- C unterstützt (wie andere Programmiersprachen auch) sogenannte Übersetzungseinheiten, d.h. ein Programm kann in mehrere separate Teile zerlegt werden, die sich getrennt übersetzen lassen.
- C unterstützt kein gehobenes Modulkonzept, sondern relativ einfache Mechanismen, wie Variablen und Funktionen aus anderen Übersetzungseinheiten benutzt werden können.
- Dennoch ist es mit einer geeigneten Vorgehensweise möglich, Schnittstellen sauber zu definieren und bis zu einem gewissen Rahmen auch die Gewährleistung der Schnittstellensicherheit zu erreichen.
- Dies ist jedoch nicht ohne Hilfsmittel zu bewerkstelligen.

deklvsdef.c

```
int f(int a, int b); /* eine Funktions-Deklaration */

/* eine Funktions-Definition */
int max(int a, int b) {
   return a > b? a: b;
}

/* eine Variablendefinition */
int i = 27;
```

- Eine Deklaration teilt dem Übersetzer alle notwendigen Informationen mit, die eine anschließende Verwendung des deklarierten Namens ermöglicht.
- Eine Definition enthält alle Informationen, die zur Erzeugung des Objekts benötigt werden:
  - Bei Funktionen gehört der Programmtext mit den Anweisungen dazu.
  - Bei Variablen schließt die Definition die Initialisierung ein (falls eine gewünscht wird).

- C hat einen globalen Namensraum für globale Variablen und Funktionen.
- Keine Variable oder Funktion darf über alle Übersetzungseinheiten und Bibliotheken hinweg mehr als einmal definiert werden.
- Es sind aber beliebig viele nicht-definierende Deklarationen einer Funktion oder Variablen zulässig.
- Die nicht-definierenden Deklarationen dienen im Kontext der Modularisierung dazu, eine Funktion oder Variable zu deklarieren, die in einer anderen Übersetzungseinheit definiert wird.

main.c

```
#include <stdio.h>

/* Deklarationen */
extern int i;
extern void f();

int main() {
   printf("Wert von i vor dem Aufruf: %d\n", i);
   f();
   printf("Wert von i nach dem Aufruf: %d\n", i);
}
```

lib.c

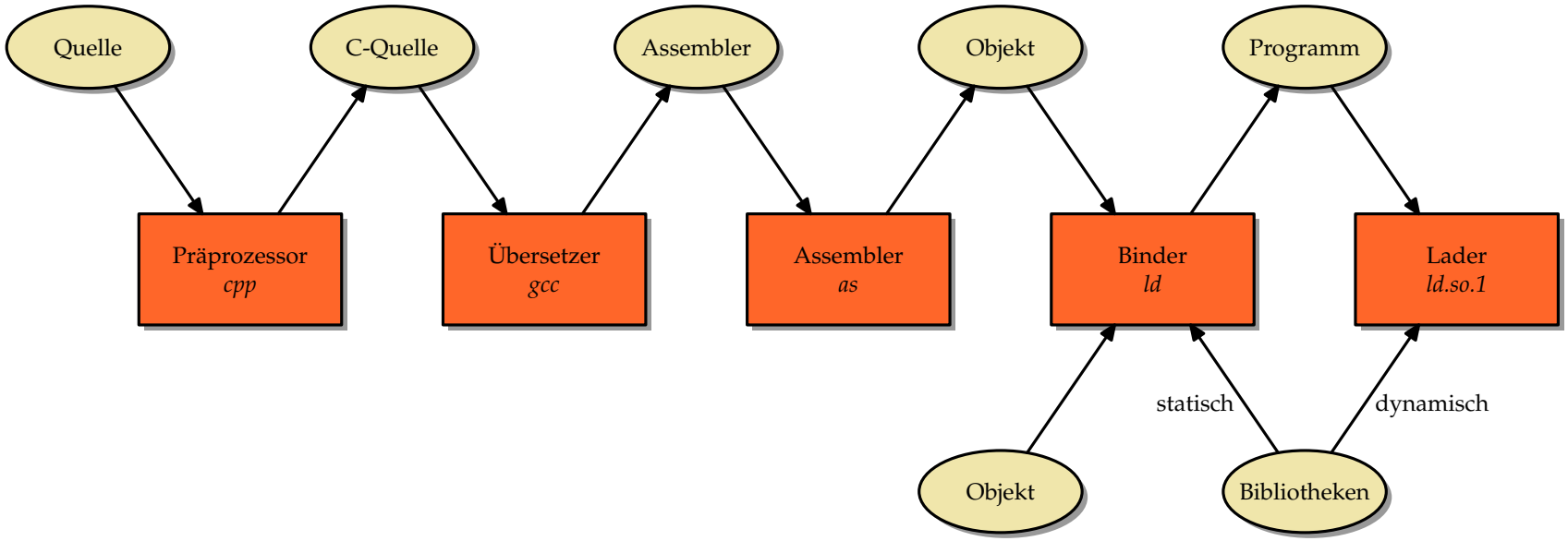
```
int i = 1; /* Definition */

void f() { /* Definition */
   ++i;
}
```

