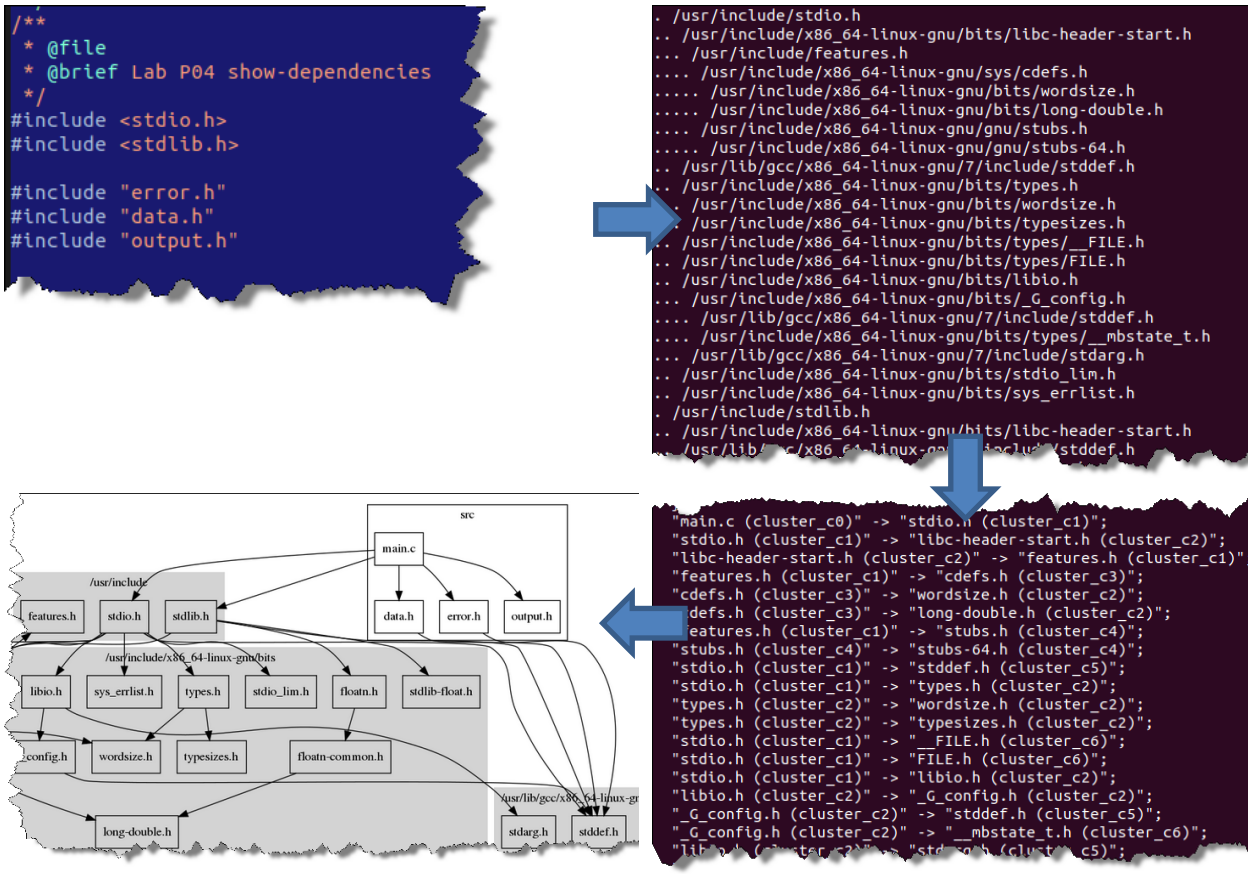
SNP: Modularisieren von C Code



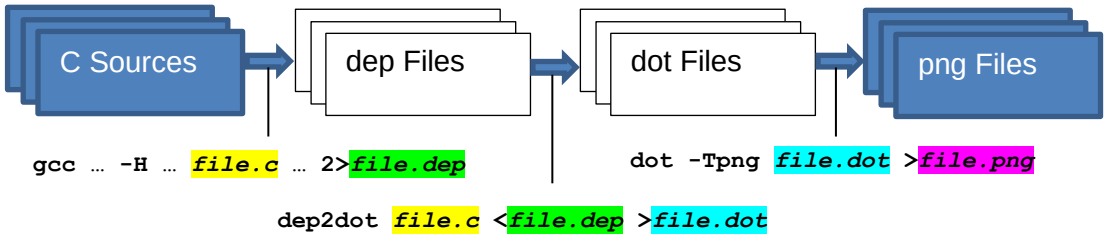
SNP: Modularisieren von C Code 1
1 Übersicht 2
2 Lernziele 2
3 Aufgabe 1: Modularisieren 2
4 Aufgabe 2: Makefile Regeln 5
5 Bewertung 6
6 Anhang 6

1 Übersicht

In diesem Praktikum werden File Abhängigkeiten dargestellt.

In der ersten Aufgabe schreiben Sie zu einem bestehenden C Programm die notwendigen Header Files plus passen das Makefile so an, dass die entsprechenden Module mit kompiliert werden.

In der zweiten Aufgabe erstellen Sie Makefile Regeln für die drei Schritte von den C Source Files zur graphischen Darstellung der Abhängigkeiten.



Im Anhang ist eine Übersicht über die verwendeten File Formate gegeben.

2 Lernziele

In diesem Praktikum lernen Sie die Handgriffe um ein Programm zu modularisieren, d.h. in mehrere Module aufzuteilen.

- Sie wissen, dass ein Modul aus einem C-File und einem passenden H-File bestehen.
