Vorlesungsbegleiter für Systemnahe Software I

Kapitel 9: Modularisierung in C

Andreas F. Borchert
12. Dezember 2024

Lernziele:

- Überblick der Techniken zur Herstellung der Schnittstellensicherheit in C
- Verständnis des Werkzeugs *make* und des Aufbaus einfacher *Makefile*-Dateien
- Fähigkeit, objekt-orientierte Techniken mit Hilfe von Funktionszeigern und abstrakten Datentypen in C umzusetzen und Module dynamisch nachzuladen

9.1 Deklarationen vs. Definitionen

Bei Funktionen und Variablen wird zwischen Deklarationen und Definitionen unterschieden. Eine Deklaration teilt dem Übersetzer alle notwendigen Informationen mit, so dass die entsprechende Variable oder Funktion anschließend verwendet werden kann. Eine Definition führt im Falle von Variablen dazu, dass entsprechend Speicherplatz belegt wird und, falls erwünscht, eine Initialisierung erfolgt. Eine Funktions-Definition schließt den Programmtext der Funktion mit ein. Mehrfache Deklarationen des gleichen Objekts sind zulässig, sofern sie genau übereinstimmen. Jedes Objekt darf aber nur ein einziges Mal definiert werden. Typdeklarationen, sei es mit **enum**, **struct**, **union** oder **typedef**, dürfen ebenfalls nur einfach erfolgen, wobei unvollständige **enum**-, **struct**- und **union**-Deklarationen ohne Angabe der Namen oder Komponenten (*members*) zulässig sind, die mit der vollständigen Definition ergänzt werden dürfen.

Einige Beispiele:

```
/* declaration of a function */
double sin(double x);

/* definition of a function */
int max(int a, int b) {
    return a >= b? a: b;
}

/* definition of a variable */
int i = 42;
```

Wie kann jedoch eine Variable nur deklariert werden, ohne sie zu definieren? Hierfür gibt es das Schlüsselwort **extern**, das einer Deklaration vorausgeht. Prinzipiell dürfte dieses Schlüsselwort auch Funktionsdeklarationen vorausgehen – aber dort ergibt sich der Unterschied aus Deklaration und Definition bereits aus der Abwesen- bzw. Anwesenheit des zugehörigen Programmtexts. Beispiel:

```
/* variable declaration */
extern int foo;
```

Im Falle einer Funktions- oder Variablen-Deklaration akzeptiert der Übersetzer eine darauf folgende Verwendung der entsprechenden Funktion oder Variablen. Dennoch muss die jeweilige Funktion oder Variable irgendwo definiert werden. Unterbleibt dies, so gibt es eine Fehlermeldung bei einem Versuch, das Programm zusammenzubauen.

Variablen können global oder lokal sichtbar sein. Mit **extern** deklarierte Variablen sind jedoch immer global. Im folgenden Beispiel wird zuerst der Wert der lokalen Variable *foo* ausgegeben. Die **extern**-Deklaration für *foo* innerhalb des lokalen Blocks bezieht sich aber auf die globale Variable namens *foo*. Entsprechend wird dort 42 ausgegeben:

```
#include <stdio.h>

int foo = 42; /* definition of a global variable */

int main(void) {
    int foo = 7; /* definition of a local variable */
    printf("foo␣=␣%d\n", foo);
    {
        extern int foo; /* declaration that refers to the global variable */
        printf("foo␣=␣%d\n", foo);
    }
}
```

9.2 Aufteilung eines Programms in Übersetzungseinheiten

Ein Programm besteht aus beliebig vielen Übersetzungseinheiten und jede Übersetzungseinheit ist eine Folge von Deklarationen und Definitionen. In C gibt es einen globalen Namensraum – alle Funktionen und alle globale Variablen gehören dazu, es sei denn, sie sind **static** deklariert. Im letzteren Fall erstreckt sich die Sichtbarkeit dann nur über eine Übersetzungseinheit.

Gegeben sei folgendes einfache Beispiel mit zwei Übersetzungseinheiten, *main.c* und *lib.c*. In ersterer wird *main* definiert, in letzterer die Variable *counter* und die Funktion *f*:

```
#include <stdio.h>

/* declarations */
extern int counter;
extern void f();
```

```
int main(void) {
    printf("value␣of␣counter␣before␣invoking␣f():␣%d\n", counter);
    f();
    printf("value␣of␣counter␣after␣invoking␣f():␣%d\n", counter);
}
```

```
/* definition of variable counter */
int counter = 1;

/* definition of function f */
void f(void) {
    ++counter;
}
```

Die Übersetzungseinheiten können in einer beliebigen Reihenfolge übersetzt werden. Dann entstehen jeweils Objekt-Dateien, die aus dem durch die Übersetzung entstandenen Maschinen-Code und einer Symboltabelle bestehen. Für Objekt-Dateien gibt es mehrere Formate, es hat sich aber weitgehend das *Executable and Linking Format* (ELF) durchgesetzt, das 1989 in Verbindung mit Unix System VR4 kam und später standardisiert worden ist,[1] jedoch nicht Teil des POSIX-Standards ist. Die Symboltabelle nennt primär die globalen Symbole, die entweder nur in einer Übersetzungseinheit oder über alle Übersetzungseinheiten hinweg sichtbar sind. Mit dem Werkzeug *nm*[2] ist es möglich, eine Symboltabelle in lesbarer Form auszugeben:

```
theon$ ls
lib.c   main.c
theon$ gcc -Wall -c main.c
theon$ gcc -Wall -c lib.c
theon$ ls
lib.c   lib.o   main.c  main.o
theon$ nm lib.o
0000000000000000 D counter
0000000000000000 T f
theon$ nm main.o
                 U counter
                 U f
0000000000000000 T main
                 U printf
theon$
```

Die Option „-c“ steht für „compile“ und veranlasst den *gcc*, auf den Zusammenbau zu einem ausführbaren Programm zu verzichten. Nach der Übersetzung entstanden die beiden Objekt-Dateien *lib.o* und *main.o*.

Die von *nm* ausgegebenen Symboltabellen bestehen aus drei Spalten, die die Adresse in hexadezimaler Darstellung, das Segment bzw. die Art des Symbols und schließlich den Namen

[1] Siehe *Executable and Linking Format* (*ELF*) *Specification*, 1995. Adresse: `https://refspecs.linuxfoundation.org/elf/elf.pdf`.

