Kapitel 12

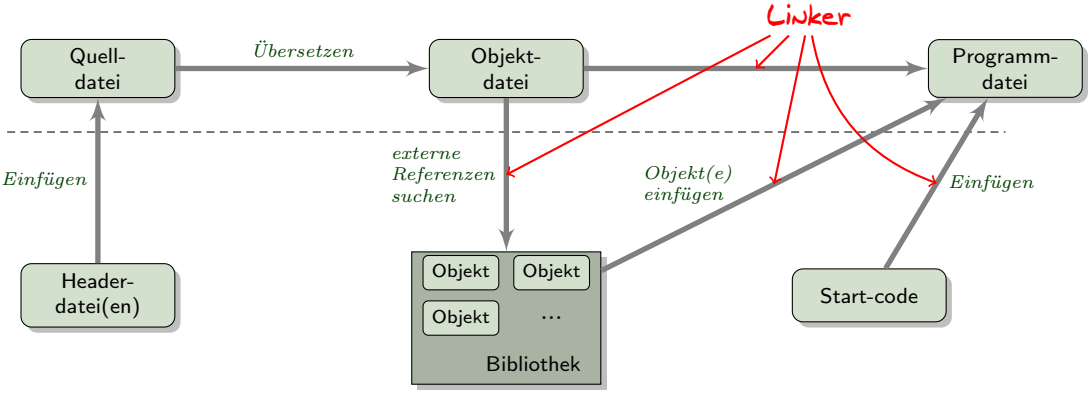
Linker und Bibliotheken

Ich habe mir das Paradies immer als eine Art Bibliothek vorgestellt

(Jorge Luis Borges)

12.1 Wiederholung: Module und Linker

- Im letzten Kapitel haben wir eine Anwendung in verschiedene Module zerlegt, die jeweils ein eigenes C-Quellfile hatten
- Erst der Linker hat die einzelnen Module zusammengefügt
- Wir wollen uns jetzt den Linker näher betrachten



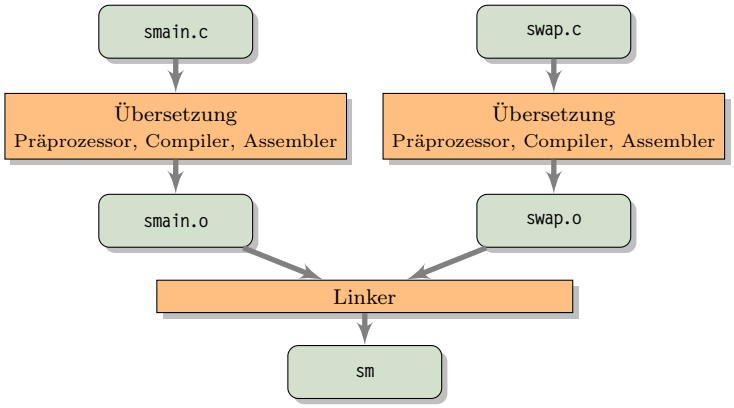
- Als Beispiel betrachten wir folgenden Code einer allgemeinen Tausch-Funktion:

```c
#include <stddef.h>          // for size_t
extern void* malloc(size_t);
extern void free(void*);

static void bytecopy(char *dest, char *src, size_t size){
  while(size > 0){
    *dest++ = *src++;
    size--;
  }
}

int swap(void *first, void *second, size_t size){
  void *tmp = malloc(size);
  if (!tmp) return 0;
  bytecopy(tmp,first,size);
  bytecopy(first,second,size);
  bytecopy(second,tmp,size);
  free(tmp);
  return 1;
}
```

