Tutorium 1

Setup und C-Basics

Vorab – Wer bin ich?

- Alexander Herbrich (aber, nennt mich ruhig Alex)
- 6. Sem. Informatik B.Sc.
- Fragen? Organisatorisches?
  - Mail at: **herbrich@tu-berlin.de**
  - Im Forum, falls Andere von der Antwort profitieren könnten
- Lösungen?
  - Die offiziellen Musterlösungen werden im ISIS-Kurs hochgeladen
  - Meine Notizen/Resourcen/Empfehlungen findet ihr in diesem Repo (GitHub @aherbrich)

Vorab - Organisatorisches

- **Anmeldung !!!**
  - Über Moses **bis 17.05.24** möglich
    - Abmeldung bis 17.05.24 möglich!
  - Wählt vorerst einen der zwei Termine; möglich sich **bis zum 15.07.24** Termin umzuentscheiden
- Benotung:
  - 50PP Klausur
  - 50PP Hausaufgaben, wovon
    - 2x 5PP Theorieaufgaben
    - 1x 10PP (kleine) Praxisaufgabe
    - 1x 30PP (große) Praxisaufgabe

Vorab – Ablauf

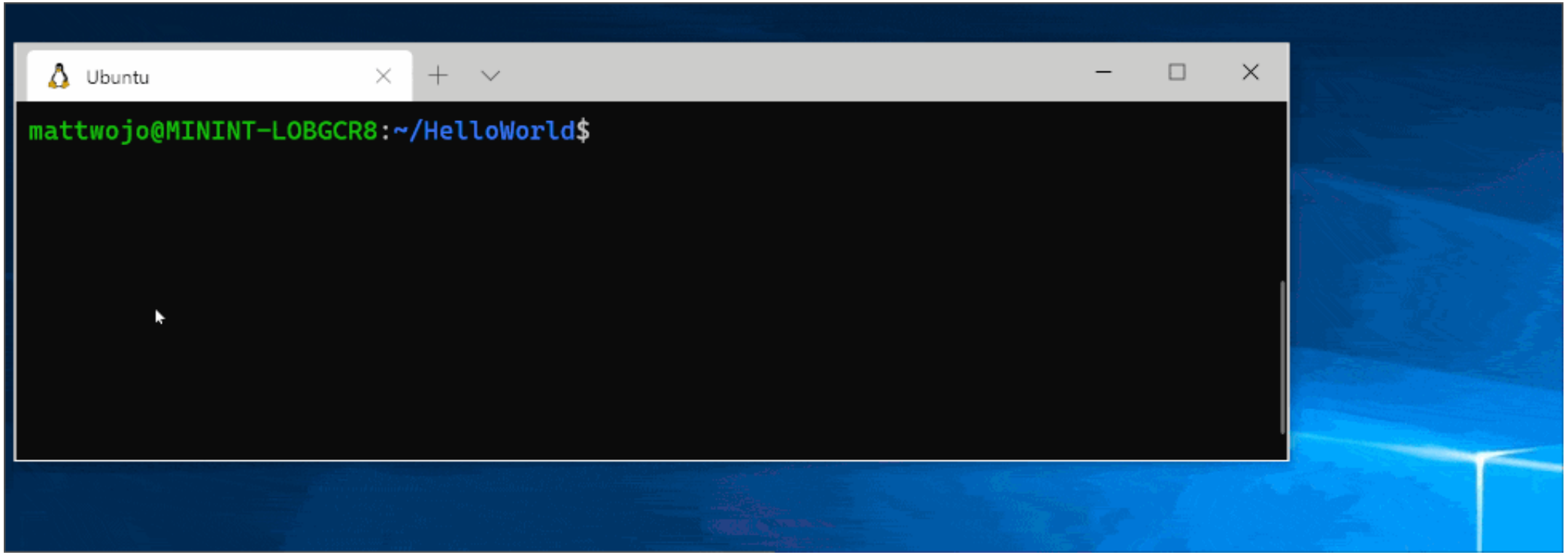
- Ablauf *Generell*
  - Die (meisten) Übungsblätter sind für 2 Tutorien konzipiert.
  - In den Tutorien besprechen wir die **Tafelübungen**
  - Falls wir Zeit haben, besprechen wir auch *teilweise die Selbststudiums-Aufgaben*
- Ablauf ***Heute***
  - Sehr entspanntes Tutorium; wir haben viel Zeit!
  - Nur *zwei* Themen:
    - Einrichtung eurer Entwicklungsumgebung
    - C-Basics / C Common Pitfalls

