Einführung in die Programmierung

Vorlesungsteil 1
Einführung und erste Schritte

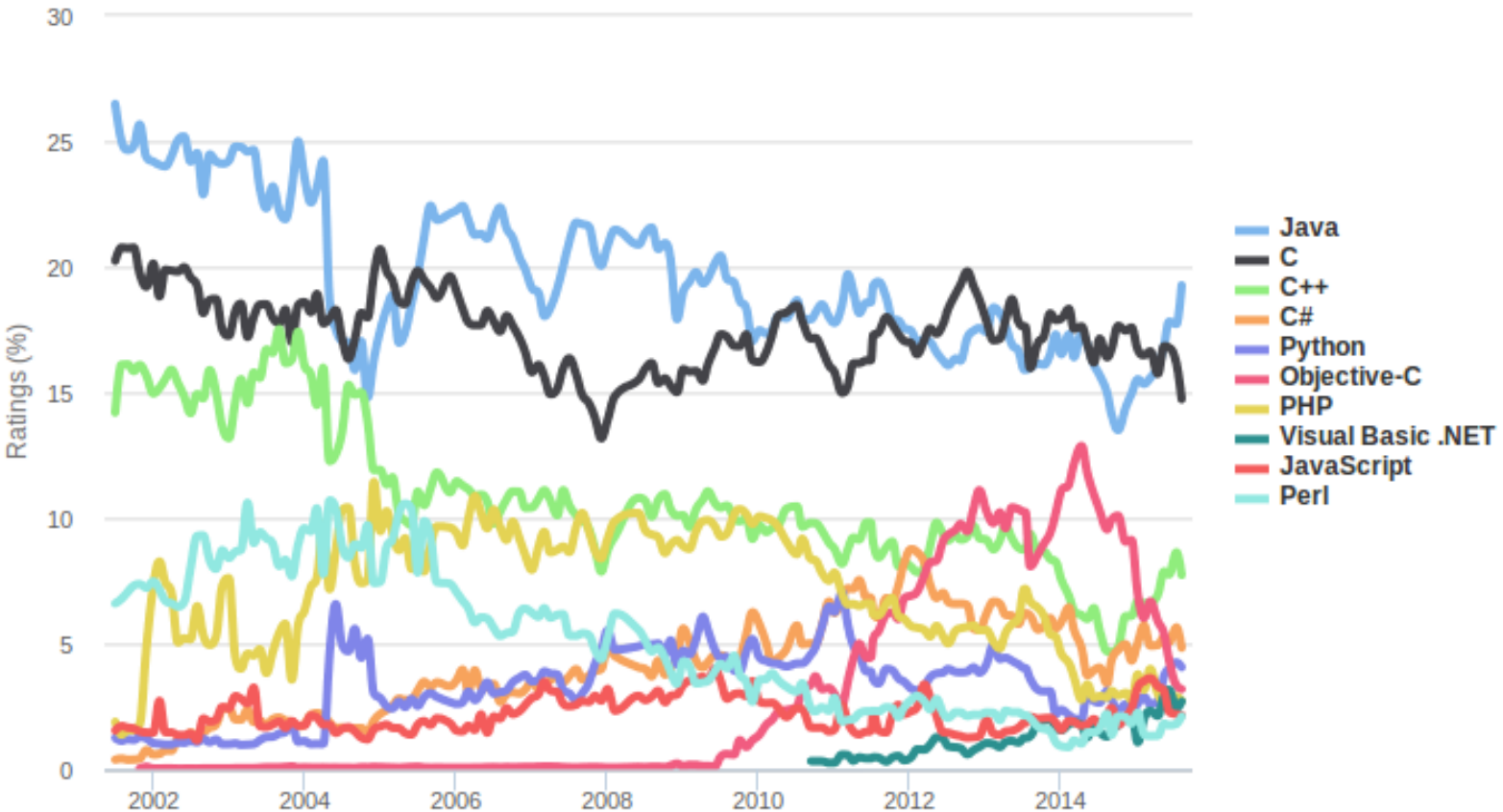
PD Dr. Thomas Hinze

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik

Wintersemester 2015/2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

C: eine der verbreitetsten Programmiersprachen



Quelle: TIOBE Index, www.tiobe.com

Marktanteil von C: ca. 15%

Einige Fakten zu C

- schon *1972* eingeführt, häufig genutzt, schnell *etabliert* und gut *standardisiert*

Einige Fakten zu C

- schon *1972* eingeführt, häufig genutzt, schnell *etabliert* und gut *standardisiert*
- Etwa *1,5 Milliarden* Arbeitsplatzcomputer und mehrere zehntausend Großrechner führen in C geschriebene Programme aus (2013).

Quelle: Christian Ullenboom. C von A bis Z. Galileo Computing, 2014

Einige Fakten zu C

- schon *1972* eingeführt, häufig genutzt, schnell *etabliert* und gut *standardisiert*
- Etwa *1,5 Milliarden* Arbeitsplatzcomputer und mehrere zehntausend Großrechner führen in C geschriebene Programme aus (2013).
- Etwa *9 Millionen* Softwareentwickler weltweit verwenden regelmäßig C.

Quelle: Christian Ullenboom. C von A bis Z. Galileo Computing, 2014

Einige Fakten zu C

- schon *1972* eingeführt, häufig genutzt, schnell *etabliert* und gut *standardisiert*
- Etwa *1,5 Milliarden* Arbeitsplatzcomputer und mehrere zehntausend Großrechner führen in C geschriebene Programme aus (2013).
- Etwa *9 Millionen* Softwareentwickler weltweit verwenden regelmäßig C.
- C verhalf der *imperativen* (befehlsorientierten) Programmierung zum Durchbruch und förderte ihre breite Anwendung

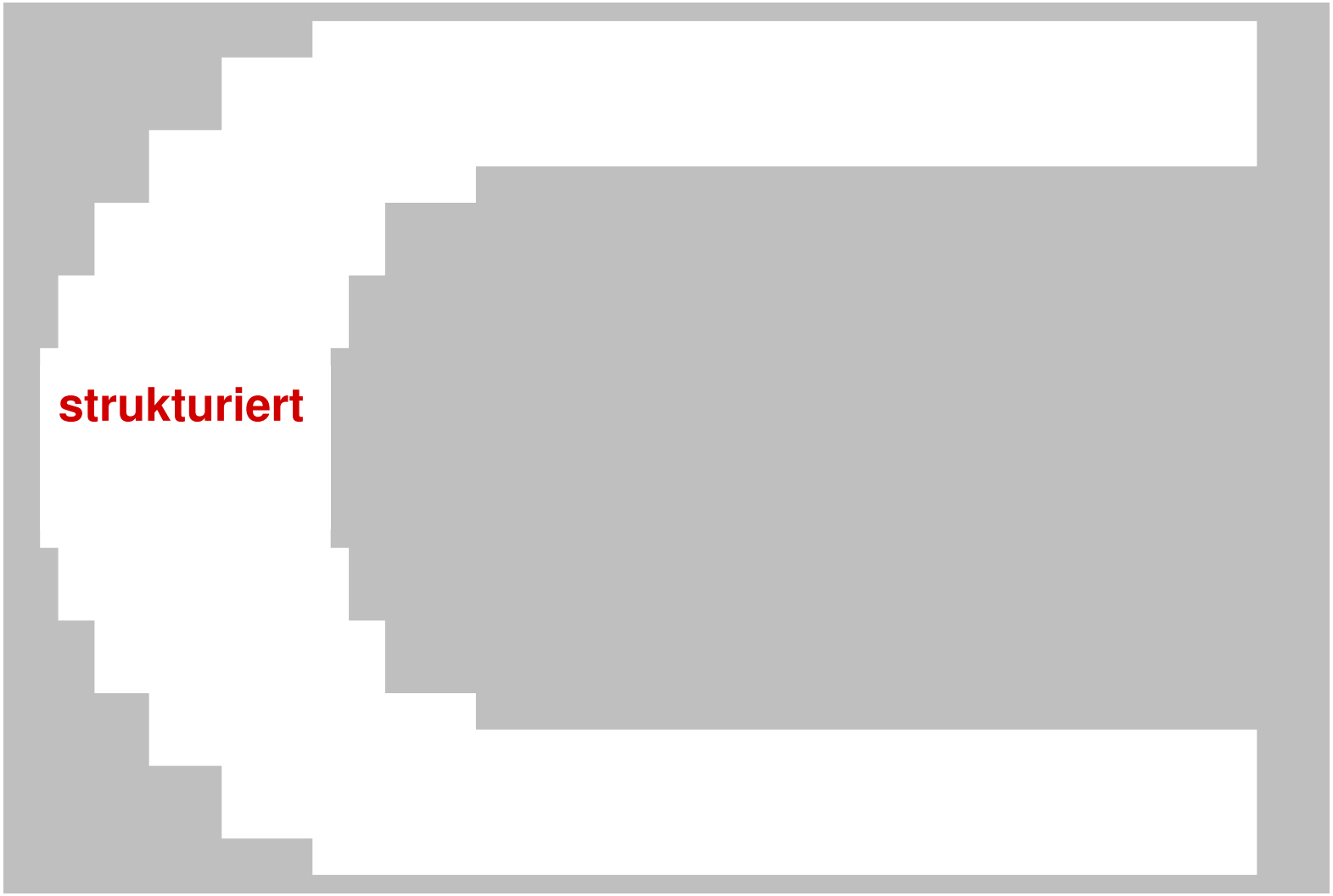
Quelle: Christian Ullenboom. C von A bis Z. Galileo Computing, 2014

Einige Fakten zu C

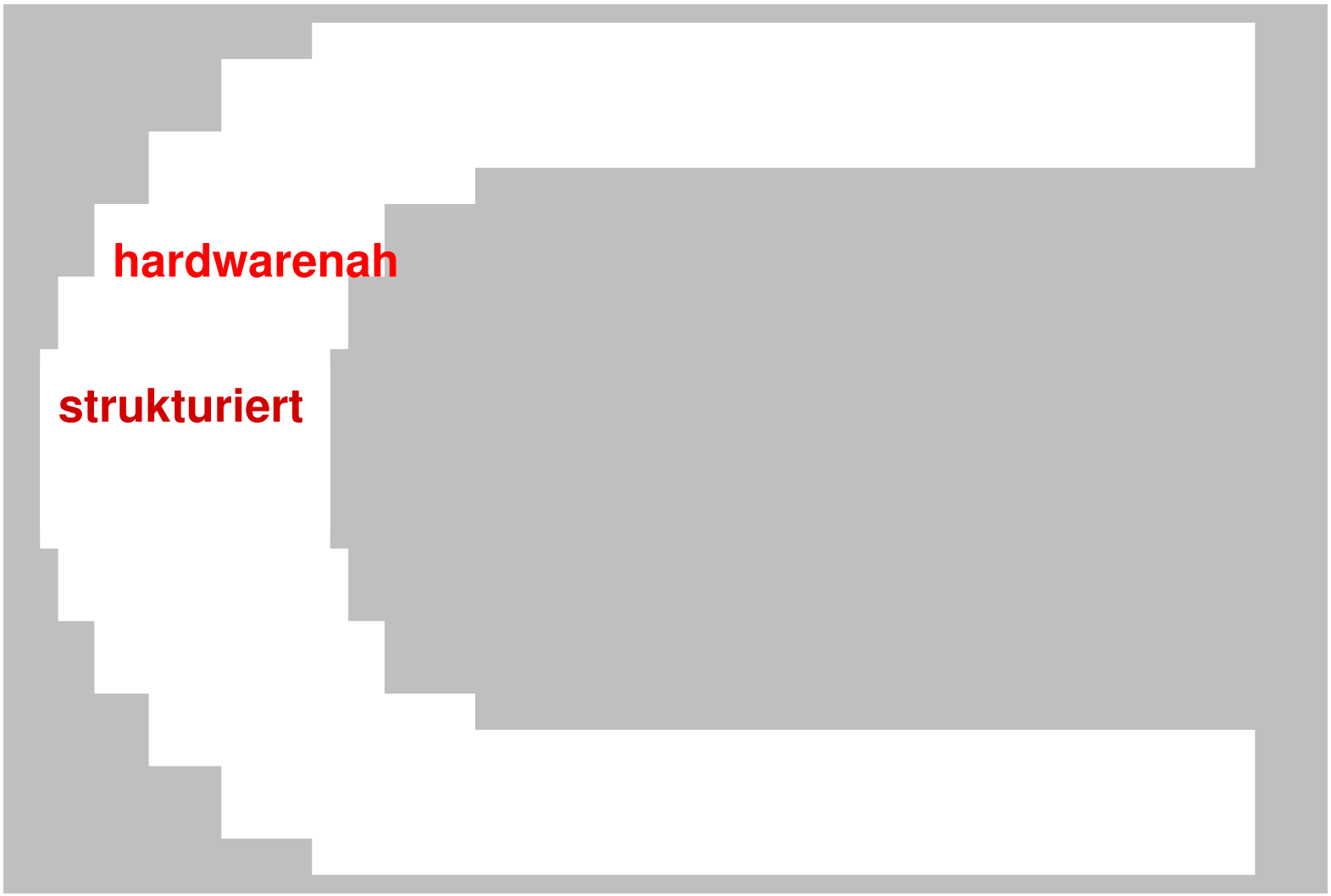
- schon *1972* eingeführt, häufig genutzt, schnell *etabliert* und gut *standardisiert*
- Etwa *1,5 Milliarden* Arbeitsplatzcomputer und mehrere zehntausend Großrechner führen in C geschriebene Programme aus (2013).
- Etwa *9 Millionen* Softwareentwickler weltweit verwenden regelmäßig C.
- C verhalf der *imperativen* (befehlsorientierten) Programmierung zum Durchbruch und förderte ihre breite Anwendung
- C ist Vorläufer zahlreicher weiterentwickelter Programmiersprachen wie Java, C++, C#, JavaScript

Quelle: Christian Ullenboom. C von A bis Z. Galileo Computing, 2014

C ist . . .



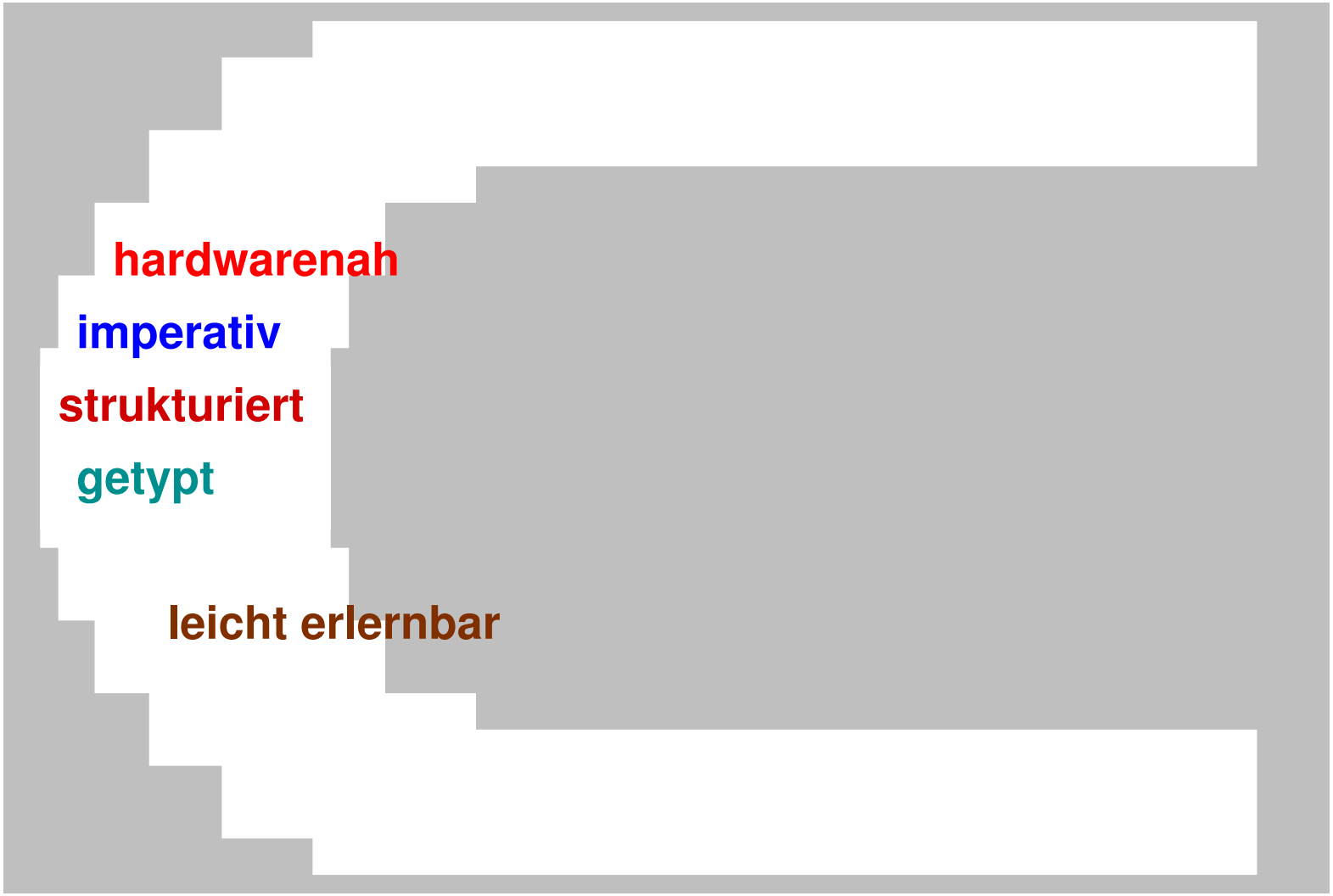
C ist . . .