[2] Gehört zum POSIX-Standard, siehe nm.

des Symbols angeben. Jede Objekt-Datei kann aus mehreren Segmenten bestehen, wozu insbesondere das Text- und das Datensegment gehören. Die Aufteilung in Segmenten erfolgt, um diese Bereiche später in getrennten Kacheln an unterschiedlichen Stellen im Adressraum mit unterschiedlichen Zugriffsprivilegien zu laden. So wird das Textsegment typischerweise nur zum Lesen und Ausführen geladen, erlaubt jedoch keine Schreiboperationen. Das Datensegment erlaubt Lese- und Schreiboperationen, verhindert jedoch das Ausführen.

Die Segmente werden durch Buchstaben gekennzeichnet, so steht „D“ für ein Datensegment und „T“ für ein Textsegment. Die gleichen Buchstaben werden auch als Kleinbuchstaben verwendet, wenn die Symbole nur lokal in einer Übersetzungseinheit definiert sind und somit nicht in Konflikt zu anderen Symbolen des gleichen Namens in anderen Übersetzungseinheiten sind. „U“ steht für „unknown“ und kommt zum Zuge, wenn nur eine Deklaration vorliegt, jedoch keine Definition. In diesem Falle werden keine Annahmen getroffen, in welchem Segment das anderswo definiert sein könnte.

Die Adressen sind jeweils nur relativ zu den einzelnen Segmenten innerhalb eines Objekts. Erst beim Zusammenbau eines Programms werden aus relativen Adressen absolute Adressen, wie sie später im Adressraum zur Anwendung kommen.

Der Zusammenbau erfolgt durch den *ld* (*loader* oder *linkage editor*), der von dem *gcc* aufgerufen wird:[3]

```
theon$ gcc -o main lib.o main.o
theon$ ls
lib.c   lib.o   main    main.c  main.o
theon$ nm main | egrep ' f$|counter|main|printf'
0000000000600c70 D counter
0000000000400812 T f
0000000000400828 T main
                 U printf@@SUNW_0.7
theon$
```

Der Zusatz „@@SUNW_0.7“ ist nicht Teil des Symbolnamens, sondern eine Versionsnummer. Diese gibt es nur in Verbindung mit zur Laufzeit nachladbaren Bibliotheken (*shared libraries*). Wenn ein Programm mit einer älteren Version übersetzt worden ist, dann wird die entsprechend ältere Fassung nachgeladen, selbst wenn eine neuere Version inzwischen zur Verfügung stehen sollte. Dies ermöglicht es, bei inkompatiblen Schnittstellenänderungen sowohl die ältere als auch die neuere Fassung zeitgleich zur Verfügung zu stellen. Diese Technik wurde bei Solaris 2.5 eingeführt und kurz danach auch von Linux übernommen.[4]

Durch den Zusammenbau wurden durch den *ld* die aus den Deklarationen hervorgehenden undefinierten Symbole in *main.o* den definierten Symbolen aus *lib.o* zugeordnet. Symbole aus der C-Bibliothek wie *printf* bleiben noch undefiniert und werden erst zur Laufzeit durch den dynamischen Lader aufgelöst.

[3] Keines dieser Kommandos wird im POSIX-Standard erwähnt. Als Übersetzer für C wird nur c17 genannt, offenbar in Bezug auf den 2017 finalisierten und 2018 veröffentlichten C-Standard: *Information technology – Programming languages – C*. International Organization for Standardization, 2018. Adresse: `https://www.iso.org/standard/74528.html`.

[4] Siehe D. J. Brown und K. Runge, „Library interface versioning in Solaris and Linux“, in *4th Annual Linux Showcase & Conference (ALS 2000)*, 2000. Adresse: `https://www.usenix.org/publications/library/proceedings/als00/2000papers/papers/full_papers/browndavid/browndavid.pdf` und *VERSION Command*, 4. Aug. 2024. Adresse: `https://sourceware.org/binutils/docs/ld/VERSION.html` (besucht am 05. 12. 2024).

Dabei ist zu beachten, dass der Zusammenbau eines Programms durch den *ld* nur auf den Symbolen beruht. Eine semantische Information wie etwa der zugehörige Typ liegt dem *ld* nicht vor bzw. wird von diesem nicht ausgewertet.[5] Entsprechend gibt es in C keine Sicherheit, dass zusammengebaute Teile auch zusammenpassen. Dies wird durch folgendes Beispiel belegt, bei dem *f* in *m1.c* als Funktion deklariert wird und in *m2.c* als Variable des Typs **double** deklariert wird:

```
theon$ cat m1.c
extern void f();
int main() {
   f();
}
theon$ cat m2.c
double f = 1.0;
theon$ gcc -Wall -c m1.c
theon$ gcc -Wall -c m2.c
theon$ gcc -o m m1.o m2.o
theon$ ./m
Segmentation Fault (core dumped)
theon$
```

Bei keinem der beiden Übersetzungsläufe kann der Übersetzer die Inkonsistenz entdecken und für den *ld* sind die Unterschiede irrelevant, so dass ohne jegliche Warnungen ein Zusammenbau möglich ist. Die Ausführung des Programms scheitert in diesem Beispiel an dem Versuch, die *double*-Variable als Funktion auszuführen. Aber ein Absturz ist nicht garantiert – insbesondere, wenn die Fehler sehr viel subtiler sind mit nicht übereinstimmenden Parameterzahlen oder Typen bei Variablen, Rückgabewerten und Parametern.

9.3 Herstellung der Schnittstellensicherheit

Der erste und wichtigste Schritt ist die Auslagerung aller Deklarationen externer Variablen und Funktionen in Header-Dateien. Dies vermeidet nicht nur die Existenz voneinander abweichender Deklarationen in unterschiedlichen Quellen, sondern ermöglicht die Überprüfung sowohl der Funktionsaufrufe oder Variablenverwendungen als auch der Funktions- oder Variablendefinitionen gegen die gleiche Deklaration. Wenn die Verwendung und auch die Definition einer Funktion oder Variablen zu der gleichen Deklaration konsistent sind, dann passen sie zusammen.