1. Setup

Einrichtung eurer Entwicklungsumgebung

Empfehlungen

- Benutzt ein Linux (oder alternativ) MacOS-basierten System
- Fall ihr Windows benutzt
  - Benutzt WSL um Linux auf eurem Rechner zu nutzen (siehe hier und hier)
  - Richtet euch VSCode für WSL ein (siehe hier)



Empfehlungen

- Werdet *comfortable* mit eurer Shell
  - Installiert euch die Shell eurer Wahl (meine Empfehlung: *zsh*)
  - Installiert euch Plugins, für styling, sytnax-highlighting, auto-complete, history etc. (wie z.B hier für die Shell *zsh* beschrieben ist)
  - Lernt die wichtigsten Shell Befehle
    - Alles, was ihr mit grafischen Oberflächen macht, könnt ihr auch mit der Shell
    - Grafische Oberflächen sind einfach nur Wrapper für Befehle, die im Terminal ausgeführt werden
    - Supergutes Video für Shell 101 hier…

2. C - Refresher

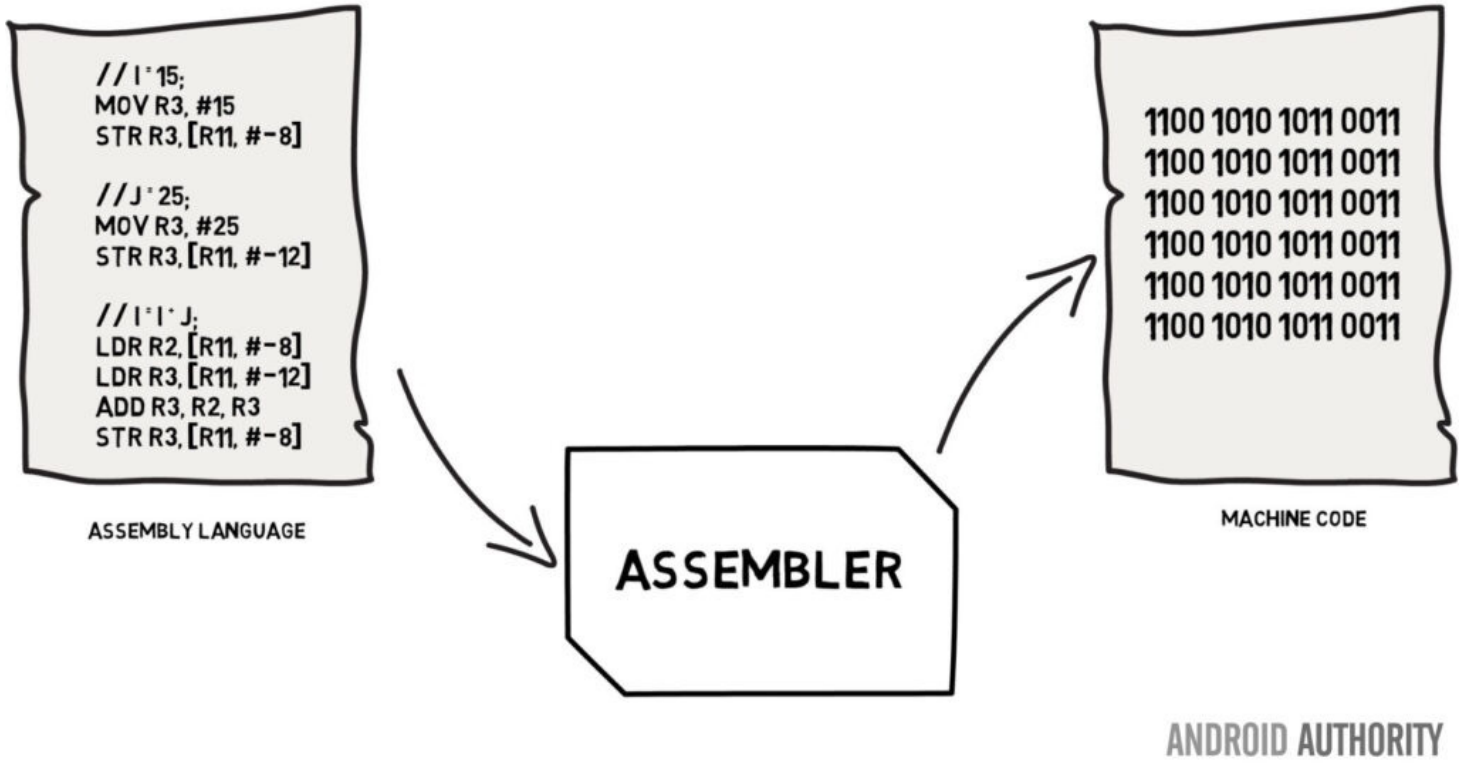
C-Basics / C Common Pitfalls

Warum C?

- Der CPU versteht nur Abfolgen von Nullen und Einsen einer bestimmen Länge
  - Sogenannte 32- bzw. 64-bit lange Wörter
  - Jedes Wort entspricht einem Befehl
- **Problem:** Äußerst schwer zu lesen, geschweige denn zu schreiben!
- + wir müssten für jeden CPU, den CPU-eigenen Befehlssatz lernen…

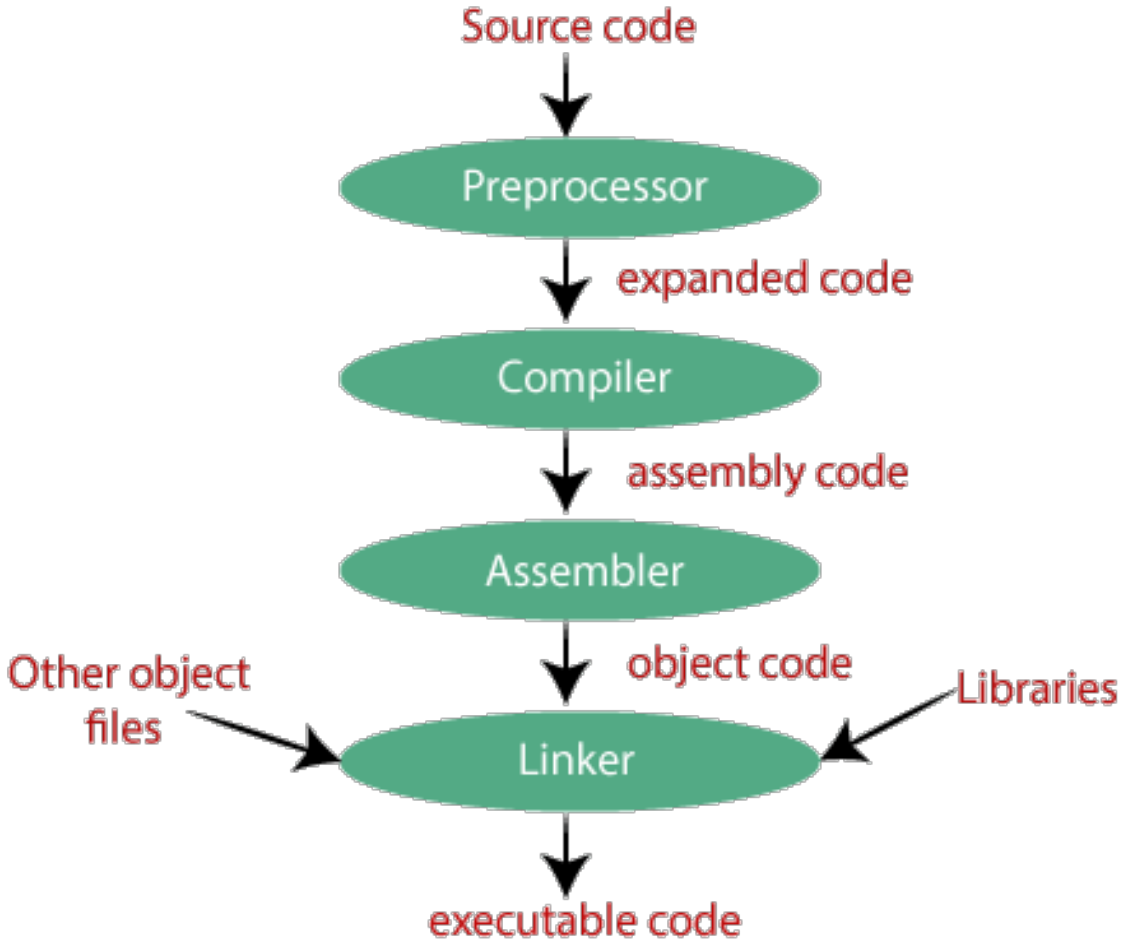
Warum C?

