**Praktikum Intro**

# Einführung in die Programmentwicklung unter Linux und Unix

**Frühlingssemester 2020**

**M. Thaler, J. Zeman**

## Überblick

In diesem Praktikum lernen Sie, wie man C-Programme unter Linux bzw. Unix entwickelt. Die wichtigsten Programme und Hilfsmittel der kommandozeilenorientierten Entwicklungsumgebung, mit denen wir arbeiten werden, sind:

- der C Compiler (gcc)
- der Debugger **gdb** mit seinem graphischen Frontend **ddd**
- die make-Utility
- die Editoren gedit, ev. vi, emacs, kate, etc.

Sie werden als Programmbeispiel selbständig ein C-Modul entwickeln und testen, das Sie später im Praktikum **MyThreads** benötigen werden:

- das Modul **`mlist`**, eine einfach verkettete Liste für die Verwaltung von Threads.

# Inhaltsverzeichnis

# 1 Einführung

## 1.1 Ziele

- Die Linux-Programmierumgebung
  - Sie können ein C-Programm schreiben, das aus mehreren Modulen besteht
  - Sie können für jedes Modul das dazugehörigen Header-File schreiben.
  - Sie können die make-Utility verwenden und für Ihre Programme ein einfaches make-File schreiben.
  - Sie können mit dem Debugger `gdb` bzw. seinem graphischen Frontend `ddd` umgehen und können Programme laden, die Programmausführung im Source-Code-Fenster verfolgen, Programme schrittweise durcharbeiten, Breakpoints setzen und Variablenwerte mit Display und Watch verfolgen.
- Ein Programm-Modul für das Praktikum **MyThreads** implementeiren
  - Sie implementieren das Modul `mlist`, eine einfach verkettete Liste, das für das Praktikum **MyThreads** als FIFO-Warteschlange benötigen werden.

## 1.2 Programmiersprache: C

Die Betriebssysteme Linux und Unix sind in C programmiert. Alle Betriebssystem Dienste und Datenstrukturen sind als C-Funktionen und C-Datentypen in entsprechenden C-Header-Files definiert. Deshalb ist es naheliegend, dass wir linuxnahe Programme auch in C schreiben. Zudem sind solche Programme i.A. nicht sehr gross, d.h. die Vorteile der objektorientierten Programmierung wie grössere Übersichtlichkeit, einfache Erweiterbarkeit und Wartung spielen hier eine untergeordnete Rolle.

Da Sie Vorkenntnisse in C und Java mitbringen, sollte die Syntax keine Schwierigkeiten bereiten. Falls Sie zur Auffrischung Tutorials benötigen, finden Sie diese auf OLAT unter Readings.

## 1.3 Durchführung und Leistungsnachweis

Es gelten grundsätzlich die Vorgaben Ihres Dozenten zur Durchführung der Praktika und zu den Leistungsnachweisen im Kurs BSy.

Die Inhalte des Praktikums gehören zum Prüfungsstoff.

# 2 Aufgabenstellungen

## 2.1 Module: Demo-Programm, make-Utility und Debugging

Arbeiten Sie die folgenden Punkte der Reihe nach durch:

1. Lesen Sie das Praktikum bitte zu Hause durch. Die Theorie und das Hintergrund-Wissen zum diesem Aufgabenteil finden Sie in Abschnitt 3.

2. Die vorbereiteten Programm `main` finden Sie mit allen dazugehörigen Files in `Intro.tgz`. Das tar-File können Sie mit `tar -xvzf Intro.tgz` auspacken, dabei wird automatisch das Verzeichnis `./Intro` mit den Unterverzeichnissen `./listen` und `./module` erzeugt.

3. Wechseln Sie ins Verzeichnis `./Intro/module` und geben Sie im Kommandozeilen-Fenster den Befehl `make` ein, damit erzeugen Sie aus den Programmquellen das ausführbare File `main.e`.

4. Starten Sie das Programm in einem Kommandozeilen-Fenster, geben Sie dazu `./main.e` ein. Das Programm können Sie mit CTRL-C sofort abbrechen (Programm abbrechen) oder mit CTRL-D das Ende der Eingabe (End Of File) signalisieren.

