Praktikum Intro

Einführung in die Programmentwicklung unter Linux und Unix

Frühjahrssemester 2012
J. Zeman, M. Thaler

Überblick

In diesem Praktikum lernen Sie, wie man C-Programme unter Linux resp. 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 Programmerbeispiel selbständig ein C-Modul entwickeln und testen, das Sie später im Praktikum **mythreads** benötigen werden:

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

Inhalt

1. Aufgabenstellung

1.1 Ziele:

- Sie beherrschen die Linux-Programmierumgebung
  - Sie sind in der Lage ein C-Programm, bestehend aus mehreren Modulen, zu schreiben.
  - Sie können für jedes Modul die dazugehörigen Header-Dateien schreiben.
  - Sie können die make-Utility verwenden und für Ihre Programme ein einfaches make-File schreiben.
  - Sie können mit dem Debugger-Programm gdb resp. seinem graphischen Frontend **ddd** umgehen und kennen die grundlegenden Debugging-Möglichkeiten: Programme laden, die Programmausführung im Source-Code-Fenster verfolgen, Programme schrittweise durcharbeiten, Breakpoints setzen, Variablenwerte mit Display und Watch betrachten.
- Programm-Module für das Praktikum **mythreads** bereitstellen
  - Sie schreiben ein Programmodul, das das Modul **mylist** (eine infach verkettete Liste) realisiert, sie wird im Praktikum **mythreads** als FIFO-Warteschlange und priorisierte Warteschlange eingesetzt werden.

1.2 Verwendete 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-Dateien definiert. Deshalb ist es zwingend, 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, Erweiterbarkeit und einfache Wartung spielen hier nur ein 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 zur Vorlesung unter. Die Grundstruktur eines C-Programmes können Sie den einführenden Beispielen in Kapitel 2.3 entnehmen.

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.

1.4 Aufgaben

1.3.1 Module: Demo-Programm, make-Utility und Debugging

1. Praktikum bitte zu Hause durchlesen. Die Theorie und das Background-Wissen zum diesem Aufgabenteil finden Sie in den Abschnitten 2.1-2.4.
2. Das Demo-Programm `main`, das im Abschnitt 2.3.1.1 (Listing auf Seite 13) beschrieben ist, finden Sie mit allen dazugehörigen Dateien auf dem Web-Server unter Praktika in der Archivdatei `Intro.tar.gz`. Das tar-File können Sie mit `tar –xvzf Intro.tar.gz` auspacken dabei wird automatisch das Verzeichnis `./Intro` mit den Unterverzeichnissen `./listen` und `./modul` erzeugt.
3. Wechseln Sie ins Verzeichnis `./Intro/module` und geben Sie dazu im Kommandozeilen-Fenster den Befehl **`make`** ein , damit erzeugen Sie damit aus den Programmquellen die ausführbare Datei `main`,.
4. Starten Sie das Demo-Programm in einem Kommandozeilen-Fenster, geben Sie dazu **`main`** ein. Abbruch des Programms mit CTRL-C (Programm beenden) oder CTRL-D (End Of File).
5. Starten Sie das Programm im Debugger mit **`ddd main`**. Beachten Sie, dass bei der Programmübersetzung für diesen Fall immer der Switch (Option) **-g** für *include debugging information* verwendet werden muss. Siehe dazu auch die Datei **`makefile`**.
6. Versuchen Sie nun die wichtigsten Debugging-Tätigkeiten: siehe dazu die Kurzanleitung in **`DDD.pdf`**:
   - einen ersten Breakpoint in den Funktionen `main` und `flaeche` (in Modul `func1.c`) setzen
   - die Variablen **`R`**, **`F`** und **`U`** betrachten
   - das Programm starten und am 1. Breakpoint mit **`Next`** oder **`Cont`** weiterfahren
   - nach dem zweiten Breakpoint schrittweise weiterfahren, etc.

   Weitere Informationen finden auf dem WEB: `http://www.gnu.org/software/ddd/`):
7. Ändern Sie den Typ der Variable **`radius`** im Modul `func1.c` zu integer:
   `my_float flaeche(int radius).`
   Beobachten Sie, was während der Programmübersetzung mit **`make`** passiert.
8. Ergänzen Sie Ihr Programm um das Modul **`func3.c`**, wo Sie eine Funktion zur Berechnung des Kugelvolumens `my_float kugelvolumen(my_float r)` implementieren. Hinweis: $V=(4/3)*\pi*r^3$).
   Im Hauptprogramm **`main`** soll natürlich das berechnete Volumen ausgegeben werden. Machen Sie alle notwendigen Änderungen in den Programmdateien und der makefile-Datei, testen Sie das Programm.
9. Sie wollen, dass das Programm genauer als mit dem Datentyp `my_float`[1] rechnet. Zudem wollen Sie die Konstante **`phi`** mit mehr als nur zwei Stellen (3.14) darstellen. Was müssen Sie alles ändern[2] ?
   Führen Sie die Änderung durch und testen Sie das Programm wieder.
   Hinweise:
   - in der Standard-Header-Datei `/usr/include/math.h` ist die Konstante `M_PI` auf 20 Dezimalstellen genau definiert
   - *gefährlicher Programmierstil, wieso? Siehe Fussnote 2*
10. Die make-Dateien `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 Makefiel `makefile.macros3` übersetzt `./main_cpp_io.c`, das für die Ein- und Ausgabe C++ IO-Streams verwendet).