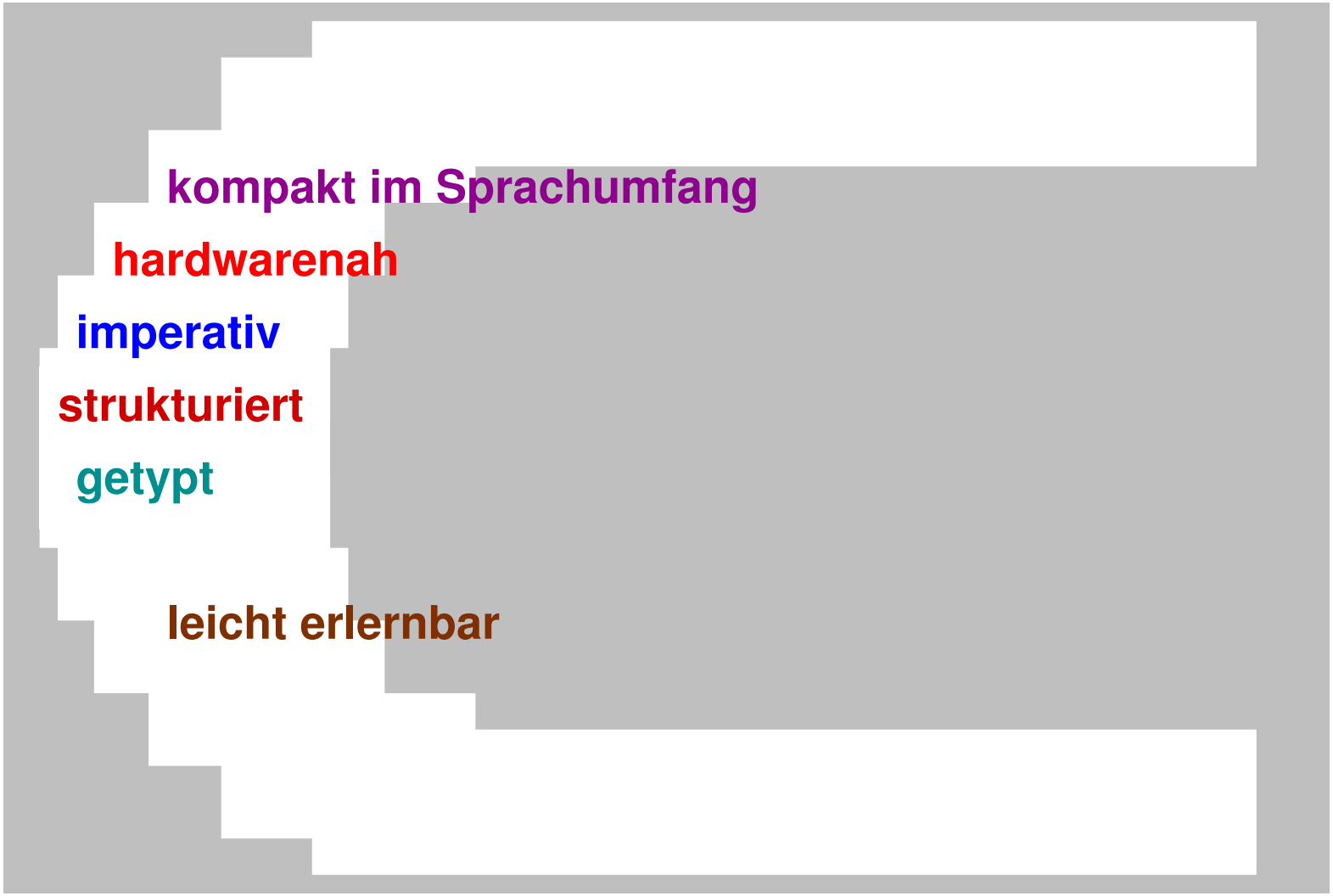
C ist . . .

C ist . . .

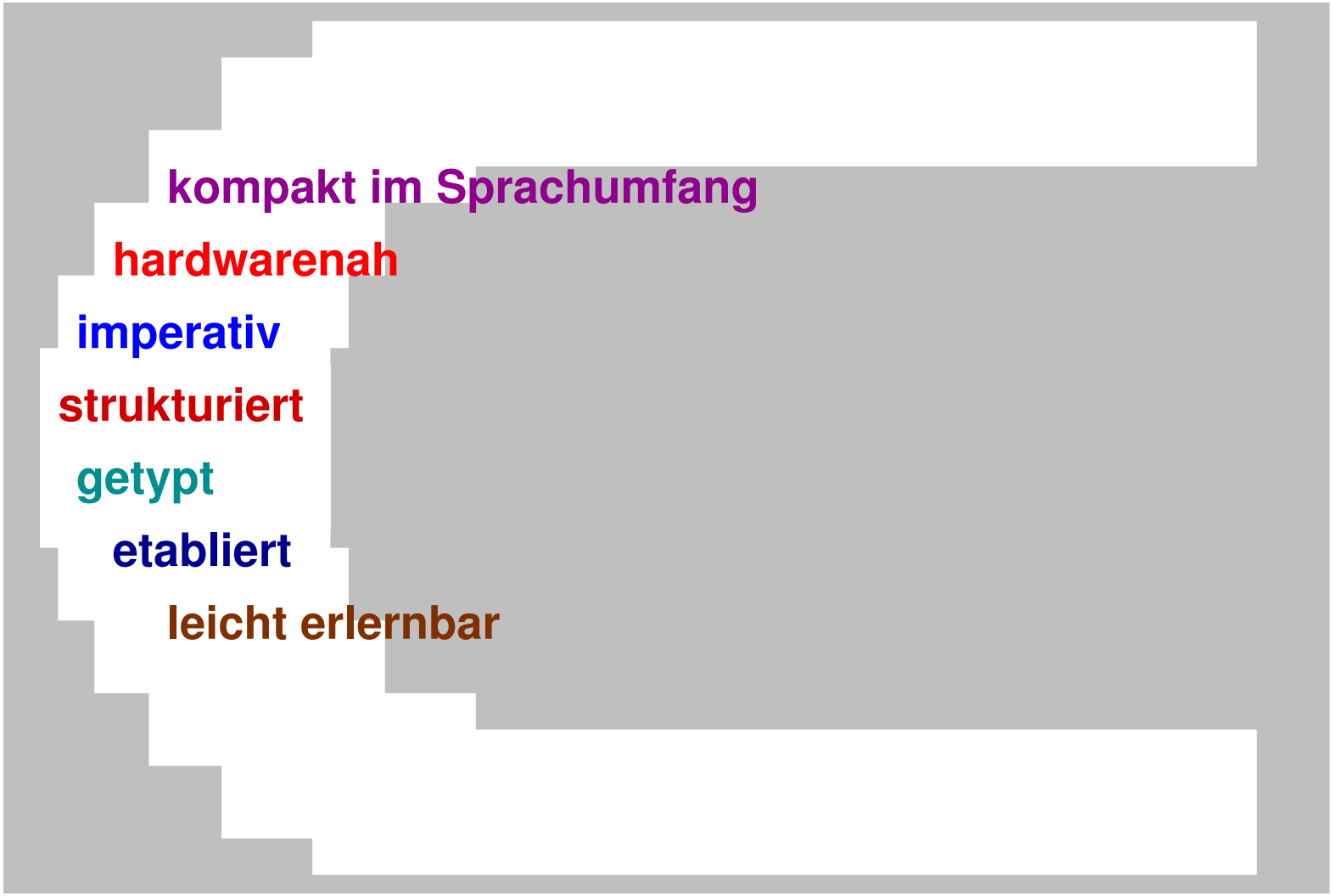
C ist . . .


hardwarenah
imperativ
strukturiert
getypt
leicht erlernbar

C ist . . .


kompakt im Sprachumfang
hardwarenah
imperativ
strukturiert
getypt
leicht erlernbar

C ist . . .


kompakt im Sprachumfang
hardwarenah
imperativ
strukturiert
getypt
etabliert
leicht erlernbar

C ist . . .

ausgerichtet auf kurze Quelltexte

kompakt im Sprachumfang

hardwarenah

imperativ

strukturiert

getypt

etabliert

leicht erlernbar

C ist . . .

ausgerichtet auf kurze Quelltexte

kompakt im Sprachumfang

hardwarenah

imperativ

strukturiert

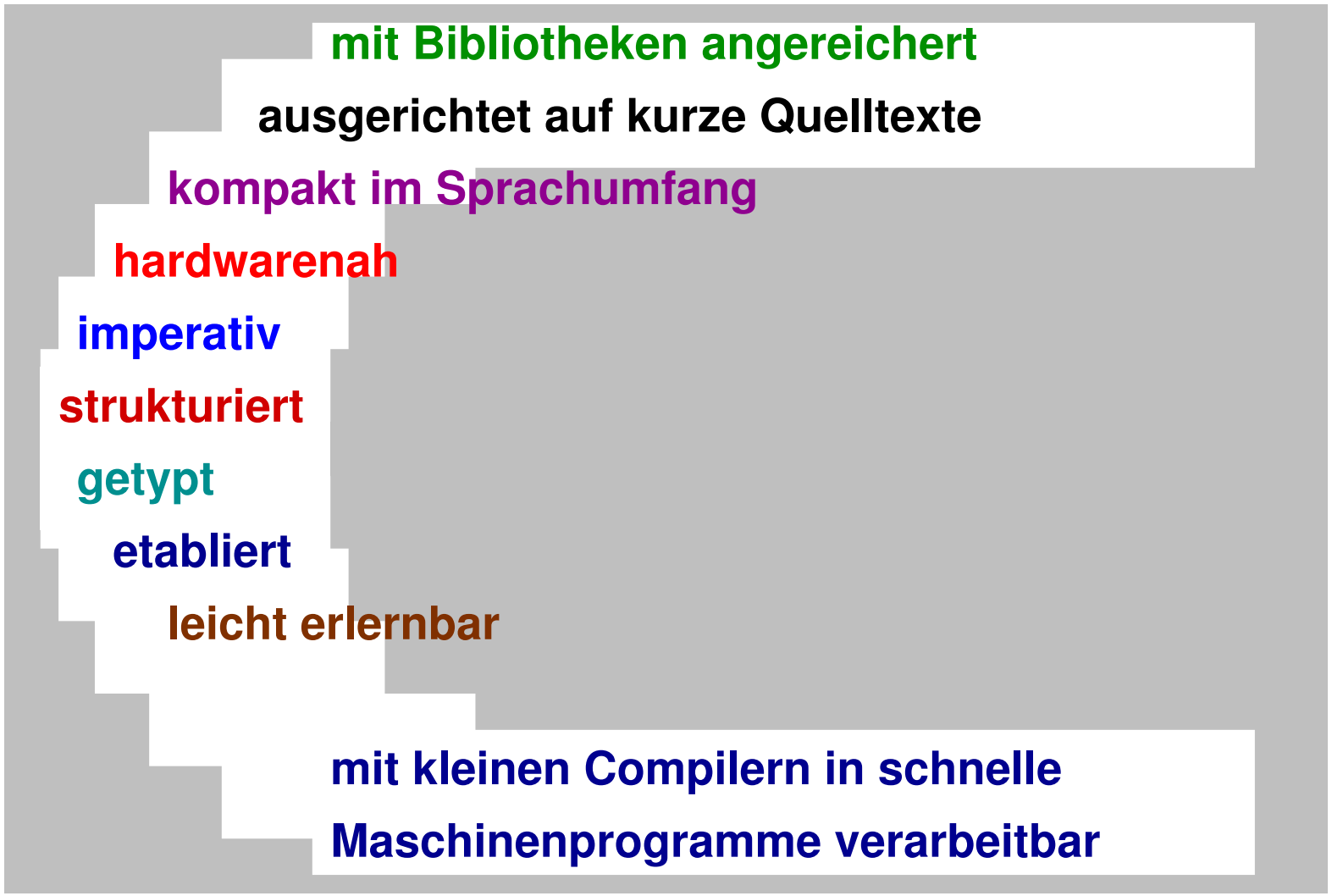
getypt

etabliert

leicht erlernbar

mit kleinen Compilern in schnelle Maschinenprogramme verarbeitbar

C ist . . .


mit Bibliotheken angereichert
ausgerichtet auf kurze Quelltexte
kompakt im Sprachumfang
hardwarenah
imperativ
strukturiert
getypt
etabliert
leicht erlernbar
mit kleinen Compilern in schnelle
Maschinenprogramme verarbeitbar

C ist . . .

mit Bibliotheken angereichert

ausgerichtet auf kurze Quelltexte

kompakt im Sprachumfang

hardwarenah

imperativ

strukturiert

getypt

etabliert

leicht erlernbar

auf Dateiebene modularisiert

mit kleinen Compilern in schnelle

Maschinenprogramme verarbeitbar

Software, die größtenteils in C geschrieben wurde

- UNIX-Betriebssystem (erstes Großprojekt 1973)
- zahlreiche Hardwareprogrammierungen (z.B. FPGAs) und Systemsimulatoren

Alle 32 Schlüsselwörter von C

C als besonders kompakte Programmiersprache

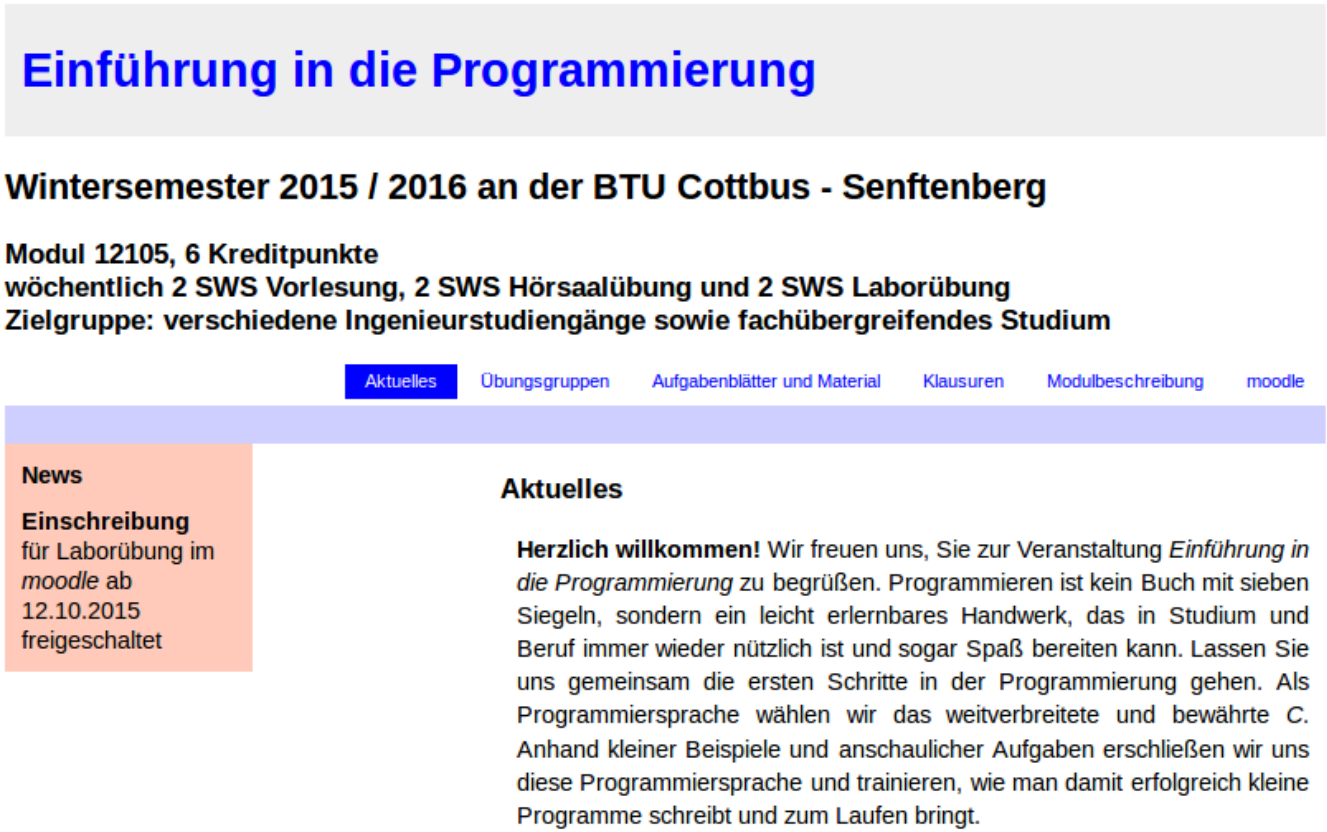
`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

im ANSI-Standard C90

ergänzt durch vorbelegte Bezeichner (z.B. `main`), Bibliotheksfunktionen (z.B. `printf`) und Operatoren (z.B. `+` und `<`)

Webseite

http://www.informatik.tu-cottbus.de/∼hinzet/eidp-wise1516/



Einführung in die Programmierung

Wintersemester 2015 / 2016 an der BTU Cottbus - Senftenberg

Modul 12105, 6 Kreditpunkte
wöchentlich 2 SWS Vorlesung, 2 SWS Hörsaalübung und 2 SWS Laborübung
Zielgruppe: verschiedene Ingenieurstudiengänge sowie fachübergreifendes Studium

Aktuelles | Übungsgruppen | Aufgabenblätter und Material | Klausuren | Modulbeschreibung | moodle

News

Einschreibung
für Laborübung im *moodle* ab 12.10.2015 freigeschaltet

Aktuelles

Herzlich willkommen! Wir freuen uns, Sie zur Veranstaltung *Einführung in die Programmierung* zu begrüßen. Programmieren ist kein Buch mit sieben Siegeln, sondern ein leicht erlernbares Handwerk, das in Studium und Beruf immer wieder nützlich ist und sogar Spaß bereiten kann. Lassen Sie uns gemeinsam die ersten Schritte in der Programmierung gehen. Als Programmiersprache wählen wir das weitverbreitete und bewährte *C*. Anhand kleiner Beispiele und anschaulicher Aufgaben erschließen wir uns diese Programmiersprache und trainieren, wie man damit erfolgreich kleine Programme schreibt und zum Laufen bringt.

- Ankündigungen (z.B. Klausurtermine)
- Übungsblätter und hilfreiche Links
- Downloadmaterial

Orga ganz kurz

https://www.tu-cottbus.de/elearning/btu

- Bitte im Laufe dieser Woche im Moodle für eine Laborübungsgruppe anmelden
- Ab 19.10. drei Veranstaltungen jede Woche: Vorlesung, Hörsaalübung, Laborübung
- **Alle weiteren Organisationsdetails nächste Woche in der Hörsaalübung**

Vorlesung Einführung in die Programmierung mit C

1. Einführung und erste Schritte . . . . . .

. . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer

Vorlesung Einführung in die Programmierung mit C

1. Einführung und erste Schritte . . . . . .
. . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer

2. Elementare Datentypen, Variablen, Arithmetik, Typecast . . . . . .
. . C als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe

Vorlesung Einführung in die Programmierung mit C

1. Einführung und erste Schritte . . . . . .
. . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer

2. Elementare Datentypen, Variablen, Arithmetik, Typecast . . . . . .
. . C als Taschenrechner nutzen, Tastatureingabe $\rightarrow$ Formelberechnung $\rightarrow$ Ausgabe

3. Imperative Kontrollstrukturen . . . . . .
. . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren

Vorlesung Einführung in die Programmierung mit C

1. Einführung und erste Schritte . . . . . .
. . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer

2. Elementare Datentypen, Variablen, Arithmetik, Typecast . . . . . .
. . C als Taschenrechner nutzen, Tastatureingabe $\rightarrow$ Formelberechnung $\rightarrow$ Ausgabe

3. Imperative Kontrollstrukturen . . . . . .
. . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren

4. Aussagenlogik in C . . . . . .
. . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren

Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe $\rightarrow$ Formelberechnung $\rightarrow$ Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe

Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe $\rightarrow$ Formelberechnung $\rightarrow$ Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel

Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe $\rightarrow$ Formelberechnung $\rightarrow$ Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
7. **Felder und Strukturierung von Daten** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen

Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe $\rightarrow$ Formelberechnung $\rightarrow$ Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
7. **Felder und Strukturierung von Daten** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf

Vorlesung Einführung in die Programmierung mit C

1. Einführung und erste Schritte .....
..... Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer

2. Elementare Datentypen, Variablen, Arithmetik, Typecast .....
.. C als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe

3. Imperative Kontrollstrukturen .....
..... Befehlsfolgen, Verzweigungen und Schleifen programmieren

4. Aussagenlogik in C .....
..... Schaltbelegungstabellen aufstellen, optimieren und implementieren

5. Funktionen selbst programmieren .....
... Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe

6. Rekursion .....
.... selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel

7. Felder und Strukturierung von Daten .....
.... effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen

8. Sortieren .....
..... klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf

9. Zeiger, Zeichenketten und Dateiarbeit .....
..... Texte analysieren, ver- und entschlüsseln, Dateien lesen und schreiben

Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe $\rightarrow$ Formelberechnung $\rightarrow$ Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
7. **Felder und Strukturierung von Daten** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeiger, Zeichenketten und Dateiarbeit** . . . . . Texte analysieren, ver- und entschlüsseln, Dateien lesen und schreiben
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank

Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe $\rightarrow$ Formelberechnung $\rightarrow$ Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
7. **Felder und Strukturierung von Daten** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeiger, Zeichenketten und Dateiarbeit** . . . . . Texte analysieren, ver- und entschlüsseln, Dateien lesen und schreiben
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank
11. **Ausblick und weiterführende Konzepte** . . . . .

