Materialpaket 01_ENV – Environment

C/C++, Autor: Prof. Dr.-Ing. Carsten Link

Version 1.3.1 March 3, 2019

Contents

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten, Fertigkeiten zur Problemlösung):

Sie können die für die C/C++-Programmierung übliche Arbeitsumgebung bedienen.

Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernegebnisse: Sie können nachweislich[1]:

- die zur Verfügung gestellte Linux-Programmierumgebung verwenden
- sich im Verzeichnisbaum mit der `bash`-shell bewegen
- Quelltexte editieren

[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen

- Programme, welche aus mehreren Quelltextdateien bestehen, übersetzen lassen
- ein C/C++-Programm, welches aus mehreren Quelltextdateien besteht, erstellen
- die Aufgaben der nötigen Werkzeuge benennen
- dieses Programm modifizieren, so dass sich ein anderes Verhalten ergibt

2 Konzepte

2.1 Vorübung Programmierumgebung

Installieren Sie VirtualBox auf ihrem Rechner und binden die virtuelle Maschine[2] ein. Fahren Sie die Maschine hoch und melden sich als Nutzer `devel` mit dem Passwort `devel` an.

Starten Sie die `bash`-shell (bzw. `LXTerminal`) und erstellen ein neues Verzeichnis, in dem Sie dann arbeiten werden. Erstellen und editieren Sie die Quelldateien mit `geany`, `gedit` oder `LeafPad`.

Auf der Kommandozeile sind folgende Befehle hilfreich:

- `pwd`: anzeige des aktuellen Arbeitsverzeichnisses
- `cd name`: wechseln in das Verzeichnis `name`. Spezielle Namen sind: `~` home directory, `..` eine Ebene höher, `.` das aktuelle Verzeichnis
- `mkdir name`: anlegen des Verzeichnisses `name`
- `touch name`: erstellt die Datei `name` oder ändert den Modifikationszeitstempel (für Backups oder `make`)
- `ls`: listet die Dateien im aktuellen Verzeichnis auf. `ls -l` gibt mehr Details
- `man name`: zeigt eine Hilfeseite zum Thema `name`. Beenden mit `q`
- `gedit name`: startet den Editor `gedit` mit der Datei `name`
- `gedit name &`: startet den Editor `gedit` mit der Datei `name` im Hintergrund
- `rm name`: löscht die Datei `name`
- `chmod +x name`: macht die Datei `name` ausführbar
- `./name`: startet die Datei `name`

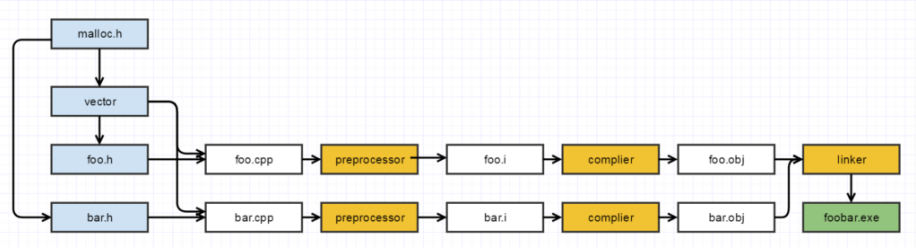
Eine kurze Einführung zum Thema Unix-Kommandozeile finden Sie hier[3] unter *Tutorial One* und *Tutorial Two*.