    Hinweise:
    a) `macros2` und `macros3` arbeiten ohne **`func3.c`**
    b) 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
```

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

[2] Im Hauptprogramm müssen Sie in der Funktion `eingabe()` bei `scanf()` die Formatanweisung `%f` auf `%lf` (long float) ändern.

1.4.2 Liste: Ein Programm in C unter Linux schreiben

1.4.2.1 Das Listenmodul

Implementieren Sie das C-Modul **`mylist`**, das eine einfach verkettete Liste mit einem Dummy Knoten realisiert und Datenobjekte vom Typ **`ThreadCB`** (Thread-Control-Block, eine Struktur definiert in `mythreads.h`) verwaltet (siehe Fig. 1). Verwenden Sie vorerst für den Zugriff auf die Thread-Control-Block Daten, die Funktionen aus dem Modul **`mythreads`** (ähnlicher Programmierstil wie bei OO).

Die Liste selbst wird mit Hilfe einer Struktur vom Typ **`TList`** verwaltet, die Knoten werden mit Strukturen vom Typ **`TNode`** implementiert. Da der Dummy Knoten auch bei einer leeren Liste vorhanden ist, wird die Listenprogrammierung vereinfacht (ein Knoten ist immer vorhanden). In `mylist` werden drei Zeiger verwendet: `head` zeigt dabei auf den Dummy-Knoten, `tail` zeigt immer letzen Knoten der Liste und `iter` wird als Iterator verwendet.

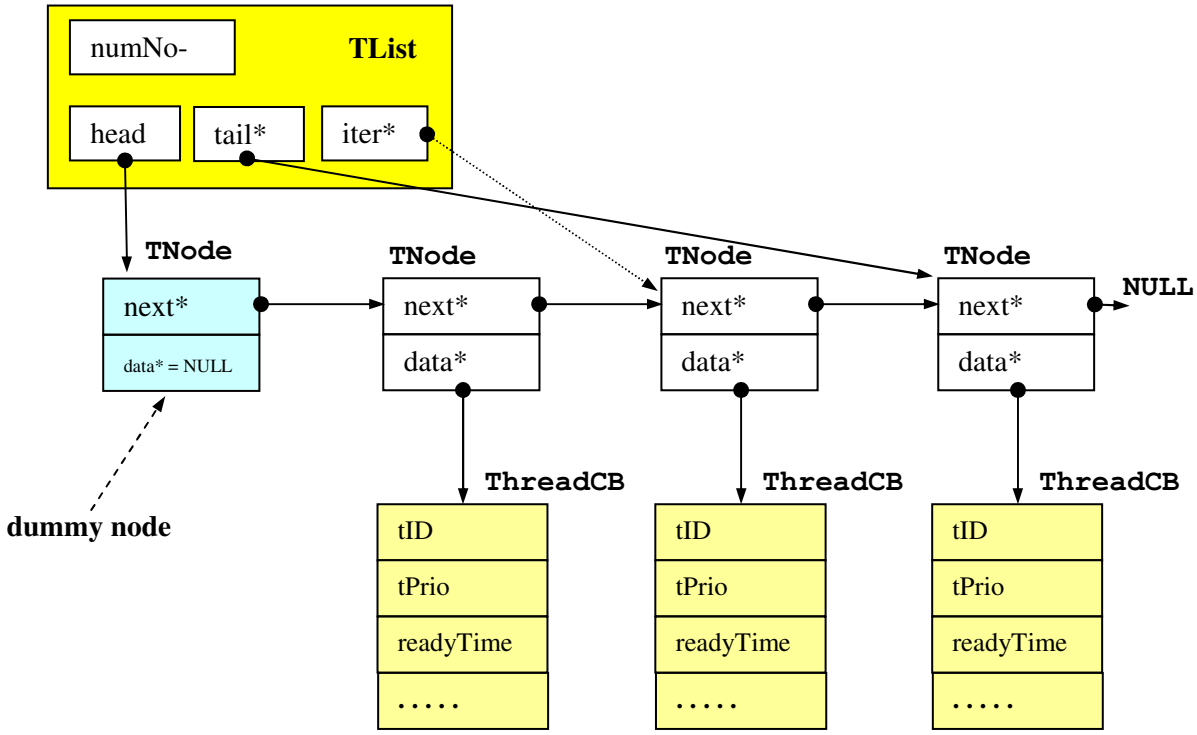


Fig. 1: Realisierung einer einfach verketteten Liste mit einem Dummy Node

Die Liste soll folgende Anwendungen unterstützen:

- **FIFO-Warteschlange**: neue Datenobjekte werden am Ende der Liste (tail) mit `tlEnqueue()` angehängt und am Kopf der Liste mit `tlDequeue()` entnommen.
- **Priorisierte Warteschlange**: neue Datenobjekte werden nach einer ihren Eigenschaften (in unserem Fall die `readyTime`) sortiert in die Liste mit `tlSortIn()` eingehängt (in aufsteigender Reihenfolge der `readyTime`) und am Kopf der Liste mit `tlDequeue()` entnommen.

Zusätzlich soll das Modul folgende Funktionen anbieten:

- Zugriff auf das erste Datenobjekt, ohne es aus der Liste zu entfernen: `readFirst()`
- Abfrage, wie viele Datenobjekte (Knoten) sich in der Liste befinden: `getNumNodes()`
- Sequentielles Durchlaufen und Lesen aller Listenelemente z.B. für Kontrollausgaben: `setPtrFirst()`, `setPtrNext()`, `readCurrent()`.

Die Datenstrukturen und Funktionen für `TNode` und `TList` sind in der Datei `mylist.h` deklariert und enthalten folgende Einträge:

```
//-----------------------------------------------------------------------------
// list node

typedef struct TNode{
    ThreadCB*     data;                 // pointer to data
    struct TNode* next;                 // pointer to next list element
} TNode;

//-----------------------------------------------------------------------------
// list header

typedef struct {
    unsigned    numNodes;               // number of list elements
    TNode*      head;                   // pointer to header node
    TNode*      tail;                   // pointer to tail node
    TNode*      iter;                   // pointer for iterartions
} TList;

//-----------------------------------------------------------------------------

TList*      tlNewList();                // setup list with dummy header node
void        tlDelList(TList* list);     // delete list including data
void        tlEnqueue(TList* list, ThreadCB* dat); // append thread data to list
ThreadCB*   tlDequeue(TList* list);     // take first element out of the list
                                        // return ptr to thread struct or NULL
void        tlSortIn(TList* list, ThreadCB* dat); // insert thread data sorted
                                                  // according ready time
ThreadCB*   tlReadFirst(TList* list);   // return ptr to the first thread data
                                        // struct in list, do not dequeue not
unsigned    tlGetNumNodes(TList* list); // return number of elements in list
void        tlSetPtrFirst(TList* list); // set iter pointer to first element
void        tlSetPtrNext(TList* list);  // move iter pointer to next element
ThreadCB*   tlReadCurrent(TList* list); // return thread data via iter pointer

//*****************************************************************************
```

Hinweis: beim Entfernen des ersten Daten-Knotens (`tlPopFirst()`) übernimmt der entfernte Knoten die Aufgabe des Dummy Knotens.

Die von der Liste verwaltete Datenobjekte werden durch das Modul **mythreads** definiert. Diese Datenstruktur wird im Praktikum **mythreads** als Thread-Controllblock für die Thread-Verwaltung benötig und speichert alle notwendigen Thread-Parameter. Im Moment arbeiten wir mit einer **vereinfachten Minimalversion**, die folgende Datenobjekte und Funktionen enthält:

```
typedef struct {
    unsigned  tID;              // thread ID
    unsigned  tPrio;            // thread priority
    unsigned  readyTime;        // timestamp when thread is ready to run
} ThreadCB;

ThreadCB* mtNewThread(unsigned id, unsigned prio, unsigned readyTime);
void      mtDeleteThread(ThreadCB* thread);

unsigned  mtGetID(ThreadCB* tcb);
void      mtSetID(ThreadCB* tcb, unsigned id);

unsigned  mtGetPrio(ThreadCB* tcb);
void      mtSetPrio(ThreadCB* tcb, unsigned prio);

unsigned  mtGetReadyTime(ThreadCB* tcb);
void      mtSetReadyTime(ThreadCB* tcb, unsigned rt);
```

1.4.2.2 Programmtest

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`-Programm).
  Hinweis: in DDD können Sie Listen graphisch darstellen, klicken Sie dazu jeweils auf die entsprechenden Pointer. Beim Steppen werden zudem die Links graphisch aufdatiert.
- Übersetzen Sie anschliessend Ihre Programm-Module mit den beiden Testprogrammen **test1** und **test2**. **Test1** führt einen Unit-Test durch, Sie erhalten entsprechende Fehlerrmeldungen, test2 erzeugt Output, den Sie mit den Musterausgaben in `test2out.txt` vergleichen sollen: die Ausgaben sollten übereinstimmen. Bei nicht übereinstimmenden Resultaten sind die Ursachen zu untersuchen und entsprechend zu begründen.

1.4.2.3 Profiling und Performance

Hier wollen wir untersuchen, wie effizient Ihre Implementation arbeitet. Ersetzen Sie dazu im `makefile` das Flag `-g` durch `-pg` und übersetzen Sie das Programm neu: zuerst `make clean`, dann `make`. Lassen Sie nun das Program `test2` einmal durchlaufen. Im Arbeitsverzeichnis befindet nun sich das File `gmon.out`, das Profiling Informationen enthält. Diese Informtionen können mit folgendem Befehl in eine Datei geschrieben werden (`gprof` benötigt dazu auch das Programmfile):