- **„Lösung“: Assembly**
  - Befehle werden nicht mehr als 32 (bzw. 64) Nullen und Einsen geschrieben
  - für jeden Befehl existiert nun eine Abkürzung (=Assembly)
  - Assembly wird von einem **Assembler** zu Maschinencode *„assembliert“*
- Gut, aber es geht noch besser?



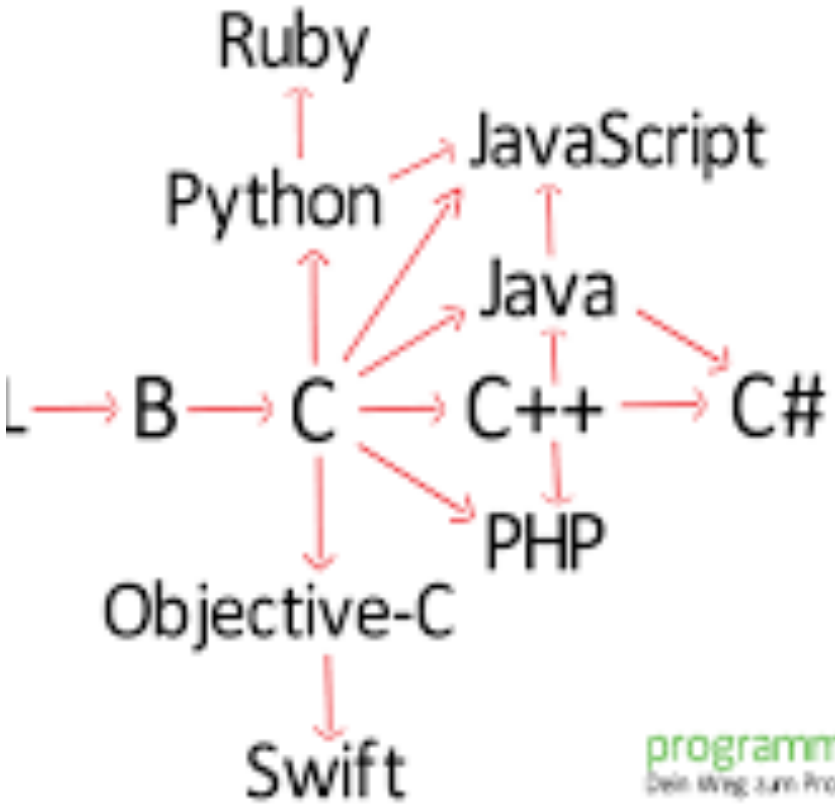
Warum C?

- Ja, es geht besser!
- Mit Hilfe von C können wir *„Menschenlesbaren“* Code schreiben, vereinfacht:
  - C-Datei wird durch **Compiler** zu Assembly kompiliert
  - Assembly wird zu Object-Code (vereinfacht: *quasi* Maschinencode) assembliert
  - Falls wir mehrere Dateien hatten: Die einzelnen Object-Code Dateien werden zu einem großem Machine-Code-File, dem *“Executable”* **gelinkt**
- ***Achtung*** *- Verwirrung: Heutige „Compiler“ beinhalten meist den (eigentlichen) Compiler UND Linker in Einem*
- Gut, aber geht es noch besser?