Folgendes Beispiel demonstriert dies für eine Funktion zur Berechnung des größten gemeinsamen Teilers, die von dem Hauptprogramm getrennt wurde. Die Header-Datei *gcd.h* enthält die Deklaration der Funktion *gcd*:

```
int gcd(int a, int b);
```

Die zugehörige Implementierung in der Datei *gcd.c* zieht die Header-Datei mit ein. Dies erlaubt dem Übersetzer, die Deklaration in der Header-Datei mit der Definition der Funktion *gcd* zu vergleichen:

[5] Typinformationen sind optional – sie werden typischerweise durch die Option „-g“ generiert und primär von den Debuggern ausgewertet.

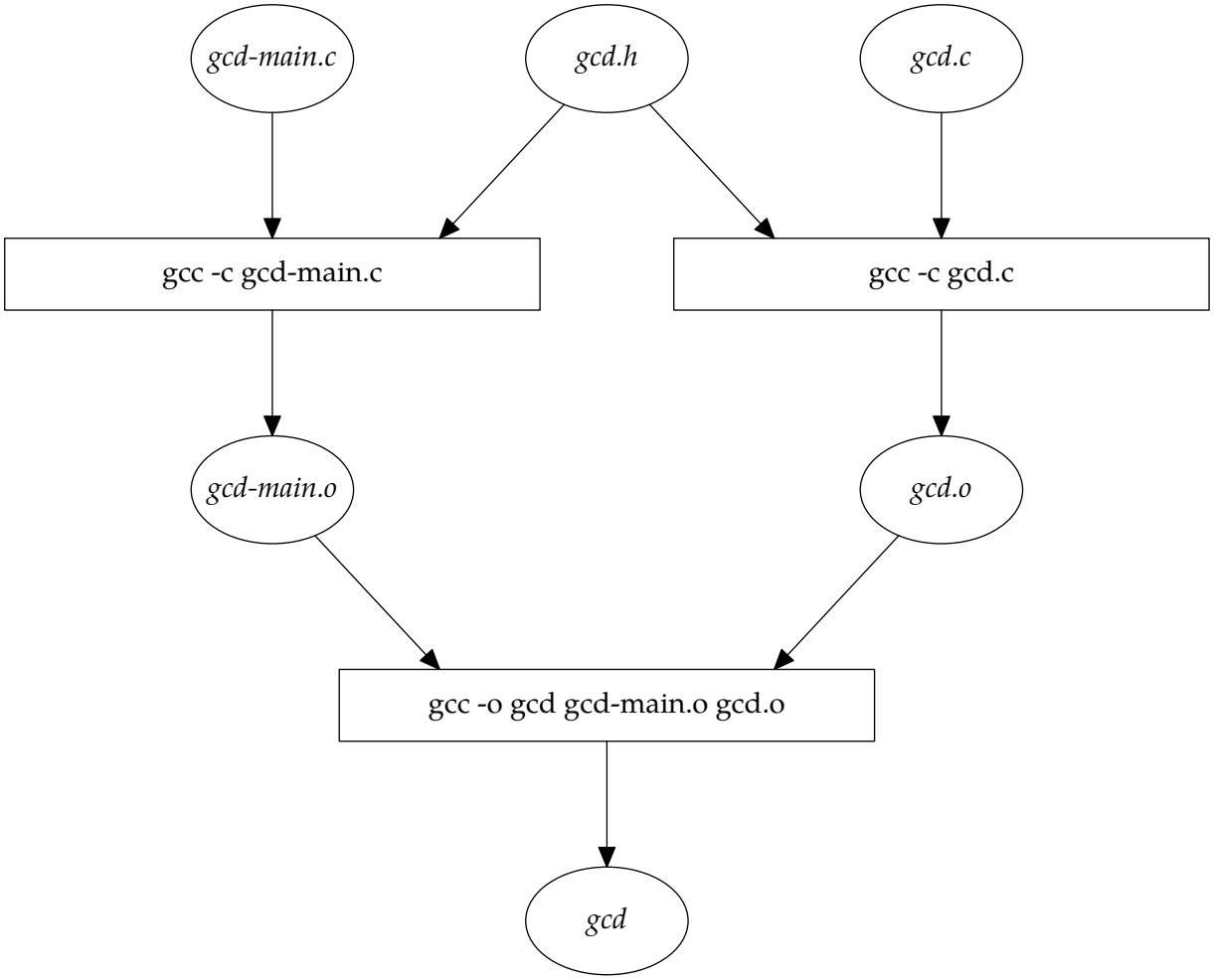

Abbildung 9.1: Übersetzungsvorgang bei ausgelagerten Deklarationen

```
#include <assert.h>

#include "gcd.h"

int gcd(int a, int b) {
    assert(a > 0 && b > 0);
    while (a != b) {
        if (a > b) {
            a -= b;
        } else {
            b -= a;
        }
    }
    return a;
}
```

Im Hauptprogramm wird ebenfalls die Deklaration aus *gcd.h* einbezogen:

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

#include "gcd.h"

int main(int argc, char* argv[]) {
    char* cmdname = *argv++; --argc;
    char usage[] = "Usage: %s a b\n";
    if (argc != 2) {
        fprintf(stderr, usage, cmdname); exit(1);
    }

    int a; int b;
    if (sscanf(argv[0], "%d", &a) != 1 ||
            sscanf(argv[1], "%d", &b) != 1) {
        fprintf(stderr, usage, cmdname); exit(1);
    }
    if (a <= 0 || b <= 0) {
        fprintf(stderr, "%s: values must be positive\n", cmdname);
        exit(1);
    }

    printf("%d\n", gcd(a, b));
}
```

Obwohl die Übersetzungen von *gcd.c* und *gcd-main.c* unabhängig voneinander erfolgen, gibt es mit *gcd.h* einen gemeinsamen Teil, der bei beiden Übersetzungen vorliegt (siehe Abb. 9.1). Innerhalb von *gcd.h* wird die Funktion *gcd* deklariert. Wenn *gcd.c* übersetzt wird, so kann diese

Deklaration mit der darauf folgenden Definition verglichen werden. Abweichungen führen hier zu einer Fehlermeldung. Analog wird der Aufruf der Funktion *gcd* innerhalb von *gcd-main.c* anhand der vorliegenden Deklaration überprüft. Auf diese Weise lässt sich sicherstellen, dass *gcd-main.o* und *gcd.o* in Bezug auf die Funktion *gcd* zueinander passen, wenn sie von *ld* zu *gcd* zusammengebaut werden:

```
theon$ gcc -Wall -c gcd-main.c
theon$ gcc -Wall -c gcd.c
theon$ gcc -o gcd gcd-main.o gcd.o
theon$ ./gcd 36 48
12
theon$
```

9.4 Neu-Übersetzungen unter Berücksichtigung der Abhängigkeiten

Die vorgestellte Lösung funktioniert nur unter einer wichtigen Voraussetzung: Wenn eine Header-Datei verändert wird, müssen alle Übersetzungseinheiten neu übersetzt werden, die diese Header-Datei direkt oder indirekt über **#include** einbeziehen. Unterbleibt dieses, dann ist die Schnittstellensicherheit nicht mehr gewährleistet, da dann die einzelnen Übersetzungseinheiten mit nicht zueinander kompatiblen Versionen einer Header-Datei übersetzt sein könnten. Die Gefahr ist durchaus gegeben, da bei einer umfangreichen Zahl von Übersetzungseinheiten und Header-Dateien es rasch unübersichtlich wird, welche C-Quellen wegen einer Schnittstellenänderung neu zu übersetzen sind.

Für dieses Problem wurde 1977 in den Bell Laboratories das Werkzeug *make* von Stuart Feldman entwickelt.[6] und es gibt zahlreiche erweiterte Fassungen, wozu insbesondere GNU make gehört.[7] Alle heutigen Varianten dieses Werkzeugs haben die wesentlichen Eigenschaften des Originals übernommen. Die Grundidee liegt darin, eine Datei namens *makefile* (oder *Makefile*) zusammen mit den Quellen im gleichen Verzeichnis anzulegen, das

- die Abhängigkeiten der Dateien (sowohl Quellen als auch die erzeugbaren Dateien) spezifiziert und
- angibt, mit welchen Kommandos bei Bedarf die erzeugbaren Dateien hergestellt werden können.

Die Abhängigkeiten und Erzeugungskommandos können für das vorgestellte Beispiel wie folgt in einem *makefile* repräsentiert werden:

```
gcd: gcd.o gcd-main.o
        gcc -o gcd gcd.o gcd-main.o
gcd.o: gcd.h gcd.c
        gcc -c gcd.c
gcd-main.o: gcd.h gcd-main.c
        gcc -c gcd-main.c
```

[6] Siehe S. I. Feldman, „Make — a program for maintaining computer programs“, *Software: Practice and Experience*, Jg. 9, Nr. 4, S. 255–265, 1979. DOI: https://doi.org/10.1002/spe.4380090402. Das Werkzeug ist Teil des POSIX-Standards, siehe make.

[7] Siehe https://www.gnu.org/software/make/.

Zeilen, die nicht mit einem Tabulator-Zeichen beginnen, nennen eine erzeugbare Datei. Es folgen ein Doppelpunkt, ggf. Leerzeichen und dann (noch auf der gleichen Zeile) die Dateinamen, wovon die erzeugbare Datei abhängt. Im konkreten Fall hängt beispielsweise *gcd* von den Dateien *gcd.o* und *gcd-main.o* ab. Die darauffolgenden Zeilen, die jeweils mindestens mit einem führenden Tabulator-Zeichen beginnen müssen, enthalten die Kommandos, mit denen die zuvor genannte Datei erzeugt werden kann. Dabei darf jeweils vorausgesetzt werden, dass die Dateien, wovon die erzeugende Datei abhängt, bereits existieren. Im obigen Beispiel wird so zu Beginn das Kommando „gcc -o gcd gcd.o gcd-main.o“ angegeben, das die Datei *gcd* erzeugen kann, sobald *gcd.o* und *gcd-main.o* vorliegen.

Beliebig vieler solcher erzeugbaren Dateien, deren Abhängigkeiten und die zugehörigen erzeugenden Kommandos können in einem *makefile* aufgeführt werden. Wenn das Kommando *make* aufgerufen wird, kann entweder die gewünschte Datei auf der Kommandozeile angegeben werden oder es wird implizit die erste im *makefile* genannte erzeugbare Datei ausgewählt (im Beispiel wäre das *gcd*).

Das Werkzeug *make* geht dann beginnend mit dem gegebenen Ziel rekursiv wie folgt vor:

1. Sei Z das Ziel. Wenn das Ziel im *makefile* nicht explizit genannt ist, jedoch als Datei existiert, dann ist nichts weiter zu tun. (Falls das Ziel wieder als Datei noch als Regel existiert, dann gibt es eine Fehlermeldung.)

2. Andernfalls ist innerhalb des *makefile* eine Abhängigkeit gegeben in der Form

   $$Z : A_1 \ldots A_n,$$

   wobei die Folge $\{A_i\}_1^n$ leer sein kann ($n = 0$). Wenn $n > 0$, dann ist der Algorithmus (beginnend mit Schritt 1) rekursiv aufzurufen für jede der Dateien $A_1 \ldots A_n$.

3. Sobald alle Dateien $A_1 \ldots A_n$ in aktueller Form vorliegen, wird überprüft, ob der Zeitstempel von Z (letztes Schreibdatum) jünger ist als der Zeitstempel jede der Dateien $A_1 \ldots A_n$.

4. Falls es ein A_i gibt, das neueren Datums ist als Z, dann werden die zu Z gehörenden Kommandos ausgeführt, um Z neu zu erzeugen.

So sieht für das obige Beispiel ein Aufruf von *make* aus, wenn alle erzeugbaren Dateien (*gcd*, *gcd.o* und *gcd-main.o*) fehlen:

```
theon$ make
gcc -c gcd.c
gcc -c gcd-main.c
gcc -o gcd gcd.o gcd-main.o
theon$ make
make: `gcd' is up to date.
theon$ ./gcd 48 112
16
theon$
```

Hier ist auch zu sehen, dass bei einem unmittelbar folgenden zweiten Aufruf von *make* nichts passiert, da *gcd* immer noch existiert und aktuell ist. Das Werkzeug *make* kann auch darum gebeten werden, die genaue Vorgehensweise zu dokumentieren:[8]

[8]Die Option „–debug“ wird von GNU make unterstützt und ist nicht Teil des POSIX-Standards.