5. Starten Sie das Programm im Debugger mit `ddd main.e`. Beachten Sie, dass bei der Programmübersetzung für diesen Fall der Switch (Option) `-g` verwendet werden muss: einbinden von Debugging Informationen. Siehe dazu auch das Files `makefile`.

6. Führen Sie nun folgende Debugging-Tätigkeiten durch:
   - einen Breakpoint in `main()` und in der Funktion `flaeche()` (Modul `func1.c`) setzen
   - die Variablen `R`, `F` und **`U`** in `main()` anzeigen
   - das Programm starten und am 1. Breakpoint mit `Next` oder `Cont` weiterfahren
   - nach dem zweiten Breakpoint schrittweise weiterfahren, etc.

   Weitere Informationen finden Sie in der Kurzanleitung `ddd.pdf` und auf der GNU Web-Seite: `http://www.gnu.org/software/ddd/`.

7. Ändern Sie den Typ der Variablen `radius` im Modul `func1.c` auf `int`:

   ```
   myfloat_t flaeche(int radius)
   ```

   Beobachten Sie, was während der Programmübersetzung mit `make` passiert.

8. Das Programm soll auch das Kugelvolumen $V = (4/3) * \pi * r^3$ berechnen. Ergänzen Sie dazu das Programm um das Modul `func3.c`, das die Funktion `myfloat_t volumen(myfloat_t r)` implementiert.

   Das berechnete Volumen soll im Hauptprogramm ausgegeben werden. Machen Sie alle notwendigen Änderungen in den Programmfiles sowie im `makefile` und testen Sie das Programm.

9. Ihr Programm soll genauer als mit Datentyp `myfloat_t`[1] rechnen. Zudem soll die Konstante `phi` mit mehr als nur zwei Stellen (3.14) darstellen. Was müssen Sie alles ändern? Führen Sie die Änderung durch und testen Sie das Programm wieder.

   Hinweise:
   - im Header-File `math.h` ist die Konstante `M_PI` auf 20 Dezimalstellen genau definiert
   - *gefährlicher Programmierstil, wieso?* Siehe Fussnote[2]

[1] Der Datentyp `myfloat_t` ist mit Hilfe von `typedef` als `float` in `mydefs.h` definiert.

[2] In der Funktion `eingabe()` muss bei `scanf()` die Formatanweisung von `%f` auf `%lf` (long float) geändert werden.

10. Die make-Files `makefile.macros2` und `makefile.macros3` im Unterverzeichnis `./extras`, enthalten Macros und implizite Bildungsregeln. Analysieren Sie, wie `make` diese Macros ersetzt und die Bildungsregeln anwendet. Das Makefile `makefile.macros2` übersetzt `./main_m2.c` und `makefile.macros3` übersetzt `./main_m3.c`.

    Hinweise:
    - `macros2` und `macros3` arbeiten ohne `func3.c`
    - da sich nur die make-Files im Verzeichnis `extras` befinden, muss `make` im Verzeichnis `module` wie folgt aufgerufen werden:

```
make -f ./extras/makefile.macros2
make -f ./extras/makefile.macros3
```

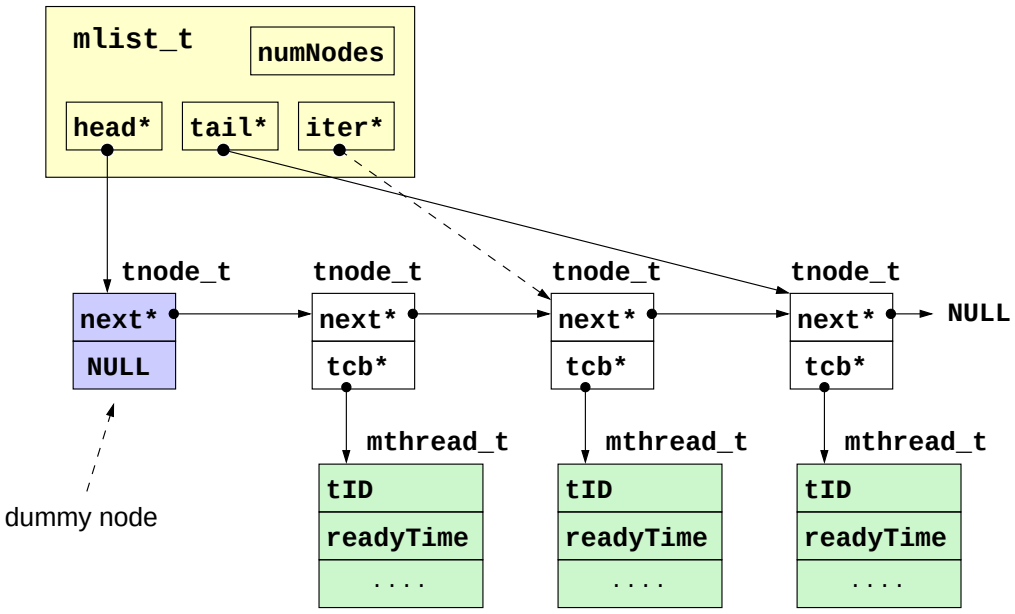
## 2.2 Liste: Ein Programm in C unter Linux schreiben