Warum C?

- Geht es besser als C? Jein!
- In Systemprogrammierung beschäftigen wir uns mit Betriebssystemen
  - Betriebssysteme sind im Kern *„einfach nur“* **komplexe Speichermanipulation**
- C ist eine **maschinennahe** Sprache, d.h.
  - Relativ geringe Abstraktion von was/wie der Rechner arbeitet, und wie wir Code schreiben
  - Kontrolle über Speicher aufs Bit genau möglich
  - Daher, **de-facto Standard** für die BS-Entwicklung
- Sprachen wie Python oder JavaScript weisen “zu hohe” Abstraktion auf



Warum C?

- **Motivation außerhalb des Kurses:**
  - C zu lernen, **beschleunigt** das Erlernen jeder anderen Programmiersprache
  - Durch das Erlernen von C erhält man Einblicke, **welche Vorteile andere Sprachen bieten** und welche Laufzeitkosten damit einhergehen
  - Das Verständnis von C ermöglicht ein **tieferes Verständnis** dafür, wie Computer tatsächlich arbeiten

C-Basics

- Wenn ihr noch NIE programmiert habt, dann schaut in den (von unserem Tutor Timo) vorbereiteten Video-Kurs vorbei!
- Schaut in den Kurs rein, wenn ihr eines der Folgenden nicht kennt:
  - Variablen und Datentypen
  - Structs (Zusammengesetzte Datentypen)
  - Arrays
  - Bedingte Codeausführung
  - Schleifen
  - Adressen und Pointer
  - Funktionen
  - Dynamsiche Speicherverwaltung
  - Chars und Strings
  - Bitweise Operationen
- Ihr findet den Kurs auch auf ISIS verlinkt!

C-Basics

- Wenn ihr noch NIE programmiert habt, dann schaut in den (von unserem Tutor Timo) vorbereiteten Video-Kurs vorbei!
- Schaut in den Kurs rein, wenn ihr eines der Folgenden nicht kennt:
  - Variablen und Datentypen
  - Structs (Zusammengesetzte Datentypen)
  - Arrays
  - Bedingte Codeausführung
  - Schleifen
  - **Adressen und Pointer**
  - **Funktionen**
  - **Dynamsiche Speicherverwaltung**
  - Chars und Strings
  - Bitweise Operationen

<- Focus heute

- Ihr findet den Kurs auch auf ISIS verlinkt!

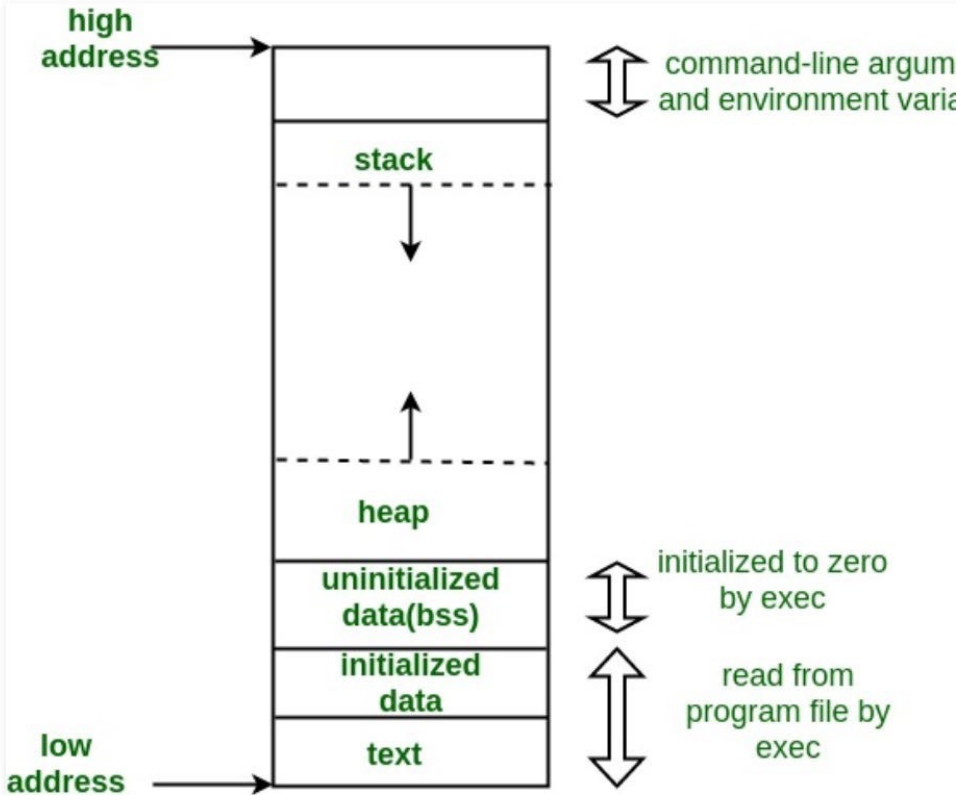
C – Aufgabe 1.2 a)