```
theon$ make --debug | sed -n '/^Updating/,$p'
Updating goal targets....
 File `gcd' does not exist.
   File `gcd.o' does not exist.
  Must remake target `gcd.o'.
gcc -c gcd.c
  Successfully remade target file `gcd.o'.
   File `gcd-main.o' does not exist.
  Must remake target `gcd-main.o'.
gcc -c gcd-main.c
  Successfully remade target file `gcd-main.o'.
Must remake target `gcd'.
gcc -o gcd gcd.o gcd-main.o
Successfully remade target file `gcd'.
theon$
```

Interessant ist dann der Fall, bei dem die durch *ggt.h* repräsentierte Schnittstelle aktualisiert wird, aber alle erzeugbaren Dateien existieren:

```
theon$ touch gcd.h
theon$ make --debug | sed -n '/^Updating/,$p'
Updating goal targets....
   Prerequisite `gcd.h' is newer than target `gcd.o'.
  Must remake target `gcd.o'.
gcc -c gcd.c
  Successfully remade target file `gcd.o'.
   Prerequisite `gcd.h' is newer than target `gcd-main.o'.
  Must remake target `gcd-main.o'.
gcc -c gcd-main.c
  Successfully remade target file `gcd-main.o'.
 Prerequisite `gcd.o' is newer than target `gcd'.
 Prerequisite `gcd-main.o' is newer than target `gcd'.
Must remake target `gcd'.
gcc -o gcd gcd.o gcd-main.o
Successfully remade target file `gcd'.
theon$
```

Eines der verbliebenen Probleme ist die korrekte Extraktion der Abhängigkeiten, d. h. wie wird verlässlich ermittelt, welche Header-Dateien direkt (oder indirekt!) von einer C-Quelle über **#include** einbezogen werden. Hierfür gibt es mehrere Werkzeuge für C, wovon das von Todd Brunhoff für X-Windows entwickelte *makedepend* die älteste Lösung ist.[9] Eine generelle Lösung gibt es nicht, da keines der Werkzeuge in den POSIX-Standard aufgenommen worden ist.

Beim *gcc* bietet sich hier auch die Option „-M“ an. Der Vorteil liegt hier darin, dass der Präprozessor von *gcc* dieses unmittelbar analysiert und dabei auch den für *gcc* konfigurierten Suchpfad für **#include**-Direktiven berücksichtigt, den davon unabhängige Werkzeuge wie *makedepend* nicht kennen. Hierbei sind jeweils in der Kommandozeile alle Übersetzungseinheiten anzugeben sowie die für den Präprozessor relevanten Übersetzungsoptionen, da diese Einfluß auf die Wahl der Header-Dateien ausüben können.

[9] Siehe J. Fulton, *Configuration management in the X Window system*, 1989. Adresse: `https://citeseerx.ist.psu.edu/document?doi=400c05711b3bffd7ef9e07108c6032767a343701`.

Die Ausgabe von „gcc -M gcd.c gcd-main.c“ ist recht umfangreich, da dann auch alle System-Header-Dateien mit aufgeführt werden. Die Ausgabe ist bei der Option „-MM“ etwas überschaubarer, da dann die System-Header weggelassen werden:

```
theon$ gcc -MM gcd.c gcd-main.c
gcd.o: gcd.c gcd.h
gcd-main.o: gcd-main.c gcd.h
theon$
```

Wie hier zu sehen, werden nur die Abhängigkeiten angegeben, aber nicht die zugehörigen Kommandos. *make* kennt bereits einige Regeln und „weiß“ daher, dass eine „.o“-Datei aus einer „.c“-Datei mit Hilfe des *cc* erzeugt werden kann.[10] Mit dem Setzen einiger Variablen wie *CC* oder *CFLAGS* bei *make* lassen sich dann diese voreingestellten Regeln anpassen:

```
theon$ cat makefile-updated
CC = gcc
CFLAGS = -Wall
gcd:            gcd.o gcd-main.o
gcd.o: gcd.c gcd.h
gcd-main.o: gcd-main.c gcd.h
theon$ make -f makefile-updated
gcc -Wall   -c -o gcd.o gcd.c
gcc -Wall   -c -o gcd-main.o gcd-main.c
gcc   gcd.o gcd-main.o   -o gcd
theon$
```

Das Aktualisieren eines *Makefile* kann auch direkt übernommen werden von Werkzeugen wie *makedepend* oder durch die auf „gcc -MM“ basierende Fassung *gcc-makedepend*:[11]

```
theon$ cat makefile-depend
CC = gcc
CFLAGS = -Wall
gcd:            gcd.o gcd-main.o
theon$ gcc-makedepend -f makefile-depend gcd.c gcd-main.c
theon$ cat makefile-depend
CC = gcc
CFLAGS = -Wall
gcd:            gcd.o gcd-main.o
# DO NOT DELETE
gcd.o: gcd.c gcd.h
gcd-main.o: gcd-main.c gcd.h
theon$
```

[10] Bei GNU make lässt sich mit dem Kommando „make -p -f/dev/null“ eine Liste aller voreingestellten Regeln ausgeben. Siehe auch R. M. Stallman, R. McGrath und P. D. Smith, *Catalogue of Built-In Rules*, 26. Feb. 2023. Adresse: `https://www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html` (besucht am 09. 12. 2024).

[11] Das Werkzeug *gcc-makedepend* ist auf unseren Rechnern installiert und steht zur Verfügung unter `https://github.com/afborchert/gcc-makedepend`.

Das Werkzeug *makedepend* hat hier die Konvention eingeführt, dass die generierten Abhängigkeiten im *Makefile* alle hinter der Zeile „# DO NOT DELETE“ stehen. Bei einem erneuten Aufruf wird dann dieser Teil durch den aktualisierten Stand ersetzt. Dieser Aufruf lässt sich auch direkt in das *Makefile* integrieren, so dass dann nur „make depend“ aufzurufen ist, wobei „depend“ ein Ziel ist, das mit keiner Datei verbunden ist. Daher wird es im folgenden *Makefile* als „.PHONY:“ deklariert:

```
theon$ cat makefile-depend2
Sources :=      gcd.c gcd-main.c
Objects :=      $(patsubst %.c,%.o,$(Sources))
CC = gcc
CFLAGS = -Wall
gcd:            $(Objects)