- Sie können Header Files korrekt strukturieren.
- Sie wissen wie **Include Guards** anzuwenden sind.
- Sie können Module im **`Makefile`** zur Kompilation hinzufügen.
- Sie können anhand einer Beschreibung Typen und Funktionen in den passenden Header Files deklarieren.
- Sie können **`Makefile`** Regeln schreiben.

Die Bewertung dieses Praktikums ist am Ende angegeben.

Erweitern Sie die vorgegebenen Code Gerüste, welche im **`git`** Repository **`snp-lab-code`** verfügbar sind.

3 Aufgabe 1: Modularisieren

Das zu ergänzende Programm **`dep2dot`** hat folgende Funktionalität:

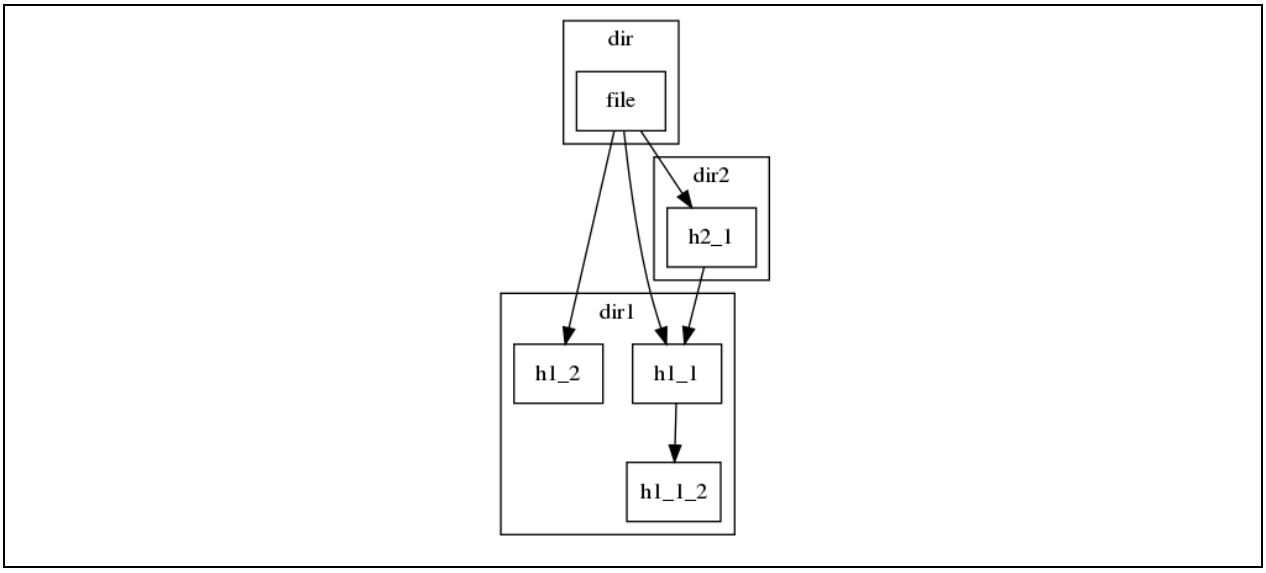
1. es liest von **`stdin`** die vom Compiler generierten Abhängigkeits-Daten in Form des **`dep`** Formates
2. das erste und einzige Command Line Argument gibt das File an für welches die von **`stdin`** gelesenen Abhängigkeiten gelten
3. auf **`stdout`** werden die Abhängigkeiten von **`stdin`** übersetzt als **`dot`** File Format ausgegeben

