Einführung in die Programmierung

MIT DER PROGRAMMIERSPRACHE C

PROF. DR. THOMAS GABEL

Überblick über die Vorlesung

1. Algorithmen, Programme und Software
2. **Einstieg in die Programmierung mit C**
3. Strukturiertes Programmieren in C
4. Effizientes Programmieren in C
5. Fortgeschrittene Aspekte der Programmierung in C

2. Einstieg in die Programmierung mit C

1. **Grundelemente eines C-Programms**
2. Variablen und Datentypen
3. Ausdrücke und Anweisungen
4. Grundlagen der Ein- und Ausgabe
5. Operatoren

Die Programmiersprache C

Geschichte / Ursprung

- Martin Richards entwickelt die Sprache BCPL
- BCPL beeinflusst 1970 die von Ken Thomspon entwickelte Sprache B.
- BCPL und B sind typenlose Sprachen und haben große Nähe zu Assembler.
- Dennis Ritchie arbeitete bei Bell Labs an dem Betriebssystem UNIX
  - 1972 Programmiersprache C
  - Unix wurde zu 95% in C geschrieben

Pro

- (relative) Low-Level-Programmierung
- geschwindigkeits- und speichereffizent
- portabel

Kontra

- unsicher (Sicherheitslücken)
- niedriger Abstraktionsgrad

Voraussetzungen

Was brauche ich, um ein **C-Programm** zu schreiben?

- Wissen, wie man programmiert
- Wissen, über welche Befehle die Programmiersprache verfügt

Besuch der Vorlesung

- einen Editor (Kwrite, Kate, vi, Emacs, Eclipse etc.)
- einen Compiler und Linker (gcc)
- viel Fleiß und Übung

Besuch der Übungen

Was ist ein **Compiler** (Übersetzer)?

- Der Compiler überprüft ein Programm auf seine syntaktische Korrektheit.
- Der Compiler übersetzt das Programm in Maschinen-Code.
- Der Compiler kann mit Hilfe des Linker fremde Bibliotheksfunktionen einbinden und ein ausführbares Programm erzeugen
- In Linux wird standardmäßig der **GCC** (GNU Compiler Collection) verwendet.
  - früherer Name: GCC = GNU C-Compiler
  - im Laufe der Zeit: Abdeckung weiterer Sprachen (C++, Fortran, etc.) durch GCC, daher die Umbenennung
  - Programmiersprachen entwickeln sich weiter, Sprachversionen existieren (C89, C99, C17, C23)
- Befehl: GCC-Aufruf zur Angabe seiner Version: `gcc -v`

Aufbau eines C-Programms

Struktur eines einfachen C-Programms

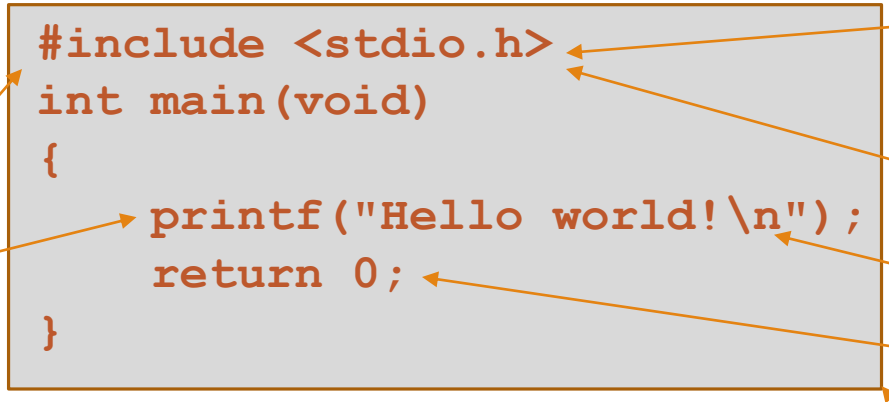
Textuelles Einfügen von Deklarationen und Definitionen aus anderen Dateien.

Bekanntmachung (Auflistung der Namen) aller noch kommenden Funktionen.

Dies wird bei Programmstart ausgeführt.

C – Hello World

Erzeuge die Datei `hello.c`, z.B. mit `kate`



Präprozessor-anweisung

Bildschirm-ausgabe

Standardein- und -ausgabe; dort ist bspw. printf definiert.

Angabe eines Pfades wäre auch möglich, z.B. `<../bin/stdio.h>`

Zeilenvorschub (new line)

Rückgabewert: 0

Datei hello.c

Übersetze das Programm

Eingabeaufforderung der Konsole (z.B. %)

```
% gcc hello.c -o hello
% ./hello
Hello world!
%
```

Ergebnis

- Der **Compiler** übersetzt das Programm in ein Object-Datei.
  - binärer Inhalt, d.h. nicht menschenlesbar, aber auch nicht ausführbar, d.h. nicht als Programm lauffähig
- Der **Linker** (Binder) fügt Objektdateien und Bibliotheken zu einen **ausführbaren Programm** (Binary, Executable) zusammen. Im obigen Beispiel macht `gcc` beides (in einem Schritt).

Vom Quellkode zum Programm

Kompilieren und Linken

- Das Kompilieren und Linken eines Programms erfolgt grundsätzlich mit dem Befehl

```
gcc [-parameter […]] dateiname […]
```

- Der Parameter `-o name` (output) bewirkt, dass das ausführbare Programm einen definierten Namen bekommt.
- Wenn diese Option (-o) fehlt, wird das Programm standardmäßig unter dem Namen `a.out` abgespeichert.
- Der Compiler übernimmt in diesem Fall das Linken gleich mit.