MAKEFILE :=     $(CURDIR)/$(firstword $(MAKEFILE_LIST))
.PHONY:         depend
depend:
                gcc-makedepend -f $(MAKEFILE) -gcc $(CC) $(CPPFLAGS) $(
                    Sources)
theon$ make -f makefile-depend2 depend
gcc-makedepend -f /home/borchert/soft9/makefile-depend2 -gcc gcc  gcd.c gcd-
    main.c
theon$ cat makefile-depend2
Sources :=      gcd.c gcd-main.c
Objects :=      $(patsubst %.c,%.o,$(Sources))
CC = gcc
CFLAGS = -Wall
gcd:            $(Objects)

MAKEFILE :=     $(CURDIR)/$(firstword $(MAKEFILE_LIST))
.PHONY:         depend
depend:
                gcc-makedepend -f $(MAKEFILE) -gcc $(CC) $(CPPFLAGS) $(
                    Sources)
# DO NOT DELETE
gcd.o: gcd.c gcd.h
gcd-main.o: gcd-main.c gcd.h
theon$
```

Hier kommen einige Erweiterungen von GNU make zum Tragen, das heute überwiegend zum Einsatz kommt.[12] Hier werden Funktionen wie beispielsweise *patsubst* unterstützt (*pattern substitute*), bei der etwa aus der Liste der mit „.c“ endenden Quellen die Liste der entsprechenden in „.o“ Objekt-Dateien erzeugt werden kann.[13]

9.5 Objekt-orientierte Techniken in C

Die dynamische Verknüpfung von Objekten mit Methoden kann in C über Funktionszeiger erfolgen. Im folgenden Beispiel haben wir einen abstrakten Funktionstyp in einer Veränderlichen

[12] Siehe R. M. Stallman, R. McGrath und P. D. Smith, *GNU Make*, 26. Feb. 2023. Adresse: `https://www.gnu.org/software/make/manual/` (besucht am 05. 12. 2024).

[13] Siehe `https://www.gnu.org/software/make/manual/html_node/Text-Functions.html`.

des Typs **double**, der mit einem Funktionszeiger verknüpft ist:

```
#ifndef FUNCTION_H
#define FUNCTION_H

/* remains opaque */
typedef struct function Function;

Function* create_function(const char* name, double (*f)(double x));
const char* get_name(Function* f);
double execute(Function* f, double x);

#endif
```

Zu beachten ist hier, dass in dieser Header-Datei die **struct** *function* bzw. der zugehörige Typname *Function* deklariert werden, aber nirgends ausgeführt wird, wie diese **struct** aussieht. Das ist zulässig, wenn letztere Informationen nicht benötigt werden. Da die Größe eines Zeigers unabhängig von der Größe des referenzierten Objekts ist, ist es möglich, mit Zeigern des Typs *Function*∗ umzugehen, obwohl die Größe der referenzierten Objekte und deren Aufbau unbekannt bleiben. Somit bleibt *Function* ein abstrakter Datentyp, dessen Innenleben verborgen bleibt. Wenn weniger Details in Header-Datei aufgenommen werden, dann ist es auch seltener erforderlich, die Header-Dateien anzupassen, was jedesmal eine Reihe von Neu-Übersetzungen nach sich ziehen würde.

Das Konstrukt mit **#ifndef**, **#define** zu Beginn und **#endif** am Ende sorgt dafür, dass die Header-Datei nicht mehrfach von dem Übersetzer bearbeitet wird. Im Englischen wird dies als *include guard* oder *wrapper #ifndef* bezeichnet. Das hierzu verwendete Symbol muss eindeutig sein und wird normalerweise von dem Dateinamen abgeleitet; so wird hier für *function.h* das Präprozessor-Symbol *FUNCTION_H* verwendet. Diese Technik ist notwendig, da es zu Mehrfach-Inklusionen der gleichen Header-Datei kommen kann und der Übersetzer eine mehrfache **typedef**-Definition für *Function* nicht akzeptieren würde, selbst wenn diese Definitionen genau übereinstimmen würden.

Alternativ wird auch häufig **#pragma** once verwendet, das bislang jedoch nicht vom C-Standard unterstützt wird, wohl aber von der Mehrheit der Übersetzer.[14] Mit **#pragma**-Anweisungen können Hinweise an den Übersetzer gegeben werden. Der Standard beschränkt nur auf solche, die mit **#pragma** STDC beginnen. Die Unterstützung anderer **#pragma**-Anweisungen ist implementierungsabhängig. Typischerweise führen nicht unterstützte Anweisungen zu Warnungen.

Ebenso außerhalb des Standards ist die #import-Direktive als Alternative zu **#include**, die Mehrfach-Inklusionen verhindert. Dies wird jedoch inzwischen als wenig glücklich angesehen, da dies die Verantwortung von dem Ersteller des Headers zu dem Nutzer verlagert.[15]

Zu dem abstrakten Datentyp *Function* stehen die Methoden *get_name* und *execute* zur Verfügung. Anders als bei objekt-orientierten Programmiersprachen muss hier jeweils der Objektzeiger des Typs *Function*∗ explizit mit übergeben werden.

[14] Siehe *pragma once*, 2023. Adresse: `https://en.wikipedia.org/wiki/Pragma_once` (besucht am 12. 12. 2023) sowie *Implementation defined behavior control*, 2023. Adresse: `https://en.cppreference.com/w/cpp/preprocessor/impl` (besucht am 12. 12. 2023).

[15] Siehe *Alternatives to Wrapper #ifndef*, 2023. Adresse: `https://gcc.gnu.org/onlinedocs/cpp/Alternatives-to-Wrapper-_0023ifndef.html` (besucht am 12. 12. 2023).

Die Funktion *create_function* wird von abgeleiteten „Klassen“ aufgerufen, die dann jeweils über einen Funktionszeiger ihre eigene Implementierung einer Methode übergeben. Hier wäre es auch denkbar, dass mehrere Funktionszeiger anzugeben sind, wenn da es noch mehr Methoden gibt, die von der jeweiligen Implementierung zu übernehmen sind.

Folgendes Beispiel zeigt eine konkrete Implementierung bzw. Ausprägung für die Sinus-Funktion:

```
#ifndef SINUS_H
#define SINUS_H

#include "function.h"

Function* get_sinus();