2.2 Toolchain

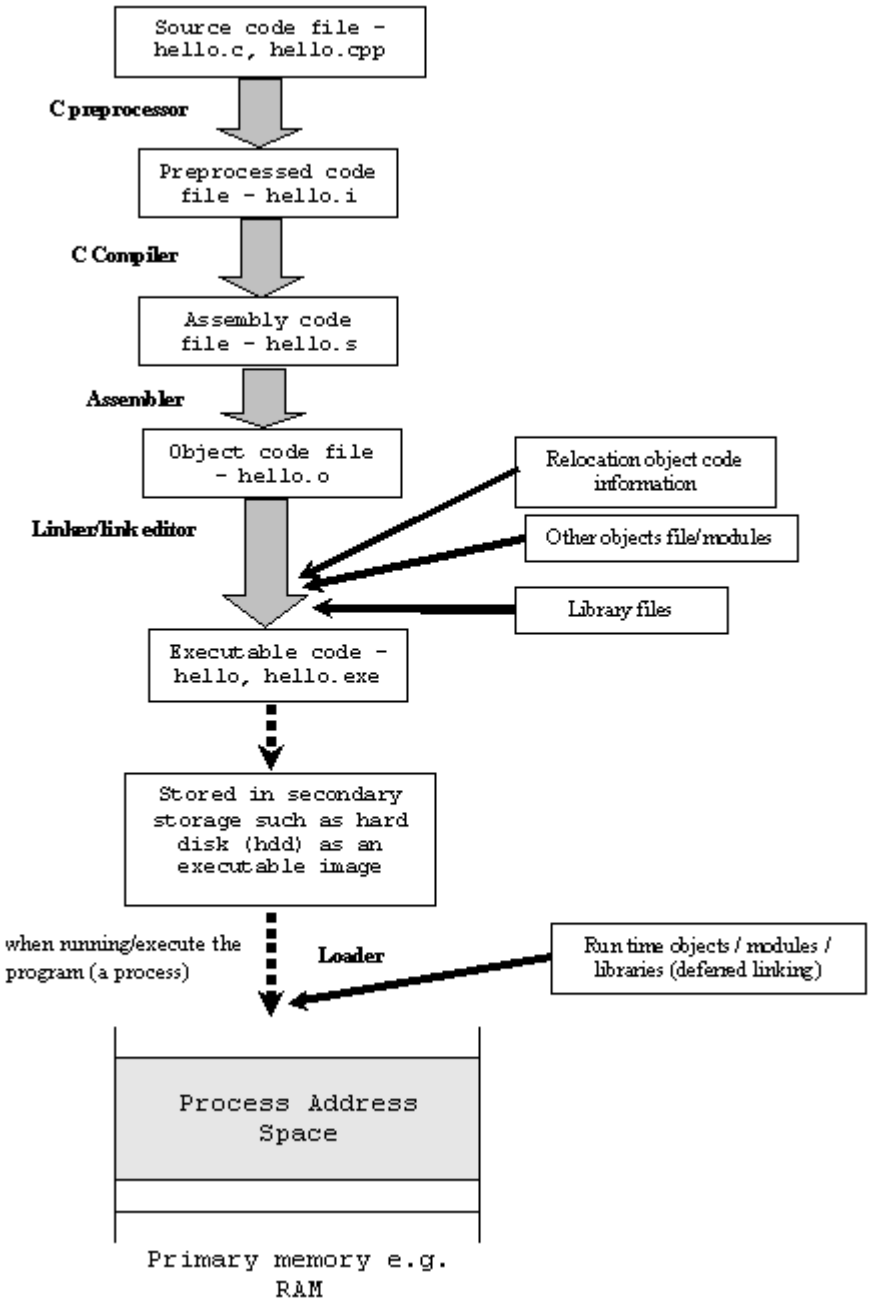
- Editor
- Preprocessor (Präprozessor)

[2] http://www.technik-emden.de/~clink/FedoraLXDE-2016-01-05.7z
[3] http://www.ee.surrey.ac.uk/Teaching/Unix/

- Compiler
- Assembler
- Linker (Binder)
- Treiber
- Include-Datei
- Quelltext-Datei
- Objekt-Datei
- Library (Bibliotheksdatei)
- C-Funktion
- Deklaration
- Definition



Die verschiedenen Werkzeuge der Werkzeugkette (Quelle: https://techblog.king.com/speeding-up-build-times-with-masterunitybulk-builds/).



Eine weiter Darstellung aus dem Artikel [TENCAL].

In den Abbildungen ist zu sehen, wie die einzelnen Werkzeuge aus Eingabedateien Ausgabedateien erstellen:

- Der Präprozessor behandelt Präprozessordirektiven. So wird beispielsweise `#include filename` die Stelle der Direktive durch den Inhalt der Datei `filename` ersetzt. Die Direktive `#define BUFSIZE 10` sorgt dafür, dass im darauffolgenden Quelltext `BUFSIZE` durch `10` ersetzt wird. Die erstellte Datei hat die Endung `.i`
- Der Compiler übersetzt den vom Präprozessor behandelten Quelltext in Assemblercode oder Maschinencode (Dateiendungen `.s` oder `.o`)
- Der Assembler übersetzt Assemblercode (assembly, Endung `.s`) in Maschinencode (Dateiendung `.o`)
- Der Linker setzt mehrere Dateien mit Maschinencode (`*.o`) zu einer ausführbaren Datei zusammen (`a.out`)
- Schließlich lädt das Betriebssystem (loader) das Programm in den Arbeitsspeicher, versorgt es mit Speicherbereichen und lässt die CPU in die Funktion `main` springen