Beispiele:

- übersetze das Programm `hello.c` und benenne das erzeugte Programm `hello`

```
gcc hello.c -o hello
```

- übersetze `kurvendiskussion.c`, füge zusätzliche Debug-Informationen (g) in das erzeugte Program und füge die C-Mathe-Bibliothek (lm) dem Programm hinzu und benenne das erzeugte Porgramm `kurdis`

```
gcc -g -lm kurvendiskussion.c -o kurdis
```

Grundelemente der Sprache C

Verwendbare Zeichen in einem C-Programm

- alphanumerische Zeichen (Buchstaben und Ziffern)
- Leerzeichen
- Zeilenendzeichen (Semikolon als Kennzeichnung des Endes einer Anweisung)
- Blockbegrenzer (geschweifte Klammern für Beginn und Ende eines Blocks)
- Sonderzeichen ($, %, &, …)
- Steuerzeichen
- Kommentarzeichen (// sowie /* … */ um Freitextkommentare in den Quelltext einzufügen)

Bestimmte Zeichen dienen als **Operatoren**

- für arithmetische Grundoperationen (+, -, *, /, %)
- für diverse Formen von Vergleichen (!=, <=, …)
- für besondere Sprachmittel ([], (), ->, …)

Bezeichner

Definition: Unter **Bezeichnern** verstehen wir Namen, die für Elemente eines Programms (für Typen, für Variablen, für Funktionen etc.) vergeben werden.

Regeln:

- erstes Zeichen eines Bezeichners ist ein Buchstabe oder das Zeichen `_` (Unterstrich)
- die folgenden Zeichen dürfen Buchstaben, Ziffern oder Unterstriche sein
- es wird zwischen Groß- und Kleinschreibung unterschieden (`a` ist etwas Anderes als `A`)
- als Bezeichner nicht erlaubt sind reservierte Schlüsselwörter der Sprache

Definition: Unter **Schlüsselwörtern** verstehen wir Wörter mit vordefinierter Bedeutung, die nicht in einer anderen Bedeutung benutzt werden dürfen.

- Beispiel: Einleitung eines Schleifenkonstrukts mit `while`.
- `auto, break, case, char, const, continue`
- `default, do, double, else, enum, extern`
- `float, for, goto, if, int, long, register`
- `return, short, signed, sizeof, static, struct`
- `switch, typedef, union, unsigned, void, volatile, while`

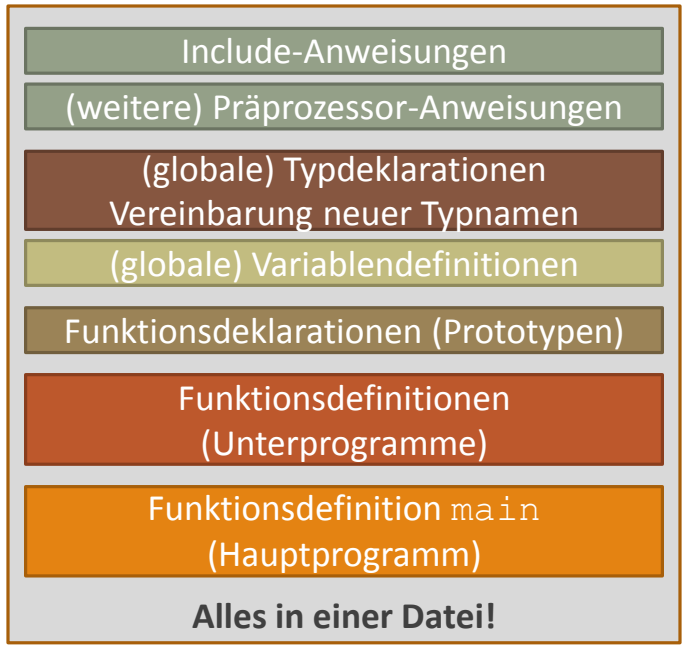
Funktionen (1)

C-Programme bestehen aus Funktionen.

- Ein C-Programm besteht aus mindestens einer Funktion.
- Die Funktionen führen die Aufgaben des Programms aus.

Funktionen

- nehmen Argumente als Eingabe entgegen
  ➔ Eingabewerte
- berechnen etwas
- geben ein Ergebnis zurück ➔ Rückgabewert



Regel: Für jedes ausführbare C-Programm muss genau eine Funktion mit dem Namen `main` existieren, die den Einstiegspunkt bezeichnet.

Beispiel:

- main-Funktion (Hauptprogramm)

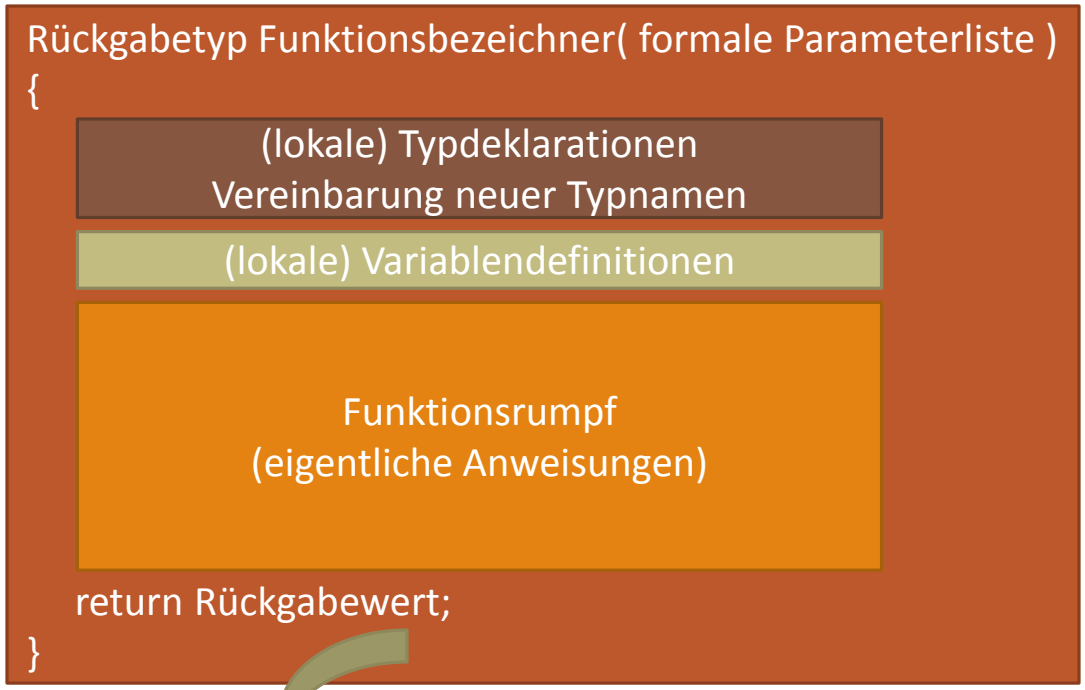
```
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

Funktionen (2)

Unterscheidung zwischen
- Funktionsdeklarationen
- Funktionsdefinitionen und
- Funktionsaufrufen

Funktionsdeklaration
- Bekanntmachung des Namens der Funktion inkl. Parameter und Art des Rückgabewertes
- auch: Funktionsprototypen genannt

Funktionsdefinition
- Ausimplementierung der Funktion
- Struktur einer Funktionsdefinition