Das **`dep`** Format und das **`dot`** Format sind im Anhang beschrieben.

Wenn die Aufgabe erfolgreich umgesetzt ist, können Sie zur manuellen Überprüfung der Funktionalität folgende Zeilen auf der Bash Shell ausführen (das **`dep.input`** File ist Teil der automatisierten Test Suite):

```
bin/dep2dot dir/file <tests/dep.input >dep.dot
dot -Tpng dep.dot >dep.png
firefox dep.png
```

Als Resultat sollte Firefox folgende Graphik darstellen:



Als Abnahme muss der Test unverändert ohne Fehler ausgeführt werden:

```
make test
...
Suite: lab test
  Test: test_fail_no_arg ...passed
  Test: test_no_dep ...passed
  Test: test_dep ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      3      3      3      0        0
             asserts   2014   2014   2014      0      n/a
```

Erweitern Sie das vorgegebenen Programm Gerüst an den bezeichneten Stellen so, dass das alle Module Fehler- und Warnungs-frei kompiliert werden. Dazu müssen Sie die Module im **`Makefile`** einbinden und zwei Header Files mit Inhalt füllen und schliesslich das **`dot`** Format generieren.

Prüfen Sie die Umsetzung beider Teilaufgaben mittels **`make test`**.

3.1 Teilaufgabe Modules einbinden, Header Files schreiben

Das Programm besteht aus folgenden Files

`Makefile`	→ **anzupassen**: Module einbinden
`src/main.c`	→ gegeben, d.h. nichts anzupassen
`src/error.h`	→ gegeben, d.h. nichts anzupassen
`src/data.h`	→ **anzupassen**: umsetzen gemäss den Angaben unten
`src/output.h`	→ **anzupassen**: umsetzen gemäss den Angaben unten
`src/data.c`	→ gegeben, d.h. nichts anzupassen
`src/output.c`	→ **anzupassen**: umsetzen gemäss den Angaben unten

1. Als erstes müssen Sie im **`Makefile`** die **`SOURCES`** Variable anpassen so dass **`src/data.c`** und **`src/output.c`** mit kompiliert werden.
2. Prüfen sie das mittels **`make -n clean default`**. Die Option **`-n`** steuert make so, dass die Aktionen nicht wirklich ausgeführt werden, sondern nur ausgegeben wird, was gemacht würde. Die Ausgabe sollte Compileraufrufe für **`src/data.c`** und **`src/output.c`** beinhalten.
3. Füllen Sie folgende Header Files an den angegebenen Stellen mit folgendem Inhalt:

`src/data.h`

- Include Guard
- Deklarieren der folgenden Typen

```
/**
 * @brief  Directory container for file entries of the dependency file.
 */
typedef struct {
  const char *name; ///< @brief the path name of the directory as given gcc.
} dir_t;

/**
 * @brief  File container for the file entries of the dependency file.
 */
typedef struct {
  const char *name; ///< @brief  The base name of the file as given by gcc.
  size_t dir;       ///< @brief  The index of the directory entry.
  size_t level;     ///< @brief  The level as given by gcc.
} file_t;

/**
 * @brief  Overall container for all directories and all files from gcc.
 */
typedef struct {
  size_t n_dirs;  ///< @brief  The number of valid entries in the dirs list.
  dir_t *dirs;    ///< @brief  The list of directories.
  size_t n_files; ///< @brief  The number of valid entries in the files list.
  file_t *files;  ///< @brief  The list of files from the dependency file.
} data_t;
```