Der gesamte Übersetzungsvorgang soll anhand eines kleinen C-Programmes illustriert werden. Zur Übersetzung wird `pcc` (The Portable C Compiler[4]) verwendet, da dieser einfache Compiler sehr übersichtlichen Assemblercode erzeugt.

Wichtig: im folgenden Abschnitt sowie in der *Vorübung Toolchain* sollen sie die Werkzeuge kennen lernen und eine Vorstellung davon entwickeln, welche Art von Zuständigkeiten diese jeweils haben. Es ist nicht nötig, jedes Detail zu verstehen – ein grober Überblick reicht völlig aus.

Das nachfolgende Programm verfügt über die Hauptfunktion `main()` (welche vom Betriebssystem im Zusammenspiel mit den Standardbibliotheken aufgerufen wird), die Funktion `sum()`, sowie eine globale Variable `int global`.

```
1  #include <stdio.h>
2
3  int global = 8150;
4
5  int sum(int a, int b){
6    int result = 451;
7    result = a + b;
8    return result;
9  }
10
11 int main(int argc, char **argv)
12 {
13   int local=4711;
14   printf("Hello, world!\nglobal=%d local=%d\n", global, local);
15   local = sum(global, local);
16   return local;
17 }
```

[4]Portable C Compiler http://pcc.ludd.ltu.se

Das obige Programm ist in der Datei `main.c` gespeichert. Übersetzt wird es mit der Datei `build.sh`:

```
#!/bin/sh
# generate main.s, main.o, b.out
pcc -O0 -S main.c
as -o main.o main.s
pcc -o b.out main.o
# generate a.out
pcc -O0 -g main.c
# generate assembly intermixed with source code
objdump -S a.out > objdump-S_a.out.txt
```

Das oben angegeben Shell Script startet mehrere Werkzeuge um diverse Übersetzungen zu erhalten.

- `pcc -O0 -S main.c` erstellt eine Assemblerdatei `main.s`
- `as -o main.o main.s` übersetzt `main.s` in die Objektdatei main.o. Diese Objektdatei enthält die ausführbaren Funktionen aus `main.c`, jedoch fehlen Funktionen aus den Standardbibliotheken (Zur Ausgabe mit `printf` und zum Starten und Beenden des Programms)
- `pcc -o b.out main.o` nutzt `pcc` als Treiber (driver), um vom Linker `ld` das vollständige (ausführbare) Programm `b.out` erstellen zu lassen.
- `pcc -O0 -g main.c` nimmt nicht den Umweg über eine Assemblerdatei; `a.out` (default name) wird direkt mittels Assembler und Linker erstellt (pcc als Treiber). Die Option `-g` sorgt dafür, dass die Ausgabe `a.out` mit Debug-Informationen versehen wird; die Option `-O0` sorgt dafür, dass der Compiler keine Optimierungen vornimmt