Legen Sie mit dynamischem Speicher eine Variable an. Geben Sie die Speicheradresse der Variable sowohl im üblichen Hex-Format, als auch als Dezimalzahl aus.

C – Aufgabe 1.2 a)

- **Statischer Speicher**
  - Zur Kompilierzeit zugewiesen
  - Feste Größe
  - Keine explizite Freigabe erforderlich
  - Schnellerer Zugriff

- **Dynamischer Speicher:**
  - Während Laufzeit zugewiesen
  - Flexibel, kann wachsen/schrumpfen
  - Mögliche Fragmentierung
  - Erfordert **explizite Freigabe** und **langsamerer Zugriff**
    - Wenn möglich statischen Speicher benutzen!



C – Aufgabe 1.2 a)

```c
int main() {



}
```

C – Aufgabe 1.2 a)

```c
int main() {
	int *meinezahl = malloc(sizeof(int));




}
```

C – Aufgabe 1.2 a)

```c
#include <stdlib.h>

int main() {
	int *meinezahl = malloc(sizeof(int));




}
```

C – Aufgabe 1.2 a)

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
	int *meinezahl = malloc(sizeof(int));
	printf("%p\n", meinezahl);



}
```

C – Aufgabe 1.2 a)

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
	int *meinezahl = malloc(sizeof(int));
	printf("%p\n", meinezahl);

	unsigned long adresse = (unsigned long) meinezahl;
	printf("%lu\n", adresse);
}
```

C – Aufgabe 1.2 b)

Deklarieren Sie auf normalem (statischem) Weg eine Variable. Geben Sie wieder die Speicheradresse im Hex-Format und als Dezimalzahl aus.

C – Aufgabe 1.2 b)

```c
#include <stdio.h>

int main() {

	printf("%p\n", ...


	printf("%lu\n", ...
}
```