```
gprof -b test2 > analysis.txt
```

Weiter Informationen finden Sie in den Manuals (`man gprof`) und auf dem WEB.

Aufgaben

- Analysieren Sie den Output im Fiel `analysis.txt` (speziell die Anzahl ***calls***). Fällt Ihnen etwas speziell auf?
- Falls ja: was kann der Grund für dieses Verhalten sein und wie lässt sich das Problem lösen? Implementieren und überprüfen Sie Ihren Änderungsvorschlag. Was haben Sie erreicht? Ist anzunehmen, dass Ihre Lösung auch im angestrebten Kontext, Prozessverwaltung für den eigenen Thread Scheduler, allgemeine Gültigkeit hat?
  Anmerkung: der Aufruf von Unterprogrammen mit Auf- und Abbau eines Stackframes ist auf der X86-Architektur relativ aufwendig (Anzahl Instruktionen) ... bei der AMD64 Bit Architektur stimmt das nur noch bedingt, weil die ersten 6 Integer Parameter in Register übergeben werden (Float Parameter über die MMX Register).

P.S. "Power Efficient Software" ist im Moment ein heisses Forschungsthema!
… vielleicht ein kleiner Beitrag

2. Theorie zu Linux-Programmierumgebung und Make

2.1 Einführung

Diese Anleitung versucht Ihnen in kürze die wichtigsten Informationen zu vermitteln, die Sie benötigen, um modular gestaltete C- und C++-Programme unter Unix (und insbesondere unter Linux) im Praktikum zu entwikkeln und zu testen. Sie ersetzt keine vollständige Anleitungen und Man-Pages, die bei Spezialfällen zu konsultieren sind.

In Teil 1 wird auf die Bedienung des C/C++-Compilers eingegangen.

In Teil 2 wird die modulare Programmierung in C- und C++ und die Verwendung der Header-Dateien erklärt.

Teil 3 befasst sich mit der *MAKE*-Utility, mit deren Hilfe die Übersetzung und das Linken von umfangreichen Programmpaketen mit vielen Modulen automatisiert werden kann.

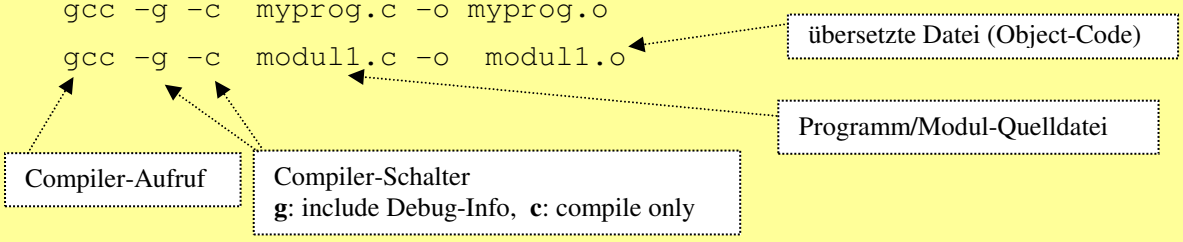
2.2 Programmübersetzung und Linking

2.2.1 Kommandozeile für den C (C++)-Compiler

Im Gegensatz zur Verwendung integrierter Umgebungen (KDevelop, Eclipse, resp. Visual-Studio unter Windows, läuft die traditionelle Programmentwicklung unter Unix/Linux wie folgt ab:

1) **Programm editieren** (mit einem beliebigen Texteditor-Programm, z2. vi, emacs, KDEdit....)
   Das Programm wird entweder im Terminal-Fenster gestartet (z.B. `vi myprog.c`) oder durch das Doppeklick (oder Menü-Wahl) unter der Xwindow-Oberfläche.
2) **Das Hauptprogramm und weitere dazugehörige Module mit einer Kommandoanweisung übersetzen**
   Bsp. für C-Compiler gcc:

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



3) **Programm binden (linking):** `gcc myprog.o  modul1.o  -o  myprog`

Der C/C++-Compiler kann einfache Programme in einem Schritt übersetzen und Linken (durch den automatischen Aufruf des Linker-Programmes (ld)) :

Bei unserem einfachen Programm, bestehend aus 2 Modulen (*myprog.c* und *modul1.c*) hätten die Schritte 2 und 3 deshalb wie folgt zusammengefasst werden können:

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

Der Nachteil dieses Vorgehen ist, dass jedes Mal alle Module zuerst neu übersetzt werden (auch solche, die nicht verändert wurden). Dies ist ineffizient und langsam, wenn man bedenkt, dass grössere Projekte mehrere Minuten bis zu einer Stunde Zeit für ihre Gesamtübersetzung benötigen.

Wenn man Module selektiv übersetzen will, dann muss man (wie im Schritt 2 gezeigt) den Compiler mit dem Schalter (-c : compile only) explizit dazu zwingen, dass nur eine Übersetzung (ohne linking) durchgeführt wird.

Beim Aufruf des Compilers ist es auch möglich, die C-Quelldateien und bereits übersetze Objektdateien als Eingangsdateien zu vermischen:

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

Der Compiler merkt anhand der Dateinamenerweiterungen (Suffix, Extension), dass nur die Datei **`myprog.c`** neu übersetzt werden muss und anschliessend mit der bereits vorhandenen, übersetzter Objektdatei **`modul1.o`** *zusammengelinkt* werden soll.

2.2.2 Die wichtigsten Standard-Suffixe für Sourcecode unter Unix

Suffix	Dateityp
.c	C-Quellprogramm
.p	Pascal-Quellprogramm
.java	Java-Quellprogramm
.o	Objektdatei: Ein einzelnes, bereits in Maschinencode übersetztes, aber noch nicht allein ablauffähiges Programm-Modul (andere Module und Programmbibliotheken fehlen)
.h	C/C++-Header-Datei (enthält Deklarationen aller aus einem Modul exportierten Funktionen, Datentypen, Klassen etc.), siehe Details im Kapitel 2.
.C	C++-Quellprogramm
.cc	C++-Quellprogramm
.cpp	C++-Quellprogramm
.s	Assembler-Quellprogramm
.cs	C#-Quellcode

2.2.3 Aufrufsyntax für die wichtigsten Unix-Compiler (unter Linux)

C-COMPILER	**gcc** [Schalter/Option(en)] Eingangsdatei(en) [Link Optionen] [o Ausgangsdatei]
C++ Compiler	**g++** [Schalter/Option(en)] Eingangsdatei(en) [Link Optionen] [o Ausgangsdatei]

Bemerkungen:

- die Eingaben in [ ] sind optional
- wenn keine Ausgangsdatei und kein **`-c`** Switch definiert wird, dann produziert der Compiler eine ausführbare Datei **`a.out`** (assembler-output)

2.2.4 Die wichtigsten Compiler-Optionen

Die detaillierte Optionen können mit dem **`man`** Befehl erfragt werden (`man gcc`, `man g++`))

-c (compile only) die angegebenen Quellprogramme nur übersetzen (compilieren) aber anschliessend nicht linken, die erzeugten Objektdateien werden nicht gelöscht.

-D*name* (Define) definiert für den Preprozessor das Symbol *name* (alternativ mit Wert: -D*name*=Wert) wie wenn es mir der #define name ...-Anweisung in jeder Quelldatei definiert wäre

-g, -ggdb (debug) fügt Debugger-Information zum erzeugten Programm bzw. zu den erzeugten Objektdateien hinzu, mit **-g** wird nur Standardinformationen hinzugefügt, **-ggdb** dagegen bewirkt, dass spezielle Informationen beigefügt werden, die nur der gdb versteht

-llibrary explizite Angabe einer Library-Datei, die das Programm benötigt und die dazugelinkt werden muss

-o (output) erlaubt die Angabe der Outputdatei (Default für die ausführbaren Dateien: a.out)

-O (Optimize) schalten den Code-Optimierer ein

-Wall aktiviert alle sinnvollen Warnungen → sicherer Code wie mit dem Syntaxprüfer `lint`

2.3 Modulare Programmierung in C und C++

Grössere Programme werden normalerweise in mehrere, separat zu übersetzende Quelldatei-Module aufgeteilt. Dabei entstehen folgende Probleme:

1. **Konsistenzerhaltung**
   Die Deklaration und Verwendung einer Funktion stimmen nicht überein. Beispiel: Eine Funktion liefert als Resultat einen Integer-Wert, bei Ihrer Verwendung wird aber ein Float-Wert als Ausgang erwartet.
2. **Management bei der Programmübersetzung und beim -Linking**
   Es müssen immer nur diejenige Module neu übersetzt werden, die verändert wurden, bei einer grösserer Anzahl Module verliert man schnell die Übersicht. Z.B. werden Fehler in verschiedenen Modulen korrigiert, die entsprechenden Module aber nicht neu übersetzt oder das Programm nicht neu gelinkt. Das Resultat: der resp. die Fehler treten immer noch auf, obwohl sie korrigiert wurden.

Das erste Problem wird in C- und C++-Programmen mit Hilfe der *Header-Dateien* gelöst. Für die Lösung des zweiten Problems hilft uns die make-Utility (Kapitel 2.4).

2.3.1 Header-Dateien

Die Header-Dateien enthalten die Deklarationen aller Datentypen, Klassen, Konstanten, Variablen und Funktionen, welche aus einem C/C++-Modul exportiert werden.

Die Header-Datei muss in alle Module mit der #include Anweisung eingebunden, die Funktionen, Variablen, etc. vom exportierenden Modul verwenden (externe Deklarationen). Zusätzlich wird sie aber auch in das dazugehörige, exportierende Modul (wo die Funktionen ausprogrammiert sind) eingebunden.

Das Einbinden der gemeinsamen Header-Datei in das ex- und importierende Modul ermöglicht, das der Compiler auch bei einer einzelnen Modul-Übersetzung mit Hilfe der Header-Datei *sieht* wie die dort deklarierten Daten in anderem Modul verwendet (resp. deklariert) werden.

2.3.1.1 Beispiel und Erklärungen zu den Header-Dateien

Das nachfolgende (triviale) C-Programm besteht aus 3 Quelldateien (Modulen). Die Modulabhängigkeiten und -Listings sind in Fig.2 (S. 3) gezeigt. Das Programm berechnet den Umfang und die Fläche eines Kreises. Die Funktionen **`umfang()`** und **`flaeche()`** wurden in separaten Modulen `funct1.c` und `func2.c` ausprogrammiert und werden von dort exportiert. Die exportierten Funktionen sind in dazugehörigen Header-Dateien `funct1.h` und `func2.c` als **`extern`** deklariert. Das Hauptprogramm im Module `main.c` bindet beide Header-Dateien ein und verwendet dann die importierten Funktionen.

Bemerkungen:

- Die `#include` Anweisung ist ein Preprocessor-Kommando. Der Inhalt der angegebenen Datei wird einfach vor der Übersetzung an die *include*-Stelle eingefügt.
- Der Dateiname in der *include* Anweisung kann entweder in <...> oder "..." eingeschlossen werden,. Bedeutung:
  - **`#include <Datei.h>`**: die Datei wird nur in den Standardverzeichnissen für Header-Dateien gesucht (z.B. /usr/include, usr/local/include etc.)
  - **`#include "Datei.h"`**: die Datei wird zuerst im aktuellen Verzeichnis und anschliessend in den Standardverzeichnissen für Header-Dateien gesucht. Dieser Variante eignet sich für eigene Header-Dateien
- Jedes C- und C++-Programm enthält mindestens die Include-Anweisungen für die Dateien, in welchen die Prototypen der verwendeten C/C++-Library-Funktionen deklariert sind (z.B. stdio.h, iostreams).

- Vermeiden von Mehrfachdeklarationen

  Da man als Autor eine Moduls keinen Einfluss auf die Reihenfolge der Include-Anweisungen in anderen Modulen hat und auch nicht weiss, ob andere lokal verwendete Standard-Libraries auch im Anwendungsprogramm mit dazugehörigen Include-Anweisung verwendet werden, besteht die Möglichkeit von mehrfachen Deklarationen, die der C/C++-Compiler mit Fehlermeldungen quittieren würde.

  Als Standardverfahren enthält deshalb jede Header-Datei folgende Konstruktion mit den Preprozessor-Anweisungen:

```
#ifndef MODULNAME
#define MODULNAME
.....
..... Deklarationen
.....
#endif
```

  Die Preprozessor-Anweisung **`#ifndef`** testet, ob eine Compiler-Konstante (hier **`MODULNAME`**) bereits definiert ist. Wenn nein, dann wird der nachfolgende Block eingefügt. Dort steht als erste Anweisung **`#define MODULNAME`**. Damit wird der Compiler instruiert, diese Konstante zu definieren. Wenn dann irgendwo später diese Header-Datei noch einmal bei der Übersetzung dieses Moduls aufgerufen wird, dann wird dieser Block ignoriert.

  Als MODULNAME kann man natürlich einen beliebigen Namen nehmen. Aus systematischen Gründen sollte man aber grossgeschriebene Name der betreffenden Datei verwenden.

- Gemeinsam verwendete Konstanten, Typendeklarationen, globale Variablen

  Normalerweise existiert immer ein Dateipaar: die Quelldatei **`*.c`** und die dazugehörige Header-Datei **`*.h`**, in welcher alle exportierten Elemente deklariert werden. Manchmal werden aber bestimmte Konstanten oder Typendeklarationen an verschiedenen Stellen im Programmmodulen und Header-Dateien benötigt. Dann sollte man eine zusätzliche Header-Datei kreieren, welche diese Deklarationen enthält und die überall mit **`#include`** eingebunden wird. Man betrachte in unserem Beispiel die Datei `mydefs.h`. Hier können z.B. zentral an einer Stelle der Typ **`my_float`** von float auf double oder die Konstante **`my_phi`** umdefiniert werden.

- **Vermischt man C++-Programme mit C-Libraries,** müssen alle externe C-Funktionen mit **`extern "C"`** deklariert werden. Sonst wird sie der Linker nicht finden, weil C++-Programme wegen Method-Overloading andere Symbolbezeichnungen für C++-Funktionen verwendet als der (einfachere) C-Compiler..

  Beispiel:

```
extern "C" void c_func(int);
```

  Man kann auch einen ganzen Block von Funktionen als extere Funktionen deklarieren:

```
extern "C" {
      void c_func(int);
      void c_f2(char *);
}
```

- Die Steuerung der Programmübersetzung mit mehreren Modulen übernimmt die make-Utility (siehe Kapitel 2.4).
- Schliesslich soll hier noch kurz zusammengefasst werden, was eine C/C++-Quelldatei und eine Header-Datei enthalten soll (gemäss Cay S. Horstmann, "Mastering C++", John Wiley, 1991).

gemeinsame Elemente	kommt in die .c- oder .cc-Datei	kommt in die Header-Datei .h
nicht initialisierte Daten	`Employee staff[100];`	`extern Employee staff[];`
initialisierte Daten	`int reportwidth = 80;`	`extern int reportwidth;`
inline-Konstanten		`const int NSTAFF = 100;`
gespeicherte Konstanten	`extern const Complex j(0,1);`	

Funktionen	`Complex log (Complex x)`
	`{........}`
		`Complex log (Complex x) ;`
		`Complex log (Complex) ;`
		`extern Complex log (Complex x) ;`

In-Line-Funktionen		`inline int min(int x, int y)` `{ return (x<y ? x : y) }`

Klassen,
Strukturen,
Unions
	`Date Date::add(int n)`
	`{.......}`
		`Class Date`
		`{int d, m, y;`
		`public`
		`int day() {return d;}`
		`Date add(int);`
		`….`
		`};`

typedef, enum		`typedef double myfloat;`

- Die Verwaltung von Header-Dateien ist fehleranfällig und muss konsequent gehandhabt werden. Es existieren heute Tools, die Header-Dateien (und auch Make-Dateien) automatisch aus den Quelldateien erzeugen können.

  Beispiele: Perl-Scripts oder integrierte Umgebungen wie z.B. Borland-CBuilder oder Visual-Studio, wo die Module durch *Projekte* verwaltet werden.

Programm-Quelldateien (Module) Header-Dateien

```
/* File:       "func1.c"
   Fach:       Betriebssysteme, ZHW, J. Zeman, Sept. 99
   In dieser Datei wird die Funktion "Flaeche" ausprogrammiert.
*/

#include "mydefs.h"
#include "func1.h"

my_float flaeche(my_float radius)
{
  return (my_phi * radius * radius);
}
```

```
#ifndef FUNC1  /* Standard-Konstruktion */
#define FUNC1

/*   File:      "func1.h",  ZHW, J. Zeman, Sept. 99
     externe Deklaration der  Funktionen, welche
     aus dem Modul func1.c exportiert werden
*/

#include "mydefs.h"   /* enthält die Deklaration vom
                         Datentyp „my_float“ */

  extern my_float flaeche(my_float);

#endif
```

```
/*
  Beispiel:   Modulare Programmierung in C (und C++),
  File:       "main.c"  (Hauptprogr. mit Headerdat. der ext. Module)
  Fach:       Betriebssysteme, ZHW, J. Zeman, Sept. 99
*/
#include <stdio.h>       /* Header-Datei für Standard-Library-Fkt.*/
#include "mydefs.h"   /* Header-Datei mitl
                         eigenen typen und Konst */
#include "func1.h"    /* Header-Datei fuer ext. Modul func1.cc */
#include "func2.h"    /* Header-Datei fuer ext. Modul func2.cc */

/* ..... Funktionsprototypen: (lokale Funktionen in diesem Modul) */
void eingabe(my_float *x);

/* ......Hauptprogramm ..... */
void main(void)
{
  my_float r ;

  eingabe(&r);
  printf("\n\n Kreisberechnung");
  printf("\n(berechnet mit phi= %f)", 3.14);
  printf("\n Radius= %f, -> Flaeche= %f, Umfang= %f",
          r ,flaeche(r), umfang(r) );
  getchar(); getchar(); /* dummy input -> warte hier */
}

void eingabe(my_float *x)
{
  printf("\n Kreisberechnung; Radius =?");
  scanf("%f", x);

}
```

```
#ifndef _MYDEFS   /* Standard-Konstruktion */
#define _MYDEFS

/*  File:      "mydefs.h",  J. Zeman, Sept. 99
    externe Typen, Konstanten, Macros und globale
    Variablendeklarationen */

  typedef float my_float;

  #define my_phi 3.14

  #endif
```

```
#ifndef _FUNC2    /* Standard-Konstruktion  */
#define _FUNC2

/*   File:      "func2.h",  ZHW,  J. Zeman, Sept. 99
     externe Deklaration der  Funktionen, welche aus dem
     Modul func2.c exportiert werden
*/

#include "mydefs.h" /* enthält die Deklaration
                       vom Datentyp „my_float“ */

  extern my_float umfang(my_float);

#endif
```

```
/* File:       "func2.c"
   Fach:       Betriebssysteme, ZHW, J. Zeman, Sept. 99
   In dieser Datei wird die Funktion "Umfang" ausprogrammiert.
*/

#include "mydefs.h"
#include "func2.h"

my_float umfang(my_float radius)
{
  return (2 * my_phi * radius);
}
```

Fig. 2: Beispiel eines Modularen C-Programmes mit dazugehörigen Header-Dateien, die Ouellendateien finden Sie auf dem WEB Server

2.4. Einführung in die *make*-Utility

Grössere Softwarepakete bestehen üblicherweise aus mehreren separat zu übersetzenden Modulen. Häufig werden auch zusammengehörende Quellen- und Objektcode-Module in separaten Verzeichnissen aufbewahrt. Dabei entsteht aber auch ein gewisser Unterhaltsaufwand:

Der Programmierer muss wissen, welche Programmteile nach einer Änderung einer Quelldatei neu übersetzt werden müssen. Nach der Übersetzung müssen alle Module zu einem ausführbaren Programm zusammengefügt werden (linking). Evtl. sind noch weitere Routine-Tätigkeiten, wie z.B. Einbinden der veränderten Module in Programmbibliotheken, Aufräumen der Verzeichnisse usw. notwendig.

Das Utility-Programm ***make*** kann alle diese Aufgaben automatisieren. Dabei geht es intelligenter vor als eine reine Batch-Datei: Aufgrund der Information über die gegenseitigen Abhängigkeiten der einzelnen Programmteile (definiert in der Steuerdatei **`Makefile`** oder **`makefile`**) und des letzen Änderungsdatums jeder Datei kann es bestimmen, welche Dateien von der Änderung betroffen sind und neu übersetzt werden müssen. Diese Dateien werden dann gemäss der im **`Makefile`** definierten Regeln und den dazugehörigen Befehlszeilen neu übersetzt und anschliessend das Programm neu *gelinkt*.

2.4.1 Struktur der Datei *makefile*

Beim Aufruf von ***make*** wird die Steuerdatei mit dem Default-Namen **`Makefile`** im aktuellen Verzeichnis eingelesen. Dieses File besteht aus Zeilenblöcken, welche durch eine oder mehrere leere Zeilen getrennt sind:

```
target1:    source1 source 2
            command-line
            [evtl. weitere command-lines]

target2:    source_x source_y
            command-line
            ……
```

Die Blöcke haben folgende Struktur:

1) Regel-Zeile (rules line)	Definition des Ziels (target:) und der Fileabhängigkeiten (Liste der Dateien, von welchen das Ziel abhängig ist)
2) Befehlszeile(n)	Definiert die Schritte, die notwendig sind, um das Zielfile auf den aktuellen Stand zu bringen (compilieren, linken etc.)

Als Beispiel betrachten wir das C- (C++)-Programm aus dem Abschnitt 2.3.1.1 (siehe S. 13), welches aus drei Modulen besteht (*main.c*, func1.c, func2.c). Für die Module func1.c und func2.c existieren die Header-Dateien (func1.h und func2.h), die die Deklarationen der exportierten Funktionen enthalten. Diese werden bei der Compilation von allen Dateien benötigt, welche die exportierten Funktionen benutzen (importieren).

Zusätzlich benötigen alle Module Konstanten und Typen, die in der Header-Datei `mydefs.h` zentral verwaltet werden.

Die Abhängigkeiten der Dateien können graphisch als ein Netzwerk dargestellt werden (Fig.3). Das dazugehörige *Makefile* ist in Fig.4 abgebildet.

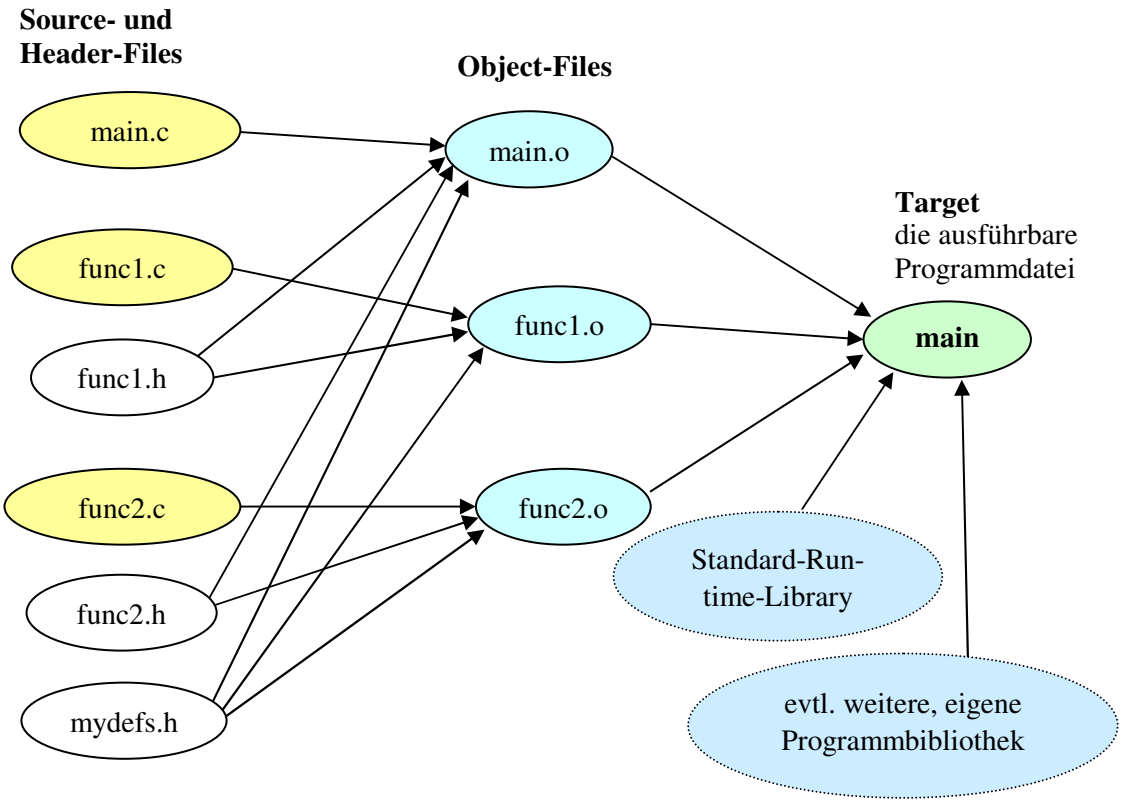


Fig.3 : Darstellung der Abhängigkeiten der Programmodule als *directed graph*

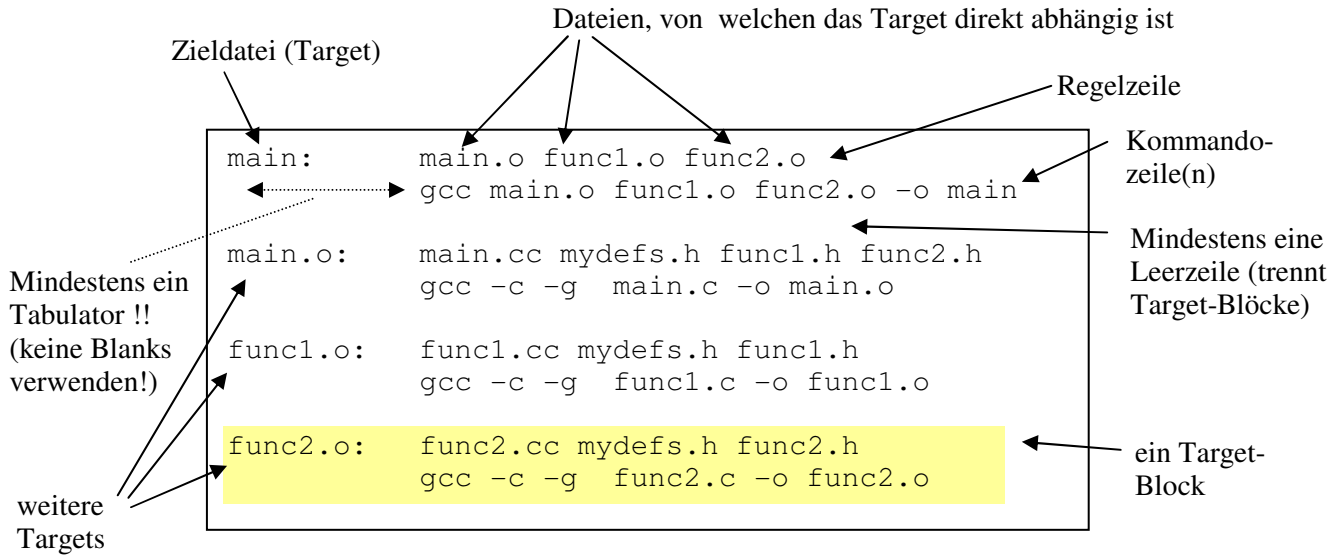


Fig.4 : Einfache Make-Datei `makefile`: textuelle Darstellung von Fig. 3

Wenn die make-Utility nur mit dem Befehl "make" aufgerufen wird, sucht sie im aktuellen Verzeichnis nach der Datei "Makefile" oder "makefile" und darin nach dem ersten Target (Ziel "main").
Sie überprüft anschliessend, von welchen Dateien dieses Target abhängig ist (main.o, func1.o und func2.o). Diese Dateien sind weiter unten als untergeordnete Ziele in der Datei `makefile` aufgeführt (zusammen mit den dazugehörigen Abhängigkeitsregel- und Kommandozeilen).

Wird bei irgendeinem Target festgestellt, dass eine der Dateien, von denen es abhängt, *jünger* als das Target-File ist, dann wird das Target gemäss der nachfolgenden Kommandozeile neu gebildet.

Diese Prüfung wird rekursiv fortgesetzt, bis alle Targets aktualisiert sind.

Bemerkungen:

- **Reihenfolge der Definitionen im `Makefile`:** Die Regelzeilen im `Makefile` beschreiben im Prinzip die Topologie des Graphen in Fig. 3: Jede Regelzeile definiert die Vorgänger eines Knotens. Dabei ist die Reihenfolge wichtig: Man geht vom Endknoten aus (im Graphen von rechts nacht links). So werden zuerst die Vorgänger des Endknotens *main* definiert , dann, weiter **unten,** folgen die Definitionen der Vorgänger der Knoten im mittleren Bereich (*main.o*, func1.*o* und func2.*o*) usw.
- **Systemweite Standard-Library-Files** werden in den Regelzeilen nicht explizit aufgeführt, obwohl das Target davon abhängig ist. Es wird angenommen, dass sich solche Files im Laufe der Projektentwicklung nicht ändern. Alle projektspezifischen *Libraries* und *Include-Files* müssen aber in den Regelzeilen erfasst werden!
- **Tabulatoren:** ***Jede Kommandozeile muss mindestens mit einem Tabulator beginnen!*** *Keine Spaces !!*
- **Zeilenfortsetzung:** Lange Regel- und Kommandozeilen können am Ende mit "\"-Zeichen auf der nächsten Zeile fortgesetzt werden.
- **Kommentarzeilen** beginnen mit # Zeichen.
- **Targetblock-Länge**: Nach einer Regelzeile können *beliebig viele Kommandozeilen* folgen (jede Zeile muss mit TABs beginnen). Auf einer Kommandozeile können auch alle *Shell*-Kommandos verwendet werden (siehe Beispiel unten). Der *Kommandozeilenblock wird durch eine Leerzeile beendet.*
- **Automatische Benennung der Oject-Files:** Ist bei einem UNIX-Compiler die Option **`-c`** (nur compilieren, ohne Linken) angewählt, so bekommt das übersetzte File automatisch den gleichen Basisnamen wie das Source-File und die Extension **`.o`** (z.B. `func1.c` → `func1.o`). In den Kommandozeilen von Fig. 4 könnte also die Option **`-o .....`** weggelassen werden

  (z.B. `gcc -g func1.c`).
- **Unabhängige Zielblöcke**: Im **`Makefile`** können auch Ziele definiert werden, die von keinem anderen File abhängig sind. Die dazugehörigen Target-Blöcke werden deshalb nie automatisch ausgeführt, sie können aber via einen ***make-****Aufruf mit expliziter Angabe des Ziels* aktiviert werden und dienen speziellen Aufgaben, die nicht bei jeder Programmänderung ausgeführt werden sollten (z.B. Aufräumen im Verzeichnis durch Löschen aller nicht mehr benötigten Zwischendateien, Übersetzung von allen Files, Backup etc.).

Beispiel:

der letzte Block:
target **"clean"**

mit dem *Aufruf*
`make clean`

werden alle (nicht mehr benötigten) Objekt-Dateien im aktuellen Verzeichnis gelöscht und dann auf dem Bildschirm eine Meldung "Verzeichnis geputzt" ausgegeben.