### 2.2.1 Das Listen-Modul

Implementieren Sie das C-Modul **`mlist`**, das eine einfach verkettete Liste mit einem Dummy Knoten realisiert und Datenobjekte vom Typ **`mthread_t`** verwaltet. Die Datenobjekte sind sogenannte Thread-Control-Blocks und im Modul `mthreads` definiert. Verwenden Sie für den Zugriff auf die Daten des Thread-Control-Blocks die Funktionen aus dem Modul `mthreads`.

Die Liste wird mit Hilfe einer Struktur vom Typ **`tlist_t`** verwaltet, die Knoten werden mit Strukturen vom Typ **`tnode_t`** implementiert. Da der Dummy Knoten auch bei einer leeren Liste vorhanden ist, wird die Programmierung der Liste vereinfacht (ein Knoten ist immer vorhanden). In `mlist` werden drei Zeiger verwendet: `head` zeigt auf den Dummy-Knoten, `tail` zeigt immer auf den letzten Knoten der Liste und `iter` wird als Iterator verwendet.

**Hinweis**: Wir verwenden hier die in der prozeduralen und objektorienten Programmierung übliche Definition von *Tail*: letztes Element der Liste.




Die Liste soll folgende Anwendungen unterstützen:

- **FIFO-Queue**: neue Datenobjekte werden am Ende der Liste (tail) mit `mlEnqueue()` angehängt und am Kopf der Liste mit `mlDequeue()` entnommen.
- **Sortierte Queue**: neue Datenobjekte werden nach einer ihren Eigenschaften (in unserem Fall die `readyTime`) sortiert in die Liste mit `mlSortIn()` eingehängt (in aufsteigender Reihenfolge der `readyTime`) und am Kopf der Liste mit `mlDequeue()` entnommen.

Zusätzlich soll das Modul folgende Funktionen anbieten:

- Zugriff auf das erste Datenobjekt, ohne es aus der Liste zu entfernen: `mlReadFirst()`
- Abfrage, wie viele Datenobjekte (Knoten) sich in der Liste befinden: `mlGetNumNodes()`
- Sequentielles Durchlaufen und Lesen aller Listenelemente z.B. für Kontrollausgaben: `mlSetPtrFirst()`, `mlSetPtrNext()`, `mlReadCurrent()`.

Die Datenstrukturen und Funktionen für `tnode` und `mlist` sind im File `mlist.h` deklariert und enthalten folgende Einträge:

```
//---------------------------------------------------------------------------
// list node
typedef struct tnode {
    mthread_t*    tcb;                // pointer to tcb (thread control block)
    struct tnode* next;               // pointer to next list element
} tnode_t;
//---------------------------------------------------------------------------
// list header (we use her type tnode_t)
typedef struct mlist {
    unsigned  numNodes;                // number of list elements
    tnode_t*  head;                    // pointer to header node
    tnode_t*  tail;                    // pointer to tail node
    tnode_t*  iter;                    // pointer for iterartions
} mlist_t;
//---------------------------------------------------------------------------
mlist_t*   mlNewList();                // setup list with dummy header node
void       mlDelList(mlist_t* list);   // delete list including data
void       mlEnqueue(mlist_t* list, mthread_t* tcb);  // append tcb to list
mthread_t* mlDequeue(mlist_t* list);   // take first element out of ist
                                       // return ptr to tcb or NULL
void mlSortIn(mlist_t* list, mthread_t* tcb, int (*cmp)(void *a, void *b));
                                       // insert tcb sorted according to
                                       // to function cmp()
mthread_t* mlReadFirst(mlist_t* list); // return ptr to first tcb in list,
                                       // but do not dequeue
unsigned   mlGetNumNodes(mlist_t* list); // return number of elements in list
void       mlSetPtrFirst(mlist_t* list); // set iter pointer to first element
void       mlSetPtrNext(mlist_t*  list); // move iter pointer to next element
mthread_t* mlReadCurrent(mlist_t* list); // return tcb by iter pointer
```