- Der Code kann mit verschiedenen Typen genutzt werden, wie folgendes Programm demonstriert:

```c
#include <stdio.h>
int swap(void *first, void *second, size_t size);

char* str1="Hello";
char* str2="World";
int a,b;

int main(){
  a=42;
  b=23;
  printf("1st string: %s, 2nd string: %s\n",str1,str2);
  swap(&str1,&str2,sizeof(str1));
  printf("1st string: %s, 2nd string: %s\n",str1,str2);
  printf("1st integer: %d, 2nd integer: %d\n",a,b);
  swap(&a,&b,sizeof(a));
  printf("1st integer: %d, 2nd integer: %d\n",a,b);
  return 0;
}
```

```
> cc -std=c99 -Wall euclid.c swap.c smain.c -o sm
```




```
> ./sm
1st string: Hello, 2nd string: World
1st string: World, 2nd string: Hello
1st integer: 42, 2nd integer: 23
1st integer: 23, 2nd integer: 42
>
```

Anmerkung

Der hier benutzte Code ist nicht unbedingt optimal. Z.B. entspricht die statische Funktion `bytecopy()` der Funktion `memcpy()` aus der Standardbibliothek. Mit Hilfe von Präprozessor-Makros und ggf. von Generics (Schlüsselwort `_Generic` ab C11) könnte man eine solche Funktion noch eleganter (ohne Übergabe der Größe) schreiben. In C++ werden dafür **Templates** benutzt.

Warum Linker?

- **Modularität**
  - Programme können als Ansammlung kleinerer Quelldateien geschrieben werden, anstatt als eine monolithische Datei
  - Es können Bibliotheken von häufig genutzten Funktionen geschrieben werden (mehr in Abschnitt 12.3), z.B. Standardbibliothek, Mathematik-Bibliothek, etc.

- **Effizienz**
  - Zeit: Es müssen nicht immer der komplette Quellcode übersetzt werden
  - Platz: Bibliotheken können eine Sammlung unterschiedlicher Funktionen enthalten, das fertige Programm enthält aber nur die Funktionen, die es benötigt

12.2 Linker

Was macht der Linker?

- **1. Auflösung von Symbolen**
  - Programme definierten und referenzieren **Symbole**, d.h. Namen von Variablen und Funktionen

`int swap() {...}`	definiert Symbol `swap`
`swap(&a,&b,sizeof(a));`	referenziert Symbole `swap`, `a` und `b`
`int a;`	definiert Symbol `a`

  - Symboldefinitionen werden vom Compiler in einer **Symboltabelle** gespeichert
    - Die Symboltabelle ist ein Array einer speziellen Datenstruktur (`struct`), die für jedes Symbol den Namen, den Typ, die Größe und den Ort enthält
  - Der Linker verknüpft jede Symbolreferenz mit genau einer Symboldefinition
- **2. Verschiebung** (*Relocation*)
  - Fügt separate Code- und Datenabschnitte in gemeinsamen Abschnitten (*Sections*) zusammen
  - Verschiebe Symbole von ihrer relativen Position im Object-File (.o) auf ihre finale absolute Speicherposition im ausführbaren File
  - Korrigiert alle Referenzen auf diese Symbole, so dass diese auf die neuen Positionen zeigen

Linkersymbole

Es gibt drei Typen von Linkersymbolen

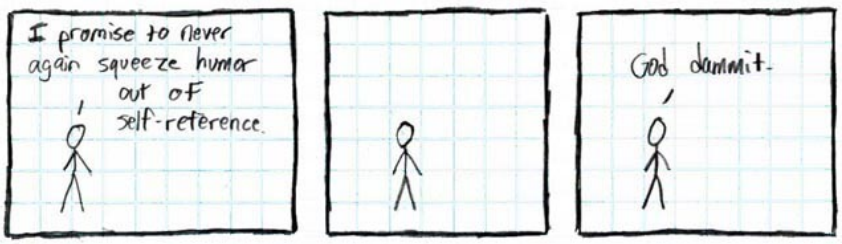
- **Globale Symbole**
  - Symbole, die in einem Modul definiert sind und durch andere Module referenziert werden können (z.B. nicht-statische C-Funktionen oder nicht-statische globale Variablen)
- **Externe Symbole**

  - Globale Symbole die in einem Modul referenziert werden, aber in einem anderen Modul definiert werden

- **Lokale Symbole**
  - Symbole, die in einem Modul definiert und ausschließlich auch dort referenziert werden, z.B. Funktionen und globale Variablen die das `static`-Attribut haben

Merke

Lokale Variablen erzeugen **keine** Linkersymbole, auch keine lokalen.