```
main:       main.o func1.o func2.o
◄···TAB···► gcc main.o func1.o func2.o -o main

main.o:     main.c mydefs.h func1.h func2.h
◄···TAB···► gcc -c -g  main.c -o main.o

func1.o:    func1.c mydefs.h func1.h
◄···TAB···► gcc -c -g  func1.c -o func1.o

func2.o:    func2.c mydefs.h func2.h
◄···TAB···► gcc -c -g  func2.c -o func2.o

clean:
◄···TAB···► rm *.o
◄···TAB···► echo "Verzeichnis geputzt"
```

2.4.2 Macros

Mit Hilfe von Macros können am Anfang eines **Makefile** Abkürzungen definiert werden, die dann in den Kommandozeilen verwendet werden. Das obige **Makefile** kann mit Macros wie folgt vereinfacht werden:

```
# ------------------------------------------------------------------
# Makefile fuer Programmbeispiel mit separaten Dateien
# Fach:     BS, J. Zeman, Sept. 99
# File:     makefile
# ------------------------------------------------------------------
#macros
CC= gcc
CFLAGS= -c -g

main:       main.o func1.o func2.o
            $(CC) main.o func1.o func2.o -o main

main.o:     main.c mydefs.h func1.h func2.h
            $(CC) $(CFLAGS)   main.c -o main.o

func1.o:    func1.c mydefs.h func1.h
            $(CC) $(CFLAGS)    func1.c -o func1.o

func2.o:    func2.c mydefs.h func2.h
            $(CC) $(CFLAGS)    func2.c -o func2.o

clean:
            rm *.o
            echo "Verzeichnis geputzt"

all:
            rm -f *.o
            make main
```

Fig.5: Makefile mit einfachen Macros.

Eine Macro-Abkürzung (Macro-Variable) wird am Anfang des Makefiles durch eine einfache Zuweisung definiert:

CC= gcc

Später kann eine Makrovariable überall in den Makefile-Anweisungen vorkommen. Dabei wird die Konstruktion **$(Macro_Namen)** durch den Inhalt der Macro-Variablen **Macro_Namen** ersetzt. Man beachte, dass dabei mehrbuchstabige Macro-Namen immer in Klammern **( )** oder **{ }** eingeschlossen werden müssen. Einbuchstabige Macro-Namen können ohne Klammern referenziert werden (**X** → **$X** oder **$(X)** oder **${X}**).

Durch die Verwendung von Macros ist es z.B. möglich, zentral, an einer Stelle im Makefile festzulegen, welcher Compiler (z.B. **gcc** oder **c++**) verwendet wird.

Im obigen **Makefile** sind auch zwei ***unabhängige Ziele*** (Targets *clean* und *build*) definiert.

Das Target **build** veranlasst, dass das Projekt frisch übersetzt und gelinkt wird. Im dazugehörigen Target-Kommandoblock werden zuerst mit Hilfe des Shell-Kommandos **touch** die Änderungszeiten aller Quelldateien auf die momentane Systemzeit gesetzt. Anschliessend ruft sich **make** selber rekursiv auf.

Da alle Quelldateien nun jünger sind als die im Verzeichnis vorhandenen dazugehörigen Objektdateien, werden sie gemäss der Make-Regel neu übersetzt und anschliessend das ganze Programm neu gebildet.

2.4.2.1 Vordefinierte Macros für explizite Bildungsregel

Nachfolgend sind einige nützliche, vordefinierte Macros aufgeführt, welche in den Make-Kommandozeilen direkt verwendet werden können. Sie stellen folgende Information dar:

$@ vollständiger Name (incl. Extension) des Targets aus der zugehörigen Regelzeile.

$? Liste von vollständigen Namen aller Quellenfiles aus der zugehörigen Regelzeile, welche *neuer* sind als das Target-File.

Weitere Anwendungen dieser Macros sind aus Fig. 6 ersichtlich.

```
# ------------------------------------------------------------------------
# Makefile fuer Programmbeispiel mit separaten Dateien
# Fach:      BS, J. Zeman, Sept. 99
# File:      makefile
# ------------------------------------------------------------------------
#macros
CC= gcc
CFLAGS= -c -g


main:       main.o func1.o func2.o
            $(CC) main.o func1.o func2.o -o $@

main.o:     main.c mydefs.h func1.h func2.h
            $(CC) $(CFLAGS)   main.c -o $@

func1.o:    func1.c mydefs.h func1.h
            $(CC) $(CFLAGS)     func1.c -o $@

func2.o:    func2.c mydefs.h func2.h
            $(CC) $(CFLAGS)     func2.c -o $@

clean:
            rm *.o
            echo "Verzeichnis geputzt"