**Hinweis**: beim Entfernen eines Daten-Knotens mit **`mlDequeue()`** wird der dazugehörige `tnode` zum neuen Dummy Knoten, der alte Dummy Knoten wird gelöscht.

**Hinweis**: die Funktion **`mlSortIn()`** verwendet eine Callback-Funktion, der Zeiger auf die zu vergleichenden Datenwerte bzw. Structs (mit zu vergleichenden Datenwerten) übergeben werden müssen. Die Callback-Funktion wird von der Applikation implementiert und muss einen Integerwert $< 0$, $= 0$ oder $> 0$ zurückgeben, je nach dem ob für die zu vergleichenden Parameter `aW,bW` gilt: `aW < bW`, `aW == bW` oder `aW > bW`.

Die von der Liste verwalteten Datenobjekte sind im Modul `mthreads` definiert. Im Moment arbeiten wir mit einer einfachen Version, die folgendes Datenobjekt definiert:

```
typedef struct mthread {
    unsigned int tID;                  // thread-ID
    unsigned int readyTime;            // time, when a a waiting thread
    usingned int tPrio;                // priority of thread
    void         *stack;               // thread stack
} mthread_t;
```

und unten stehende Funktionen zur Verfügung stellt:

```
mthread_t*   mtNewThread(unsigned id, tprio_t prio, unsigned readyTime);
void         mtDelThread(mthread_t *tcb);
unsigned     mtGetID(mthread_t* tcb);
void         mtSetReadyTime(mthread_t* tcb, unsigned rtime);
unsigned     mtGetReadyTime(mthread_t* tcb);
```

### 2.2.2 Programmverifikation: Unit Tests

Da Sie die Liste in späteren Praktikumsversuchen benötigen werden, ist ein einwandfreies Funktionieren Ihres Codes ein **MUSS**!

- Testen Sie zuerst die einzelnen Programmteile mit eigenen kleinen Testprogrammen. Verfolgen Sie evtl. Fehler mit dem Debugger ddd.

  Hinweis: in ddd können Sie Listen graphisch darstellen, klicken Sie dazu jeweils auf die entsprechenden Pointer. Beim Steppen werden zudem die Links graphisch aufdatiert.

- Testen Sie anschliessend Ihre Programm-Module mit dem Programm **`test1.e`**, das einen Unit-Test durchführt: Sie erhalten entsprechende Fehlermeldungen.

- Das Programm `test2.e` erzeugt eine Ausgabe, die Sie mit der Ausgabe im File `test2out.txt` vergleichen können. Leiten Sie dazu die Ausgabe von Programm **`test2.e`** in eine Datei um mit **`test2.e > test2.txt`** und vergleichen Sie die Files mit **`diff test2.txt test2out.txt`**.

  Wieso stimmen die Resultate nicht überein? Analysieren Sie dazu das Programm `test2.c`: was passiert genau und welcher Fehler (auf logischer Ebene) wird hier gemacht?

  Hinweis: starten Sie das Programm auch mit einem Parameter, z.B. `test2.e 1` und analysieren Sie den Output. Achtung: wenn Sie ein älteres Linux verwenden ändert sich hier eventuell nichts.

### 2.2.3 Programmverifikation: Memory Leaks

Bei der C/C++ Softwarentwicklung ist der Programmierer für die Freigabe von dynamisch allozierten Speicher verantwortlich (kein Garbage Collector), tut er dies nicht, steht dieser Speicher nicht mehr zur Verfügung. Bei langlebigen Programmen führen diese sogenannten *Memory Leaks* dazu, dass irgendwann kein Speicher mehr alloziert werden kann.