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
http://xkcd.com/33/

```c
#include <stddef.h>          // for size_t
extern void* malloc(size_t);
extern void free(void*);

static void bytecopy(char *dest, char *src, size_t size){
  while(size > 0){
    *dest++ = *src++;
    size--;
  }
}

int swap(void *first, void *second, size_t size){
  void *tmp = malloc(size);
  if (!tmp) return 0;
  bytecopy(tmp,first,size);
  bytecopy(first,second,size);
  bytecopy(second,tmp,size);
  free(tmp);
  return 1;
}
```

extern

lokal

global

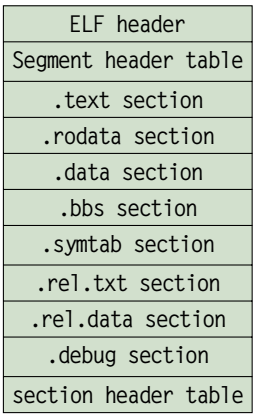
Der Linker weiß nichts von temp

- Mit Hilfe von Programmen wie `readelf`, `objdump` oder `nm` können die Symboletabellen ausgelesen werden

```
> nm -fs swap.o
Symbols from swap.o:

Name                  Value           Class        Type         Size             Line  Section

bytecopy             |0000000000000000|   t  |             FUNC|000000000000003d|     |.text
free                 |                |   U  |           NOTYPE|                |     |*UND*
malloc               |                |   U  |           NOTYPE|                |     |*UND*
swap                 |000000000000003d|   T  |             FUNC|000000000000008a|     |.text
> nm -m smain.o
Symbols from smain2.o:

Name                  Value           Class        Type         Size             Line  Section

a                    |0000000000000004|   C  |           OBJECT|0000000000000004|     |*COM*
b                    |0000000000000004|   C  |           OBJECT|0000000000000004|     |*COM*
main                 |0000000000000000|   T  |             FUNC|00000000000000c1|     |.text
printf               |                |   U  |           NOTYPE|                |     |*UND*
str1                 |0000000000000000|   D  |           OBJECT|0000000000000008|     |.data
str2                 |0000000000000008|   D  |           OBJECT|0000000000000008|     |.data
swap                 |                |   U  |           NOTYPE|                |     |*UND*
>
```

Executable and Linkable Format (ELF)

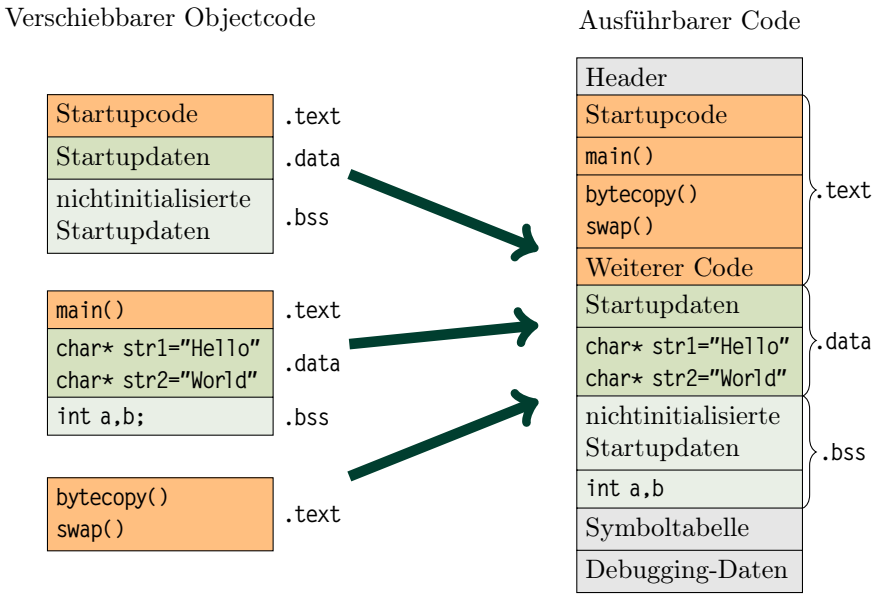
- Es gibt verschiedene Datenformate für Objektdateien
  - Z.B.: a.out, COFF, Mach-O, PE
  - Betrachten hier ELF
- **ELF** – Executable and Linkable Format
- Ursprünglich von AT&T für System V entwickelt
- Heute bei vielen Betriebssystemen verwendet, u.a. BSD und Linux
- ELF wird für verschiedene Arten von Binärdateien eingesetzt:
  - Verschiebbare Objectfiles (`.o`)
  - Ausführbare Programmfiles
  - Geteilte Objectfiles (`.so`)