- Deklaration der Funktion

```
/**
 * @brief            Entry function to read the dependency data from stdin.
 * @param root [IN]  The name of the root file.
 * @return           The container of the read data from stdin.
 */
const data_t data_read_all(const char *root);
```

- Fehlende Includes

`src/output.h`

- Include Guard
- Deklaration der Funktion

```
/**
 * @brief           Produces DOT output of the dependencies given in data.
 * @param data [IN] Container of the dependency data.
 */
void output_dot(const data_t data);
```

- Fehlende Includes

4. Kompilieren Sie das ganze mittels **`make clean default`**. Es sollten keine Compiler Fehler auftreten. Nur folgende Warnung wird erscheinen:

```
...
warning: ‘print_node’ defined but not used [-Wunused-function]
...
```

Zu beachten: Das Programm ist noch nicht funktionstüchtig, da noch die Umsetzung der folgenden Teilaufgabe fehlt.

3.2 Teilaufgabe Generieren von DOT Output Format

Implementieren Sie im File **`src/output.c`** an den angegebenen Stellen

- die Ausgabe der individuellen Abhängigkeit (**`a -> b`**) gemäss Vorgabe im Code
- die Ausgabe der File Knoten gemäss Vorgaben im Code
- die Ausgabe der File Knoten innerhalb der Directory Cluster gemäss Vorgabe im Code

Zu beachten:

- Benutzen Sie für obige Ergänzungen die Funktion

```
/**
 * @brief  Writes the node name of the given file.
 * @param  file [IN]  The file for which to write the node name.
 * @remark The dependency data contain duplicates of file entries
 *         (the node name must be unique for the path and the *basename* of
 *          the files).
 */
static void print_node(file_t file);
```

- Die Ausgabe muss auf das Zeichen genau den Vorgaben entsprechen damit die Tests (**`make test`**) erfolgreich durchlaufen.

Wenn die beiden obigen Teilaufgaben erfolgreich umgesetzt sind, laufen die Tests ohne Fehler durch und der manuellen Tests wie in der Einleitung angegeben zeigt das aufgeführte **`png`** File in Firefox.

4 Aufgabe 2: Makefile Regeln

Nachdem das Programm in Aufgabe 1 umgesetzt ist, geht es nun darum, im **`Makefile`** Regeln zu definieren welche die einzelnen Schritte von den Source Files zu den **`png`** Files ausführen.

Prüfen Sie schliesslich die Umsetzung beider Teilaufgaben mittels
`make dep-clean dep && firefox src/*.png`.

4.1 Neue Regeln hinzufügen

Führen Sie im **`Makefile`** an den angegebenen Stellen folgende Ergänzungen durch

- definieren Sie eine Variable **`DEPFILES`** deren Inhalt die Liste alle **`Einträge`** der Variable **`SOURCES`** ist, wobei bei allen die Endung **`.c`** durch die Endung **`.c.png`** ersetzt ist
- fügen Sie zum **Pseudo-Target** **`.PHONEY`** das Target **`dep`** dazu – dies besagt, dass das später folgenden Target **`dep`** nicht ein File repräsentiert (ohne dieses Setting würde **`make`** gegebenenfalls nach einem File mit Namen **`dep`** suchen um zu entscheiden ob es inkrementell gebildet werden muss)
- schreiben Sie das Target **`dep`** gemäss der Beschreibung im Makefile
- schreiben Sie die Suffix Regel für die Übersetzung von **`.png`** ← **`.dot`** gemäss Vorgabe im **`Makefile`** (als Inspiration, siehe auch die **`%.c.dep: %.c`** Suffix Regel weiter unten im **`Makefile`**) – erklären Sie was die Regel macht
- schreiben Sie die Suffix Regel für die Übersetzung von **`.dot`** ← **`.dep`** gemäss Vorgabe im **`Makefile`** – erklären Sie was die Regel macht

Die Umsetzung der obigen Änderungen sind erfolgreich, wenn Sie folgende Shell Command Line erfolgreich ausführen können und in Firefox die Abhängigkeiten der C-Files von den Include Files dargestellt wird.
`make dep-clean dep && firefox src/*.png`.