main.c

```
extern int i;
extern void f();
```

- Das Schlüsselwort **extern** hat zwei Funktionen:
  - Es macht klar, dass es sich um eine Deklaration und nicht um eine Definition handelt.
  - Es stellt fest, dass es sich um einen Namen aus dem globalen Namensraum handelt.
- Wenn wir mit Variablen- oder Funktionsnamen Objekte einer anderen Übersetzungseinheit ansprechen wollen, müssen diese Namen im globalen Namensraum sein.
- Jede Übersetzungseinheit, die diese Namen benutzen möchte, benötigt eine passende Deklaration.
- Ob Deklaration und Definition aus zwei verschiedenen Übersetzungseinheiten zueinander passen, wird nicht überprüft.

Übersetzung und Zusammenbau eines Programms 234



- Der Übersetzer „sieht“ jeweils nur eine Übersetzungseinheit mitsamt allen per **#include** hereinkopierten Deklarationen.
- Der Binder (*ld*) und der dynamische Lader (*ld.so.1*) arbeiten nur mit Namen und der Adressen, die sie repräsentieren.
- Es gibt also keine Überprüfung, ob die Definition und die Nutzung eines Namens entsprechend den Regeln von C konform zueinander sind.
- Entsprechend besitzt C keine Schnittstellensicherheit.