Literatur: Wikibook C-Programmierung

http://de.wikibooks.org/wiki/C-Programmierung

$\Longrightarrow$ kostenfrei, gut verständlich, klar gegliedert, viele Beispiele

Vertiefend: Galileo Computing – C von A bis Z

http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/

$\Longrightarrow$ nützliche Praxistipps, themenübergreifende Beispiele

Vorbereitungen

Um eigene C-Programme schreiben und ausführen zu können, brauchen wir als Software-Grundausstattung

- einen **Texteditor** (Empfehlung: **Notepad++**)

Vorbereitungen

Um eigene C-Programme schreiben und ausführen zu können, brauchen wir als Software-Grundausstattung

- einen **Texteditor** (Empfehlung: **Notepad++**)
- einen **C-Compiler** (Empfehlung: `gcc`)

Vorbereitungen

Um eigene C-Programme schreiben und ausführen zu können, brauchen wir als Software-Grundausstattung

- einen **Texteditor** (Empfehlung: **Notepad++**)
- einen **C-Compiler** (Empfehlung: `gcc`)
- ein **Terminal-Fenster**
  (heißt in Windows: Eingabeaufforderung)
  zum Ausführen compilierter C-Programme

Vorbereitungen

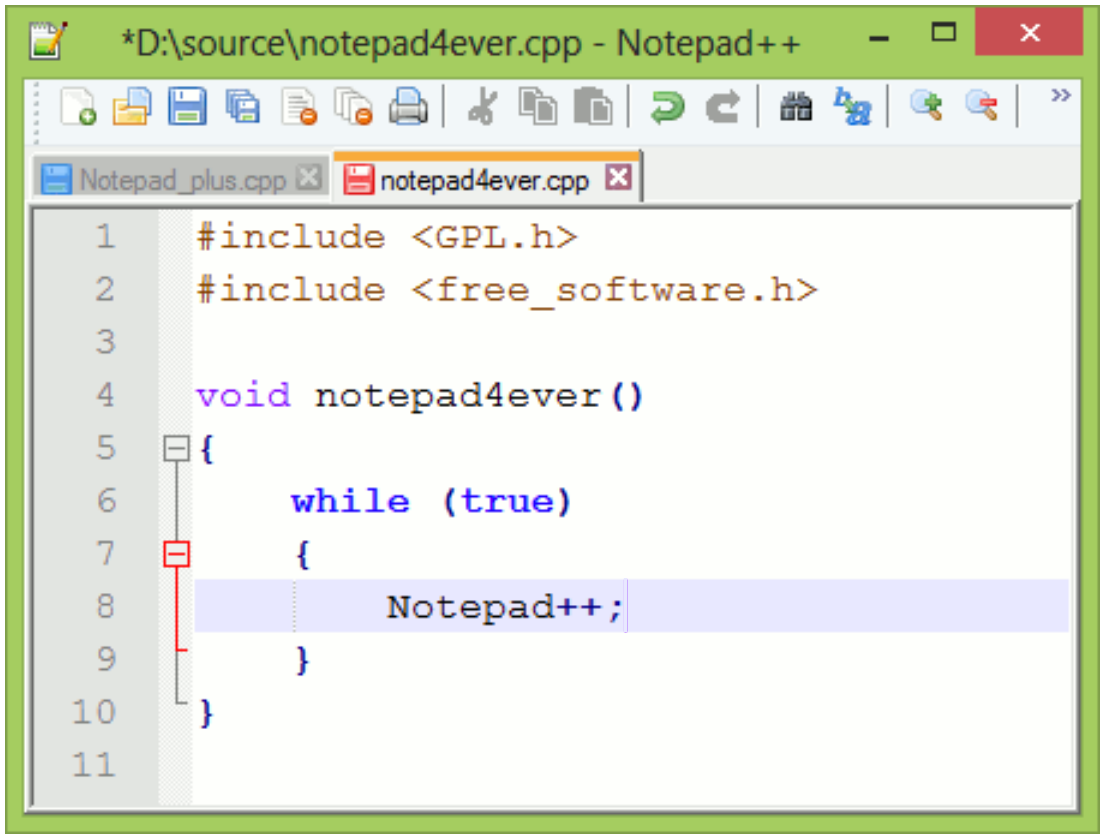
Um eigene C-Programme schreiben und ausführen zu können, brauchen wir als Software-Grundausstattung

- einen **Texteditor** (Empfehlung: **Notepad++**)
- einen **C-Compiler** (Empfehlung: `gcc`)
- ein **Terminal-Fenster**
  (heißt in Windows: Eingabeaufforderung)
  zum Ausführen compilierter C-Programme
- (oder alternativ eine integrierte C-Entwicklungsumgebung wie **xcode** für Mac OS oder **Orwell Dev C/C++** für Windows)

Notepad++ zum Eingeben der C-Programmquelltexte

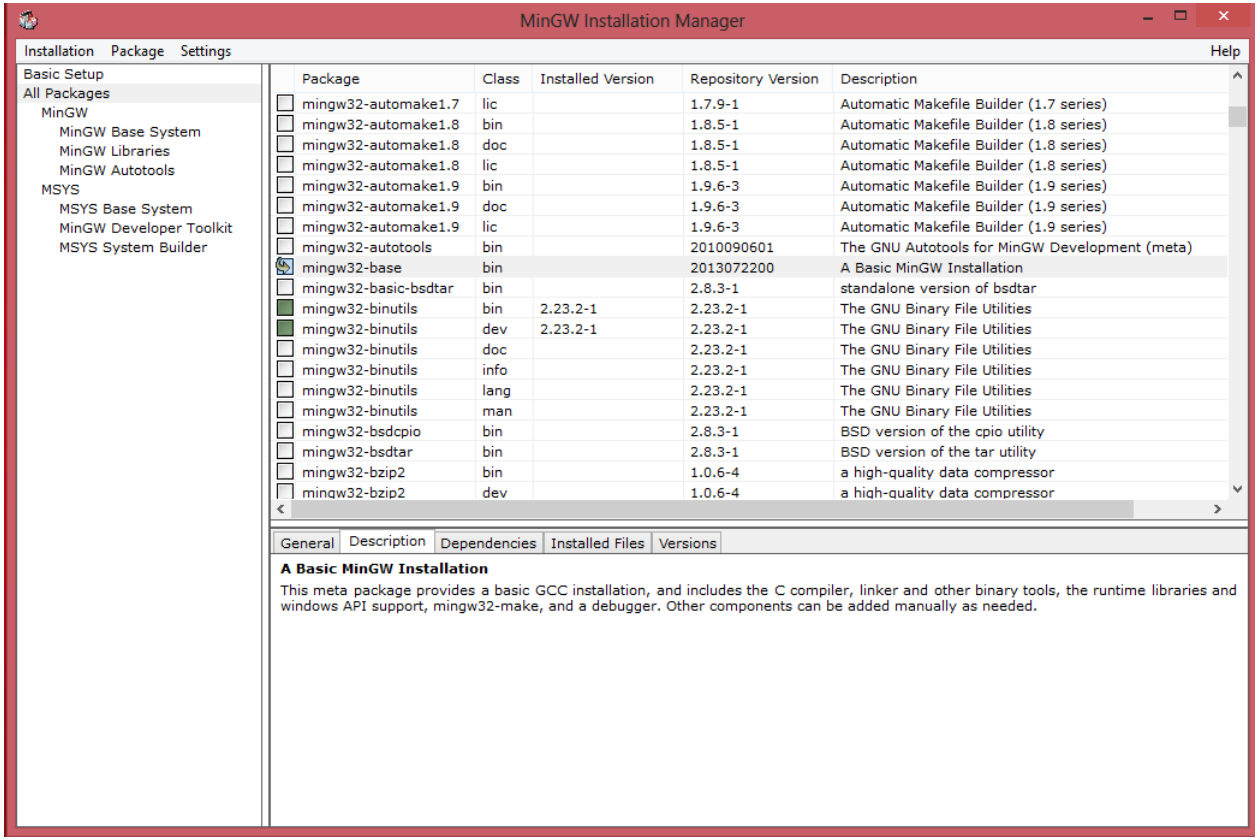
http://www.notepad-plus-plus.org

flexibler Unicode-fähiger Editor mit Syntax-Hervorhebung



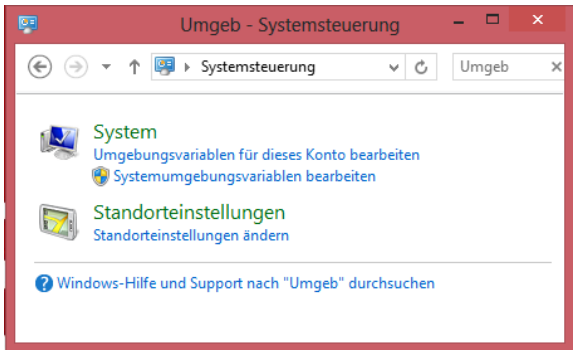
GNU C-Compiler (unter Windows: MinGW)

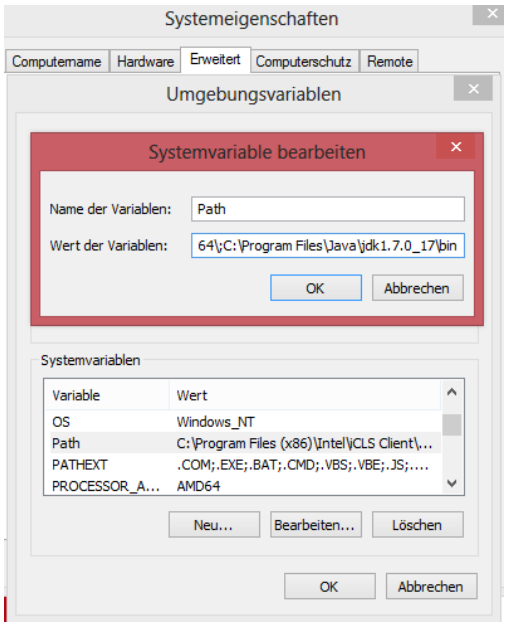
http://sourceforge.net/projects/mingw/files/Installer/



Runterladen und starten: **`mingw-get-setup.exe`**

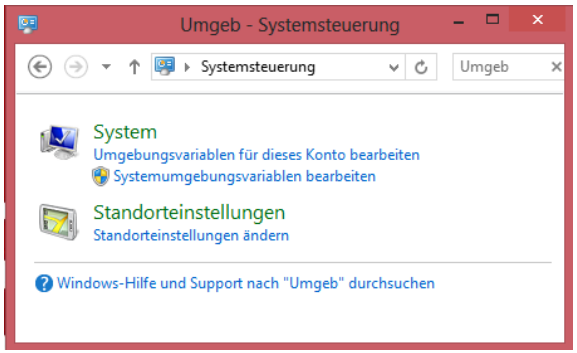
Umgebungsvariable „Path“ setzen

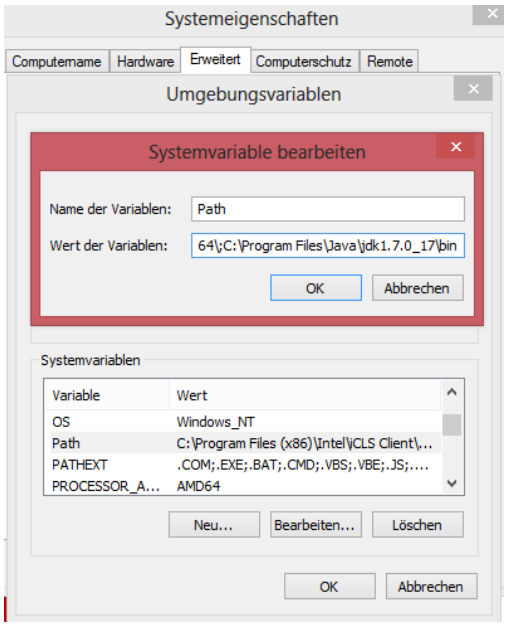




Viele Programme (z.B. den C-Compiler `gcc`) ruft man aus einem Ordner heraus auf, in welchem sich die Programmdatei selbst nicht befindet.

Umgebungsvariable „Path“ setzen

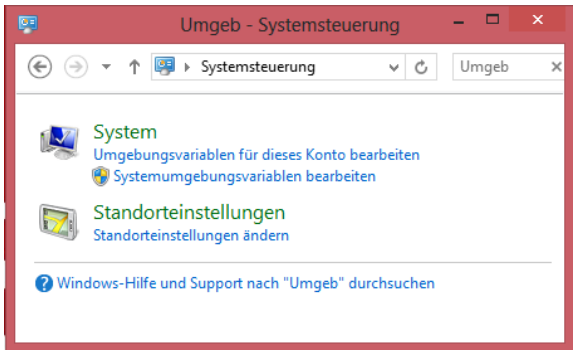


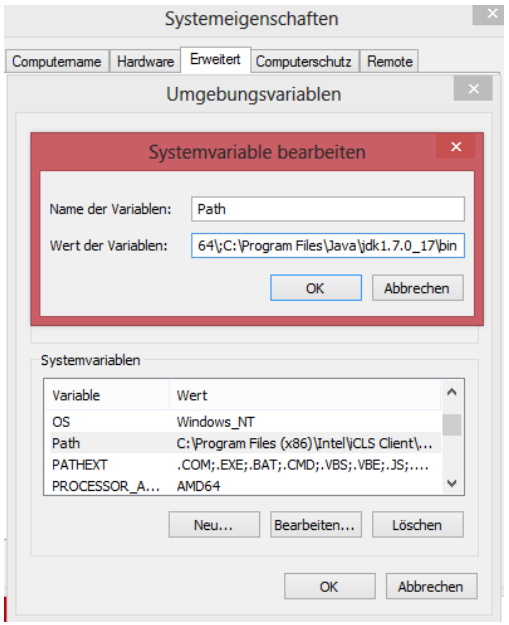


Viele Programme (z.B. den C-Compiler `gcc`) ruft man aus einem Ordner heraus auf, in welchem sich die Programmdatei selbst nicht befindet.

Damit die ausführbare Programmdatei gefunden wird, muss ihr Verzeichnispfad in der Umgebungsvariablen „Path“ eingetragen sein.

Umgebungsvariable „Path“ setzen





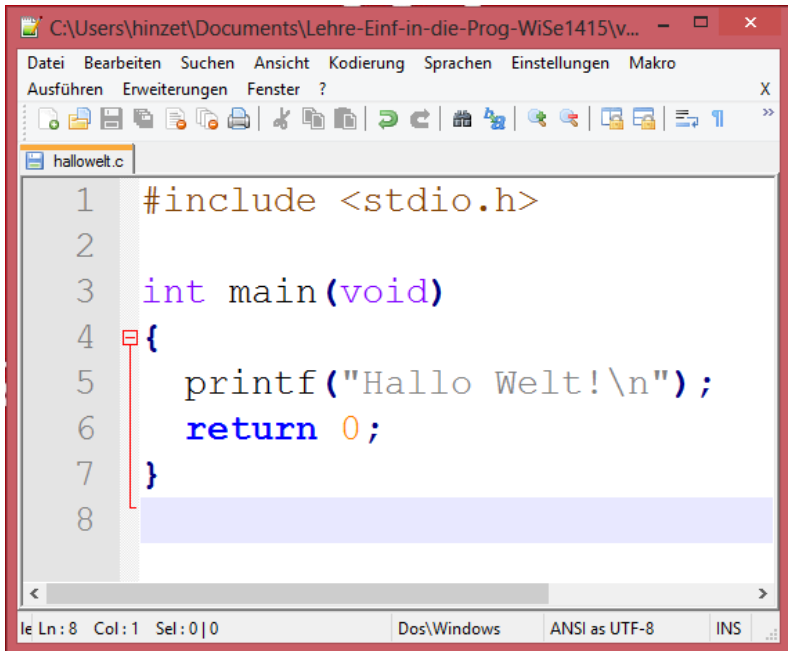
Viele Programme (z.B. den C-Compiler `gcc`) ruft man aus einem Ordner heraus auf, in welchem sich die Programmdatei selbst nicht befindet.

Damit die ausführbare Programmdatei gefunden wird, muss ihr Verzeichnispfad in der Umgebungsvariablen „Path“ eingetragen sein.

Systemsteuerung $\longrightarrow$ Umgebungsvariablen $\longrightarrow$ Systemumgebungs-variablen bearbeiten. Mehrere Pfade durch „`;`“ trennen

Ein erstes C-Programm: `hallowelt.c`

Im Texteditor eingeben und als **`hallowelt.c`** speichern:



- Terminal-Fenster öffnen und in das Verzeichnis wechseln, in welchem **`hallowelt.c`** liegt
- Compilieren durch **`gcc hallowelt.c`**
- Dadurch entsteht neue Datei **`a.exe`**
- Ausführen dieser Datei durch **`a.exe`**

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Jede C-Quelltextdatei enthält eine Sammlung von **Funktionen**.

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Jede C-Quelltextdatei enthält eine Sammlung von **Funktionen**.
- Die Funktion mit dem Namen `main` wird als erste aufgerufen.

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Jede C-Quelltextdatei enthält eine Sammlung von **Funktionen**.
- Die Funktion mit dem Namen `main` wird als erste aufgerufen.
- Es werden keine Argumente übergeben, dafür steht `void`.

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Jede C-Quelltextdatei enthält eine Sammlung von **Funktionen**.
- Die Funktion mit dem Namen `main` wird als erste aufgerufen.
- Es werden keine Argumente übergeben, dafür steht `void`.
- Funktionswert (Ergebnis) ist eine ganze Zahl, dafür steht `int`.

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Der Funktionsrumpf wird in geschweifte Klammern { } gesetzt.

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Der Funktionsrumpf wird in geschweifte Klammern { } gesetzt.
- Dort Schritt für Schritt beschrieben, was die Funktion tun soll.

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Der Funktionsrumpf wird in geschweifte Klammern { } gesetzt.
- Dort Schritt für Schritt beschrieben, was die Funktion tun soll.
- Guter Stil: Programmblöcke zwischen { und } einrücken.