- **ELF Header**: Wortgröße, Byteordnung, Dateityp, Plattform, etc.
- **Segment-Header-Tabelle**:
  - nötig für ausführbare Dateien
  - Seitengröße, Sections, Segmentgröße
- **.text**: (Maschinen-)Code
- **.rodata**: Nur-lese-Daten (Sprungtabellen, ...)
- **.data**: initialisierte globale Variablen
- **.bbs**: uninitialisierte globale Variablen[1]
  - wird im Speicher mit 0 initialisiert
- **.symtab**: Symboltabelle
- **.rel.text**: Verschiebeinformation für Code
  - Adressen von Befehlen, die beim Verschieben im Speicher modifiziert werden müssen
  - Instruktionen für die Verschiebung
- **.rel.data**: Verschiebeinformation für Variablen
- **.debug**: Information für symbolisches Debugging (`cc -g`)
- **Section header table**: Offset und Größe für jeden Abschnitt

[1] `.bbs` steht für *block started by symbol*, ein Pseudobefehl im Assembler für IBM 704

Verschiebung von Code und Daten



Das Problem globaler Variablen

- Beim Linken werden nur die Symbole betrachtet, es wird ***keine*** Typüberprüfung durchgeführt
- Betrachten folgenden Code:

```c
#include <stdio.h>

int a;
int b;

void printab1(){
  printf("1: a=%d, b=%d\n",
         a,b);
}

void setab1(){
  a=42;
  b=23;
}
```

```c
#include <stdio.h>

double a;
int b;

void printab2(){
  printf("2: a=%0.0f, b=%d\n",
         a,b);
}

void setab2(){
  a=42.0;
  b=23;
}
```

- Die gemeinsame Verwendung führt jedoch zu Problemen:

```c
void printab1();
void printab2();
void setab1();
void setab2();

int main(){
  setab1();
  printab1();
  setab2();
  printab2();
  printab1();
  return 0;
}
```

```
> cc -Wall -Wextra -c -o global1.o global1.c
> cc -Wall -Wextra -c -o global2.o global2.c
> cc -Wall -Wextra -c -o global.o global.c
> cc global1.o global2.o global.o -o global
>
> ./global
1: a=42, b=23
2: a=42, b=23>
1: a=0, b=23
>
      Uuups!
```

- Daher sollten folgenden Regeln beachtet werden:

Wenn es möglich ist, **vermeiden** Sie globale Variablen!

- Andernfalls:
  - Nutzen Sie wenn möglich `static`!
  - Initialisieren Sie globale Variablen bei der Definition!
  - Nutzen Sie `extern` für externe Variablen!

12.3 Bibliotheken

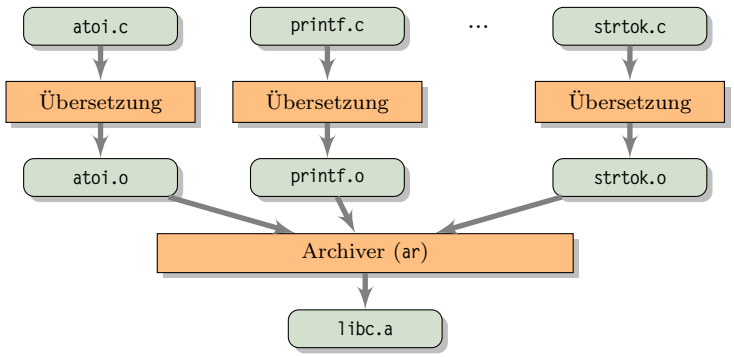
- Wie kann nachnutzbarer Code „paketiert“ werden?
  - Ein- und Ausgabe, Mathe-Funktionen, ...
- Mit den bisherigen Tool gibt es zwei Möglichkeiten:

1. Alle Funktionen kommen in ein großes Quellfile und landen in einem großen Objectfile
   - ➡ Programmierer linken dieses große Objectfile zu ihren Programmen
   - ➡ Zeit- und platzineffizient
2. Jede Funktion kommt in ein eigenes Quellfile
   - ➡ Programmierer linken explizit ausgesuchte Objectfiles zu ihren Programmen
   - ➡ Ist zwar effizienter, aber sehr aufwendig für den Programmierer

Lösung

- **Lösung**: Nutzung von **Archiven**, auch bekannt als (statische) **Bibliotheken**
- Archive ist eine einzelne Datei, die eine Sammlung von mehreren Objectfiles enthält, zusammen mit einem Index
- Linker wird so erweitert, dass er bei der Auflösung von unaufgelösten Referenzen in einem oder mehreren Archiven nachschaut
- Wenn ein Symbol in einem Archive gefunden wird, kopiert der Linker das entsprechende Objectfile und linkt es zum Programm
- Zum Erzeugen und Managen von Archiven gibt es ein eigenes Tool: `ar`

Erzeugung eines Archivs



```
> ar rs libc.a atoi.o printf.o ... strtok.o
```

- Der Archiver erlaubt inkrementelle Updates
- Das geänderte Quellfile wird compiliert und im Archiv ersetzt

Gemeinsam genutzte Bibliotheken

- Jedes System enthält eine Reihe Bibliotheken
- Der C-Standard verlangt mindestens zwei:
  - `libc.a` oder `libgcc.a`[2]: Standardbibliothek, Ein- und Ausgabe, Speicherverwaltung, Signalbehandlung, ...
  - `libm.a`: Gleitkomma-Mathematik
- Auf dem Server für Bonusaufgaben besteht...
  - `libgcc.a` aus 221 Objectdateien, die 3 MByte einnehmen
  - `libm.a` aus 460 Objectdateien, die 2,1 MByte einnehmen
- Die Standardbibliothek (und Startup-Code) werden (solange nicht die Optionen „`-nodefaultlibs`“ bzw. „`-nostartfiles`“ angegeben werden) automatisch gelinkt

Nutzung von (statischen) Bibliotheken

- Der Linker kann beliebige Archive (auch eigene) nutzen
- Optionen:
  - `-L`*`pfad`*: Linker sucht in *pfad* nach Bibliotheken; die Option kann mehrmals angegeben werden
  - `-l`*`name`*: Linker durchsucht die Archiv-Datei `lib`*`name`*`.a` bei der Symbolauflösung
- **Beispiel**: der folgende Aufruf sagt dem Linker, dass die Gleitkomma-Mathebibliothek eingebunden werden soll:

```
> cc -o myprog mycode.c -lm
```

- Eigene Bibliotheken machen werden genauso behandelt:

```
> cc -c swap.c
> ar rs libsw.a swap.o
> ar: creating libsw.a
```

[2] GNU Compiler-Suite

```
> cc -o sm smain.c -L. -lsw
```

- Bei einigen Compilern (z.B. gcc) spielt die Reihenfolge der Optionen eine Rolle, da die Symbolauflösung strikt von links nach rechts erfolgt:

```
> gcc -o sm smain.c -L. -lsw
/tmp/cc3DPrOG.o: In function ‘main’:
smain2.c:(.text+0x48): undefined reference to ‘swap’
smain2.c:(.text+0x99): undefined reference to ‘swap’
collect2: error: ld returned 1 exit status
```