```
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

Funktionsaufruf
- Aufruf einer Funktion von anderer Stelle des Programm (im Sinne eines Unterprogramms)
- Übergabe von Parametern, Entgegennahme eines zurückgegebenen Rückgabewertes

Kommentare in C-Programmen (1)

Kommentare sind grundsätzlich sehr wichtig!

- werden vom Compiler ignoriert, d.h. sie werden nicht mit übersetzt
- erhöhen die Wartbarkeit und Erweiterbarkeit des geschriebenen Quellcodes

Sinn:

- Erklärungen, wie Funktionen benutzt werden
- Erklärungen, wie Funktionen funktionieren
- Erklärungen von allem dessen, was nicht offensichtlich ist.

Wer liest Kommentare?

- jeder der den Code modifiziert
- der Programmierer in ein paar Wochen/Monaten/Jahren/Jahrzehnten

Später: Doxygen als nützliches Tool für Kommentare in C/C++

Kommentare in C-Programmen (2)

Beliebige erläuternde Texte können in C-Programmen eingefügt werden.

- zur Programmbeschreibung
- zur Erläuterung dessen, was der Programmierer sich gedacht hat

Regeln:

- einzeilige und mehrzeilige Kommentare werden unterschiedlich eingeleitet/beendet
- einzeilige Kommentare werden durch `//` eingeleitet und müssen stets hinter dem Quelltext stehen
- mehrzeilige Kommentare werden durch `/*` … `*/` eingegrenzt (dürfen auch einzeilig sein)

```
/* Dies ist ein Kommentar */

/*
   Kommentare koennen
   Mehrere Zeilen umfassen
*/

// Dies ist (seit C99, 1999) auch ein C-Kommentar!
```

Kommentare in C-Programmen (3)

Beispiel:

```
/*
 * area: finds area of circle
 * arguments: r: radius of circle
 * return value: the computed area
 */
double area(double r)
{
    double  pi = 3.1415926;
    return (pi * r * r);
}
```

Variablennamen

- sprechende Bezeichner

```
double x;           /* what does x mean? */
double distance;    /* better */
```

- … sind nicht immer notwendig

```
int loop_index;     /* bad */
int i;              /* good */
```

2. Einstieg in die Programmierung mit C

1. Grundelemente eines C-Programms
2. **Variablen und Datentypen**
3. Ausdrücke und Anweisungen
4. Grundlagen der Ein- und Ausgabe
5. Operatoren

Variablen und Datentypen (1)

Regel: Jegliche Daten in C haben einen Typ, ihren **Datentyp**.

- Ein Datentyp „sagt“ dem Rechner, wie der dem Befehl folgende Speicherinhalt interpretiert werden soll.
  - Z.B. als Repräsentation einer natürlichen Zahl oder als Zeichenkette
- Es gibt in C elementare und zusammengesetzte Datentypen.

Beispiele (elementare Datentypen):

- Typ zur Repräsentation ganzer Zahlen: `int`
- Typ zur Repräsentation von Zeichen (Buchstaben, Ziffern, Sonderzeichen): `char`
- Typen zur Repräsentation von Fließkommazahlen: `float, double`

Definition: **Variablen** werden genutzt, um Daten zu speichern.

- Die Variablen sind „das Gedächtnis“ eines Programms.
- Regel: Variablen **müssen** vor ihrer Verwendung deklariert werden.
- Stil der Variablendeklaration: `Datentyp Variablenname;`

Beispiel:

- `int a;`

Variablen und Datentypen (2)

Variablendeklarationen:

```
int i;          /* name = i  type = int */
char c;         /* name = c  type = char */
double d;
float some_float = 3.14;
```

Verwendete Bezeichner (Identifier): `i, c, d, some_float`

Optionale Deklaration mehrerer Variablen gleichen Typs:

```
int i, j, k2;
char c1, c2, c_2;
int i;   /* ERROR:
            Variable i already defined. */
```