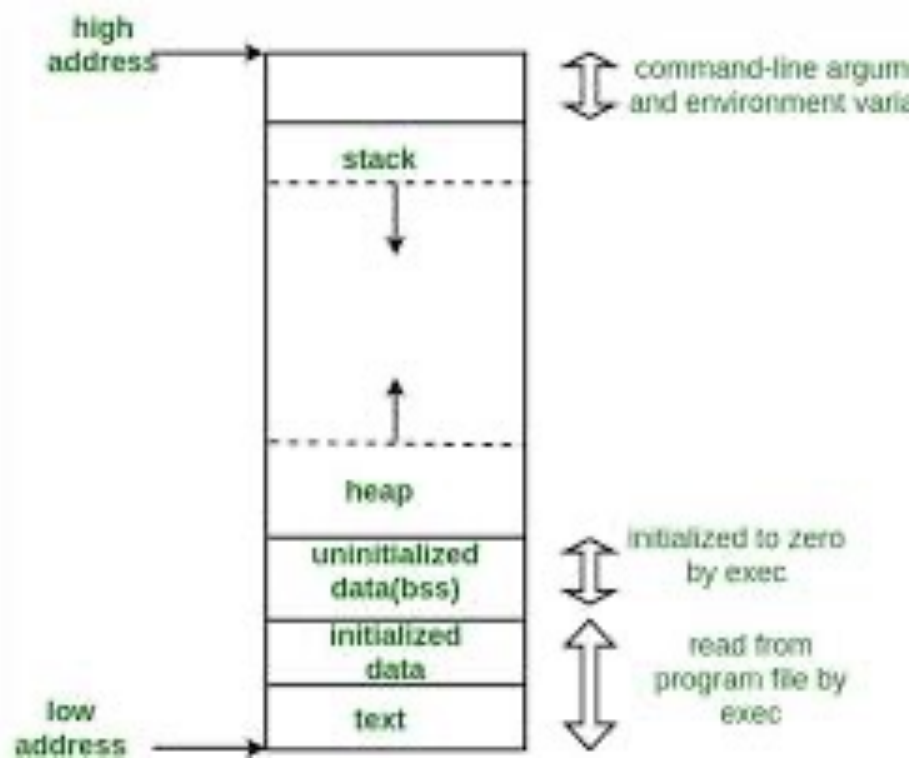
C – Aufgabe 1.2 b)

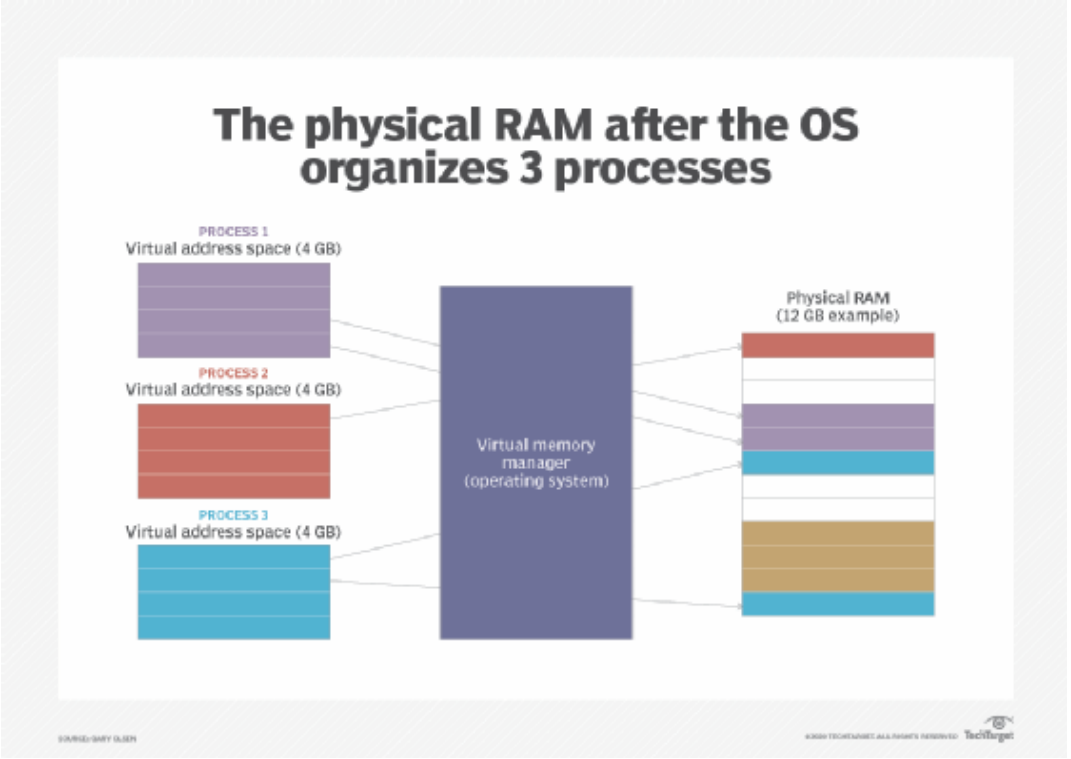
```c
#include <stdio.h>

int main() {
	int meinezahl;
	printf("%p\n", &meinezahl);

	unsigned long adresse = (unsigned long) &meinezahl;
	printf("%lu\n", adresse);
}
```

C – Aufgabe 1.2 c)

Erklären Sie, inwiefern und warum sich die Adressen unterscheiden. Sind die Adressen bei jeder Programmausführung gleich?