Dynamische Bibliotheken

- Statische Bibliotheken haben einige Nachteile:
  - Vervielfachung des Codes im Massenspeicher (die Standardbibliothek wird von [nahezu] jedem Programm genutzt)
  - Vervielfachung des Codes im Hauptspeicher
  - Kleine Fehlerbehebungen im Bibliothekscode erfordern ein explizites Neulinken **jedes** Programmes
- **Lösung**: **Dynamische Bibliotheken** (*shared libraries*)
  - Objectfiles werden erst zur Ladezeit oder Laufzeit geladen und gelinkt
  - Code wird nur einmal in Speicher geladen und mehrmals genutzt (shared/virtueller Speicher ➡ Vorlesung Betriebssysteme)
  - Standard in modernen Systemen u.a für C-Standardbibliothek
    - Linux: `libc.so`, Windows: `mscrt.dll`, macOS: `libSystem.dylib` bzw. `libc.dylib`
- Erzeugung einer dynamischen Bibliothek

```
> gcc -c -Wall -Werror -fpic swap.c
> gcc -shared -o libsw.so swap.o
```

- Bedeutung der Optionen:
  - `-fpic`: Erzeugung von positionsunabhängigen Code (*position independent code*) ➡ keine „Umrechnung“ beim Laden/Linken nötig
  - `-shared`: Erzeugt eine unabhängige Bibliothek
- Bibliothek einbinden:

```
> gcc smain2.c -o sm  -L. -lsw
```

- *Achtung!* Ein dynamisch gelinktes Programm kann nicht „einfach so“ ausgeführt werden:

```
> ./sm
./sm: error while loading shared libraries: libsw.so: cannot open shared object file: No
 such file or directory
```

- Der Lader sucht nach dynamischen Bibliotheken in vordefinierten Verzeichnissen, u.a. die in der Systemvariable `LD_LIBRARY_PATH` Pfaden