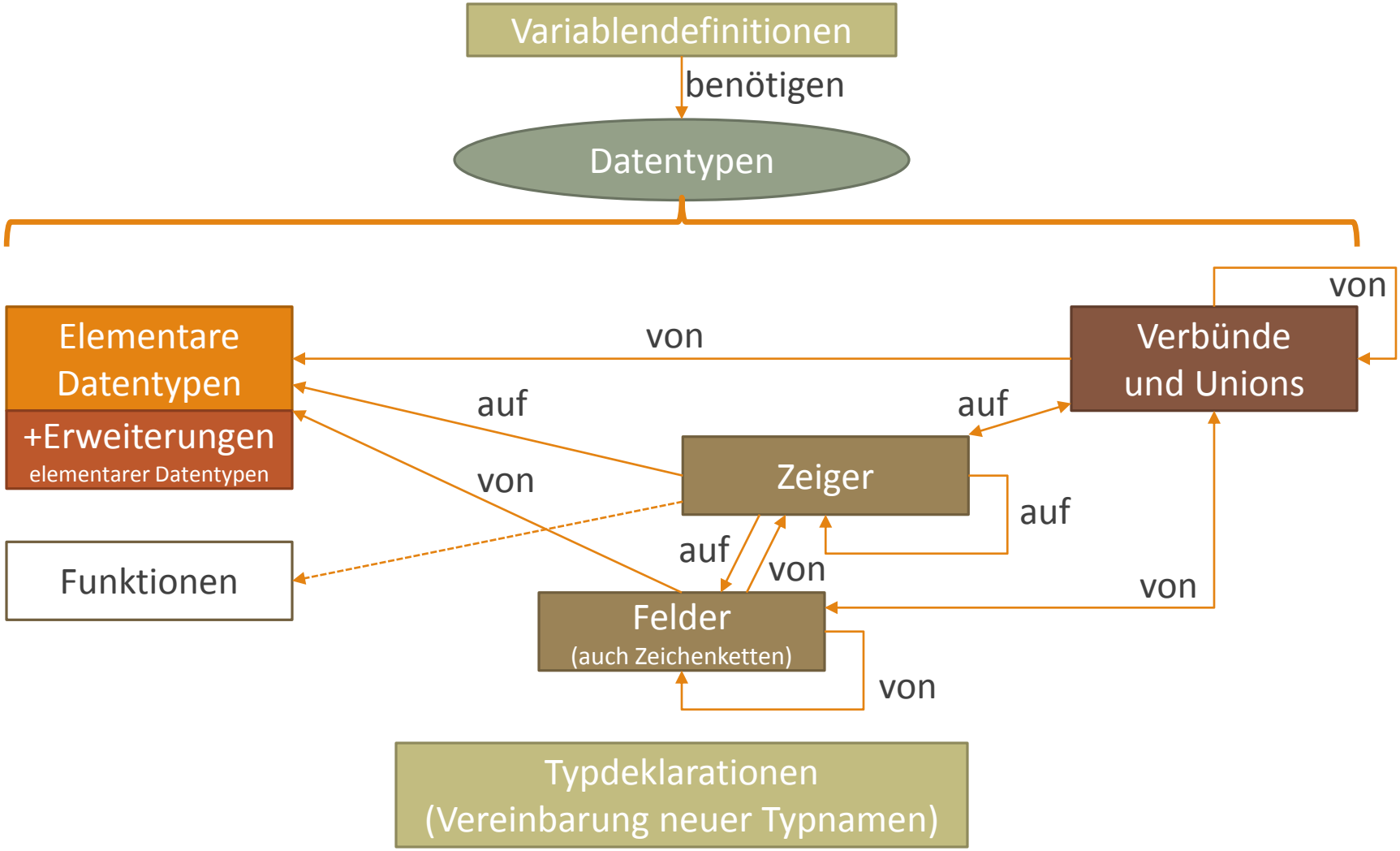
Optionale Initialisierung

- Variablen dürfen bei ihrer Deklaration mit einem Initialwert belegt werden (`some_float`).

Definition: Unter **Konstanten** verstehen wir Variablen, die einen festen Wert haben, d.h. die ihren Wert zur Laufzeit des Programms nie ändern.

- für ihre Deklaration steht das Schlüsselwort `const` zur Verfügung
- Beispiel: `const int value = 42;`

Überblicksfolie: Datentypen in C



Elementare Datentypen (1)

Frage: Welchen Speicherplatz (im Hauptspeicher) beansprucht eine Variable eines bestimmten Types?

- Der Operator **`sizeof`** liefert als Wert die Länge einer Variablen eines beliebigen Datentyps in Byte.
- z.B.: **`sizeof( int )`** → 4 Byte

Elementare Datentypen im Überblick

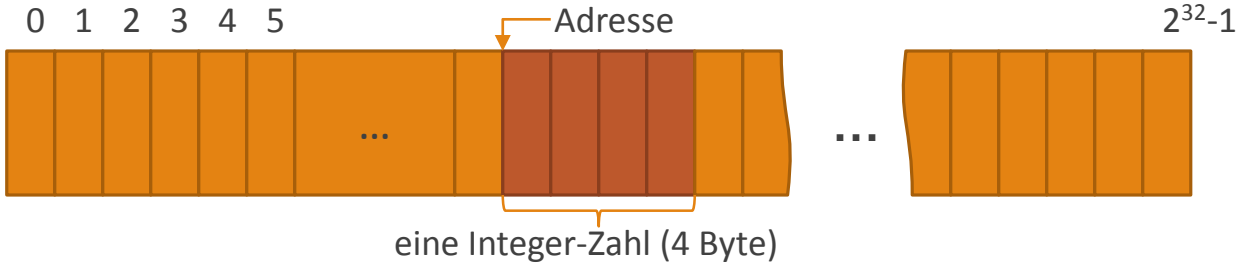
Datentyp	Wertebereich	Größe (in Byte)	Bedeutung
`char`	$-2^7+1 \ldots 2^7-1 = -127 \ldots 127$	1	ein Zeichen
`int`	$-2^{31}+1 \ldots 2^{31}-1$	4	ganze Zahl
`float`	ca. $3.4*10^{38}$ (7-8 Stellen Genauigkeit)	4	reelle Zahl
`double`	ca. $1.7*10^{308}$ (15-16 Stellen Genauigkeit)	8	reelle Zahl
`void`			leere Menge von Werten

- Anmerkung: exakte Werte abhängig von Rechnerarchitektur und Compiler

Zwischenschub: Speicherorganisation

(Einfache) Speicherorganisation

- Hauptspeicher besteht aus Zellen
- jede Zelle ist 1 Byte groß
- Speicherzellen sind linear angeordnet und fortlaufend nummeriert
  ➔ jede Zelle hat eine Adresse (eine Nummer)