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Der Funktionsrumpf wird in geschweifte Klammern { } gesetzt.
- Dort Schritt für Schritt beschrieben, was die Funktion tun soll.
- Guter Stil: Programmblöcke zwischen { und } einrücken.
- Runde Klammern ( ) für Argumente, geschweifte für Blöcke.

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Funktion `printf` bewirkt formatierte Bildschirmausgabe

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Funktion `printf` bewirkt formatierte Bildschirmausgabe
- `printf` ist in der Funktionsbibliothek `stdio.h` definiert.

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Funktion `printf` bewirkt formatierte Bildschirmausgabe
- `printf` ist in der Funktionsbibliothek `stdio.h` definiert.
- Bibliothek zur Nutzung mit `#include` einbinden (Präprozessorteil am Quelltextanfang)

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Funktion `printf` bewirkt formatierte Bildschirmausgabe
- `printf` ist in der Funktionsbibliothek `stdio.h` definiert.
- Bibliothek zur Nutzung mit `#include` einbinden (Präprozessorteil am Quelltextanfang)
- Auszugebende Zeichenkette als Argument in `"  "` setzen, `\n`: Zeilenumbruch

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Schlüsselwort `return` beendet Funktion

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Schlüsselwort `return` beendet Funktion
- Funktionswert (hier die Zahl `0`) wird zurückgegeben

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Schlüsselwort `return` beendet Funktion
- Funktionswert (hier die Zahl `0`) wird zurückgegeben
- Rückgabewert als Fehlerstatus auffassen (`0`: kein Fehler)

Bestandteile des Programmquelltextes

```
#include <stdio.h>

int main(void)
{
  printf("Hallo Welt!\n");
  return 0;
}
```

- Schlüsselwort `return` beendet Funktion
- Funktionswert (hier die Zahl `0`) wird zurückgegeben
- Rückgabewert als Fehlerstatus auffassen (`0`: kein Fehler)
- Am Ende jeder Anweisung im Block steht ein Semikolon `;`

Kommentare im Quelltext

```
#include <stdio.h>

int main(void)
{
  /* Das ist ein
     Kommentar ueber
     mehrere Zeilen. */
  printf("Hallo Welt!\n");
  return 0;
}
```

- **Kommentare** zur Verbesserung der Lesbarkeit von Quelltexten

Kommentare im Quelltext

```
#include <stdio.h>

int main(void)
{
  /* Das ist ein
     Kommentar ueber
     mehrere Zeilen. */
  printf("Hallo Welt!\n");
  return 0;
}
```

- **Kommentare** zur Verbesserung der Lesbarkeit von Quelltexten
- können an beliebigen Zeilen eingefügt werden durch `/*  */`

Kommentare im Quelltext

```
#include <stdio.h>

int main(void)
{
  /* Das ist ein
     Kommentar ueber
     mehrere Zeilen. */
  printf("Hallo Welt!\n");
  return 0;
}
```

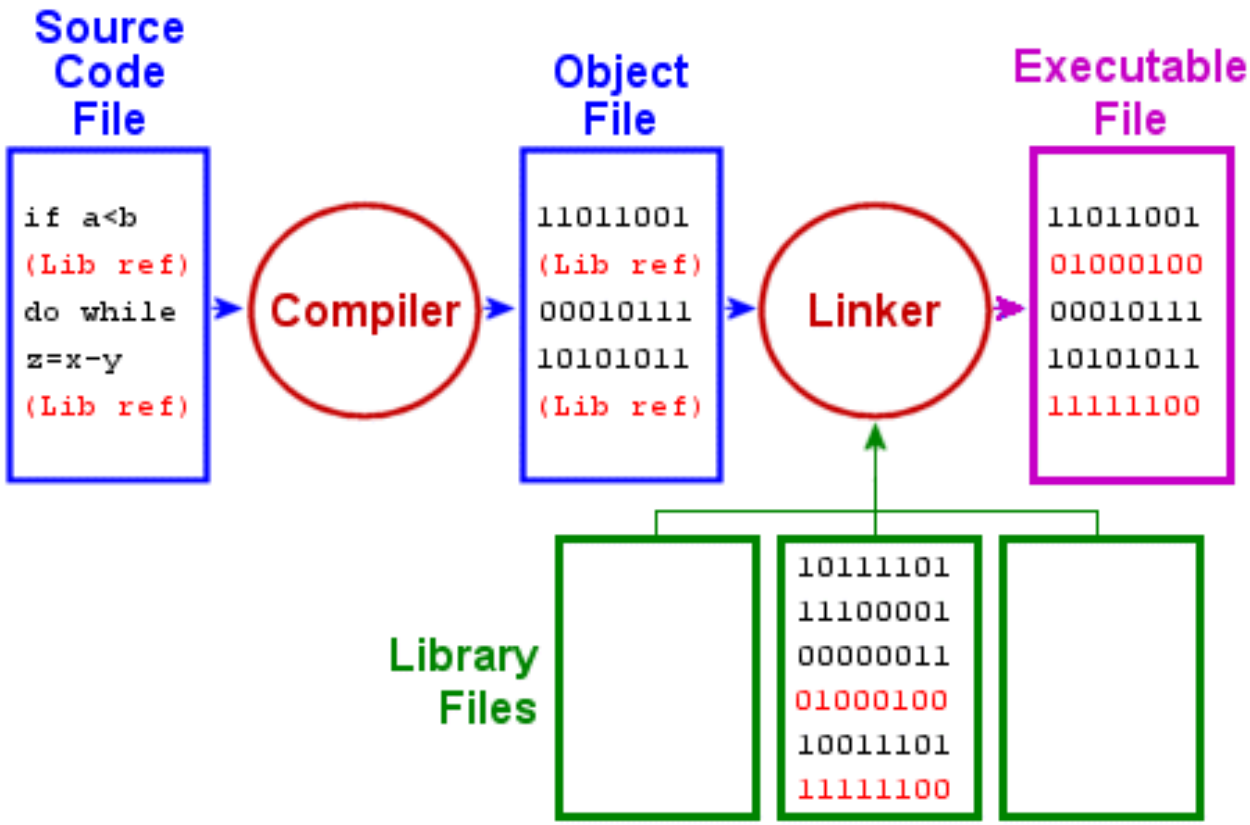
- **Kommentare** zur Verbesserung der Lesbarkeit von Quelltexten
- können an beliebigen Zeilen eingefügt werden durch `/*  */`
- eignen sich, um Quelltextteile vom Compilieren auszuschließen

Kommentare im Quelltext

```
#include <stdio.h>

int main(void)
{
  // Einzeiliger Kommentar
  printf("Hallo Welt!\n");
  return 0;
}
```

- einzeilige Kommentare mit `//` einleiten
- Kommentare werden beim Compilieren ignoriert

Compilieren und Ausführen von C-Programmen



- **Compiler** mit integriertem **Linker** erzeugt ausführbares **Maschinenprogramm** ("`.exe`")
- vorgefertigte Bibliotheksfunktionen werden dabei einbezogen

Fragen . . .

- Warum so *„umständlich“*?

Fragen . . .

- Warum so *„umständlich“*?
- Weshalb kann man den C-Quelltext `hallowelt.c` nicht „unmittelbar“ ausführen, sondern muss erst den *C-Compiler* bemühen und eine von ihm erzeugte ausführbare Datei (z.B. `a.exe`) starten?

Fragen . . .

- Warum so *„umständlich“*?
- Weshalb kann man den C-Quelltext `hallowelt.c` nicht „unmittelbar“ ausführen, sondern muss erst den *C-Compiler* bemühen und eine von ihm erzeugte ausführbare Datei (z.B. `a.exe`) starten?

$\Longrightarrow$ Um das zu verstehen, brauchen wir eine grobe Vorstellung davon, was im Inneren des Computers passiert und wie er arbeitet.

Typische Komponenten eines Personal Computers

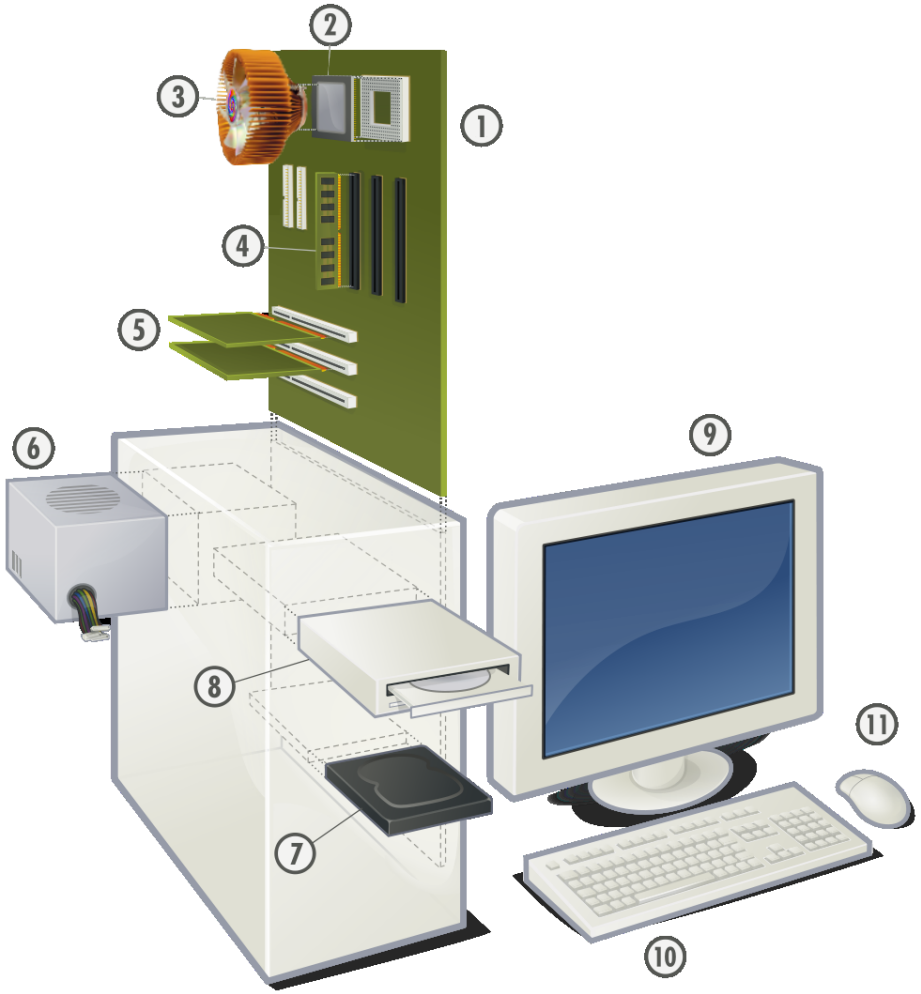


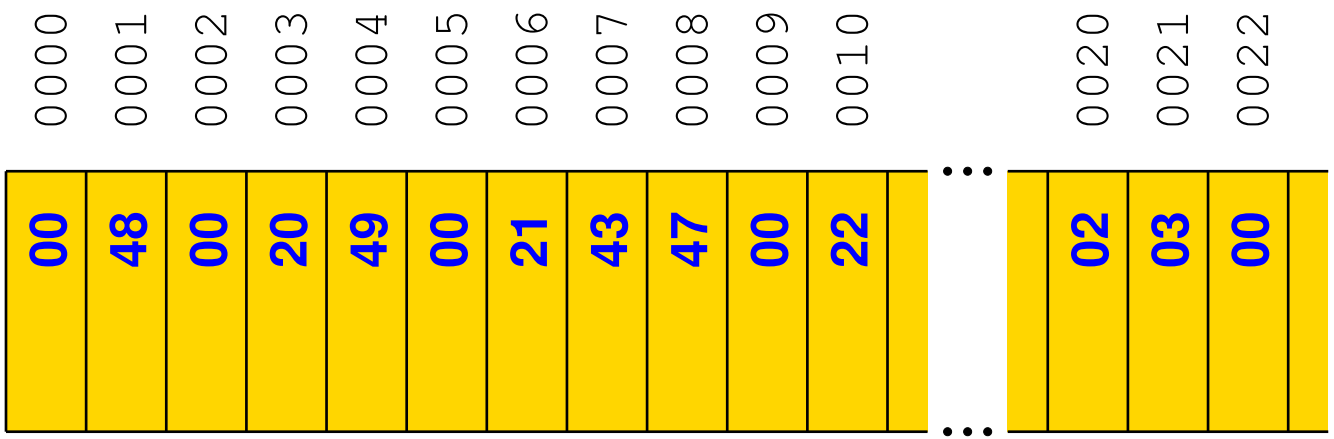
Bild: www.wikipedia.de

1. Hauptplatine (Motherboard)
2. Hauptprozessor (CPU)
3. Prozessorkühler
4. Arbeitsspeicher (RAM)
5. Grafikkarte und Netzwerkkarte
6. Netzteil
7. Festplattenlaufwerk (HDD)
8. Optisches Laufwerk (z.B. DVD-Brenner)
9. Monitor
10. Tastatur
11. Maus

Arbeitsspeicher

organisiert als virtueller Speicher

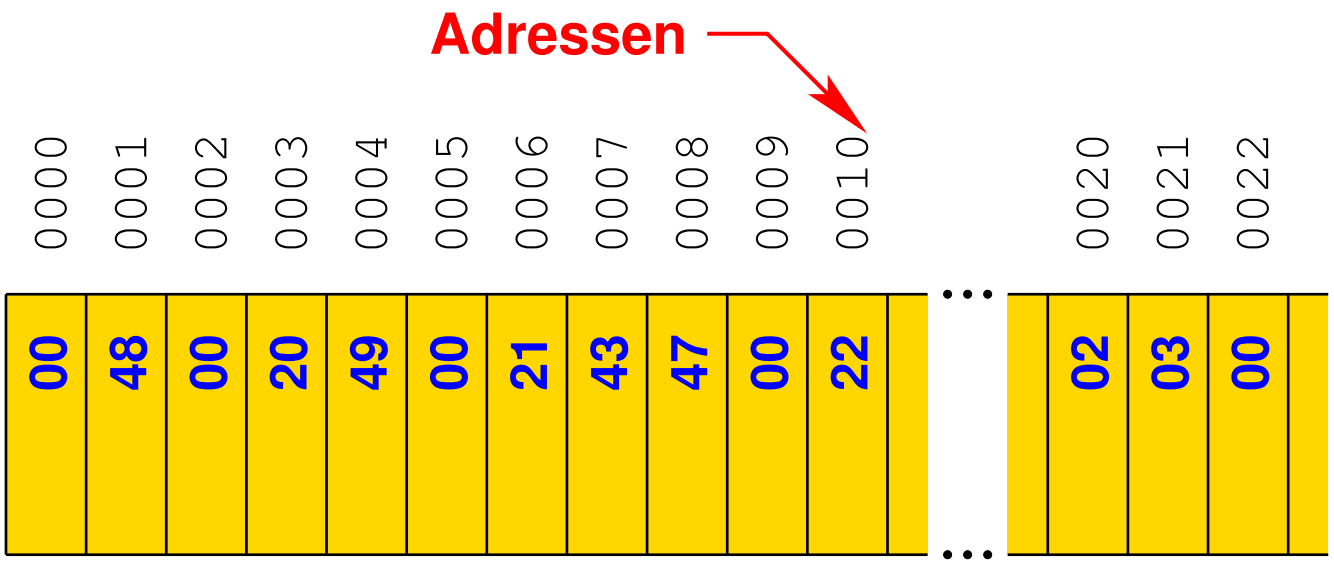
- Speicherzellen von 0 an fortlaufend durchnummeriert



Arbeitsspeicher

organisiert als virtueller Speicher

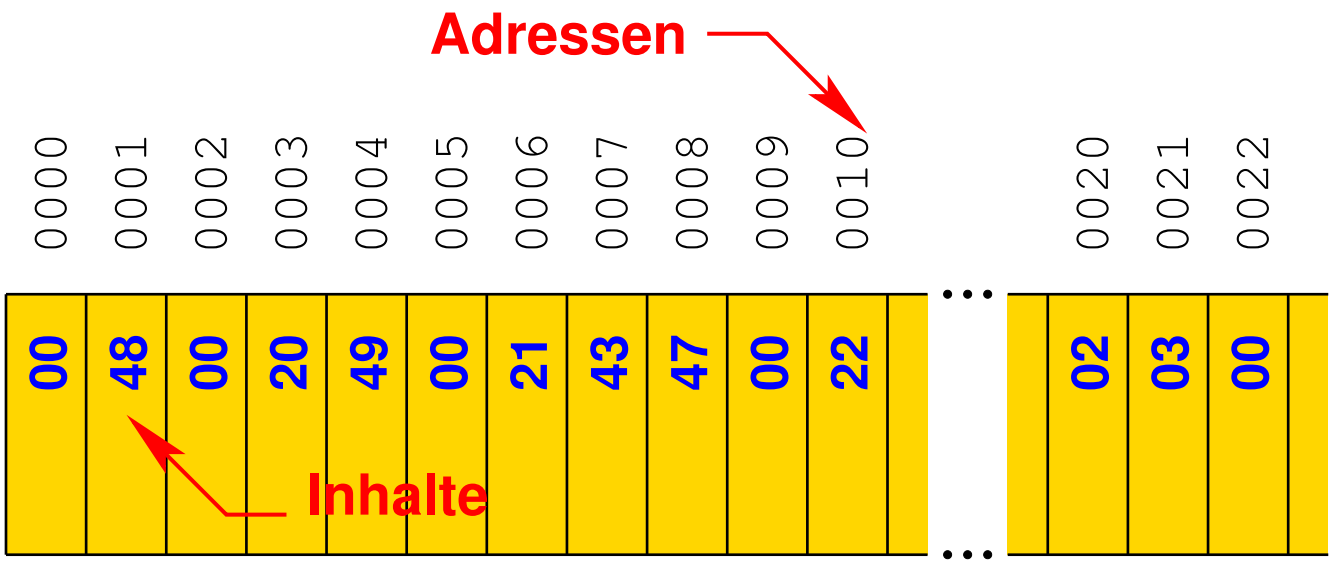
- Speicherzellen von 0 an fortlaufend durchnummeriert
- Jede Speicherzelle besitzt eine eigene *Adresse*, über die sie angesprochen wird