Das Kommando `objdump -S` stellt den Assemblercode wieder her und mischt diesen mit dem ursprünglichen Quelltext (falls debug info in der ausführbaren Datei vorhanden ist).

```
08048468 <sum>:
#include <stdio.h>

int global = 8150;

int sum(int a, int b){
 8048468:   c8 08 00 00             enter  $0x8,$0x0
  int result = 451;
 804846c:   c7 45 fc c3 01 00 00    movl   $0x1c3,-0x4(%ebp)
  result = a + b;
 8048473:   8b 45 08                mov    0x8(%ebp),%eax
 8048476:   03 45 0c                add    0xc(%ebp),%eax
 8048479:   89 45 fc                mov    %eax,-0x4(%ebp)
  return result;
 804847c:   8b 45 fc                mov    -0x4(%ebp),%eax
```

```
 804847f:   89 45 f8                mov    %eax,-0x8(%ebp)
 8048482:   eb 00                   jmp    8048484 <sum+0x1c>
}
 8048484:   8b 45 f8                mov    -0x8(%ebp),%eax
 8048487:   c9                      leave
 8048488:   c3                      ret
 8048489:   8d 76 00                lea    0x0(%esi),%esi

0804848c <main>:


int main(int argc, char **argv)
{
 804848c:   c8 08 00 00             enter  $0x8,$0x0
  int local=4711;
 8048490:   c7 45 fc 67 12 00 00    movl   $0x1267,-0x4(%ebp)
  printf("Hello, world!\nglobal=%d local=%d\n", global, local);
 8048497:   ff 75 fc                pushl  -0x4(%ebp)
 804849a:   ff 35 1c a0 04 08       pushl  0x804a01c
 80484a0:   68 58 85 04 08          push   $0x8048558
 80484a5:   e8 06 fe ff ff          call   80482b0 <printf@plt>
 80484aa:   83 c4 0c                add    $0xc,%esp
  local = sum(global, local);
 80484ad:   ff 75 fc                pushl  -0x4(%ebp)
 80484b0:   ff 35 1c a0 04 08       pushl  0x804a01c
 80484b6:   e8 ad ff ff ff          call   8048468 <sum>
 80484bb:   83 c4 08                add    $0x8,%esp
 80484be:   89 45 fc                mov    %eax,-0x4(%ebp)
  return local;
 80484c1:   8b 45 fc                mov    -0x4(%ebp),%eax
 80484c4:   89 45 f8                mov    %eax,-0x8(%ebp)
 80484c7:   eb 00                   jmp    80484c9 <main+0x3d>
}
 80484c9:   8b 45 f8                mov    -0x8(%ebp),%eax
 80484cc:   c9                      leave
 80484cd:   c3                      ret
 80484ce:   66 90                   xchg   %ax,%ax
```

Es ist zu sehen, dass beide Funkionen `main()` und `sum()` mit `enter` beginnen und mit `leave` enden. Dies dient dem Auf- bzw. Abbau des Aktivierungsrecords[5]. Dadurch erhalten lokale Variablen Speicherplatz und rekursive Aufrufe sind möglich, ohne Daten von anderen Ausprägungen der jeweiligen Funktionen zu überschreiben. Funktionsaufrufe werden mit `call` umgesetzt, die aufgerufene Funktion lässt die CPU mit `ret` zum Aufrufer zurückspringen. Hierzu hat `call` die Adresse des nachfolgenden Befehls auf den Call Stack gelegt und ret lädt

[5] https://en.wikipedia.org/wiki/Function_prologue

diesen in den Instruction Pointer.

Im nächsten Materialpaket (02_MENT) wird näher auf den Call Stack eingegangen.

2.3 Vorübung Toolchain

Erweitern Sie das oben angegebene Programm um eine lokale Variable `lineLocator` und weisen in möglichst vielen Quelltextzeilen dieser Variable den Macro-Wert `__LINE__` zu. Übersetzen Sie mit `pcc -O0 -S` und inspizieren die Ausgabe.

1. Welche Veränderungen ergeben sich, wenn Sie lokale Variablen hinzufügen? Verwenden Sie `diff` oder `Diffuse Merge Tool`, um Änderungen hervorheben zu lassen.

3 Material zum aktiven Lernen

Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C/C++Programm) erstellt, welches dann auf mehrere Arten Modifiziert wird. Insbesondere die Modifikationen ermöglichen es dem Lernenden (und auch dem Lehrenden), die Qualität des Kompetenzerwerbs bzgl. dieses Materialpakets bewerten zu können.

3.1 Aufgabe: Grundgerüst

Erstellen Sie ein Programm, das aus den Dateien `main.c, func1.h, func1.c` besteht. In `func1.h`wird eine Funktion deklariert, die in `func1.c` definiert wird und diese Signatur hat:

```
1 int func1(int x); // returns y = f(x) for value x. f(x) is a polynominal function.
```

In der `main()`-Funktion wird die Funktion `func1()` für die Werte `0..20` aufgerufen.

Lassen Sie die Quelldateien von `clang` getrennt übersetzen (siehe Compileroption `-c`) und lassen das ausführbare Programm aus den Objektdateien zusammenbinden (`clang func1.o main.o`). Hierzu können Sie ein `bash`-script verwenden[6], um sich Tipparbeit zu ersparen.