- erforderlicher Speicherplatz zum Ablegen von Variablen: mehrere benachbarte Speicherzellen
  - z.B. 4 Byte (also 4 Zellen) für eine Integer-Variable
- Speicherverwaltung übernehmen das Betriebssystem und der Compiler
  - nicht der Benutzer
- Speicheradressenverwaltung: mit 4 Byte (also 32 Bit) lassen sich 2^{32} Speicherplätze adressieren ➔ $2^{32} = 4x2^{30} = 4$ Gigabyte (Adressen von 0 bis 2^{32}-1)

Elementare Datentypen (2)

Datentyp `char`

- dient zur Darstellung eines Zeichens wie `'a'` oder `'d'` (ASCII-Zeichentabelle).
  - Wichtig: Verwendung von Hochkommata zur Darstellung im Quellkode
- kann aber auch zur Darstellung der Zahlen von -128…127 verwendet werden
- Vorteil: Arithmetische und logische Operationen sind auf ASCII-Zeichen durchführbar
- Sonderregel für nicht darstellbare Sonderzeichen: `\Sonderzeichen`
- Beispiel:

```
printf("Hallo Welt!\n");
```

Datentyp `int`

- dient der Darstellung der ganzen Zahlen (arithmetische Operationen)
- unterstützt auch logische Operationen
  - TRUE bedeutet 1 bzw. alle Werte, die nicht 0 sind
  - FALSE ist 0
- Unterstützung verschiedener Zahlensysteme
  - Dezimalsystem (Basis 10) standardmäßig
  - Oktalsystem (Basis 8) durch Voranstellen von 0 → z.B. 046 wird als 38 erkannt
  - Hexadezimalsystem (Basis 16) durch Voranstellen von 0x → z.B. 0x4B wird als 75 erkannt

Elementare Datentypen (3)

Fließkommatypen `float` und `double`

- `float` mit 7-8 Stellen Genauigkeit reicht für kaufmännische Berechnungen nicht aus
- `double` mit 15-16 Stellen Genauigkeit unterstützt Verarbeiten von zweistelligen Millionenbeträgen ohne Fehler
- `long double` mit 20 Stellen geeignet für astronomische Berechnungen

Darstellung von Fließkommazahlen im Quellkode

- als Dezimalzahl mit Punkt:

```
float a = 3.14;
float b = .42;
float c = 42.0;
```

- als Ganzzahl mit Exponenten (e oder E):

```
float d = 1e6;  // 10^6
float e = 5E-1; // 5*10^-1=0.5
float f = -3e-2;// -3*10^-2=-0.03
```

- Kombination beider Varianten:

```
float g = 0.314e1;
float h = 31.4E-1;
```

Elementare Datentypen (4)

Leerer Typ `void`

- `void` hat keinen Typ und auf ihm sind keinerlei Operationen definiert
- Verwendung
  - Funktionen, die keinen Parameter haben
  - Funktionen, die keinen Wert zurückgeben
  - generische (allgemeine) Zeiger, die explizit in einen Zeiger eines bestimmten Typs umgewandelt werden müssen
- Beispiele:

```
void myFunction1( int a )
{
  // ...
}

int myFunction2( void )
{
  // ...
}
```

Erweiterungen elementarer Datentypen

Durch Angabe zusätzlicher Schlüsselwörter können die elementaren Datentypen modifiziert werden.

- **`signed`** und **`unsigned`**: Angabe, ob nur positive oder sowohl positive als auch negative Zahlen dargestellt werden sollen (mit oder ohne Vorzeichen)
  - verwendbar für **`int`** und **`char`**
  - **`signed`** ist die Standardeinstellung
- **`short`** und **`long`**: verkleinert bzw. vergrößert den Wertebereich
  - beide verwendbar für **`int`**
  - für **`double`** ist nur **`long`** verwendbar (Vergrößerung des Wertebereichs)

Datentyp / Variable	Wertebereich	Größe (in Bytes)
`unsigned char`	0 … 255	1
`unsigned int`	0 … 2^{32}-1	4
`short int`	$-2^{15}+1$ … $2^{15}-1$	2
`long int`	$-2^{63}+1$ … $2^{63}-1$	8
`long double`	$1.9*10^{-4951}$ … $1.1*10^{4932}$	16

- abhängig von Rechnerarchitektur (konkrete Werte für x86_64)

2. Einstieg in die Programmierung mit C

1. Grundelemente eines C-Programms
2. Variablen und Datentypen
3. **Ausdrücke und Anweisungen**
4. Grundlagen der Ein- und Ausgabe
5. Operatoren

Ausdrücke in C

Definition: Ein **Ausdruck** ist eine Folge von Operanden und Operatoren.

- Operanden können sein: Konstanten, Variablen, Funktionsaufrufe oder andere Ausdrücke
- Jeder Ausdruck hat einen Wert (einen Rückgabewert).

Definition: Durch **Operatoren** werden Operanden zu komplexeren Ausdrücken verknüpft. Ein Operator ist bestimmt durch

- seine Funktion (das was er tut)
- die Anzahl der Operanden, auf die er angewendet wird
- die Datentypen seiner Operanden
- den Datentyp seines Rückgabewertes (seines Ergebnisses)

Beispiel:

- `i + 2 * j` ➔ ist ein Ausdruck (hat einen Wert)
- `+` und `*` sind darin Operatoren; `i`, `2` und `j` sind Operanden
- Frage: Was ist der Wert des Ausdrucks, wenn i aktuell den Wert 3 hat und j den Wert 5?

Anweisungen und Zuweisungen

Definition: **Anweisungen** sorgen für die grobe Struktur eines Programmteils und geben diesem den logischen und funktionalen Überbau. Eine C-Anweisung wird durch ein Semikolon abgeschlossen.