Arbeitsspeicher

organisiert als virtueller Speicher

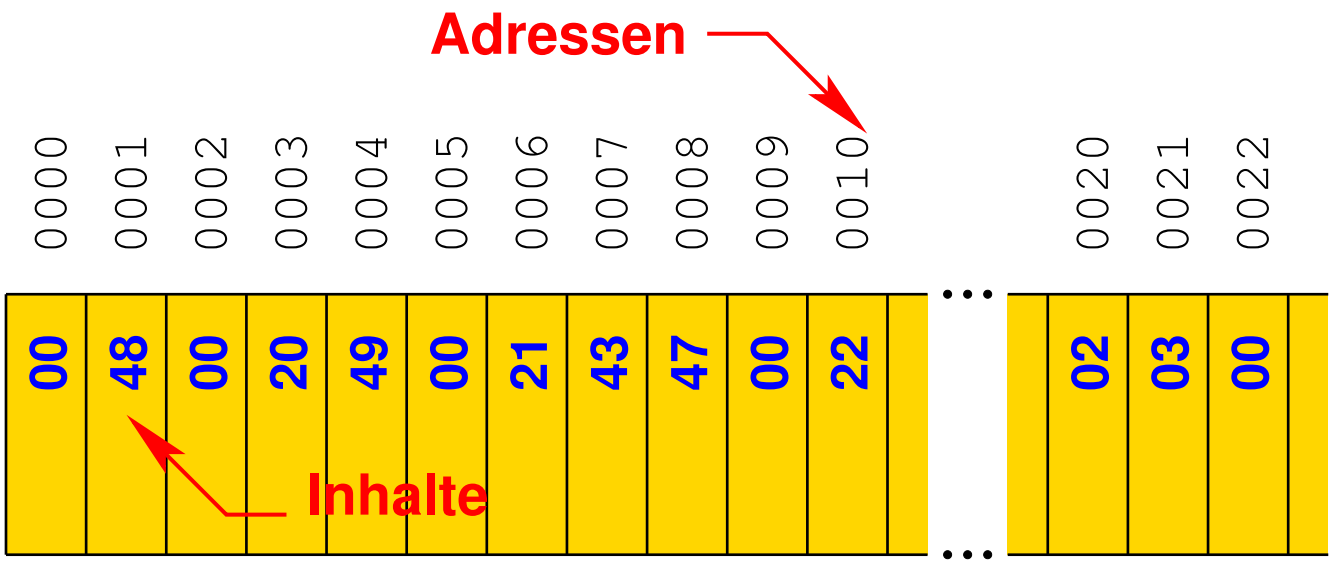
- Speicherzellen von 0 an fortlaufend durchnummeriert
- Jede Speicherzelle besitzt eine eigene *Adresse*, über die sie angesprochen wird
- In jeder Speicherzelle ist ein Zahlenwert (typischerweise 0 . . . 255 als Bitkette mit 8 Bit) abgelegt



Arbeitsspeicher

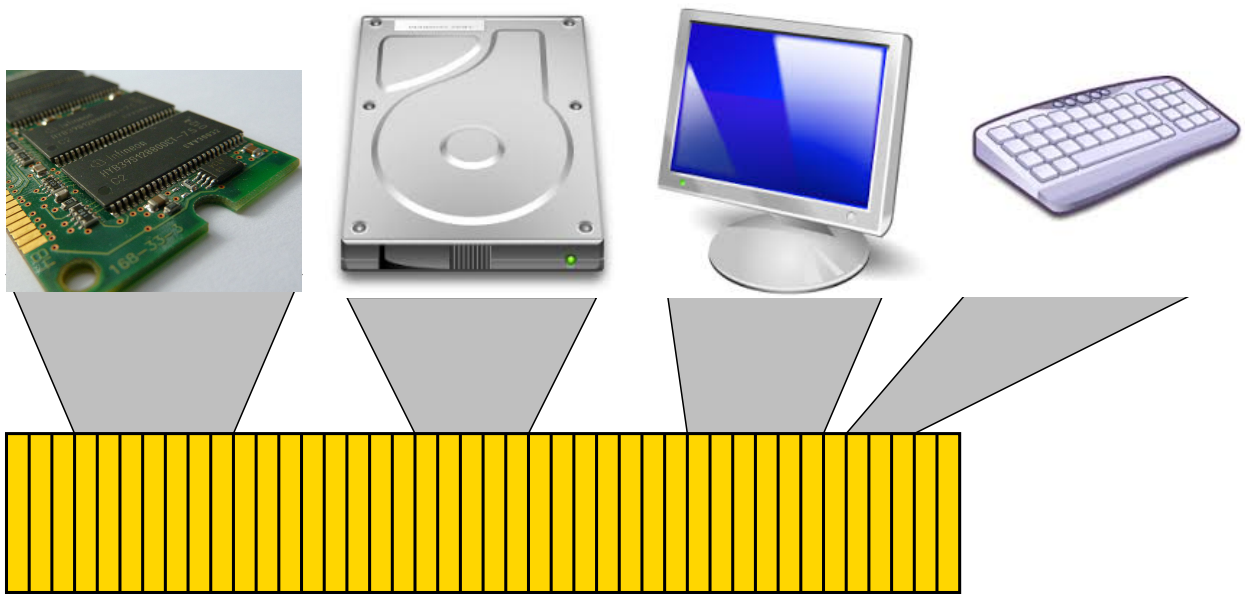
organisiert als virtueller Speicher

- Speicherzellen von 0 an fortlaufend durchnummeriert
- Jede Speicherzelle besitzt eine eigene *Adresse*, über die sie angesprochen wird
- In jeder Speicherzelle ist ein Zahlenwert (typischerweise 0 . . . 255 als Bitkette mit 8 Bit) abgelegt
- Beispiel: Bitkette 00110000 entspricht der Zahl
  $0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 48$



Arbeitsspeicher (II)

- Virtueller Speicher in Fragmente unterteilt
- Peripheriegeräte wie z.B. Festplatte eingebunden
- Physischer Speicher über Hardwarekomponenten verteilt
- Datenaustausch über ein *Bussystem* technisch realisiert

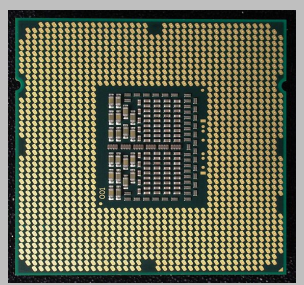


(Abbildung stark vereinfacht)

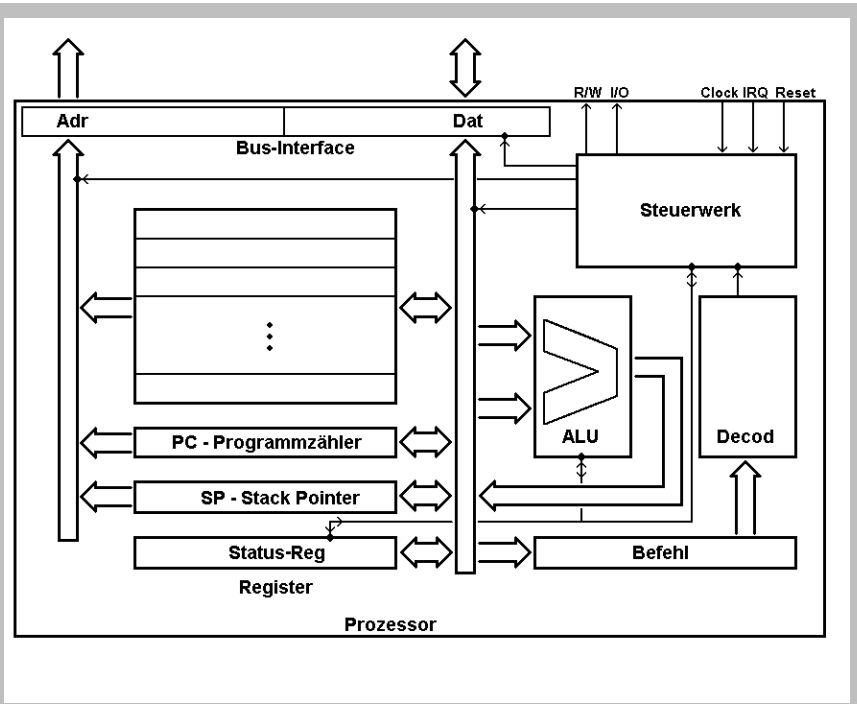
Hauptprozessor

Central Processing Unit, CPU

- kann den (virt.) Speicher gezielt auslesen und beschreiben
- besitzt einen *Befehlssatz* aus *Maschinenbefehlen*
- Jeder Prozessortyp hat eigenen Befehlssatz
  (leider kaum bis gar nicht kompatibel)

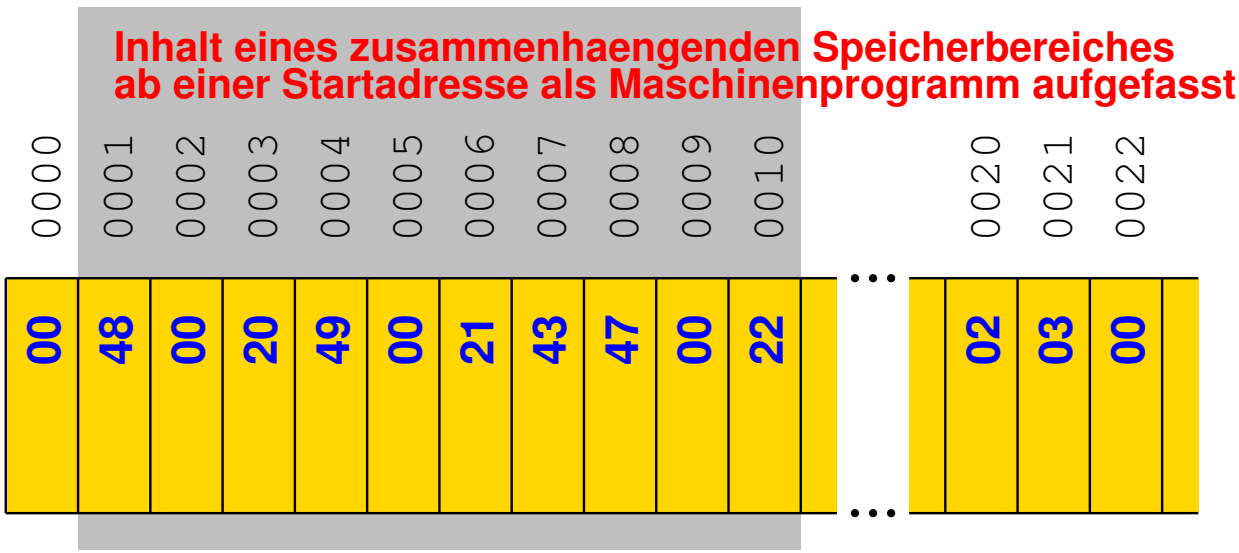


Intel i7
Ober– und Unterseite
www.wikipedia.de



Maschinenprogramm im Speicher

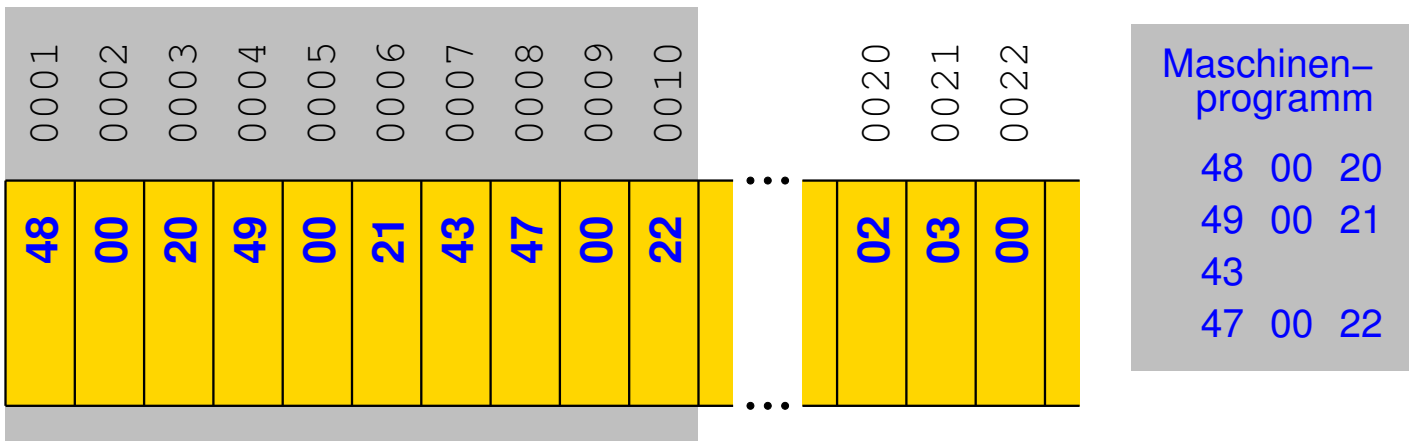
- Speicher enthält ausführbare Programme (`*.exe`) und Daten
- Jedes ausführbare Programm (Maschinenprogramm) besteht aus Maschinenbefehlen
- Startadresse des Maschinenprogramms ist klar definiert



Ein Beispiel-Maschinenprogramm

für einen fiktiven Prozessortyp

- Jeder im Befehlssatz definierte Befehl hat einen *Befehlscode*, eine fest zugeordnete Zahl (hier zwischen 40 und 49)
- Nach dem Befehlscode kann eine Anzahl von *Operanden* folgen, wobei durch den Befehl festgelegt wird, ob und wieviele Operanden es sind



Befehle des Beispiel-Maschinenprogramms

Programm zur Addition zweier Zahlen aus dem Speicher

Maschinenprogramm

```
48 00 20    LDA      $0020
49 00 21    LDB      $0021
43          ADD B
47 00 22    STORE A  $0022
```

Befehle des Beispiel-Maschinenprogramms

Programm zur Addition zweier Zahlen aus dem Speicher

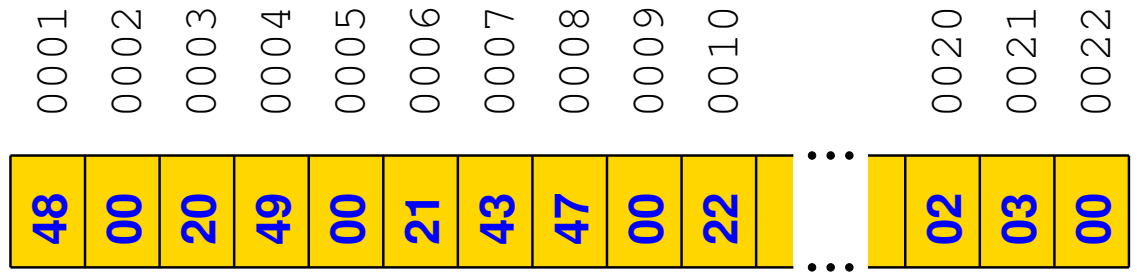
Maschinenprogramm

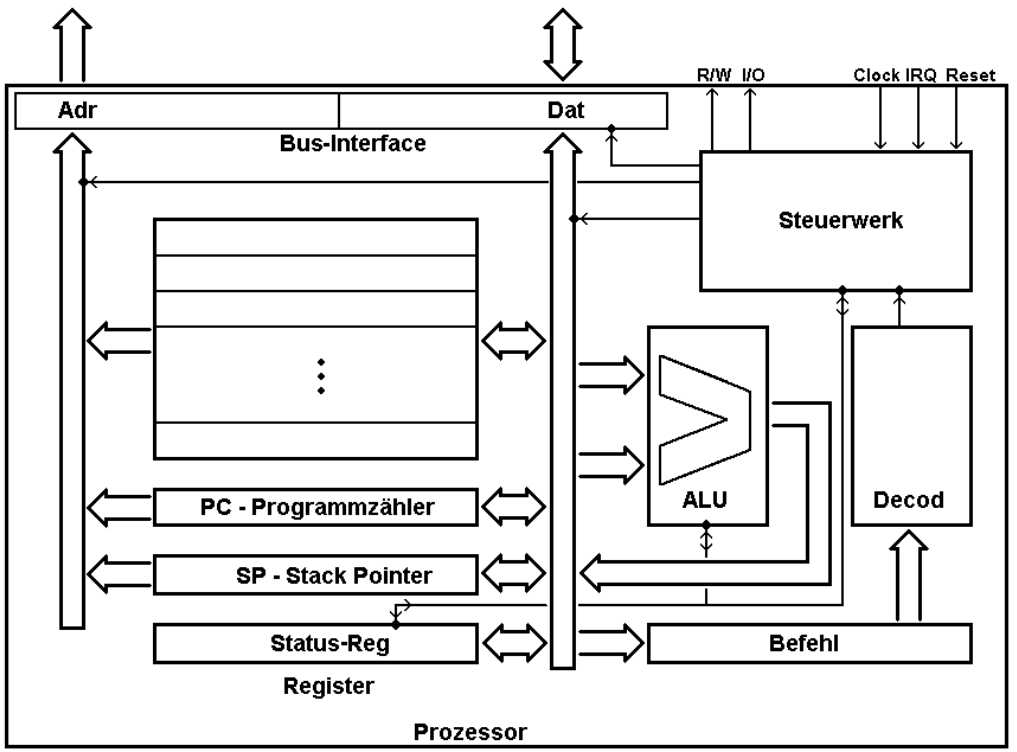
```
48 00 20    LDA       $0020
49 00 21    LDB       $0021
43          ADD B
47 00 22    STORE A   $0022
```

$\Longrightarrow$ Was bewirken die einzelnen Befehle und wie werden sie im Prozessor ausgeführt?