Um zu überprüfen, ob ein Programm ein Memory Leak hat, gibt es z.B. das Tool `Valgrind`, das feststellen kann, ob gleich viel Speicher freigegen wird, wie alloziert wurde. Valgrind instrumentiert dazu das Testprogramm mit einer eigenen Version von `malloc()` und `free()`.

**Aufgaben**

- Falls Valgrind auf Ihrem System nicht installiert ist, holen Sie das bitte nach.

- Das Testprogramm `test3` hilft, Ihre Implementierung auf Memory Leaks zu überprüfen. Damit `test3` richtig funktioniert, muss der Test `test1` fehlerfrei durchlaufen worden sein.

  Übersetzen Sie das Programm mit **`make leakTest`** und starten Sie das Programm wie folgt:

  ```
  valgrind --leak-check=full ./test3.e
  ```

  Falls Valgrind Leaks entdeckt, stopfen Sie diese bitte.

Hinweis: Damit `Valgrind` richtig funktioniert dürfen die Flags `-g` bzw. `-pg` nicht gesetzt sein.

### 2.2.4 Profiling und Performance

Wir wollen nun untersuchen, wie effizient Ihre Implementation arbeitet. Im `makefile` muss dazu die Compiler Option `-pg` eingefügt werden (wird hier automatisch gemacht) und übersetzen Sie das Programm `test4.c` mit:

```
make profileTest
```

Starten Sie das Programm `./test4.e`. Es erzeugt im Arbeitsverzeichnis das File **`gmon.out`** mit Profiling Informationen.

Die Profiling Information können Sie mit folgendem Befehl in ein File schreiben (`gprof` benötigt dazu auch das Programmfile):

```
gprof -p -b ./test4.e > analysis.txt
```

Weitere Informationen zu `gprof` finden Sie im Manual (`man gprof`) und auf dem WEB.

**Aufgaben**

- Analysieren Sie den Output im File `analysis.txt`, speziell die Anzahl Calls und die Rechenzeit der Getter-Funktion `mtGetReadyTime()`. Was fällt auf?

  Hinweis: in `mlist.c` (Funktion `mlSortIn()`) müssen Sie die Getter-Funktion `mtGetReadyTime()` für den Zugriff auf die `readyTime` verwenden (sonst wird die Funktion nicht aufgelistet).

- Im Header-File `mthread.h` haben wir für die Getter-Funktion `mtGetReadyTime()` ein Makro definiert. Wenn Sie das Kommentarzeichen in der Zeile `//#define MACRO` entfernen, wird ein Macro aktiviert, das die Getter-Funktion durch einen direkten Zugriff auf die `readyTime` ersetzt.

  Vergleichen Sie die Anzahl Calls der Funktionen aus beiden Varianten und die kumulierte Rechenzeit (2. Spalte unten). Was fällt bei der Rechenzeit auf (siehe auch Anmerkungen) ?

  ***Wichtige Anmerkungen***

  - Zur Bestimmung der Anzahl Calls wird der Code *instrumentiert*, d.h. in jeder Funktion wird Code zum Zählen der Aufrufe eingebaut: das Resultat ist deshalb korrekt.
  - Die Rechenzeiten werden mit *Sampling* angenähert (eine Instrumentierung würde zuviel Overhead erzeugen). Dabei wird der Programmzähler periodisch abgefragt und daraus zusammen mit der Anzahl Calls die Rechenzeiten geschätzt. Die Samplingperiode beträgt im Moment 10ms (0.01s): wie sind unter diesem Gesichtspunkt die Angaben der Rechzeiten zu bewerten? Ziehen Sie dazu auch die Anzahl Instruktionen (Grösse) von Getter-Funktionen wie `mtGetReadyTime()` in Betracht.
  - der Aufruf von Unterprogrammen mit Auf- und Abbau eines Stackframes ist bei der 32-Bit X86-Architektur relativ aufwendig, bei der AMD64 Bit Architektur gilt das nur noch bedingt, weil die ersten 6 Integer-Parameter in den Registern rdi, rsi, rdx, rcx, r8, r9, übergeben werden, Float-Parameter in MMX Registern.