Arten von Anweisungen:

- leere Anweisung: `;`
  - tut nichts, ist aber auch nicht verboten
- Ausdrucksanweisung: `i + 2 * j - 10;`
  - Auswertung der Operatoren (hier: Addition, Multiplikation, Subtraktion) mit den entsprechenden Operanden erfolgt
  - nicht sinnvoll, denn: Was passiert mit dem Ergebnis?
- Zuweisung: `i = j * k;`
  - Wert des Ausdrucks auf der rechten Seite wird der Variablen auf der linken Seite zugewiesen!
  - Achtung: Eine Zuweisung ist auch ein Ausdruck (und zwar einer, der den Wert des zugewiesenen Wertes hat!).
- Kontrollanweisungen: Steuern den Programmfluss → später
  - Es gibt verschiedene Kontrollanweisungen in C, die wir allesamt in einem Folgekapitel kennenlernen werden.

Beispiele:

```
int i = 10;
int j = 20;
i = 2 + i * j;
j = j % 2;
```

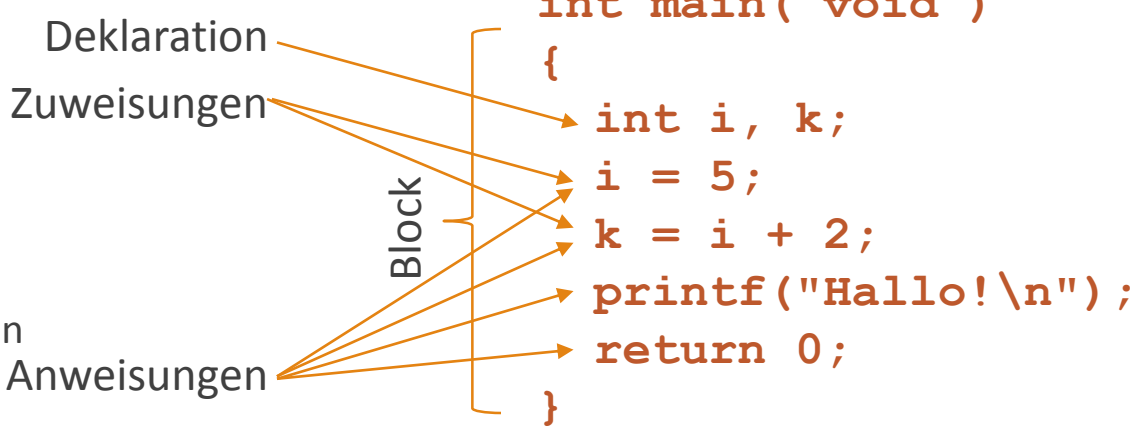
Blöcke

Definition: Eine Zusammenfassung von Deklarationen (z.B. Variablendeklarationen) und Anweisungen bezeichnet man als **Block**.

- Ein Block wird in C immer in geschweiften Klammern ({ … }) eingeschlossen.
- Die Anweisungen in einem Block werden sequentiell abgearbeitet.
- Jede Funktion repräsentiert einen Block (und kann weitere Unterblöcke enthalten).

Beispiel:

- Frage: Welche Operatoren kommen im Beispiel vor?
- Frage: Was sind im Beispiel Operanden?
- Frage: Was sind Ausdrücke im Beispiel?
  - Antwort: `k=i+2;` ist auch ein Ausdruck, und zwar einer; der den Wert von k hat.



2. Einstieg in die Programmierung mit C

1. Grundelemente eines C-Programms
2. Variablen und Datentypen
3. Ausdrücke und Anweisungen
4. **Grundlagen der Ein- und Ausgabe**
5. Operatoren

Ein- und Ausgabe

Wir haben bereits eine Zeile ausgegeben:

```
printf("Hallo!\n");
```

Die Funktionen für die Ein- und Ausgabe sind in der Datei `stdio.h` deklariert.

- Will man Ein- und Ausgabe in seinem Programm verwenden, müssen die dazu verfügbaren Funktionen bekannt gemacht und dafür die Datei `stdio.h` zu Programmbeginn inkludiert werden.

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

Unter dem Begriff **formatierte Ein-/Ausgabe** verstehen wir, dass Daten in Übereinstimmung mit ihren Datentypen ein-/ausgegeben werden.

- formatierte Eingabe: mit `scanf()`
- formatierte Ausgabe: mit `printf()`

Bemerkung:

- Die Ein- und Ausgabe einzelner Zeichen ist auch möglich, aber verglichen mit der formatierten Ein-/Ausgabe umständlicher in der Benutzung.
- Hier: nur 3 Folien zu den Grundlagen der Ein-/Ausgabe; mehr Details in VL-Kapitel 3

Ausgabe mit `printf()` (1)

Die Ausgabe mit der Funktion `printf()` dient der formatierten Ausgabe von Variablen und Konstanten (Grundtypen).

- Wenn die Ausgabe nicht „umgelenkt“ wird, so erfolgt sie nach `stdout`, also standardmäßig auf die Konsole / das Terminal, in dem das Programm gestartet wurde.
- Syntax: **`printf(formatstring [, parameter]);`**

Der **Formatstring** besteht aus

- dem Ausgabetext
- und ggf. den Formatanweisungen

Text im Formatstring wird unverändert wiedergegeben.

Formatanweisungen bestehen aus einem Prozentzeichen und einem (oder zwei) Zeichen, das den Datentyp der auszugebenden Variable spezifiziert.

- Beim Ausgeben wird jede Formatanweisung mit einem der Parameter der `printf()`-Anweisung besetzt.
- Eine Formatanweisung ist also nur ein Platzhalter.
- Beispiel: **`printf("Berechneter Wert: %d", v);`**

Ausgabe mit `printf()` (2)

Ausgabe auf den Bildschirm:

```
int a = 5;
float pi = 3.14159;
char c = 'T';
printf("Ergebnisse: a = %d, pi = %f, c = %c\n", a, pi, c);
```

Substitution erfolgt: Ersetzen von `%d`, `%f` und `%c` durch Werte

- `%d` und `%i` integer (dezimal)
  - `%u` unsigned integer
- `%f` und `%lf` Fließkommazahl (floating point, float und double)
- `%x`, `%X` hexadezimale Zahl
- `%c` einzelne Zeichen (Character, char)
- `%s` Zeichenkette (String, d.h. ein Feld von Zeichen, char-Array)
- `%p` Zeiger (pointer)
- `%%` Ausgabe des Prozentzeichens (%)
- `\n` neue Zeile, `\\` für Rückstrich (Bachslash), `\"` für Anführungsstriche

Eingabe mit `scanf()`

`scanf()` liest von der Konsole analog `printf()` zur Ausgabe

„Ungewöhnliche“ Syntax …

```
int val;
scanf("%d", &val);
```

`scanf()` ändert den Wert der Variablen in der Liste der Argumente!