#endif
```

In diesem einfachen Beispiel handelt es sich um sogenannte Singletons, d. h. wir benötigen nur ein einziges Objekt, das die Sinus-Funktion repräsentiert. Die Implementierung kann die wie folgt erfolgen, wobei die Funktion *sin* aus ⟨math.h⟩ als Funktionszeiger übergeben wird:

```
#include <math.h>
#include "sinus.h"

Function* get_sinus(void) {
    static Function* sinus = 0;
    if (!sinus) {
        sinus = create_function("sin", sin);
    }
    return sinus;
}
```

Da wir nur einen einzigen Singleton benötigen, reicht es aus, nur ein einziges Objekt mit *create_function* zu erzeugen. Mit der als **static** deklarierten Variable *sinus* in der Funktion *get_sinus* legen wir fest, dass die Lebenszeit der Variablen von dem ersten Aufruf von *get_sinus* bis zum Ende der Programmausführung währt – es gibt also nur eine einzige Instanz. Diese Variable ist jedoch nur innerhalb des umgebenden Blocks sichtbar.

Polymorphe Objekte können in einer polymorphen Datenstruktur verwaltet werden wie beispielsweise einer *FunctionRegistry* im folgenden Beispiel:

```
#ifndef FUNCTION_REGISTRY_H
#define FUNCTION_REGISTRY_H

#include <stdbool.h>

#include "function.h"
```

```
typedef struct FunctionRegistry FunctionRegistry;

FunctionRegistry* create_registry();
bool add_function(FunctionRegistry* freg, Function* f);
Function* lookup(FunctionRegistry* freg, const char* name);
void delete_registry(FunctionRegistry* freg);

#endif
```

Die Aufgabe dieser abstrakten Containers ist es, viele Funktionsobjekte zu verwalten und einen Abruf über den Namen zu ermöglichen. Wie zuvor bleibt die interne Repräsentierung der **struct** *FunctionRegistry* verborgen.

Eine triviale Implementierung auf Basis einer linearen Liste könnte dann wie folgt aussehen:

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

#include "function-reg.h"

typedef struct FunctionRegistryEntry {
    Function* f;
    struct FunctionRegistryEntry* next;
} FunctionRegistryEntry;

struct FunctionRegistry {
    FunctionRegistryEntry* entries;
};

FunctionRegistry* create_registry(void) {
    FunctionRegistry* freg = malloc(sizeof(FunctionRegistry));
    if (freg) {
        freg->entries = 0;
    }
    return freg;
}

bool add_function(FunctionRegistry* freg, Function* f) {
    FunctionRegistryEntry* entry = malloc(sizeof(FunctionRegistryEntry));
    if (!entry) return false;
    entry->f = f;
    entry->next = freg->entries; freg->entries = entry;
    return true;
}

Function* lookup(FunctionRegistry* freg, const char* name) {
    FunctionRegistryEntry* entry = freg->entries;
    while (entry && strcmp(get_name(entry->f), name)) {
```

```
        entry = entry->next;
    }
    if (entry) {
        return entry->f;
    } else {
        return 0;
    }
}

void delete_registry(FunctionRegistry* freg) {
    if (freg) {
        FunctionRegistryEntry* entry = freg->entries;
        while (entry) {
            FunctionRegistryEntry* next = entry->next;
            free(entry);
            entry = next;
        }
        free(freg);
    }
}
```

Und so könnte eine Anwendung aussehen:

```
    FunctionRegistry* freg = create_registry();
    if (freg) {
        add_function(freg, get_sinus());
        add_function(freg, get_cosinus());

        for(;;) {
            printf(": ");
            char* line = readline(stdin);
            if (!line) break;
            int fname_start = 0; int fname_end = 0; double x = 0;
            if (sscanf(line, " %n%*s%n %lf", &fname_start, &fname_end, &x) != 1 ||
                    fname_start == fname_end) {
                printf("usage: function value\n");
                free(line);
                continue;
            }
            char* fname = line + fname_start; line[fname_end] = 0;
            Function* f = lookup(freg, fname);
            if (f) {
                double result = execute(f, x);
                printf("%g\n", result);
            } else {
                printf("no such function: %s\n", fname);
            }
```

```
        free(line);
      }
      delete_registry(freg);
    }
```

9.6 Schnittstelle für das dynamische Nachladen

Der POSIX-Standard bietet Funktionen an, die das dynamische Nachladen von Modulen ermöglichen.[16] Die folgenden Funktionen werden in ⟨dlfcn.h⟩ deklariert:

```
void* dlopen(const char* pathname, int mode);
char* dlerror(void);
void* dlsym(void* restrict handle, const char* restrict name);
int dlclose(void* handle);
```

Die Funktion *dlopen* lädt ein dynamisches Modul, typischerweise mit der Dateiendung „.so“ (*shared object*), dessen Pfadname bei *pathname* spezifiziert wird.

Der Parameter *mode* legt zwei Punkte unabhängig voneinander fest:

- Wann werden die Symbole aufgelöst? Entweder sofort (*RTLD_NOW*) oder so spät wie möglich (*RTLD_LAZY*). Letzteres wird normalerweise bevorzugt.
- Sind die geladenen globalen Symbole für später zu ladende Module sichtbar (*RTLD_GLOBAL*) oder wird ihre Sichtbarkeit lokal begrenzt (*RTLD_LOCAL*)? Hier wird zur Vermeidung von Konflikten typischerweise *RTLD_LOCAL* gewählt.

Wenn das Laden nicht klappt, kann *dlerror* aufgerufen werden, um eine passende Fehlermeldung abzurufen. Wenn es klappt, wird ein **void**-Zeiger als sogenannter *handle* zurückgeliefert, der insbesondere für die Funktionen *dlsym* und *dlclose* verwendet werden kann.

Die Funktion *dlsym* erlaubt es, Symbolnamen in Adressen zu konvertieren. Im Falle von Funktionen lässt sich auf diese Weise ein Funktionszeiger gewinnen. Zu beachten ist hier, dass nur bei C-Funktionen davon ausgegangen werden kann, dass der C-Funktionsname dem Symbolnamen entspricht. Bei C++ ist dies wegen dem *name mangling* ausgeschlossen.

Mit *dclose* kann ein nicht mehr benötigtes Modul wieder entfernt werden.