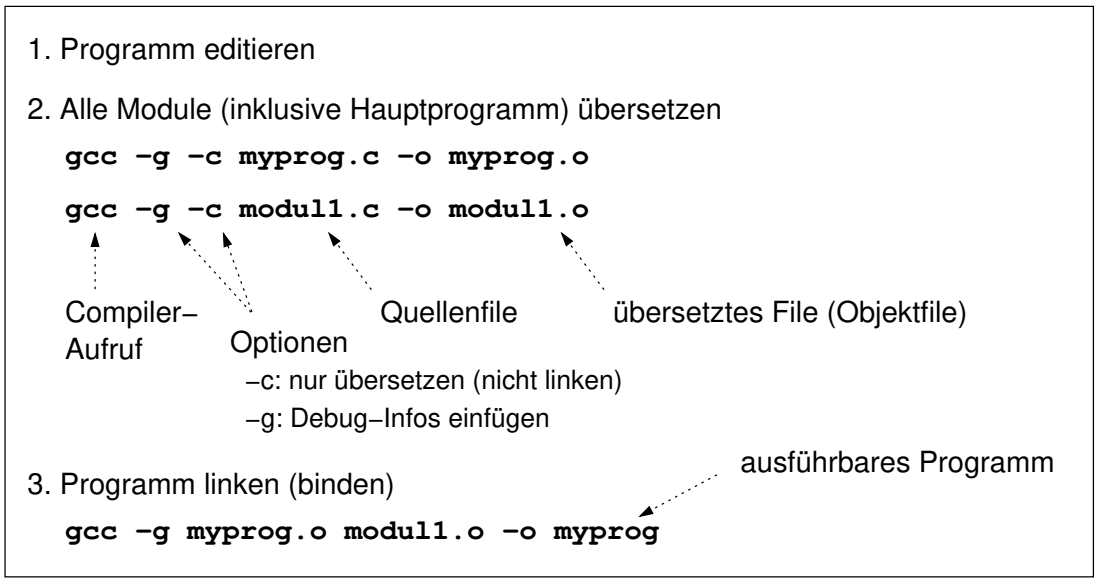
# 3 Linux/Unix-Programmierumgebung und Make

## 3.1 Einführung

Mit dieser Kurzanleitung möchten wir die wichtigsten Informationen repetieren bzw. vermitteln, die für Programmentwicklung unter Linux/Unix wichtig sind, ohne integrierte Entwicklungsumgebungen wie Eclipse, etc. zu verwenden.

## 3.2 Programmübersetzung und Linking

Die traditionelle Programmentwicklung unter Unix/Linux läuft wie folgt ab:




Der C/C++-Compiler kann einfache Programme auch in einem Schritt übersetzen und Linken. Bei unserem einfachen Programm mit den Modulen `myprog.c` und `modul1.c` könnten die Schritte 2 und 3 deshalb wie folgt zusammengefasst werden:

```
gcc -g myprog.c modul1.c -o myprog
```

Der Nachteil dieses Vorgehen ist, dass jedes Mal alle Module neu übersetzt werden, auch solche, die nicht verändert wurden, was ineffizient und langsam ist. Es ist ebenfalls möglich C-Quellenfiles und bereits übersetze Files (Objektfiles) zu mischen:

```
gcc -g myprog.c modul1.o -o myprog
```

Der Compiler merkt anhand der File-Erweiterungen (Suffixes, Extensions), dass nur das File `myprog.c` neu übersetzt werden muss und dann mit dem bereits vorhandenen Objektfile `modul1.o` gelinkt werden kann.

### 3.2.1 Compiler-Aufruf

Angaben in eckigen Klammern sind optional.

| | | |
|---|---|---|
| C-Compiler | gcc | [Optionen] Quellenfiles [Libraries] [-o Ausgangsfile] |
| C++-Compiler | g++ | [Optionen] Quellenfiles [Libraries] [-o Ausgangsfile] |

### 3.2.2 Die wichtigsten Compiler-Optionen

Die wichtigsten Compiler-Optionen sind unten aufgeführt, weitere Informationen finden Sie in den Manuals und auf dem Internt.

| | |
|---|---|
| -c | nur Quellenprogramm übersetzen, kein Linken |
| -std=gnu99 | C99 Standard verwenden, erlaubt z.B. die Verwendung von `for (int i = 0; i < N; i++)`, etc. |
| -o *file* | Ouput in File *file* schreiben, Default für ausführbares File: `a.out`, Default für Objektfile: `sourcefile.o` |
| -g | Information für Debugger hinzufügen |
| -l*library* | Library *library* beim Linken hinzufügen, muss neuerdings hinten vor Outputfile stehen |
| -O | schaltet Code-Optmierung ein |
| -Wall | aktiviert alle sinnvoller Warnungen |
| -D*sym=val* | definiert *sym* für Pre-Prozessor, wie `#define N 5` (=*val* ist optional) |