Beachte das `&val` in `scanf()`!

Genaue Erklärung später, wenn wir über Zeiger (Pointer) reden.

- Regel: `&` ist notwendig, wenn `int`, `doubles`, etc. gelesen werden, aber nicht bei Zeichenketten (Strings).

Beispiel

Nutzung der Ein- und Ausgabe:

```
#include <stdio.h>

int main(void)
{
  int summand1, summand2, summe;
  printf("Geben Sie den ersten Summanden ein: ");
  scanf("%d", &summand1);
  printf("Geben Sie den zweiten Summanden ein: ");
  scanf("%d", &summand2);
  summe = summand1 + summand2;
  printf("Die Summe von %d und %d ist %d.\n", summand1,
    summand2, summe );
  return 0;
}
```

Der C-Präprozessor

Frage: Was macht folgende Zeile

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

Vor dem Übersetzen des Programms wird der Präprozessor aktiv und behandelt alle Zeilen startend mit #

- hier: Einfügen von Funktionsdefinionen aus einer Bibliothek (Standardein-/ausgabe)
- Es wird nicht die Implementierung der Funktion eingefügt!
- konkret: Einbindung von `stdio.h` ermöglicht, die Funktion `printf()` zu benutzen.

Der Linker fügt die Implementierung (in Form einer Object-Datei) hinzu.

Zusätzlicher Schritt beim Übersetzen (vgl. Folie „Vom Quellkode zum Programm“)

1. cpp: Präprozessor
2. gcc: Übersetzen (kompilieren) zu Object-Code
3. gcc (ld): Linken (Zusammenfügen mehrerer Object-Code-Dateien zum ausführbaren Programm)

➔ `gcc` erledigt hier alles für uns

2. Einstieg in die Programmierung mit C

1. Grundelemente eines C-Programms
2. Variablen und Datentypen
3. Ausdrücke und Anweisungen
4. Grundlagen der Ein- und Ausgabe
5. **Operatoren**

Operatoren (1)

In C lassen sich Operatoren funktional in folgende Gruppen unterteilen:

- arithmetische Operatoren
- Zuweisungsoperatoren
- Inkrement- und Dekrementoperatoren
- relationale (vergleichende) Operatoren
- logische Operatoren
- bitweise Operatoren
- sonstige Operatoren

Es existieren 5 arithmetische Operatoren, die immer zweistellig sind.

- D.h. sie operieren auf zwei Operanden und liefern ein Ergebnis.
- Jeder Operand muss ein Ausdruck sein! A und B sind im Folgenden Ausdrücke …
- Additionsoperator → `A + B`
- Subtraktionsoperator → `A - B`
- Multiplikationsoperator → `A * B`
- Divisionsoperator → `A / B` (bei Integer-Division: Nachkommastellen werden abgeschnitten)
- Restwertoperator → `A % B` (nur für char und int definiert)

Operatoren (2)

Zuweisungsoperatoren weisen das Ergebnis der rechten Seite der linken Seite zu.

- Wenn die Operanden unterschiedlichen Datentypen angehören, aber solchen Datentypen die (mit mehr oder weniger „Verlusten“ ineinander umgewandelt werden können, sogenannte kombatible Datentypen), so erfolgt eine automatische Konvertierung.
  - Beispiel: Zuweisung eines Fließkommawertes (3.14) an eine Integer-Variable (Ganzzahl).
- Beispiel: **`A = B`**

Kombinationen aus Zuweisungsoperator und arithmetischem Operator

- Additionszuweisung → **`A += B`** (entspricht `A = A + B`)
- Subtraktionszuweisung → **`A -= B`** (entspricht `A = A - B`)
- Multiplikationszuweisung → **`A *= B`** (entspricht `A = A * B`)
- Divisionszuweisung → **`A /= B`** (entspricht `A = A / B`)
- Restwertzuweisung → **`A %= B`** (entspricht `A = A % B`)
- weitere Zuweisungskombinationsoperatoren existieren (&=, |=, <<=, >>=, etc.)
- Vorteil: kurzer, schneller Code, weil der Variablenwert nur einmal ermittelt werden muss

Operatoren (3)

Inkrement- und Dekrementoperatoren

- Erhöhen bzw. Verringern eines Variablenwertes um 1
- Variante 1: Postfix-Inkrementoperator **A++** und Postfix-Dekrementoperator **A--**
  - Rückgabewert ist der Wert von A
  - Nach Bestimmung des Rückgabewertes wird A um 1 erhöht/verringert.
  - Beispiel:

```
int a, b;
a = 5;   // a is 5 now
b = a++; // b is now 5, too.
         // a is also incremented to 6.
```

- Variante 2: Präfix-Inkrementoperator **++A** und Präfix-Dekrementoperator **--A**
  - Zunächst wird der Wert von A um eins erhöht/verringert.
  - Erst dann wird der Wert von A zurückgegeben.
  - Beispiel:

```
int a, b;
a = 5;   // a is 5 now
b = ++a; // a is incremented to 6.
         // a's value is returned and assigned
         // to b. So, b is 6 as well.
```

Operatoren (4)

Beispiele zu Inkrement-/Dekrementoperatoren

- ++ und −− können als Präfix und Postfix verwendet werden

Achtung: Unterschiedliche Bedeutung

```
int a = 0;
a++;  /* OK */
++a;  /* OK */
```

Oben haben beide Ausdrücke die gleiche Bedeutung, aber

```
int a, b, c;
a = 10;
b = ++a;  /* What is b? */
          /* Both, a and b, are 11. */
c = a++;  /* What is c? */
          /* c is 11, but a is 12. */
```

Operatoren (5)

Relationale (vergleichende) Operatoren dienen dem Vergleich von Ausdrücken.