C – Aufgabe 1.2 c)





C – Aufgabe 1.2 d)

Betrachtet den folgenden Code. Welche Zahl wird gedruckt?

C – Aufgabe 1.2 d)

```c
#include <stdio.h>

void upgrade(int meinezahl) {
	meinezahl = 18;
}

int main() {
	int meinezahl = 5;
	upgrade(meinezahl);
	printf("%d\n", meinezahl);
}
```

C – Aufgabe 1.2 e)

Betrachtet den folgenden Code. Welche Zahl wird gedruckt?

C – Aufgabe 1.2 e)

```c
#include <stdio.h>
#include <stdlib.h>

void upgrade(int *meinezahl) {
	*meinezahl = 18;
}

int main() {
	int *meinezahl = malloc(sizeof(int));
	*meinezahl = 5;
	upgrade(meinezahl);
	printf("%d\n", *meinezahl);
}
```

C – Aufgabe 1.2 f)

Betrachtet den folgenden Code. Welche Zahl wird gedruckt?

C – Aufgabe 1.2 f)

```c
#include <stdio.h>
#include <stdlib.h>

void upgrade(int *meinezahl) {
	meinezahl = 18;
}

int main() {
	int *meinezahl = malloc(sizeof(int));
	*meinezahl = 5;
	upgrade(meinezahl);
	printf("%d\n", *meinezahl);
}
```

C – Aufgabe 1.2 g)

Betrachtet den folgenden Code… Was sind t1, t2 und t3?

C – Aufgabe 1.2 g)

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct mystruct {
        int zahl1;
        int zahl2;
        int zahl3;
} mystruct;

int main() {
        mystruct test1;
        mystruct *test2 = malloc(sizeof(mystruct));
        int t1 = sizeof(mystruct);
        int t2 = sizeof(test1);
        int t3 = sizeof(test2);
        printf("t1: %d\nt2: %d\nt3: %d\n", t1, t2, t3);
}
```

C – Aufgabe 1.2 h)

Sie schreiben folgenden Code. Ihr Programm stürzt jedoch ab. Was ist passiert?

C – Aufgabe 1.2 g)

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
	int *meinezahl;
	meinezahl = 60412;
	printf("%d\n", *meinezahl);
}
```

C – Meine Tipps

Im Folgenden ein (kleines) Sammelsurium an weiteren häufigen Fehlerquellen

Achtung!

- Arrays…
- Char vs. String
- Uninitialized Values & Pointers
- Leaky memory
- Data structure alignment
- Bitfields in Structs

Empfehlung

- Kompiliert mit so vielen Warnung wie möglich!
- Benutzt mindestens…
  - -Wall, -Wextra
  - clang -Wall -Wextra -o my_executable my_c_file.c
- Noch besser, zusätzlich noch…
  - -Werror (verhindert die Kompilierung unter Warnungen)
  - -Wshadow -Wmissing-declarations -Wno-unused-parameter -Wshift-overflow -Wformat-security -Wnull-dereference -Wstack-protector -Walloca -Warray-bounds -Wimplicit-fallthrough -Wliteral-conversion -Wcast-qual -Wundef -Wstrict-prototypes -Wswitch-default -Wcast-align etc.

3. Wichitige Tools

gdb, valgrind, make und co.

Wichtige Tools

- **gdb (mac: lldb)** – Time to debug!
- **valgrind (mac: leaks)** – Time to find memory leaks!
- **make** – Make your life easier!

- Am besten ihr schaut euch Guides/Tutorials an, sobald ihr die Programme benötigt
- Make werdet ihr selber nicht benötigen, aber nice to know, wenn man sich dafür interessiert

Danke, für die Aufmerksamkeit! Bis nächste Woche!