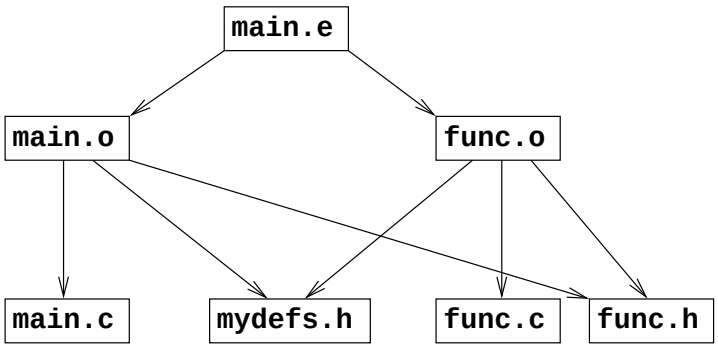
## 3.3 Programmübersetzung mit make

Grössere Softwarepakete bestehen üblicherweise aus mehreren einzelnen Modulen. Häufig werden auch zusammengehörende Quellen- und Objektcode-Module in verschiedenen Verzeichnissen abgelegt. Dabei entsteht Unterhaltsaufwand: der Programmierer muss wissen, welche Programmteile nach der Änderung einer Quelldatei neu übersetzt werden müssen und welche nicht. Nach dem Übersetzen müssen dann die einzelnen Module zu einem ausführbaren Programm zusammengefügt werden (Linking). Unter Umständen müssen noch weitere Routine-Tätigkeiten ausgeführt werden, wie z.B. das Ersetzen geänderter Module in Programmbibliotheken, das Aufräumen von Verzeichnissen, das Umkopieren von Daten, etc.

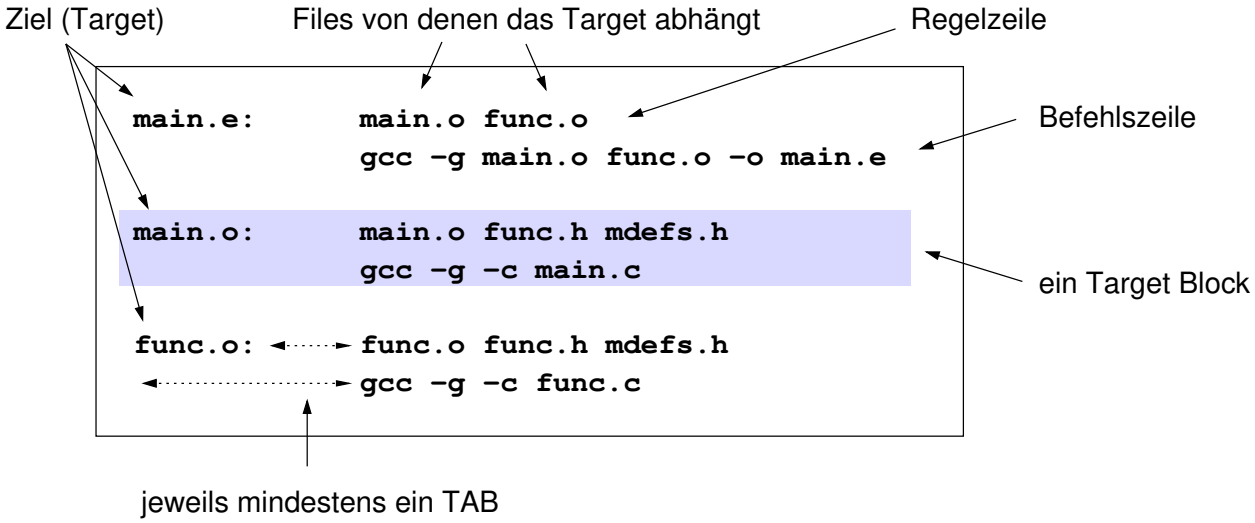
Das Programm `make` hilft diese Aufgaben zu automatisieren. Dabei geht es intelligent vor: aufgrund von Information über die gegenseitigen Abhängigkeiten einzelner Programmteile und des Änderungsdatums jedes Files, kann `make` bestimmen, welche Dateien neu übersetzt werden müssen und welche nicht. Die dazu benötigten Informationen werden in einem Beschreibungsfile mit Namen `makefile` oder `Makefile` vom Programmierer abgelegt und basieren auf Regeln und den dazugehörigen Befehlen.