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
> .sm
1st string: Hello, 2nd string: World
1st string: World, 2nd string: Hello
1st integer: 42, 2nd integer: 23
1st integer: 23, 2nd integer: 42
```

12.4 Einsatz von Bibliotheken

- Wenn Funktionen aus fremden Bibliotheken genutzt werden sollen, sollte man immer genau wissen, was der **Effekt** der Funktion ist

  ➡ Man kann meist nicht „mal schnell“ im Quellcode nachsehen

- Manche Bibliotheken sind lediglich eine Sammlung thematisch zusammengehöriger Funktionen

  – Beispiel: `libmath`

- Andere Bibliotheken stellen ein komplettes **Framework** dar, bei denen die Funktionen nur in einem Zusammenspiel sinnvoll verwendet werden können

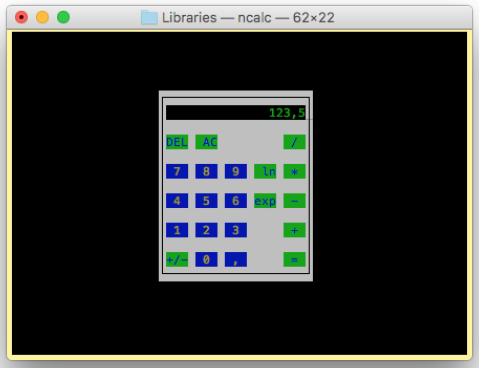
  – Beispiel: viele GUI-Frameworks, z.B. `GTK+` oder `Qt`

  – In diesem Fall hilft mitunter die Dokumentationen zu den einzelnen Funktionen nur wenig

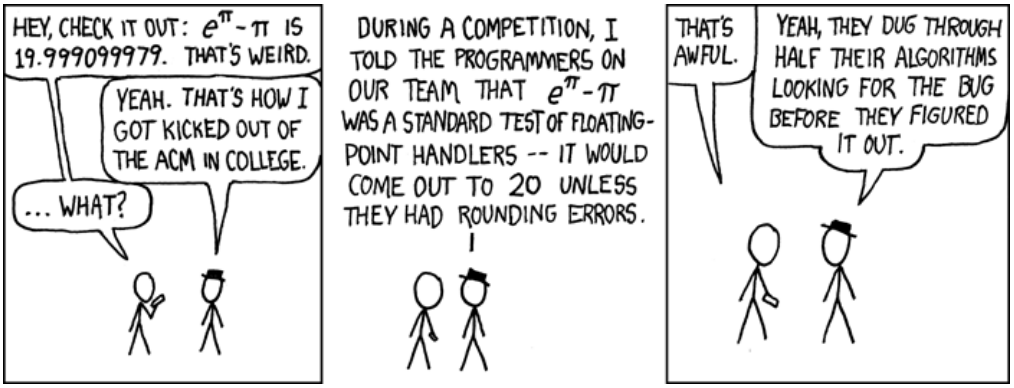
  – Häufig gibt es dann Dokumentationen für die gesamte Bibliothek oder Tutorials, HowTos etc.

Fallstudie: Taschenrechner

- Wir betrachten als Beispiel eine Anwendung, die `ncurses`- und die `math`-Bibliothek nutzt: einen „Taschenrechner“
- Der Rechner beherrscht die Grundrechenarten, zusätzlich noch logarithmieren die Exponentialfunktion
- Die Anwendung läuft zwar im Terminal, aber hat die Anmutung eines „richtigen“ Taschenrechners:



- Es geht nicht um die komplette Programmentwicklung, wir diskutieren nur einige Aspekte der Bibliotheksnutzung



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
http://xkcd.com/217/

Ncurses

- Es wird die Bibliothek `ncurses` genutzt
- Die `ncurses`-Bibliothek nutzt die Möglichkeiten der ANSI-Terminals
  - Weiterentwicklung der `curses`-Bibliothek
  - existiert auf für die meisten UNIXe, macOS, Windows, DOS
- `ncurses` stellt (je nach Version) zwischen ca. 800-1000(!) Funktionen zur Verfügung
- Viele Funktionen existieren in verschiedenen Versionen, Namensprefixe geben Auskunft über verwendete Parameter
  - Z.B. hat eine Funktion, die den Prefix „`w`“, immer ein Parameter von Typ `WINDOW*`, ein Prefix „mv“ bewegt den Cursor, etc.

Ncurses – Konzepte

- Kennt ein Konzept von **Fenstern**, die jedoch nicht überlappend sein dürfen ➡ Erweiterung mit `panel`-Bibliothek
- Ausgaben werden erst wirksam, wenn `wrefresh()` für das Fenster aufgerufen hat – häufige Quelle von Fehlern
- Benötigt umfangreiche **Initialisierung**
  - Echo bei Eingabe, Terminalmode, Nutzung von Funktionstasten, etc. ➡ ebenfalls Quelle von Fehlern
- **Farben**/Attribute
  - Farbattribute müssen mit Kombination von Vordergrund- und Hintergrundfarbe initialisiert werden
  - Nutzung über (fenster-)globale Umschaltung **vor** der eingentlichen Ausgabe

```
const int mycolor=1;
init_pair(mycolor,COLOR_GREEN,COLOR_BLACK);
wattrset(win, mycolor);
```

Erkenntnisse

- In dieser Anwendung werden für die GUI etwa doppelt Codezeilen wie die eigentliche Rechenlogik (gesamtes Programm im Anhang des Skripts)
- Bei Frameworks spielen viele Komponenten zusammen, so dass mitunter das Erlernen der Nutzung dem einer (manchmal nicht so) kleinen Programmiersprache nahekommt
- Häufig werden bestimmte Programmieransätze und -stile diktiert (z.B. Ereignisschleifen)

Aufgaben

Aufgabe 12.1

Ein Kryptogramm oder Alphametik ist ein Rätsel, das eine mathematische Gleichung oder ein Gleichungssystem unbekannter Zahlen bildet, deren Ziffern durch Buchstaben ersetzt wurden.

Lösen Sie folgende Kryptogramme:

```
  BLAU      CROSS      PLANET
 +LILA     +ROADS     + EARTH
 -----     ------     -------
 BRAUN     DANGER     ROTATES
```

Aufgabe 12.2

Schreiben Sie ein Programm, das Kryptogramme löst!

Aufgabe 12.3

Schauen Sie sich den Code für das Taschenrechnerprogramm an und erweitern Sie es so, dass der Rechner um Klammern ergänzt wird (und korrekte Klammerrechnung beherrscht)!