ggt.h

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

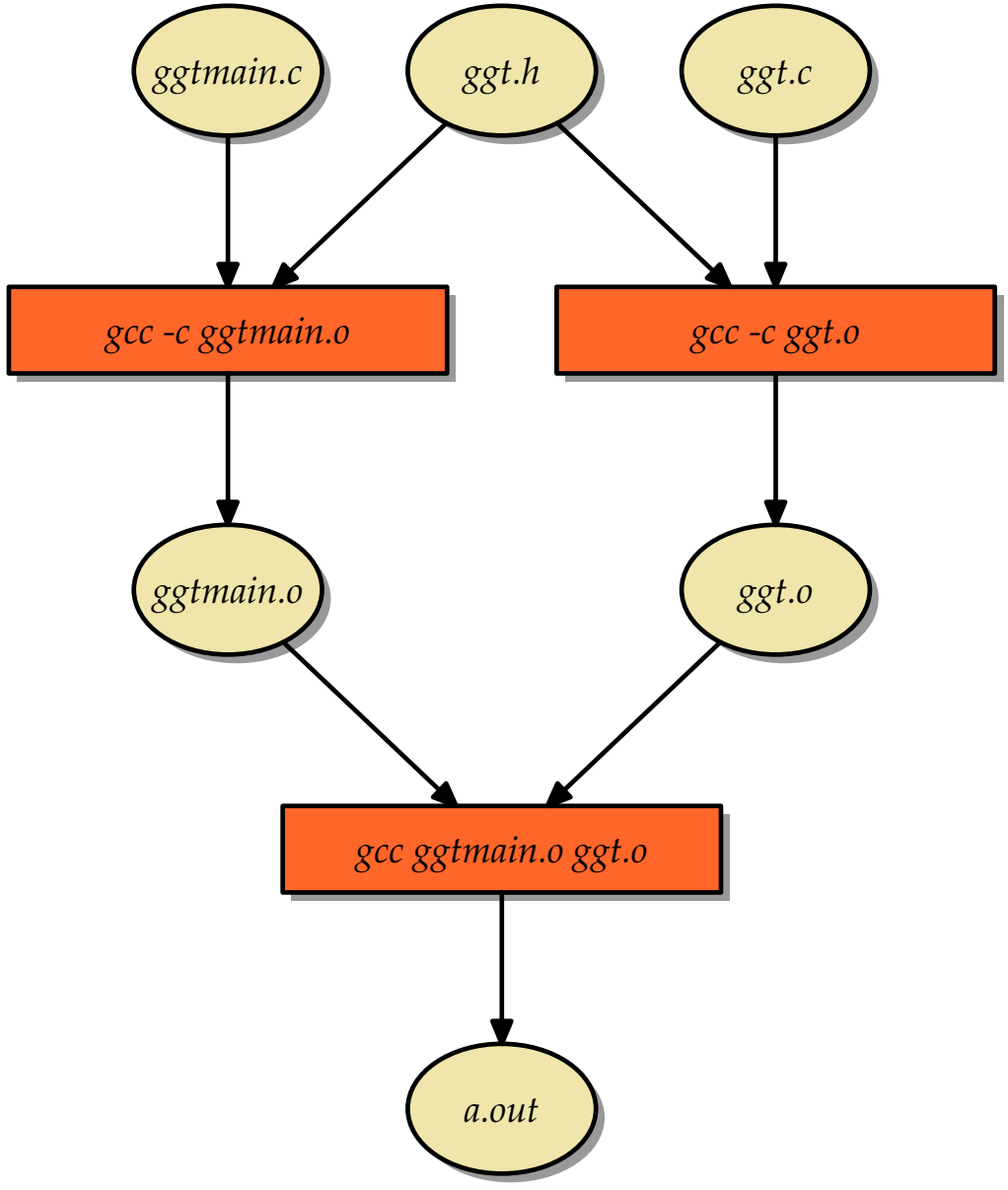
ggt.c

```
#include "ggt.h"

int ggt(int a, int b) {
   while (a != b) {
      if (a > b) {
         a -= b;
      } else {
         b -= a;
      }
   }
   return a;
}
```

- In *ggt.h* wird die Funktion *ggt* deklariert, in *ggt.c* wird sie definiert.
- Zwar könnte das **#include** wegfallen, aber dann fiele auch die Überprüfung weg, ob die Deklaration in *ggt.h* mit der Definition in *ggt.c* übereinstimmt.

ggtmain.c

```
#include <stdio.h>
#include <stdlib.h>
#include "ggt.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 = atoi(argv[0]);
   int b = atoi(argv[1]);
   if (a > 0 && b > 0) {
      printf("%d\n", ggt(a, b));
   } else {
      fprintf(stderr, usage, cmdname);
      exit(1);
   }
}
```


ggtmain.c
ggt.h
ggt.c
gcc -c ggtmain.o
gcc -c ggt.o
ggtmain.o
ggt.o
gcc ggtmain.o ggt.o
a.out

- Die Header-Datei *ggt.h* wurde sowohl bei der Übersetzung von *ggt.c* als auch der Übersetzung von *ggtmain.c* gelesen.
- Auf diese Weise lässt sich Schnittstellensicherheit erreichen, denn wenn
  - die Nutzung in *ggtmain.c* mit der Deklaration in *ggt.h* konform geht und
  - die Definition von *ggt* in *ggt.c* mit der Deklaration in *ggt.h* übereinstimmt,
  - haben wir die Sicherheit, dass die Nutzung in *ggtmain.c* mit der Definition in *ggt.c* konform geht.