### 3.3.1 Beispiel

Die folgende Firgur zeigt den Abhängigkeitsbaum eines einfachen Softwareprojektes mit zwei Modulen (`main` und `func`) und einem Headerfile (`mydefs.h`):




Das dazugehörige `makefile` hat folgende Form:




Beim Aufruf von `make`, liest `make` das File `makefile` und baut den Abhängigkeistbaum auf, ausgehend von ersten Target. Dann werden alle Befehlszeilen von unten nach oben im Baum ausgeführt, die Files in der Regelzeile enthalten, die jünger als das Target sind.

### 3.3.2 Make für Fortgeschrittene

Make unterstützt verschiedene Möglichkeiten, die die Anwendung vereinfachen und flexibilisieren. Dazu gehören Makros und implizite Bildungsregeln.

Hier ein Beispiel, wie wir in ähnlicher Form in den Praktika verwenden:

```
#-------------------------------------------------------
# Make für Fortgeschrittene
CC   = gcc
CFL  = -std=gnu99 -g
LIBS = -lm
OBJ  = main.o fun1.o fun2.o fun3.o

main:   ${OBS} mdefs.h
        ${CC} ${CFL} ${OBJ} ${LIBS} -o $@.e

.c.o:
        ${CC} ${CFL} $<

clean:
        @rm -f *.o *.e

all:
        @make clean
        @make
#-------------------------------------------------------
```

In diesem Beispiel werden selbst- und vordefinierte Makros verwendet:

| | |
|---|---|
| `CC = gcc` | selbst definiertes Makro |
| `${CC}` | wird ersetzt durch die entsprechende Definition, hier `gcc` |
| `$@` | Name des aktuellen Targets (hier `main`), damit heisst das Outputfile `main.e` |
| `$*` | Name des aktuellen Targets ohne Suffix |
| `#` | leitet Kommentar ein |

Die folgende implizite Regel oder sogenannte Suffixregel übersetzt Files mit Suffix `.c` aufgrund der Befehlszeilen in Objektfiles mit Suffix `.o`:

```
.c.o:   ${CC} ${CFL} $<
```

Die Targets `clean` und `all` räumen das Verzeichnis auf bzw. übersetzen sämtliche Files neu. Das `@`-Zeichen vor einer Befehlszeile schaltet die Ausgabe dieser Befehlszeile aus (wird aber dennoch ausgeführt).

### 3.3.3 Optionen zu `make`

**`Make`** kann mit oder ohne Optionen und Parametern aufgerufen werden:

**Einfacher Aufruf:** `make`

In diesem Fall sucht `make` im aktuellen Verzeichnis nach dem File `makefile` oder `Makefile` und führt die entsprechenden Aktionen aus.

**Aufruf mit Optionen und Parametern:** `make [Optionen] [Parameter]`

Die wichtigsten Optionen sind:

| | |
|---|---|
| `-n` | *no execution*, die Befehle werden angezeigt aber nicht ausgeführt |
| `-d` | *debugging*, während Ausführung werden alle Entscheidungen angezeigt |
| `-p` | `print`, zeigt alle Makrodefinitionen und Abhängigkieitsbeschreibungen an |
| `-f` *myfile* | verwendet *myfile* anstelle von `makefile` |

Parameter werden wie folgt übergeben (Anführungszeichnen notwendig):

| | |
|---|---|
| `"CFLAGS = -g -O"` | das Makro `CFLAGS` wird auf `-g -O` gesetzt, die Definition im `makefile` wird überschrieben |

### 3.3.4 Zusammenfassung

Make ist ein mächtiges Werkzeug und erlaubt dem Programmierer das Kompilieren von Programmen zu vereinfachen. Neben den hier vorgestellten Möglichkeiten stehen viele weitere Funktionen zur Verfügung, die wir hier aber nicht weiter besprechen möchten, siehe dazu die entsprechende Literatur.