[6] Mit `chmod 755 <file name>` können Sie das Script ausführbar machen

3.2 Aufgabe: Modifikationen

Weiter unten ist eine Liste mit Modifikationen gegeben, die zwei Zwecken dienen: 1) Sie dienen als Richtschnur für das Praktizieren und Üben der Inhalte dieses Materialpakets. 2) Die Modifikationen können im Rahmen eines Testats verwendet werden, womit Studiernde nachweisen können, dass sie den Stoff dieses Materialpakets beherrschen.

Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (2 - 5) vor Zuschauern (Testatsituation) umsetzen können. Konkret sollen Sie im Testat in der Lage sein, das gegebene Grundgerüst um mindestens eine zufällig ausgewählte Modifikation zu erweitern. Bereiten Sie dazu auf ihrer Arbeitsumgebung ein Verzeichnis vor, welches ausschließlich das Grundgerüst enthält. Im Testat sollen Sie die `bash` und `gedit` verwenden.

Modifikationen:

1. Eine zweite C-Funktion `func2()` mit einer anderen Polynomfunktion einbauen
2. Die Rückgabewerte in einer `for` oder `while`-Schleife aufaddieren (also iterativ implementieren)
3. `for` durch `while`-Schleife ersetzen (oder umgekehrt)
4. Ermitteln, wie viele Nullstellen in dem abgesuchten Bereich liegen (Rückgabewert `== 0`)
5. Bauen Sie eine Funktion `recurse()`, welche sich selbst und `func1()` aufruft und dabei die Rückgabewerte ausgibt. Dabei wird in `main()` `recurse(20)` aufgerufen; `recurse()` ruft sich selbst mit den Werten `19` bis `0` auf
6. **Fortgeschritten** Geben Sie während der obigen Rekursion die Werte in umgekehrter Reihenfolge aus, jedoch ohne die Aufrufparameter zu verändern

3.3 Verständnisfragen

Nach Bearbeitung des Kapitels "Konzepte", der Erstellung des Grundgerüsts sowie dem Üben der Modifikationen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten.

1. Schreiben Sie kurze Glossar-Einträge für alle an der Übersetzung/Erstellung eines Programms beteiligten Komponenten bzw. Phasen.
2. Ist es möglich, von einer Quelldatei aus eine Funktion aufzurufen, welche in einer anderen Quelldatei definiert ist? Begründung!
3. Welche Aufgaben übernimmt der Compiler?
4. Was ist der Unterschied zwischen einer ausführbaren Datei (z.B. `a.out`) und einer Objektdatei (`.obj`)?
5. Was ist der Unterschied zwischen einer Bibliotheksdatei (`.a`) und einer ausführbaren Datei (z.B. `a.out`)?

6. Was ändert sich im Assemblercode (`objdump -S`), wenn Sie einer Funktion eine lokale Variable hinzufügen?
7. Welche Art von Fehler von welchem Werkzeug ergibt sich, wenn `#include "func1.h"` in `main.c` fehlt, dort aber `func1()` aufgerufen wird?
8. Welche Art von Fehler von welchem Werkzeug ergibt sich, wenn `#include "func1.h"` in `main.c` vorhanden ist und `int func1(int)` in `func1.h` deklariert ist, aber `func1.c` keine Definition von `int func1(int)` enthält?

4 Nützliche Links

- Sektion 3 der Linux Manual Pages (z. B. `man 3 printf`)
- Clang Man Page online[7] oder per `man clang`
- TENOUK'S BUFFER OVERFLOW 3 An Assembly Language[8]
- Tenouk's C & C++ Site[9]
- Wolfgang Schröder, C++-Tutor[10]
- Herbert Schildt, C The Complete Reference online[11] (Vorsicht: enthält Fehler)
- Bjarne Stroustrup's FAQ http://www.stroustrup.com/bs_faq.html

5 Literatur

- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition
- [TENCAL] Tenouk: COMPILER, ASSEMBLER, LINKER AND LOADER: A BRIEF STORY[12]

[7] http://clang.llvm.org/docs/CommandGuide/clang.html
[8] http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1b.html
[9] http://www.tenouk.com/Sitemap.html
[10] http://www.cpp-tutor.de/cpp/
[11] https://docs.google.com/file/d/0B3OzFFMgEP0tU3RVcmh2Wm5ZUWs/edit?pref=2&pli=1
[12] http://www.tenouk.com/ModuleW.html