5 Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden.

Aufgabe	Kriterium	Gewicht
1	Sie können das funktionierende Programm inklusive funktionierende Tests demonstrieren und erklären.	
	Teilaufgabe Modules einbinden, Header Files schreiben	2/4
	Teilaufgabe Generieren von DOT Output Format	1/4
2	Sie können das funktionierende Makefile demonstrieren und erklären.	
	Neue Regeln hinzufügen	1/4

6 Anhang

6.1 Verarbeitung und verwendete File Formate

Das Programm in diesem Praktikum ist Teil für die graphische Darstellung von **`#include`** File Abhängigkeit von C Files.

Den ersten Schritt für die Darstellung der **#include** File Abhängigkeiten bietet der Compiler. Der Compiler kann mittels der **-H** Command Line Option auf **stderr** ein Text File generieren, welches die tatsächlich verwendeten Header Files auflistet. Zusätzlich wird im Resultat die Verschachtelungstiefe der Includes angegeben.

Im zweiten Schritt übersetzt das Programm (**dep2dot**) dieses Praktikums solche Dependency Files (**dep**) in eine Text Repräsentation der Abhängigkeiten (**dot**) welche in graphische Darstellung (**png**) übersetzt werden kann.

Als Tool zur Übersetzung der **dot** Files in das **png** Format dient das **dot** Tool. Dieses Tool muss gegebenenfalls installiert werden:

```
sudo apt install graphviz
```

Die **png** Files können dann z.B. in der Programm Dokumentation integriert werden (Darstellung zu Test Zwecken z.B. mittels **firefox file.png**).

6.2 dep File

Siehe: **man gcc**

```
-H  Print the name of each header file used, in addition to other
    normal activities.  Each name is indented to show how deep in the
    #include stack it is. [...]
```

Das File wird auf **stderr** ausgegeben.

Beispiel File (für Abhängigkeiten des **main.c** Files des **dep2dot** Programms)

```
. /usr/include/stdio.h
.. /usr/include/x86_64-linux-gnu/bits/libc-header-start.h
... /usr/include/features.h
.... /usr/include/x86_64-linux-gnu/sys/cdefs.h
..... /usr/include/x86_64-linux-gnu/bits/wordsize.h
..... /usr/include/x86_64-linux-gnu/bits/long-double.h
.... /usr/include/x86_64-linux-gnu/gnu/stubs.h
..... /usr/include/x86_64-linux-gnu/gnu/stubs-64.h
.. /usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h
.. /usr/include/x86_64-linux-gnu/bits/types.h
... /usr/include/x86_64-linux-gnu/bits/wordsize.h
... /usr/include/x86_64-linux-gnu/bits/typesizes.h
.. /usr/include/x86_64-linux-gnu/bits/types/__FILE.h
.. /usr/include/x86_64-linux-gnu/bits/types/FILE.h
.. /usr/include/x86_64-linux-gnu/bits/libio.h
... /usr/include/x86_64-linux-gnu/bits/_G_config.h
.... /usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h
.... /usr/include/x86_64-linux-gnu/bits/types/__mbstate_t.h
... /usr/lib/gcc/x86_64-linux-gnu/7/include/stdarg.h
.. /usr/include/x86_64-linux-gnu/bits/stdio_lim.h
.. /usr/include/x86_64-linux-gnu/bits/sys_errlist.h
. /usr/include/stdlib.h
.. /usr/include/x86_64-linux-gnu/bits/libc-header-start.h
```

```
.. /usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h
.. /usr/include/x86_64-linux-gnu/bits/floatn.h
... /usr/include/x86_64-linux-gnu/bits/floatn-common.h
.... /usr/include/x86_64-linux-gnu/bits/long-double.h
.. /usr/include/x86_64-linux-gnu/bits/stdlib-float.h
. src/error.h
.. /usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h
. src/data.h
.. /usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h
. src/output.h
Multiple include guards may be useful for:
/usr/include/x86_64-linux-gnu/bits/stdlib-float.h
/usr/include/x86_64-linux-gnu/bits/sys_errlist.h
/usr/include/x86_64-linux-gnu/bits/typesizes.h
/usr/include/x86_64-linux-gnu/gnu/stubs-64.h
/usr/include/x86_64-linux-gnu/gnu/stubs.h
```

