
...
```

```
Fak_main.o: Fak_main.c Fak.h
```

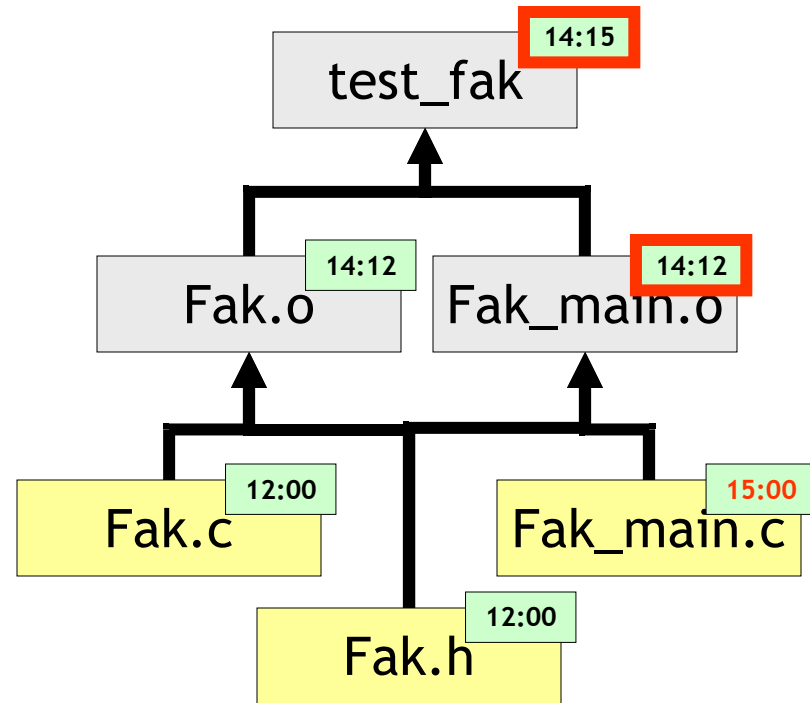
```
...
```

- Hier existieren zu Anfang nur die .c- und .h-Dateien
- Make beginnt also mit dem Erzeugen der .o-Dateien aus den .c-Dateien
- Dann sind die Abhängigkeiten für **test_fak** erfüllt und der Linker kann dieses erstellen



Abhängigkeiten

- Regeln zu einem Target werden solange abgearbeitet bis
 - ein Fehler auftritt (z.B. Compilerfehler)
 - das angegebene Target gebaut wurde
- Was passiert, wenn sich eine Quelldatei ändert?
 - **Datum und Uhrzeit** der letzten Änderung der Datei werden zugrunde gelegt um Abhängigkeiten zu definieren
 - **Ist ein Ziel älter als** eine seiner **Voraussetzungen, so muss** von dieser Stelle an im Baum aufwärts alles **neu gebaut werden** (rechts: rot umrandet)
- **Beispiel**
 - Wendet man das Kommando **touch** also z.B. auf Fak_main.c an, so muss **make** Fak_main.o und test_fak neu erzeugen
 - Eine Änderung von Fak.h erzeugt alle .o-Dateien und test_fak neu



Hinweis:

Der Unix-Befehl **touch** ändert das Modifikationsdatum/-uhrzeit einer Datei.

Makefile - Beispiel

Kommentarzeilen
beginnen mit "#"

Erstes Target: test_fak
(Target =
explizite Regel)

Zeilen nach
Regeldefinition
enthalten
auszuführende
Befehle
und müssen mit
einem TAB
beginnen!

```
# 06_Makefile_Fak_1

test_fak: 06_Fak.o 06_Fak_main.o
    gcc -o test_fak 06_Fak.o 06_Fak_main.o

06_Fak.o: 06_Fak.c 06_Fak.h
    gcc -c 06_Fak.c

06_Fak_main.o: 06_Fak_main.c 06_Fak.h
    gcc -c 06_Fak_main.c
```

Nach dem ":" in der
Regeldefinition
stehen die
Abhängigkeiten

```
>make -f 06_Makefile_Fak_1
gcc -c 06_Fak.c
gcc -c 06_Fak_main.c
gcc -o test_fak 06_Fak.o 06_Fak_main.o
```

Makros

- Makrodefinitionen in Makefiles haben prinzipiell die folgende Syntax
`<name>=<value>`
- Damit wird eine neue Makro (eine Variable mit dem Namen `<name>`) eingeführt, auf die mit `$ (<name>)` zugegriffen werden kann

- **Beispiel:**

```
CC=gcc
```

definiert eine Variable die den Namen des C-Compilers enthält

In expliziten Regeln kann man nun dieses Makro verwenden:

```
test_fak: Fak.o Fak_main.o
```

```
$(CC) -o test_fak Fak.o Fak_main.o
```

Durch Ändern dieses Makros lässt sich so an allen Stellen im Makefile ein neuer Compiler einführen.

Makros

- Makrodefinitionen auch Musterersetzung auf den Werten existierender Makros vornehmen
- Dies kann nützlich sein um z.B. in einer Definition die Dateiendungen oder Pfade zu ersetzen
- **Beispiel:**

```
OBJECTS=$(SOURCES : .c = .o)
```

ersetzt alle Suffixe `.c` mit `.o` im Wert von `SOURCES` und weist das Ergebnis `OBJECTS` zu

Hat z.B. `SOURCES` den Wert "test.c class.c", so wird nach der Zuweisung `OBJECTS` auf "test.o class.o" gesetzt

- Diese Art von Substitutionen ist in Makefiles recht häufig anzutreffen

Makros

- **make** definiert eine Reihe von besonderen Makros, die nur innerhalb der aktuellen Regel gültig sind:

<code>\$@</code>	Ziel der Regel
<code>\$<</code>	Erste Abhängigkeit der Regel
<code>\$\$</code>	Alle Abhängigkeiten durch Leerzeichen getrennt
<code>\$\$?</code>	Alle Abhängigkeiten, die neuer sind als das Ziel

- **Beispiel:**

Mit diesen Variablen kann man viele Regeln vereinfachen:

```
test_fak: Fak.o Fak_main.o
    $(CC) -o $@ $$

Fak.o: Fak.c Fak.h
    $(CC) -c $<
```

Implizite Regeln

- **Implizite Regeln** halten Makefiles kurz
- Alle .c-Dateien werden z.B. mit dem gleichen Kommando ("gcc -c") in Objektdateien umgewandelt
- Statt für jede Objektdatei **explizit** anzugeben, wie sie übersetzt wird, definiert man einmal allgemein wie aus .c-Dateien .o-Dateien werden
- Eine **implizite Regel passt auf Muster**
- Implizite Regeln sind ähnlich wie explizite Regeln aufgebaut, als Ziel wird aber ein Muster angegeben
- **Beispiel:**

```
.c.o:
```

```
$(CC) -c $<
```

- Regeln werden in der **Reihenfolge der Definition** abgearbeitet: ist eine implizite Regel vor einer expliziten Regel definiert, so wird die implizite Regel verwendet ⇒ am Ende definieren!
- Besitzt das Makefile keine explizite Regel für Fak.o, so sucht **make** nach einer impliziten Regel (und hier also nach Fak.c), um obige Regel zur Anwendung zu bringen

Sonstiges

- **Includes**
 - Mit dem Kommando "`include Dateiname`" kann eine Datei (analog zum C-Präprozessor!) in das Makefile eingebunden werden
- **Ausgabe**
 - Steht am Anfang eines Kommandos ein "@", so wird das Kommando bei der Ausführung nicht angezeigt
- **Abbruch**
 - **make** bricht beim ersten Fehler (d.h. ein Kommando, das bei der Ausführung nicht 0 zurückgibt) ab
 - Steht vor dem Kommando ein "-", so werden Fehler ignoriert
- **Lange Zeilen**
 - Ist eine Zeile zu lang oder zu übersichtlich, so kann man sie mit einem "\" am Ende der Zeile umbrechen
- **Beispiel:**

```
OBJECTS=\
  test1.o\
  test2.o

-include test.mak
test:
  @echo "Echo 1: $(OBJECTS) "
  echo "Echo 2: $(OBJECTS) "
```

```
>make
Echo 1: test1.o test2.o
echo "Echo 2: test1.o test2.o"
Echo 2: test1.o test2.o
```

Abhängigkeiten bestimmen

- Abhängigkeiten entstehen in der Regel durch das Einbinden anderer Dateien mit `#include`
- Einen Teil dieser Abhängigkeiten kann man automatisch vom Compiler finden lassen
- Ruft man `gcc` mit der Option `-M` auf, so werden alle eingebundenen Dateien in einem direkt für Makefiles verwendbaren Format ausgegeben:

```
>gcc -M 06_Fak.c 06_Fak_main.c
06_Fak.o: 06_Fak.c
06_Fak_main.o: 06_Fak_main.c /usr/include/stdio.h \
/usr/include/_ansi.h \
/usr/include/newlib.h /usr/include/sys/config.h \
...
/usr/include/sys/sysmacros.h /usr/include/stdint.h \
/usr/include/endian.h /usr/include/sys/stdio.h \
/usr/include/sys/cdefs.h 06_Fak.h
```

- Die Vielzahl der Dateien kommt daher, dass die Standardheader wie `stdio.h` ihrerseits wieder eine ganze Reihe anderer Header inkludieren

Abhängigkeiten bestimmen

- Da man in der Regel Abhängigkeiten von **Systemheadern** nicht benötigt (sie werden vom Benutzer nicht verändert!), gibt es die Option "**-MM**", die diese Dateien außer Acht lässt:

```
>gcc -MM 06_Fak.c 06_Fak_main.c
06_Fak.o: 06_Fak.c
06_Fak_main.o: 06_Fak_main.c 06_Fak.h
```

- Die derart erzeugten Regeln können einfach an ein Makefile angehängt werden und legen dort die Abhängigkeiten fest
- Existieren mehrere Targets mit dem gleichen Namen, so werden die Abhängigkeiten nicht überschrieben, sondern addiert:

```
test_fak: Fak.o
test_fak: Fak_main.o
```

ist also gleichbedeutend mit

```
test_fak: Fak.o Fak_main.o
```

Abhängigkeiten automatisch finden

- Viele Makefiles lassen die Abhängigkeiten automatisch bestimmen und in eine Datei `.Dependencies` schreiben
- Dazu existiert in der Regel ein Ziel namens `depend`
- Dieses Ziel besitzt keine Abhängigkeiten, aber andere Ziele hängen oft davon ab, um sicherzustellen, dass die Abhängigkeiten automatisch konstruiert werden
- "make depend" baut diese Abhängigkeiten, mit include werden sie in das Makefile selbst eingebunden

- **Beispiel:**

```
-include .Dependencies
```

```
depend:
```

```
gcc -MM $(SOURCES) > .Dependencies
```

- Das "-" vor dem include stellt dabei sicher, dass **make** nicht abbricht, wenn `.Dependencies` noch nicht erzeugt wurde (ansonsten ist "make depend" nicht ausführbar!)

Clean

- Die meisten Makefiles enthalten ein spezielles Ziel - `clean` -, um angelegte Dateien auch wieder zu entfernen
- Dieses Ziel besitzt in der Regel keine Abhängigkeiten
- Mit "`make clean`" kann man temporär erzeugte Objektdateien löschen, z.B. um den Quellcode zusammenzupacken ohne große Objektdateien mit einzupacken
- Auch zum Zusammenpacken existieren oft entsprechende Targets
- **Beispiel:**

```
clean:
```

```
    -@rm $(OBJECTS) $(EXECUTABLES)
```

```
    -@rm .Dependencies
```

Beispiel

```
# 06_Makefile_Fak_2
CC=gcc
CFLAGS=-O3
LDFLAGS=

EXECUTABLE=test_fak
SOURCES=\
    06_Fak.c\
    06_Fak_main.c

OBJECTS=$(SOURCES:.c=.o)

$(EXECUTABLE): $(OBJECTS)
    $(CC) $(LDFLAGS) -o $@ $^

clean:
    -rm $(OBJECTS) $(EXECUTABLE)

%.o:%.c
    $(CC) $(CFLAGS) -c $<
```

Make - Tipps

- Achten Sie auf **Tabs und Leerzeichen!**
- Kommandos nach einem Ziel müssen mit einem Tab an erster Stelle eingerückt sein - nicht mit Leerzeichen!
- Erhalten Sie eine Fehlermeldung ähnlich zu

```
Makefile:14: *** missing separator. Stop.
```

so ist vermutlich ein Leerzeichen statt einem Tab eingebaut

- Um sich die Inhalte von **Variablendefinitionen** und alle definierten **Regeln** ausgeben zu lassen, können Sie "**make -p**" verwenden; die Ausgabe ist aber recht unübersichtlich
- Eine Alternative ist ein **Debugging-Ziel**:

```
debug:
```

```
    echo $(OBJECTS)
```

```
    echo $(SOURCES)
```

gibt nach Aufruf von "**make debug**" die Inhalte der Variablen auf der Konsole aus

- Wenn Sie nicht sicher sind, ob Ihr Makefile das richtige tun wird - besonders, wenn Sie ein "clean"-Target implementieren, rufen Sie "**make -n**" auf; alle Kommandos werden angezeigt, aber nicht ausgeführt!

Literatur

Literatur zu diesem Teil der Vorlesung:

[1] Dokumentation zu GNU make:

<http://directory.fsf.org/make.html>

[2] Dokumentation zum GCC:

<http://gcc.gnu.org/onlinedocs/>

[3] Ein sehr gutes Tutorial zu Makefiles von der University of Hawaii:

<http://www.eng.hawaii.edu/Tutor/Make/>

[4] Manpage zu nm ("man nm")

[5] Manpage zu ar ("man ar")