Wenn das Funktionsbeispiel auf dynamisches Nachladen von Funktionen umgestellt wird, dann entfallen die entsprechenden Header-Dateien und es wird ein einheitlicher Funktionsname verwendet (hier *get_function*), dessen Adresse jeweils mit Hilfe von *dlsym* ermittelt wird. Die Aufgabe dieser Funktion ist es, jeweils das Singleton-Objekt zu liefern:

```
#include <math.h>
#include "function.h"

Function* get_function(void) {
    static Function* sinus = 0;
    if (!sinus) {
```

[16] Siehe dlopen, dlerror, dlsym und dlclose.

```
        sinus = create_function("sin", sin);
    }
    return sinus;
}
```

Die Funktion *lookup* der *FunctionRegistry* könnte dann dahingehend erweitert werden, bislang unbekannte Funktionen dynamisch nachzuladen:

```
Function* lookup(FunctionRegistry* freg, const char* name) {
    FunctionRegistryEntry* entry = freg->entries;
    while (entry && strcmp(get_name(entry->f), name)) {
        entry = entry->next;
    }
    if (entry) return entry->f;

    // attempt to load it dynamically
    size_t sofile_len = strlen(name) + 6;
    char* sofile = malloc(sofile_len);
    if (!sofile) return 0;
    strcpy(sofile, "./"); strcat(sofile, name); strcat(sofile, ".so");
    void* handle = dlopen(sofile, RTLD_LAZY | RTLD_LOCAL);
    if (!handle) return 0;
    Function* (*gf)(void) = (Function* (*)(void)) dlsym(handle, "get_function");
    if (!gf) {
        dlclose(handle); return 0;
    }
    Function* f = gf();
    if (!add_function_with_handle(freg, f, handle)) {
        dlclose(handle); return 0;
    }
    return f;
}
```

Hierbei merken wir uns den *handle* jeweils, damit diese Module auch wieder entfernt werden können:

```
typedef struct FunctionRegistryEntry {
    Function* f;
    void* handle;
    struct FunctionRegistryEntry* next;
} FunctionRegistryEntry;

struct FunctionRegistry {
    FunctionRegistryEntry* entries;
};
```

```
void delete_registry(FunctionRegistry* freg) {
    if (freg) {
        FunctionRegistryEntry* entry = freg->entries;
        while (entry) {
            FunctionRegistryEntry* next = entry->next;
            if (entry->handle) {
                dlclose(entry->handle);
            }
            free(entry);
            entry = next;
        }
        free(freg);
    }
}
```

Die zu verwendenden Optionen beim Übersetzen und beim Zusammenbau können variieren. Folgendes *Makefile* (für GNU-make) unterstützt Solaris, Linux und MacOS:

```
MainSource := ${shell grep -l '\<main␣*(' $(wildcard *.c)}
CoreSources := $(MainSource) function.c function-reg.c
SharedSources := $(filter-out $(CoreSources),$(wildcard *.c))
Sources := $(CoreSources) $(SharedSources)
SharedObjects := $(patsubst %.c,%.so,$(SharedSources))
CoreObjects := $(patsubst %.c,%.o,$(CoreSources))
Objects := $(SharedObjects) $(CoreObjects)
Target := $(patsubst %.c,%,$(MainSource))
System := $(shell uname -s)

CC = gcc
CFLAGS = -g -std=gnu17
LDLIBS = -lm
LDFLAGS = -rdynamic
SHARED = -fPIC -shared
ifeq ($(System),Linux)
LDLIBS += -ldl
endif
ifeq ($(System),Darwin)
LDFLAGS += -undefined dynamic_lookup
endif

.PHONY: all
all: $(Target) $(SharedObjects)

$(Target): $(CoreObjects)

$(SharedObjects): %.so: %.c
                        $(CC) -o $@ $(SHARED) $(CFLAGS) $(LDFLAGS) $< $(LDLIBS)
```

```
.PHONY: clean depend realclean

clean:
                        rm -f $(Objects)
realclean: clean
                        rm -f $(Target)
ifeq ($(System),Darwin)
                        rm -fr $(patsubst %,%.dSYM,$(SharedObjects))
endif

depend: $(Sources)
                        gcc-makedepend -gcc $(CC) $(CPPFLAGS) $(Sources)
```

So sieht der Zusammenbau und eine Ausführung auf der Theon aus:

```
theon$ make depend
gcc-makedepend -gcc gcc  testit.c function.c function-reg.c cosinus.c sinus.c
theon$ make
gcc -g -std=gnu17   -c -o testit.o testit.c
gcc -g -std=gnu17   -c -o function.o function.c
gcc -g -std=gnu17   -c -o function-reg.o function-reg.c
gcc -rdynamic  testit.o function.o function-reg.o  -lm -o testit
gcc -o cosinus.so -fPIC -shared -g -std=gnu17 -rdynamic cosinus.c -lm
gcc -o sinus.so -fPIC -shared -g -std=gnu17 -rdynamic sinus.c -lm
theon$ ls
cosinus.c       function-reg.h  function.h      sinus.c         testit.c
cosinus.so      function-reg.o  function.o      sinus.so        testit.o
function-reg.c  function.c      Makefile        testit
theon$ ./testit
: sinus 3.14
0.00159265
: cosinus 3.14
-0.999999
: theon$
theon$
```

Die Option „-fPIC“ bittet den Übersetzer, Maschinen-Code zu erzeugen, der an beliebiger Stelle in den Adressraum abgebildet werden kann (*position independent code*). Bei einigen Architekturen wie beispielsweise der x86-Architektur ist das grundsätzlich erfüllt. Die Option „-shared“ bittet darum, ein *shared object* zu erzeugen, das dynamisch nachgeladen werden kann. Wenn diese nachladbaren Module Funktionen benötigen, die im Hauptmodul enthalten sind (wie die Funktion *create_function* in diesem Beispiel), dann muss dies mit der Option „-rdynamic“ zusammengebaut werden, damit die entsprechenden Symbole auch zur Laufzeit auffindbar sind. Bei Linux ist zusätzlich noch „-ldl“ anzugeben, da die Funktionen zum Nachladen nicht in der Standard-Bibliothek enthalten sind. Und unter MacOS ist es normalerweise nicht zulässig, *shared objects* zu bauen mit Verweisen zu unbekannten Symbolen. Dies wird mit der Option „-undefined dynamic_lookup“ ermöglicht.