- Die Schnittstellensicherheit steht und fällt aber damit, dass bei einer Änderung einer Header-Datei (wie etwa *ggt.h*) die Übersetzungseinheiten neu übersetzt werden, die direkt oder indirekt *ggt.h* einbeziehen.
- Unterbleibt dies, kann es zu Inkonsistenzen kommen.
- Beispiel: *ggt.c* wird übersetzt, *ggt.h* verändert, dann *ggtmain.c* entsprechend angepasst und übersetzt. Wenn dann *ggt.o* und *ggtmain.o* zusammengebaut werden, fällt es nicht mehr auf, dass beide Teile nicht zusammenpassen.

- Stuart Feldman entwickelte 1977 in den Bell Laboratories ein Werkzeug namens *make*, um das Problem zu lösen.
- (2003 erhielt er hierfür den *Software System Award* der ACM.)
- Die prinzipielle Idee ist, dass in einem *makefile* die Abhängigkeiten und die Kommandos zur erneuten Erzeugung einer Datei zusammengestellt werden und dass dann anhand der Existenz und Zeitstempel aller beteiligten Dateien von *make* automatisiert bestimmt wird, was zu tun ist, um eine Datei unter Berücksichtigung aller Abhängigkeiten korrekt neu zu erzeugen.

makefile

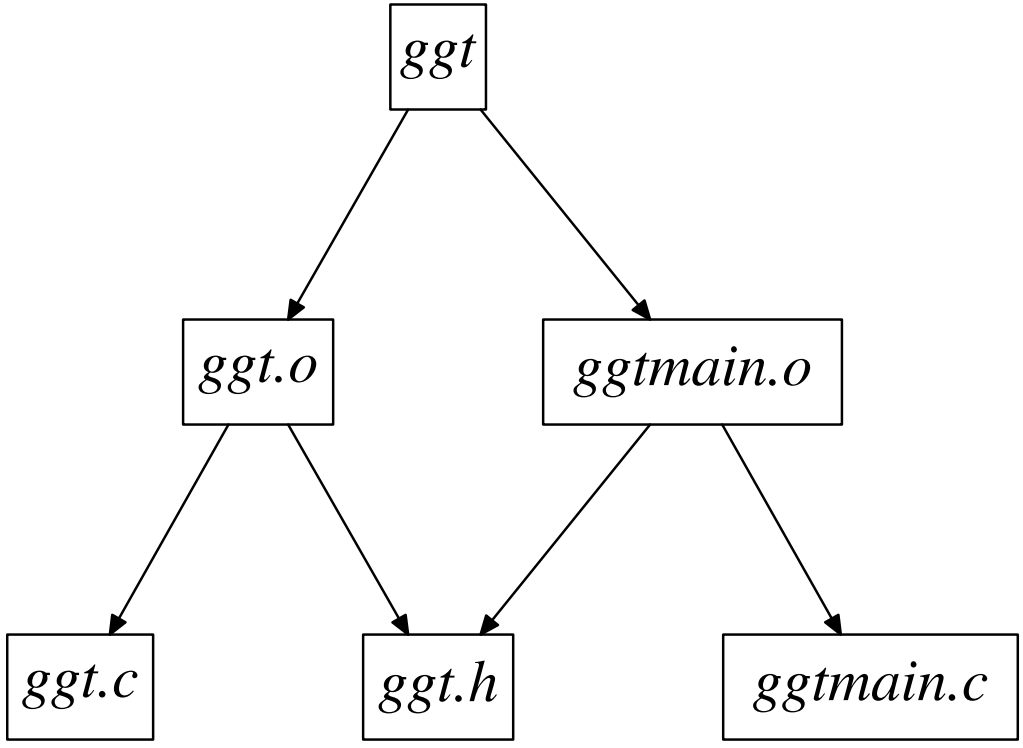
```
ggt:            ggt.o ggtmain.o
                gcc -o ggt ggt.o ggtmain.o
ggt.o:          ggt.h ggt.c
                gcc -c -Wall -std=gnu11 ggt.c
ggtmain.o:      ggt.h ggtmain.c
                gcc -c -Wall -std=gnu11 ggtmain.c
```