Programmstart

Programmzähler auf Startadresse setzen

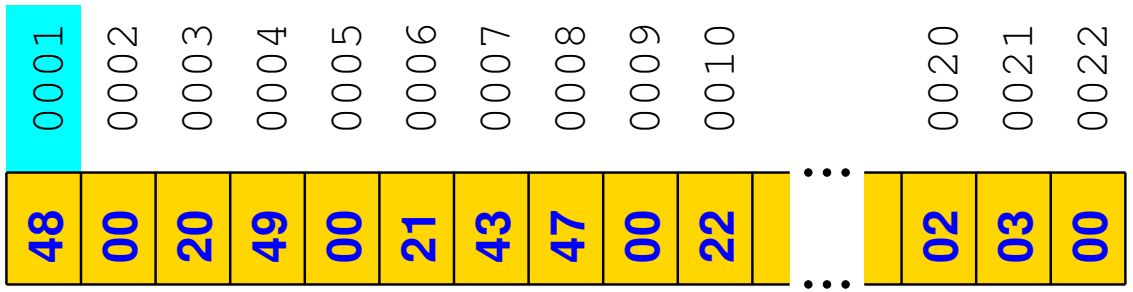


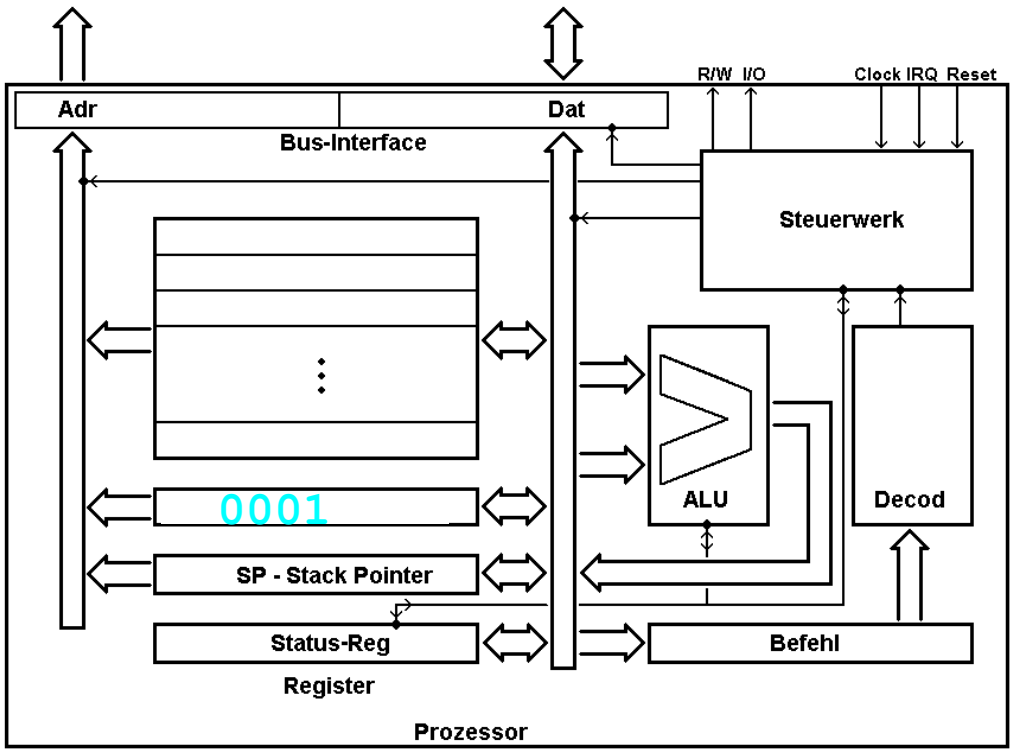



```
LDA     $0020
LDB     $0021
ADD B
STORE A $0022
```

Programmstart

Programmzähler auf Startadresse setzen

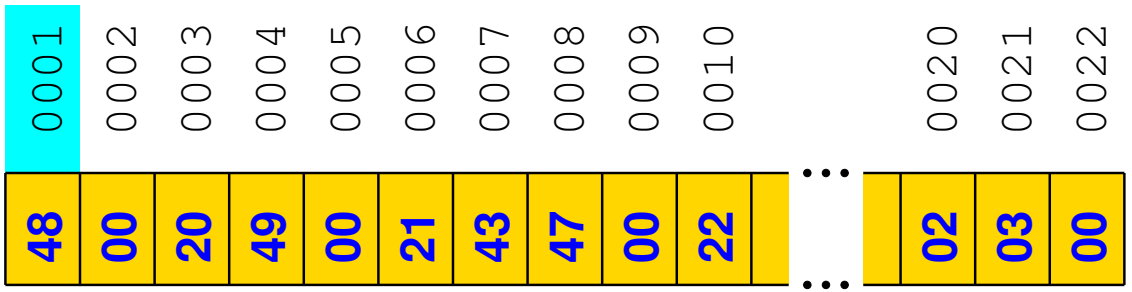




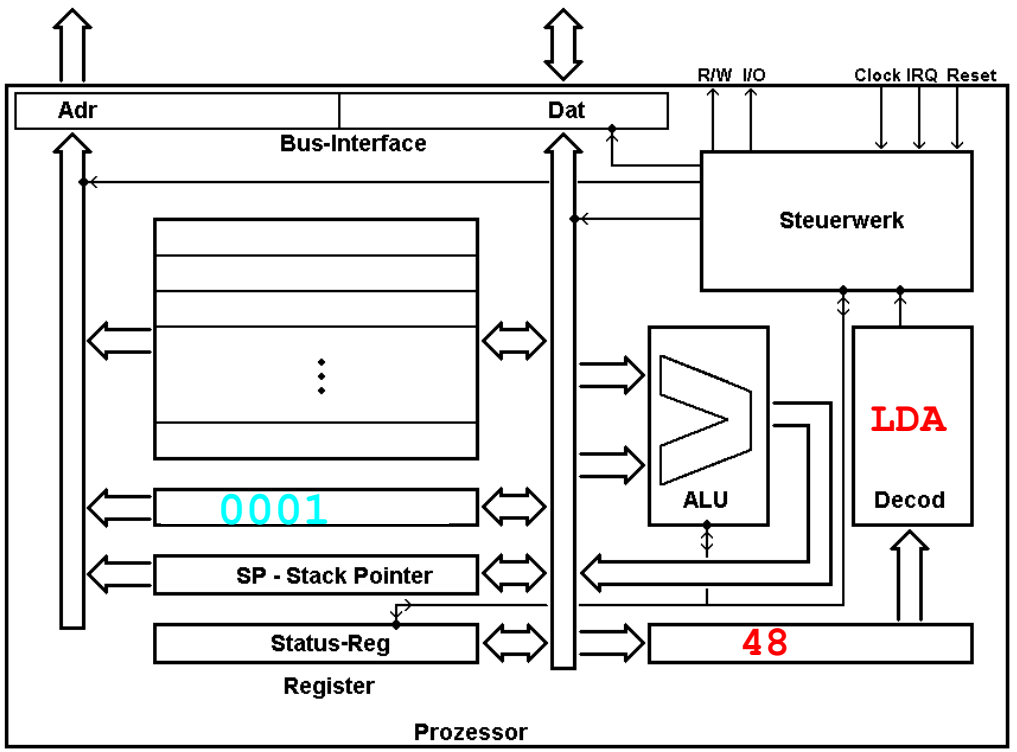

```
LDA     $0020
LDB     $0021
ADD B
STORE A $0022
```

Befehlsabarbeitung

Befehl lesen und decodieren

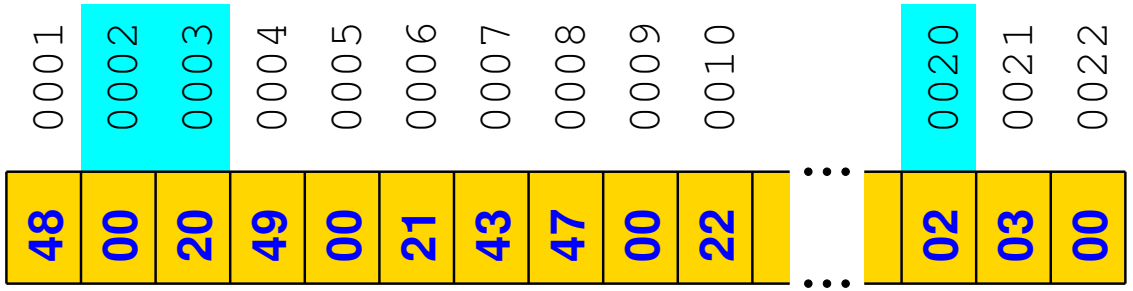



```
LDA     $0020
LDB     $0021
ADD B
STORE A $0022
```

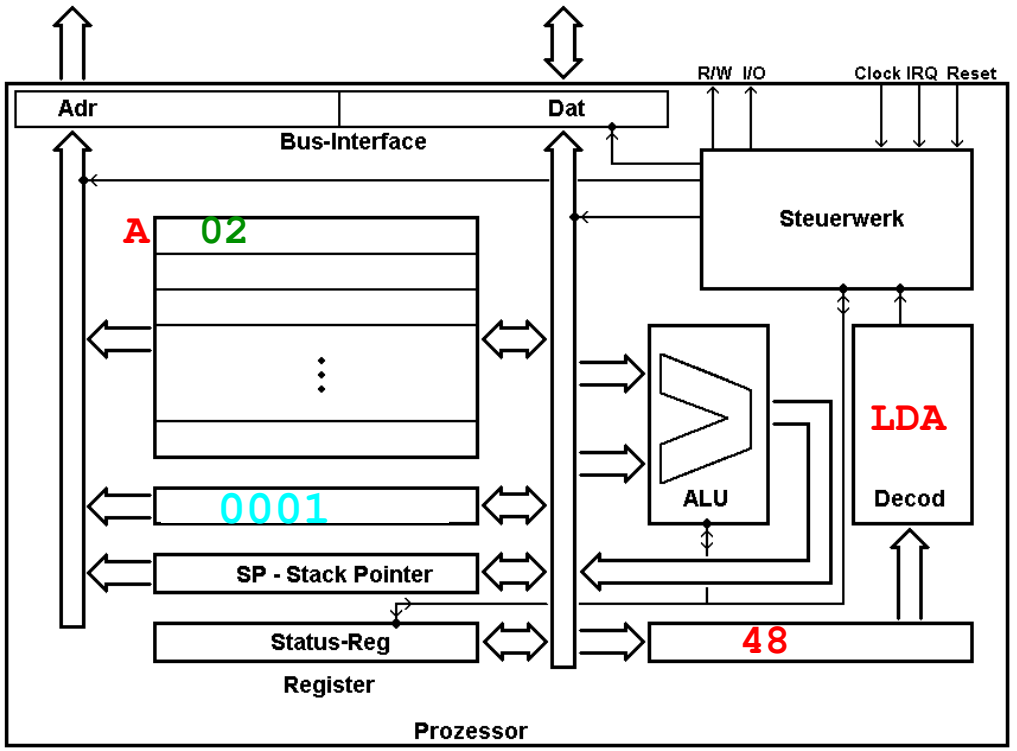


Befehlsabarbeitung

Operanden einlesen und im Register A ablegen

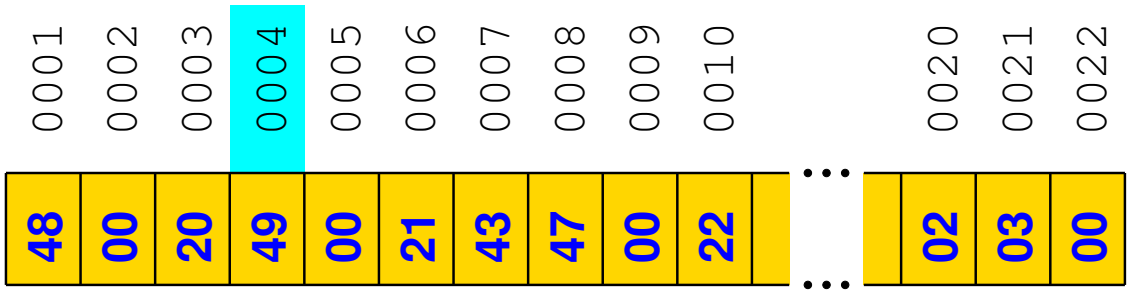



```
LDA     $0020
LDB     $0021
ADD B
STORE A $0022
```

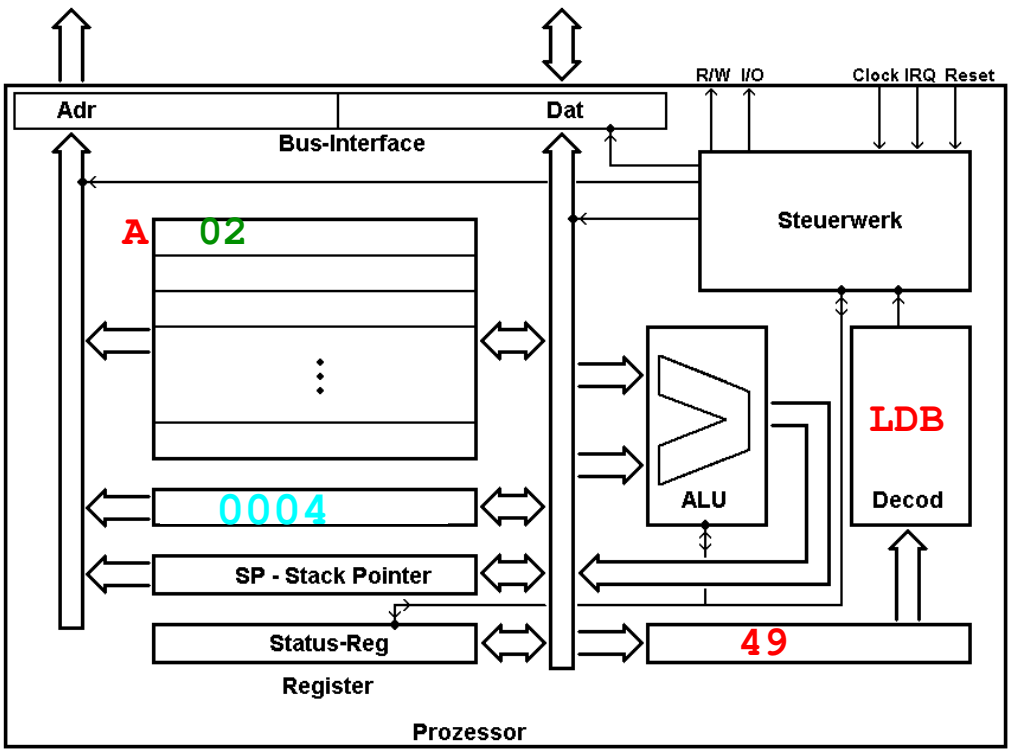


Befehlsabarbeitung

Befehl lesen und decodieren

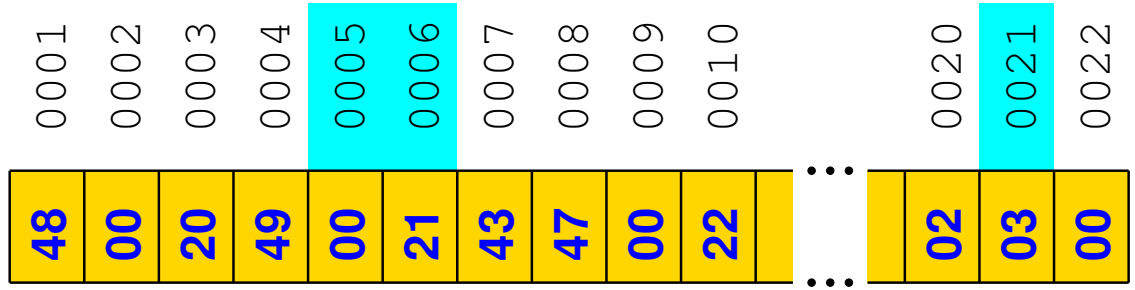


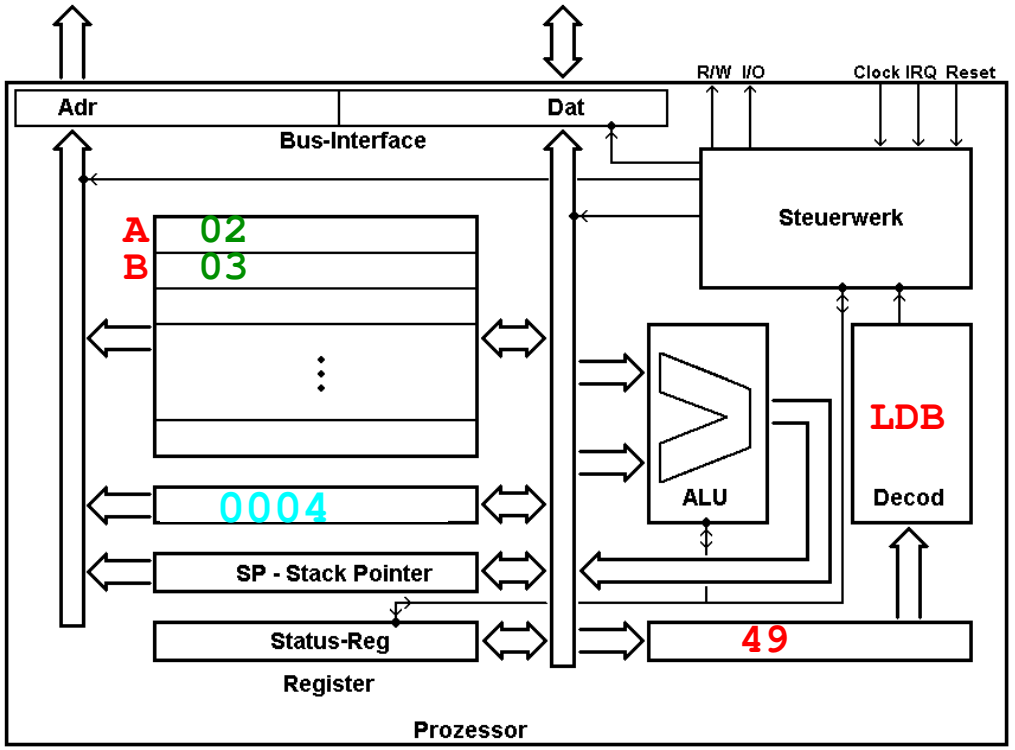

```
LDA     $0020
LDB     $0021
ADD B
STORE A $0022
```



Befehlsabarbeitung

Operanden einlesen und im Register B ablegen

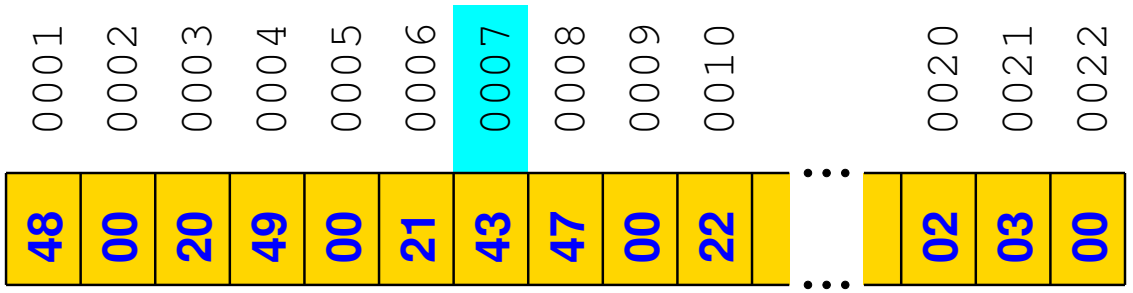




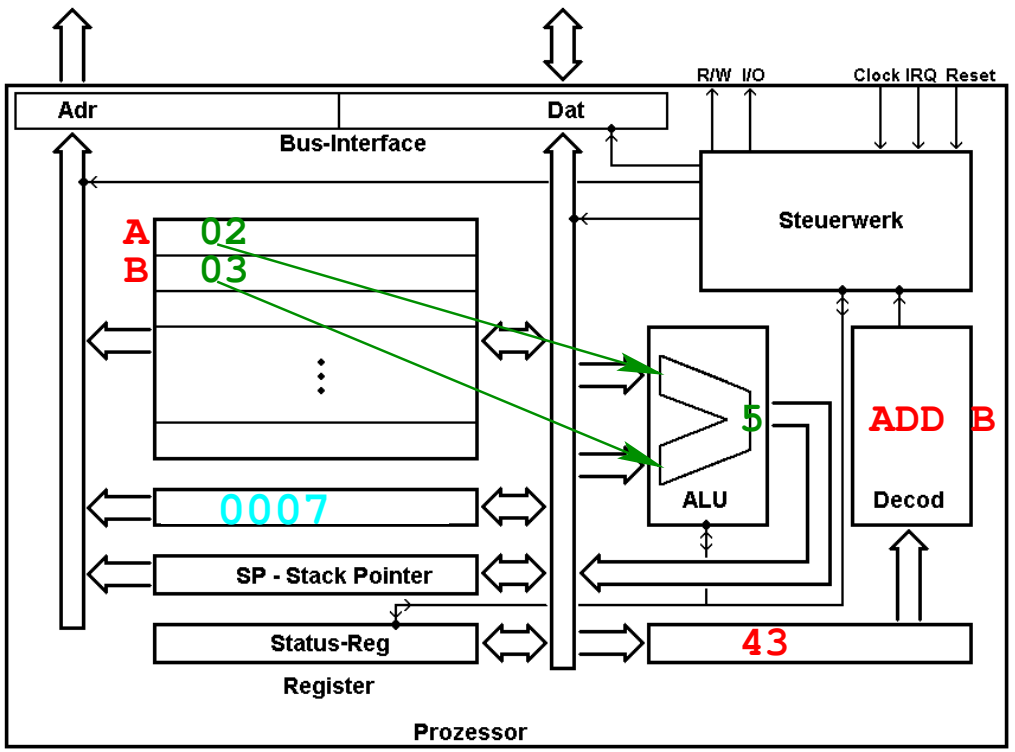

```
LDA     $0020
LDB     $0021
ADD B
STORE A $0022
```