all:
            rm -f *.o
            make main
```

Fig.6: Makefile mit weiteren Macros.

2.4.3 Aufruf des *make*-Programmes

Make kann mit Switches und mit oder ohne zusätzlichen Parametern aufgerufen werden:

1) Einfachster Aufruf ohne Parameter: ***make***
2) Aufruf mit Parametern: ***make -Switch(es) Parameter***

Wirkung: Mit den Switches kann das Verhalten von **`make`** beeinflusst werden.

Beispiele: **`make -n -d -p -f mymakefile "FLAGOPT = -g"`**

Wirkung: -n : (no execution) ***make*** testet die Abhängigkeiten und Modifizierungszeiten der Files, führt aber die Kommandos nicht aus. Dies ist besonders für das Testen eines *Makefile* interessant.

-d : (debugging) *make* beschreibt alle während der Ausführung getroffenen Entscheidungen.

-p : (print) *make* zeigt alle Macrodefinitionen und Zielbeschreibungen.

-f : (filename) Das nachfolgend definierte Steuerfile (im Beispiel `mymakefile`) wird anstatt des default-Files `makefile` oder `Makefile` gelesen.

"Macro_Name = Text" : Definition eines Macros auf der Kommandozeile. Im obigen Beispiel wird ein Macro FLAGOPT definiert, welches zusätzliche Compiler-Switches spezifiziert. Auf diese Art kann z.B. ohne Aenderung des `Make-`

`file` ein Debugging-Switch temporär bei der Übersetzung angegeben werden.

2.4.4 Make all, make build

Die meisten Makefiles beinhalten ein Target `all:` oder `build:`, das eine vollständige Neuübersetzung des gesamten Programms bewirkt. Früher wurde dazu mit dem Befehl `touch *.c, touch *.cc`, etc., das Datum aller Quellenfiles aktualisiert was eine Neuübersetzung zur folge hat. Moderne Editoren überprüfen jedoch dauernd, ob sich die editierten Files ändern und geben dann eine Aufforderung zu Neuladen aus. Um das umgehen, können auch die Object-Files gelöscht werden und so eine Neuübersetzung provoziert werden.

2.5 *make* für Fortgeschrittene

2.5.1 Universelle Make-Datei

Mit Hilfe weiterer vordefinierter Macros und implizierten resp. expliziten Bildungsregeln können universelle Make-Dateien für Gruppen von Programmen erstellt werden, die ähnlich gebildet werden und die gleichen Libraries benutzen. In einer solchen Make-Datei müssen dann nur die Namen der Programmquelldateien als Macros neu definiert werden. Beispiel:

```
CMP=            gcc
CMPFLAGS=       -g
TARGET1=        ProcA10_1
TARGET2=        ProcA10_2

LIBNAME=        -lpthread

compile:
                make $(TARGET1)
                make $(TARGET2)

$(TARGET1):     $(TARGET1).o
                $(CMP)  $(TARGET1).o $(LIBNAME) $(LDFLAGS) -o $(TARGET1)

$(TARGET2):     $(TARGET2).o
                $(CMP)  $(TARGET1).o $(LIBNAME) $(LDFLAGS) -o $(TARGET1)

.c.o:
                $(CMP) -c $(CMPFLAGS) $<
```

Make sucht das erste Target, hier **`compile`**, und führt anschliessend die beiden Befehlszeilen aus. Hier ruft sich make selbst auf, einmal mit dem Target **`TARGET1`** und einmal mit dem Target **`TARGET2`**. Die beiden Targets sind mit Hilfe von Marcros definiert und bilden die beiden Programme **`ProcA10_1`** und **`ProcA10_2`**.

2.5.2 Explizite und implizite Bildungsregeln

Bis jetzt haben wir bei den Bildungsregelen die Datei-Namen explizit angegeben. Beispiel:

```
func1.o:   func1.c func1.h mydefs.h
           $(CC) $(CFLAGS) func1.c -o $@
```

Fig.8: Makefile (einfachste Variante mit expliziten Bildungsregeln)

Implizite Regeln (auch Suffixregeln genannt) verallgemeinern die expliziten Regeln, indem sie sich auf alle Dateien mit gleichen File-Extensions beziehen. Beispiel:

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

Fig.9: Implizite Regel oder Suffixregel

Hier werden alle Dateien mit Extension **.c** resp. .cc aufgrund der Befehlszeile in Dateien mit Extension **.o** übersetzt. Das vordefinierte Macro **$<** setzt den Namen der aktuellen Quellendatei in die Befehlszeile ein.

2.5.2.1 Vordefinierte Macros für implizite Regeln.

Die folgenden zwei Macros sind in den implizierten Regeln nützlich (nur in Befehlszeilen erlaubt):

- **$<** vollständiger Name der aktuellen Quellendatei
- **$*** Basisname (ohne Extension) der aktuellen Zieldatei

2.5.2.2 Default Suffixregeln

Für die meisten gängigen Extensions sind entsprechende Suffixregeln vordefiniert und müssten eigentlich nicht angegeben werden. Um sicherzustellen, dass die Programme so übersetzt werden, wie Sie wollen und aus Kompatibilitätsgründen sollte jedoch auf den Einsatz dieser Möglichkeit verzichtet werden.

2.5.2.3 Diskussion

Die Verwendung von Suffixregeln haben jedoch auch ihre Nachteile: die Handhabung von Header-Dateien ist nicht möglich, da ja verschiedene Source-Dateien im Allgemeinen auch verschiedene Header-Dateien einbinden. Änderungen an Header-Dateien werden in diesem Fall nicht berücksichtigt, was unter Umständen zu schwierig auffindbaren Fehlern führen kann.

2.5.3 Substitution innerhalb von Macros

Auch innerhalb von können Substitutionen gemacht werden, die Anweisung $(MacroName:Text1=Text2) ersetzt im Macro **MacroName** jeden String **Text1** durch den String **Text2**.

Beispiel:

```
FILES = func1.cc func2.cc main.cc
main:       $(FILES:.c=.o)
            $(CC) $(FILES:.c=.o) -o $@
.c.o:
            $(CC) -c $(FILES)
```

Hier muss nur eine Liste der Source Files mit Extension **.c** angegeben werden, die Liste der Objektfiles (Extension **.o** wird mit Hilfe einer Substitution erzeugt. Anstelle des Macros **$(FILES)** in der Suffixregel kann auch das Macro **$<** verwendet werden.

2.5.4 Projekte mit mehreren `makefiles`

Befehlszeilen in **`makefiles`** können beliebige Shell-Befehle enthalten und nicht nur Anweisungen zum Kompilieren von Programmen. Hier das Beispiel, das aus den vier Programmen **`modify`**, **`append`**, **`next`** und **`select`** besteht. Die Quellen der Programme **`modify`** und **`append`** sind im Verzeichnis **`manip`** abgelegt, von **`next`** und **`select`** in **`choose`**. Nach der Neukompilation müssen die Programme nach /usr/local/bin kopiert werden.

```
addrsys:   manip/*.c mani/*.h choose/*.c choose/*.c
           cd manip; make
           cd choose; make
           cp manip/modify manip/append \
              choose/next choose/select /usr/local/bin
           echo "AddrSys update :" `date` > updateDate
```

Wenn in den beiden Verzeichnissen **`manip`** oder **`choose`** eine Datei geändert wurde, werden die **`makfiles`** in diesen Verzeichnissen abgearbeitet und nach dem Übersetzen die Programme nach **`/usr/local/bin`** kopiert. Der Backsalsh **`\`** bedeutet, dass die Zeile fortgesetzt wird, damit wird der Zeilenumbruch wird überlesen (ein Befehl kann nur auf einer Zeile angegeben werden).

Am Schluss wird das Aktuelle Datum in der Datei **`updateDate`** abgespeichert.