- Zeilen, die nicht mit einem Tab beginnen, nennen eine erzeugbare Datei (hier *ggt*, *ggt.o* und *ggtmain.o*).
- Dahinter folgt jeweils ein Doppelpunkt, Leerzeichen und noch auf der gleichen Zeile die Abhängigkeiten. *ggtmain.o* hängt hier beispielsweise von *ggtmain.c* und *ggt.h* ab.
- Die darauffolgenden Zeilen beginnen jeweils mit mindestens einem Tab und nennen dann die Kommandos, um die Datei neu zu erzeugen.

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 weder 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 ($n = 0$). 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 jeder der Zeitstempel 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.



- Die Abhängigkeiten lassen sich als gerichteter, antizyklischer Graph (DAG) darstellen.
- Die Quelldateien werden dann durch Blattknoten dargestellt (d.h. Knoten, von denen keine Kanten ausgehen).

- Wenn wir mit *make* oder einem vergleichbaren Werkzeug arbeiten, sind wir einen wesentlichen Schritt weiter. Aber dennoch haben wir nur dann Schnittstellensicherheit, wenn die Abhängigkeiten vollständig im *makefile* erfasst sind.
- Bei größeren Projekten lässt sich das nicht mehr „per Hand“ aktuell und korrekt halten.
- Hier helfen Werkzeuge, die die Abhängigkeiten automatisiert aus den C-Quellen extrahieren und das *makefile* entsprechend aktualisieren.
- Bekannt ist hier das Werkzeug *makedepend*, das sich aber nicht immer konform zum *gcc* verhält.
- Die Abhängigkeiten lassen sich mit Hilfe der Option „-M“ durch den *gcc* ermitteln.
- Unser *gcc-makedepend* bietet die Funktionalität von *makedepend* auf Basis von *gcc -M* an.

- Die auf obersten Ebene (d.h. nicht verschachtelt) deklarierten Variablen und Funktionen sind normalerweise global im Namensraum sichtbar, d.h. auch von anderen Übersetzungseinheiten aus nutzbar.
- Wenn das vermieden werden soll, können Variablen und Funktionen auch **static** deklariert werden.
- Wenn innerhalb eines Blocks eine Variable **static** deklariert wird, lebt sie wie eine globale Variable, bleibt aber nur lokal innerhalb des umgebenden Blocks sichtbar.

Im Normalfall bietet sich folgende Vorgehensweise an:

- Das gesamte Programm ist in geeignete Teile mit jeweils durchdachten Schnittstellen zu zerlegen.
- Abgesehen von dem Hauptprogramm sollte jede Übersetzungseinheit als Paar aufgesetzt werden mit einer Header-Datei mit denen nach außen hin sichtbaren Deklarationen und der zugehörigen Implementierung.
- Spätestens wenn Header-Dateien Typdeklarationen enthalten, sind sie mit **#ifndef** etc. gegen eine Mehrfachinklusion zu schützen.
- Jede Header-Datei sollte selbst per **#include** alle notwendigen Typdeklarationen aus anderen Header-Dateien hereinkopieren.
- Jede Implementierung sollte mindestens die zugehörige Header-Datei mit **#include** hereinkopieren. Alles, was nicht in der Header-Datei genannt wird, sollte mit **static** garantiert privat gehalten werden.
- Deklarationen fremder Übersetzungseinheiten sollten immer konsequent mit **#include** aus der entsprechenden Header-Datei übernommen werden.