Befehlsabarbeitung

Befehl lesen und decodieren




```
LDA     $0020
LDB     $0021
ADD B
STORE A $0022
```

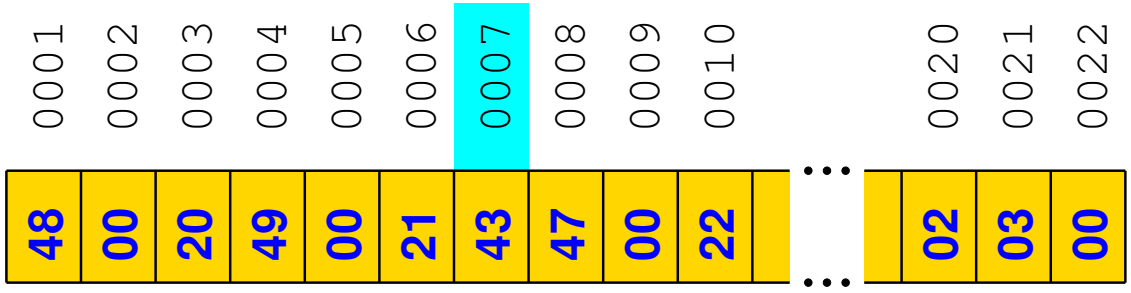


ALU:
Arithmetic
Logical
Unit

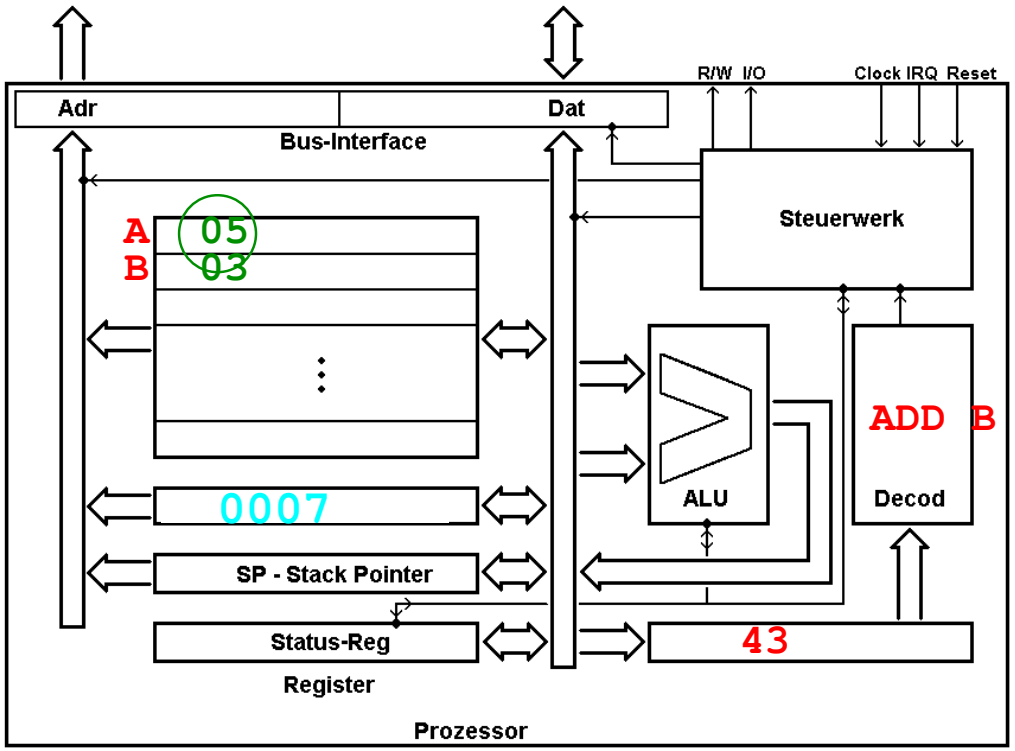
"2 + 3 = 5"

Befehlsabarbeitung

Register A und B addieren, Ergebnis in Register A ablegen




```
LDA     $0020
LDB     $0021
ADD B
STORE A $0022
```

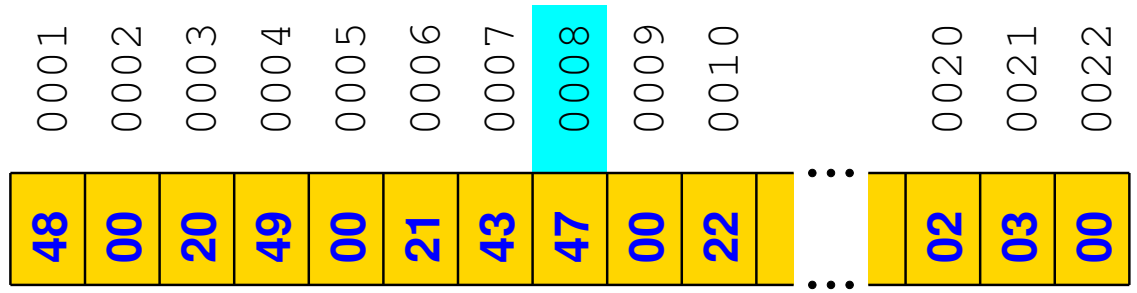


ALU:
Arithmetic
Logical
Unit

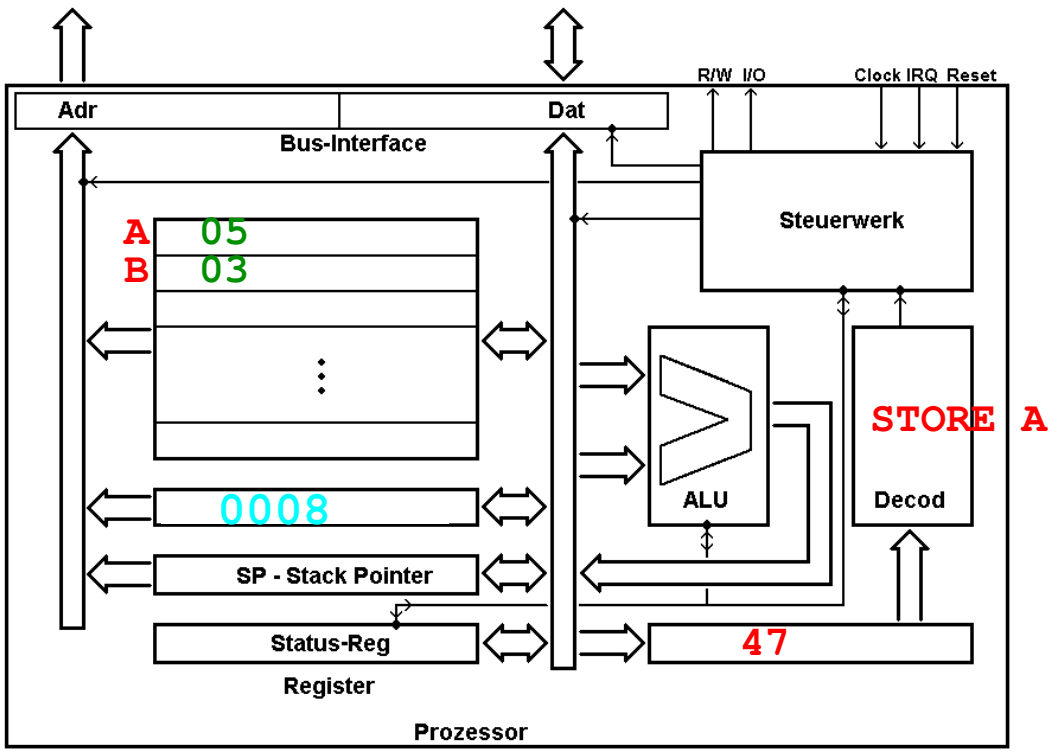
"2 + 3 = 5"

Befehlsabarbeitung

Befehl lesen und decodieren

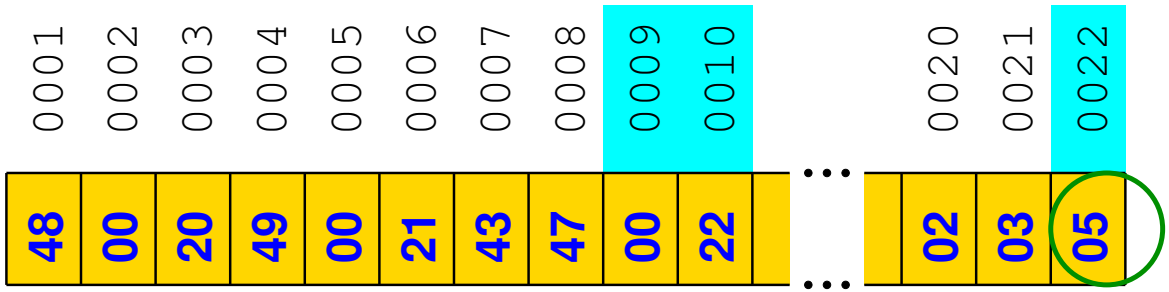



```
LDA     $0020
LDB     $0021
ADD B
STORE A $0022
```

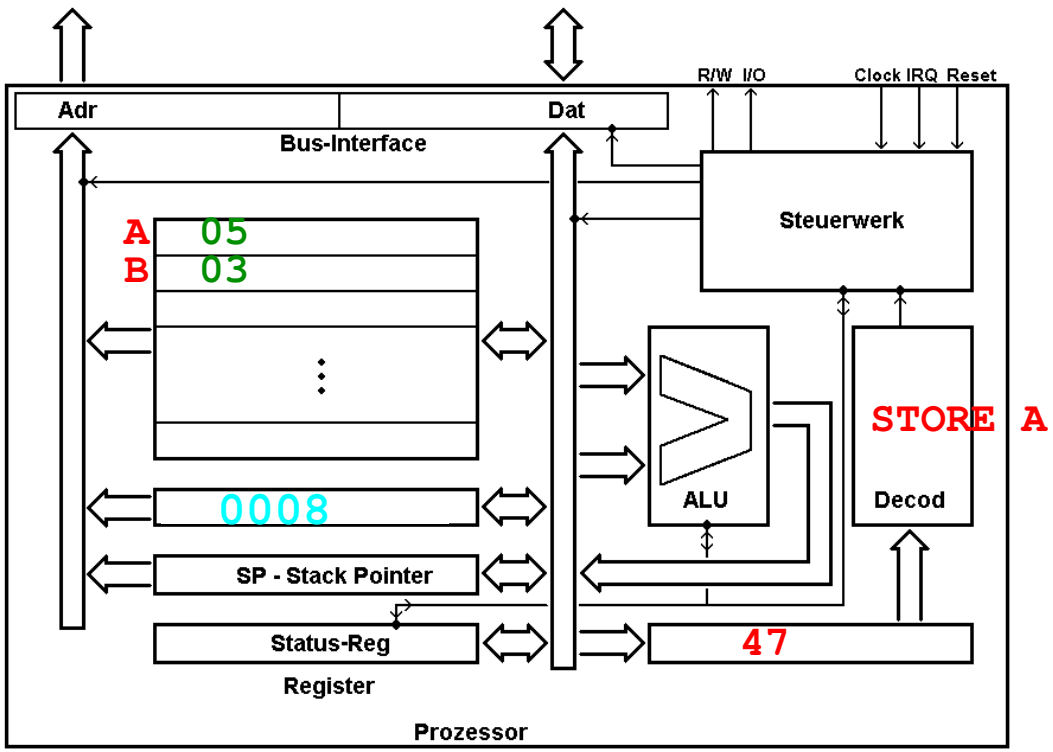


Befehlsabarbeitung

Operanden einlesen, Inhalt von A in Speicher schreiben




```
LDA     $0020
LDB     $0021
ADD B
STORE A $0022
```



Zentrale Befehlsschleife im Prozessor

1. Befehl lesen
2. Befehl decodieren
3. Operanden einlesen
4. Befehl ausführen
5. zum nächsten Befehl springen

Zentrale Befehlsschleife im Prozessor

1. Befehl lesen
2. Befehl decodieren
3. Operanden einlesen
4. Befehl ausführen
5. zum nächsten Befehl springen

- Befehlsschleife immer wieder durchlaufen
- Nach Programmende Fortsetzung der Befehlsschleife im Betriebssystem

Wollen wir auf dieser Ebene programmieren?

Wollen wir auf dieser Ebene programmieren?

Nein!

Wollen wir auf dieser Ebene programmieren?

Nein!

Der C-Compiler sorgt dafür, dass unsere C-Programme (Quelltexte) in entsprechende Maschinenprogramme umgewandelt werden, die dann auf dem genutzten Prozessor ausführbar sind.

Wollen wir auf dieser Ebene programmieren?

Nein!

Der C-Compiler sorgt dafür, dass unsere C-Programme (Quelltexte) in entsprechende Maschinenprogramme umgewandelt werden, die dann auf dem genutzten Prozessor ausführbar sind.

$\Longrightarrow$ Wir haben jetzt eine grobe Vorstellung davon, was im Computer passiert.

Wollen wir auf dieser Ebene programmieren?

Nein!

Der C-Compiler sorgt dafür, dass unsere C-Programme (Quelltexte) in entsprechende Maschinenprogramme umgewandelt werden, die dann auf dem genutzten Prozessor ausführbar sind.

$\Longrightarrow$ Wir haben jetzt eine grobe Vorstellung davon, was im Computer passiert.

In der Anfangszeit der Computertechnik (bis in die 1960er Jahre) wurden Maschinenprogramme von Hand erstellt („Assemblercode“), es gab damals noch keine (ausgereiften) Compiler.

Geschichte der Programmiersprache C (I)

1966 *Martin Richards* stellt am *Massachusetts Institute of Technology* (MIT) die **Basic Combined Programming Language** (BCPL) vor.

- BCPL kennt bereits lokale Variablen und Kontrollstrukturen, aber noch keine Typen.
- Zu jeder Variable muss vom Programmierer noch der Speicherort angegeben werden.

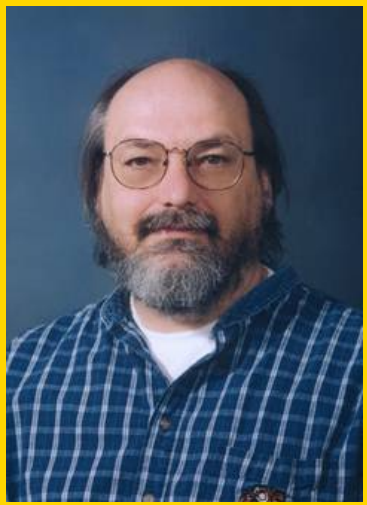
Martin Richards
www.cl.cam.ac.uk/~mr10/

```
GET "LIBHDR"

LET START () BE
$(WRITES ("Hello, world!*N") $)
```

Geschichte der Programmiersprache C (II)

1969 *Ken Thompson* und *Dennis Ritchie* präsentieren mit **B** eine Weiterentwicklung von BCPL, bei der der Speicherort von Variablen automatisch zugewiesen wird.

- Der Interpreter ist kürzer als 8 KByte (!).
- B unterscheidet keine Variablentypen.



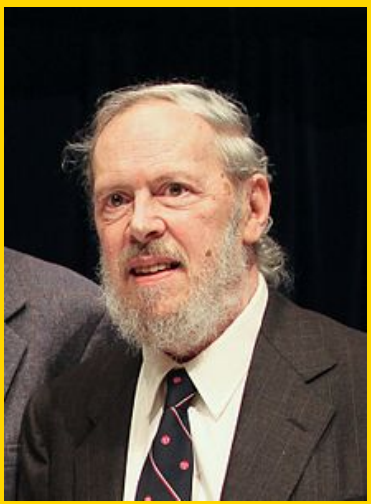
Ken Thompson

www.bell–labs.com

```
main() {
  auto c;
  auto d;
  d=0;
  while(1) {
    c=getchar();
    d=d+c;
    putchar(c);
  }
}
```

Geschichte der Programmiersprache C (III)

1972 *Dennis Ritchie* führt an den *Bell Laboratories* **C** als Weiterentwicklung von B ein. Erstmals gibt es unterschiedliche Typen für Variablen



Dennis Ritchie

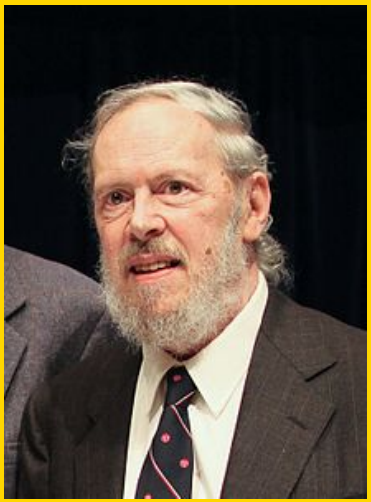
www.bell–labs.com

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

int main(void)
{
    printf("Hallo Welt!\n");
    return EXIT_SUCCESS;
} /* end main() */
```

Geschichte der Programmiersprache C (III)

1972 *Dennis Ritchie* führt an den *Bell Laboratories* **C** als Weiterentwicklung von B ein. Erstmals gibt es unterschiedliche Typen für Variablen

1988 Das *American National Standards Institute* (ANSI) normiert und standardisiert Sprachumfang und Syntax von C. Erster Standard: C90



Dennis Ritchie

www.bell–labs.com

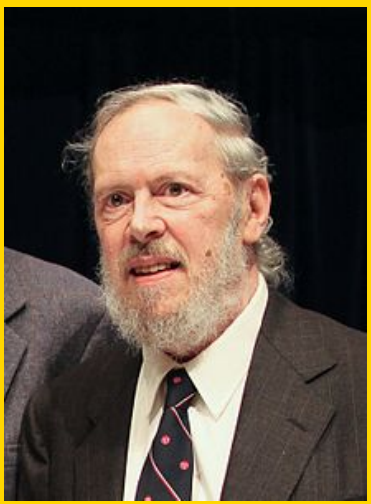
```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("Hallo Welt!\n");
    return EXIT_SUCCESS;
} /* end main() */
```

Geschichte der Programmiersprache C (III)

1972 *Dennis Ritchie* führt an den *Bell Laboratories* **C** als Weiterentwicklung von B ein. Erstmals gibt es unterschiedliche Typen für Variablen

1988 Das *American National Standards Institute* (ANSI) normiert und standardisiert Sprachumfang und Syntax von C. Erster Standard: C90

1999 Erweiterter Standard C99 mit Elementen aus C++



Dennis Ritchie

www.bell–labs.com

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

int main(void)
{
    printf("Hallo Welt!\n");
    return EXIT_SUCCESS;
} /* end main() */
```