6.3 `dot` File

Graphviz ist ein mächtiges Tool-Set welches Graphen, definiert in einem **`dot`**-Text File, automatisch anordnet und in **`png`**, **`gif`** und andere Formate übersetzt.

Siehe die offizielle Web-Page https://www.graphviz.org/.

Es gibt als Teil dieses Tool-Sets verschiedene Übersetzer. Der hier verwendete ist der Basisübersetzer: **`dot`**.

Das **`dot`**-File Format kennt viele Möglichkeiten die Knoten und Kanten eines Graphen und deren Anordnung anzugeben.

Der Vorteil eines solchen Tool-Sets ist, dass man den Inhalt (den Graphen) einfach definieren kann und sich nicht um das komplexe Problem der ansprechenden Visualisierung kümmern muss.

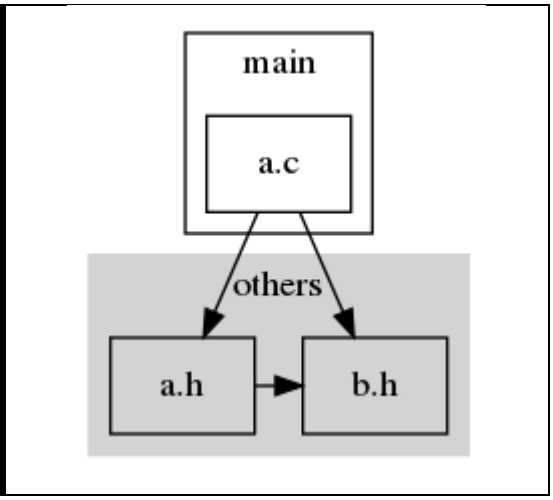
Beispiel File (**`dot -Tpng sample.dot > sample.png`**)

```
digraph G {
   node [shape=box]
   A [label="a.c"];
   B [label="a.h"];
   C [label="b.h"];

   subgraph cluster_c0 {
      label="main"; color=black;
      A;
   }

   subgraph cluster_c1 {
      label="others"; style=filled; color=lightgrey;
      { B; C; rank=same; }
   }

   A -> B;
   A -> C;
   B -> C;
}
```



6.4 `png` File

Das **`png`** Format ist ein verlustfrei komprimiertes Raster Graphik Format. Es wird oft in Web Pages verwendet.

6.5 Verwendete zusätzliche Sprach Elemente

Sprach Element	Beschreibung

`typedef struct {`
`  const char *name;`
`} name_t;`
	Das **`struct`** Feld **`const char *name`** steht hier für den Beginn eines Character Strings (d.h. ein Array von Character Elementen)
	Das Feld kann als Argument in Funktionen wie **`printf`** oder **`strcmp`**, etc. verwendet werden.
	Zugriff auf einzelne Elemente des Strings erfolgt via die Array-Index Syntax: **`name[index]`**.
	Für dieses Praktikum sollte diese Erklärung genügen. Weitergehende Details werden in Folge-Vorlesungen und Praktika behandelt.

`typedef struct {`
`  size_t n_names;`
`  name_t *names;`
`} container_t;`
	Das **`struct`** Feld **`name_t *names`** steht hier für den Beginn eines Arrays von **`name_t`** Elementen.
	Zugriff auf einzelne Elemente erfolgt via die Array-Index Syntax: **`names[index]`**.
	Mehr dazu in Folge-Vorlesungen und Praktika.