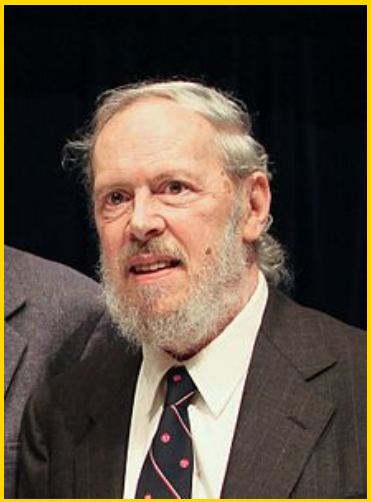
Geschichte der Programmiersprache C (III)

1972 *Dennis Ritchie* führt an den *Bell Laboratories* **C** als Weiterentwicklung von B ein. Erstmals gibt es unterschiedliche Typen für Variablen

1988 Das *American National Standards Institute* (ANSI) normiert und standardisiert Sprachumfang und Syntax von C. Erster Standard: C90

1999 Erweiterter Standard C99 mit Elementen aus C++

2011 Nochmals erweiterter Standard C11, umfasst jetzt u.a. Multithread-Handling und dynamische Speicherallokation



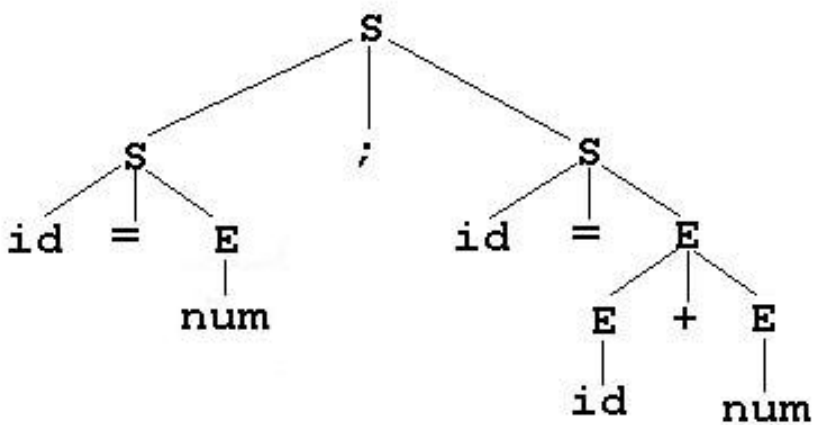
Dennis Ritchie

www.bell–labs.com

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

int main(void)
{
    printf("Hallo Welt!\n");
    return EXIT_SUCCESS;
} /* end main() */
```

Wie funktioniert ein Compiler?

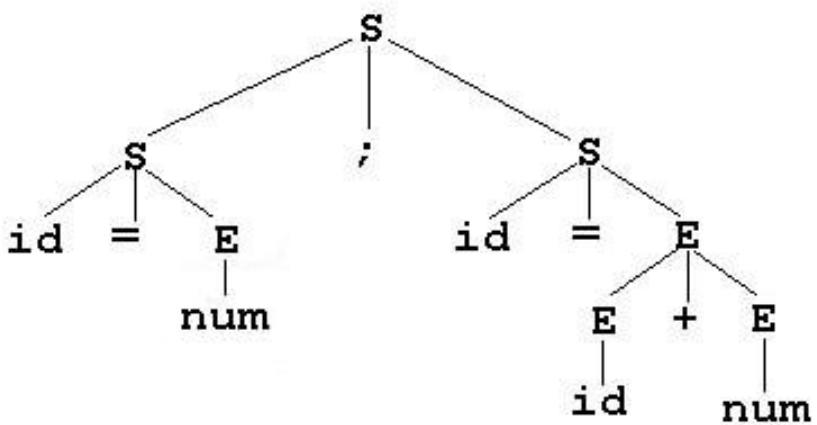
- compile $\longrightarrow$ engl. zusammentragen, zusammenführen



abstrakter Syntaxbaum für `id = num; id = id + num`

Wie funktioniert ein Compiler?

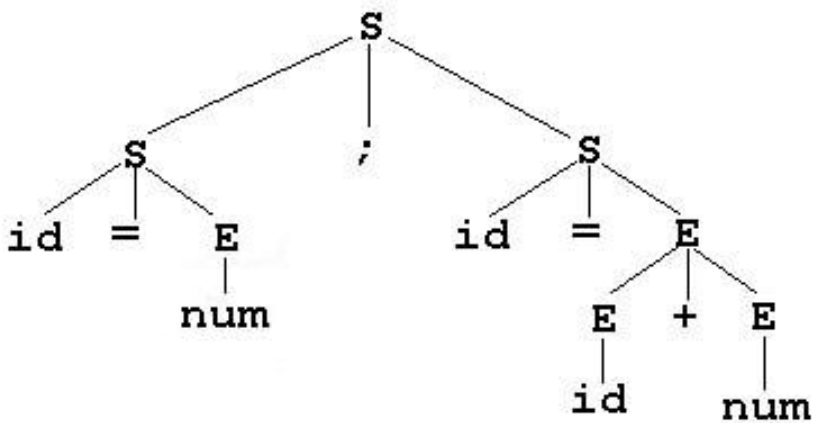
- compile $\longrightarrow$ engl. zusammentragen, zusammenführen
- übersetzt ein in einer Hochsprache geschriebenes Programm in
  - prozessorabhängigen
  - ausführbaren
  - Maschinencode (Bytecode)



abstrakter Syntaxbaum für `id = num; id = id + num`

Wie funktioniert ein Compiler?

- compile $\longrightarrow$ engl. zusammentragen, zusammenführen
- übersetzt ein in einer Hochsprache geschriebenes Programm in
  - prozessorabhängigen
  - ausführbaren
  - Maschinencode (Bytecode)
- Übersetzungsprozess gliedert sich in verschiedene Phasen, begleitet von Optimierungsschritten



abstrakter Syntaxbaum für `id = num; id = id + num`

Arbeitsschritte beim Compilieren

```
#include <stdio.h>

int main(void)
{
  int tag = 5;
  int monat = 11;
  int jahr = 2012;

  printf("Datum: %d.%d.%d\n",
           tag, monat, jahr);
  return 0;
}
```

datum.c

Arbeitsschritte beim Compilieren

1. **Lexikalische Analyse (Scanner):**
   Erkennung von *Token*

```c
#include <stdio.h>

int main(void)
{
  int tag = 5;
  int monat = 11;
  int jahr = 2012;

  printf("Datum: %d.%d.%d\n",
         tag, monat, jahr);
  return 0;
}
```

datum.c

Token:
Schlüsselwörter,
Bezeichner,
Zahlen,
Zeichenketten,
Operatoren

Arbeitsschritte beim Compilieren

1. **Lexikalische Analyse (Scanner):**
   Erkennung von *Token*
2. **Syntaktische Analyse (Parser):**
   Korrektheit der Deklarations- und Steuerstrukturen prüfen gemäß Grammatikdefinition

```c
#include <stdio.h>

int main(void)
{
  int tag = 5;
  int monat = 11;
  int jahr = 2012;

  printf("Datum: %d.%d.%d\n",
         tag, monat, jahr);
  return 0;
}
```

datum.c

Arbeitsschritte beim Compilieren

1. **Lexikalische Analyse (Scanner):**
   Erkennung von *Token*
2. **Syntaktische Analyse (Parser):**
   Korrektheit der Deklarations- und Steuerstrukturen prüfen gemäß Grammatikdefinition
3. **Semantische Analyse (Attributierung):**
   Typverträglichkeit der Variablen und Operatoren prüfen, Parametrisierung von Funktionen prüfen u.ä.

```c
#include <stdio.h>

int main(void)
{
  int tag = 5;
  int monat = 11;
  int jahr = 2012;

  printf("Datum: %d.%d.%d\n",
           tag, monat, jahr);
  return 0;
}
```

datum.c

Arbeitsschritte beim Compilieren

1. **Lexikalische Analyse (Scanner):** Erkennung von *Token*
2. **Syntaktische Analyse (Parser):** Korrektheit der Deklarations- und Steuerstrukturen prüfen gemäß Grammatikdefinition
3. **Semantische Analyse (Attributierung):** Typverträglichkeit der Variablen und Operatoren prüfen, Parametrisierung von Funktionen prüfen u.ä.
4. **Codeerzeugung:** maschinennahe Objektdatei durch Termersetzung gewinnen und optimieren

Maschinencode (hexadezimal)	zugehöriger Assembliercode	zugehöriger C-Code
55 48 89 E5	`push rbp` `mov rbp, rsp`	`int main() {`
C7 45 FC 02	`mov DWORD PTR [rbp-4], 2`	`int a = 2;`
C7 45 F8 03	`mov DWORD PTR [rbp-8], 3`	`int b = 3;`

8B 45 F8
8B 55 FC
01 D0
89 45 F4
	`mov eax, DWORD PTR [rbp-8]`
	`mov edx, DWORD PTR [rbp-4]`
	`add eax, edx`
	`mov DWORD PTR [rbp-12], eax`
		`int c = a + b;`

8B 45 F4	`mov eax, DWORD PTR [rbp-12]`	`return c;`
5D C3	`pop rbp` `ret`	`}`

Arbeitsschritte beim Compilieren

1. **Lexikalische Analyse (Scanner):** Erkennung von *Token*
2. **Syntaktische Analyse (Parser):** Korrektheit der Deklarations- und Steuerstrukturen prüfen gemäß Grammatikdefinition
3. **Semantische Analyse (Attributierung):** Typverträglichkeit der Variablen und Operatoren prüfen, Parametrisierung von Funktionen prüfen u.ä.

Maschinencode (hexadezimal)	zugehöriger Assembliercode	zugehöriger C-Code
55 48 89 E5	`push rbp` `mov rbp, rsp`	`int main() {`
C7 45 FC 02	`mov DWORD PTR [rbp-4], 2`	`int a = 2;`
C7 45 F8 03	`mov DWORD PTR [rbp-8], 3`	`int b = 3;`

8B 45 F8
8B 55 FC
01 D0
89 45 F4
	`mov eax, DWORD PTR [rbp-8]`
	`mov edx, DWORD PTR [rbp-4]`
	`add eax, edx`
	`mov DWORD PTR [rbp-12], eax`
		`int c = a + b;`

8B 45 F4	`mov eax, DWORD PTR [rbp-12]`	`return c;`
5D C3	`pop rbp` `ret`	`}`

4. **Codeerzeugung:** maschinennahe Objektdatei durch Termersetzung gewinnen und optimieren
5. **Linken:** Zusammenführen von Objektdatei(en) mit Laufzeitbibliothek zum ausführbaren Maschinenprogramm

Compiler

- Führt verschiedene Prüfungen zur *Korrektheit* des Quellcodes aus

Compiler

- Führt verschiedene Prüfungen zur *Korrektheit* des Quellcodes aus
- Verstöße gegen Syntaxregeln werden als *Syntaxfehler* gemeldet

Compiler

- Führt verschiedene Prüfungen zur *Korrektheit* des Quellcodes aus
- Verstöße gegen Syntaxregeln werden als *Syntaxfehler* gemeldet
- Bei formal korrekten, aber möglicherweise problematischen Anweisungen werden *Warnungen* angezeigt

Compiler

- Führt verschiedene Prüfungen zur *Korrektheit* des Quellcodes aus
- Verstöße gegen Syntaxregeln werden als *Syntaxfehler* gemeldet
- Bei formal korrekten, aber möglicherweise problematischen Anweisungen werden *Warnungen* angezeigt
- Bei korrektem Quellcode erzeugt der Compiler zu jedem Quellfile genau ein Objektfile

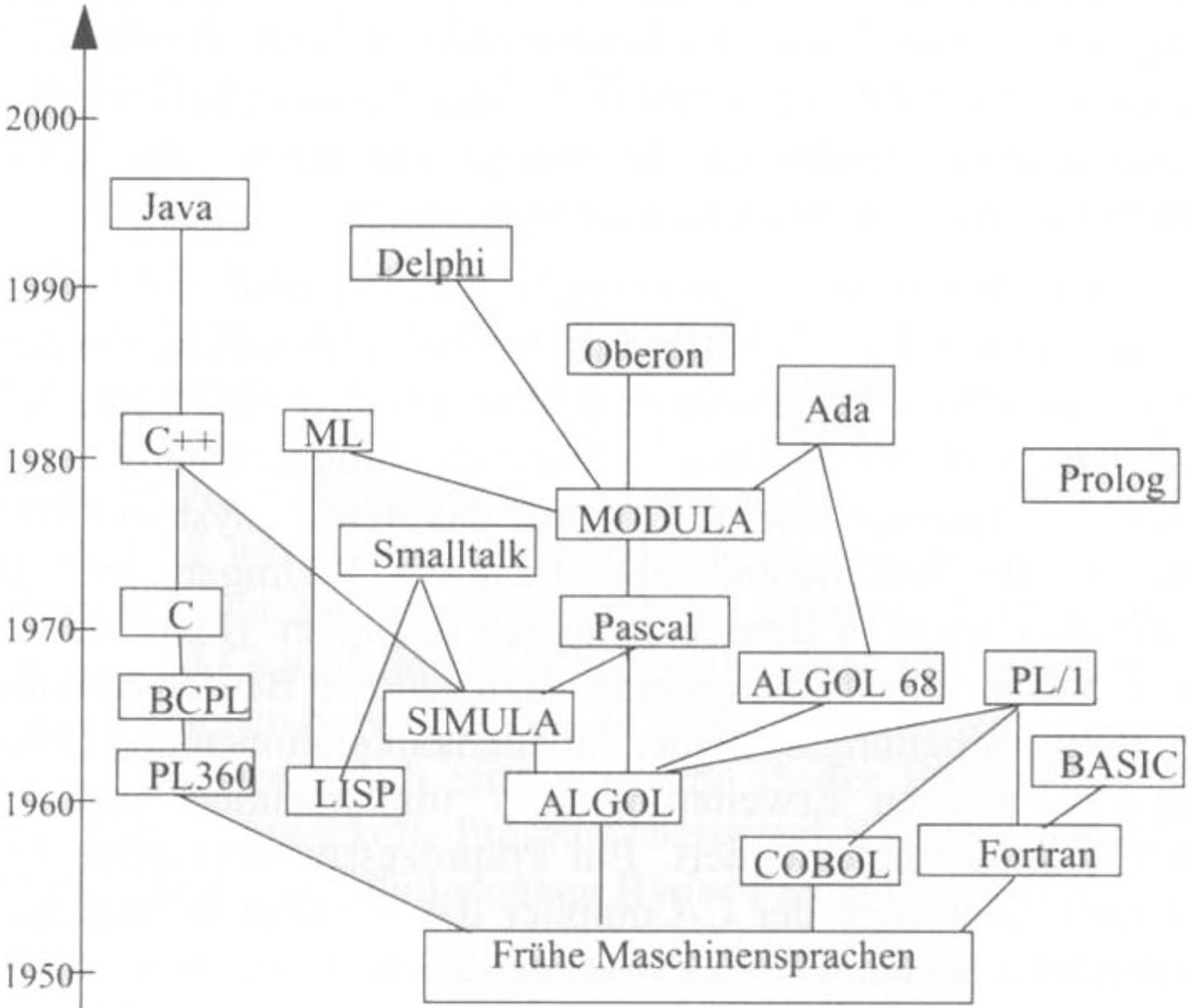
Compiler

- Führt verschiedene Prüfungen zur *Korrektheit* des Quellcodes aus
- Verstöße gegen Syntaxregeln werden als *Syntaxfehler* gemeldet
- Bei formal korrekten, aber möglicherweise problematischen Anweisungen werden *Warnungen* angezeigt
- Bei korrektem Quellcode erzeugt der Compiler zu jedem Quellfile genau ein Objektfile
- Ein C-Programm besteht i.d.R. aus mehreren Quellfiles

Compiler

- Führt verschiedene Prüfungen zur *Korrektheit* des Quellcodes aus
- Verstöße gegen Syntaxregeln werden als *Syntaxfehler* gemeldet
- Bei formal korrekten, aber möglicherweise problematischen Anweisungen werden *Warnungen* angezeigt
- Bei korrektem Quellcode erzeugt der Compiler zu jedem Quellfile genau ein Objektfile
- Ein C-Programm besteht i.d.R. aus mehreren Quellfiles
- Compiler kann nur die innere Korrektheit des Quellfiles überprüfen, er führt keine (oder nur rudimentäre) Prüfungen auf mögliche *Laufzeitfehler* (z.B. Division durch 0, Overflow, Speicherüberlauf) durch

Evolution der Programmiersprachen



Programmierparadigmen

Programmiersprachen nutzen verschiedene Grundkonzepte

Programmierparadigmen

Programmiersprachen nutzen verschiedene Grundkonzepte

Imperative Sprachen verwenden Abfolgen von **Befehlen**, die schrittweise Daten (Variablenwerte) verarbeiten. Das Programm legt fest, in welcher Reihenfolge welche Verarbeitungsschritte ausgeführt werden.
Beispiele: Fortran, Cobol, **C**, Basic, Perl, ...

Programmierparadigmen

Programmiersprachen nutzen verschiedene Grundkonzepte

Imperative Sprachen verwenden Abfolgen von **Befehlen**, die schrittweise Daten (Variablenwerte) verarbeiten. Das Programm legt fest, in welcher Reihenfolge welche Verarbeitungsschritte ausgeführt werden.
Beispiele: Fortran, Cobol, **C**, Basic, Perl, ...

Deklarative Sprachen **beschreiben** das Ergebnis. Daraus leitet ein Übersetzerprogramm die benötigten Verarbeitungsschritte und ihre Abfolge her.
Beispiele: funktionale (Haskell, ...) und logische Sprachen (Prolog, ...)

Programmierparadigmen

Programmiersprachen nutzen verschiedene Grundkonzepte

Imperative Sprachen verwenden Abfolgen von **Befehlen**, die schrittweise Daten (Variablenwerte) verarbeiten. Das Programm legt fest, in welcher Reihenfolge welche Verarbeitungsschritte ausgeführt werden.
Beispiele: Fortran, Cobol, **C**, Basic, Perl, ...

Deklarative Sprachen **beschreiben** das Ergebnis. Daraus leitet ein Übersetzerprogramm die benötigten Verarbeitungsschritte und ihre Abfolge her.
Beispiele: funktionale (Haskell, ...) und logische Sprachen (Prolog, ...)

Objektorientierte Sprachen sind (zumeist) eine Erweiterung imperativer Sprachen, bei denen Daten und auf ihnen operierende Methoden als eigenständige Einheiten (Objekte) behandelt werden. Objekte kommunizieren zum Datenaustausch miteinander.
Beispiele: C++, Java, JavaScript, Python, ...