- Rückgabewert ist ein **Wahrheitswert** (wahr oder falsch)
- Regel: Ein Wert **ungleich null** wird als **wahr** (true) betrachtet.
- Regel: Ein Wert **gleich null** wird als **falsch** (false, unwahr) betrachtet.
- Der Rückgabewert gibt an, ob die Bedingung erfüllt ist oder nicht.

Arten relationaler Operatoren:

- Gleichheitsoperator → `A == B`
  - Wenn A ungleich B ist, ist der Rückgabewert 0, ansonsten ist er ungleich null.
- Ungleichheitsoperator → `A != B`
- Kleiner-/Größervergleich → `A < B`, `B > A`
- Kleiner-Gleich-/Größer-Gleich-Vergleich → `A <= B`, `A >= B`

Achtung: Beliebte Fehlerquelle: Zuweisung (=) und Gleichheitsoperator (==) nicht verwechseln!

Operatoren (6)

Logische Operatoren verknüpfen Wahrheitswerte miteinander.

- logisches Und → `A && B`
- logisches Oder → `A || B`
- logische Negation → `!A`
- Anmerkung: Zur Sicherheit immer Klammerung benutzen.
- Beispiel:

```
int a=1, b=0, c=1;
int erg = ( b || (a&&c) );
```

Bitweise Operatoren dienen dazu, Manipulationen an einzelnen Bits von Variablen vorzunehmen.

- hardwarenahe Programmierung
- bitweises Und → `A & B`
- bitweises Oder → `A | B`
- bitweises exklusives Oder → `A ^ B`
- bitweises Negation → `~A`
- Link-/Rechtsschiebe-Operator → `A << 2, A >> 1`

Gruppierung und Assoziativität (1)

Definition: Unter **Gruppierung** versteht man die Auswertungsreihenfolge innerhalb eines komplexen Ausdrucks.

- Beispiel: `(a * 3 - b++) / (2 * j + k * k)`
- mathematische Vorrangregeln werden umgesetzt
- Inkrement-/Dekrementoperatoren mit höherer Priorität
- Klammerungen mit noch höherer Priorität
- Tipp: Klammerung benutzen!
- siehe folgende Überblicksfolie

Definition: Unter dem Begriff der **Assoziativität** verstehen wir die Fragestellung, ob Operatoren gleicher Gruppierungspriorität von rechts nach links oder von links nach rechts ausgewertet werden.

- Beispiel: `a*b*c*d` → `(a*(b*(c*d)))` oder `(((a*b)*c)*d)`
- Regel: Die meisten Operatoren in C sind linksassoziativ.
- Ausnahme: Zuweisungsoperatoren sind rechtsassoziativ.
  - Beispiel: `A=B=C` entspricht `A=(B=C)` und nicht `(A=B)=C`

Gruppierung und Assoziativität (2)

Operatoren	Auswertungsreihenfolge bei gleicher Priorität
`()` Funktonsaufruf, Klammerung `[]` Felder `->` `.` Strukturen	von links
`!` Not `++` `--` In-/Dekrement `+` `-` Vorzeichen `*` `&` Zeiger `(typ)` Cast (Typumwandlung) `sizeof` Typgröße	von rechts
`*` `/` `%` Arithmetik	von links
`+` `-` Arithmetik	von links
`<` `<=` `>` `>=` Vergleiche	von links
`==` `!=` Gleichheit/Ungleichheit	von links
`&&` logisches Und	von links
`||` logisches Oder	von links
`?:` Fragezeichen-Operator	von rechts
`=` `+=` `-=` `*=` `/=` `%=` Zuweisungen	von rechts

Operatoren: Abschlussbemerkungen

Frage: Wie wird `i = 2 + i * j;` ausgewertet?

a) `i = (2 + i) * j;`

b) `i = 2 + (i * j);`

Antwort: * bindet stärker als +, also b)

Besser: Benutze Klammerung (), um eine andere Interpretation zu erzwingen.

Beispiele:

- boole'sche Logik

```
int bool1, bool2, bool3, bool4;
bool1 = 0;                  /* false */
bool2 = !bool1;             /* bool2 --> true */
bool3 = bool1 || bool2;     /* value? */
bool4 = bool1 && bool2;     /* value? */
```

- unärer Minus-Operator:

```
int var1 = 10;
int var2;
var2 = -var1;
```

Typumwandlungen (Casts) (1)

Problem: Ein Operator wird aufgerufen für Operanden verschiedenen Typs.

- Beispiele: `2 * 3.4` oder `42 + 'a'` oder `double v = 1701;`

Drei Lösungen:

1. **Fehlermeldung** ← bemerkt durch den Compiler
2. **implizite Typumwandlung** (engl. cast) ← realisiert durch den Compiler
   - gemäß bestimmter Regeln
   - aber Spezialfälle existieren, ggf. compilerabhängig
   - Beispiele:

```
double r = 5 / 2;
r = 5 / 2.0;
int i, j;
r = i/j;
```

```
char a = r;
j = r;
a += 1;
```

3. **explizite Typumwandlung** (engl. cast) ← zu realisieren durch den Programmierer
   - sollten sparsam verwendet werden
   - sind nicht immer sinnvoll
   - Beispiele:

```
r = (double)i / j; //unsauber
r = (double)i / (double)j; //sauber
i = (int)r;
```

Typumwandlungen (Casts) (2)

Zusammenfassung:

- Ziel: Umwandlung von Daten unterschiedlichen Typs

```
int i = 10;
float f = (float)i;
double d = (double)i;
```

`(float)` etc. sind so genannte Typkonvertierungsoperatoren (Type Conversion Operators, Casts).

- Der Compiler tätigt diese bei Bedarf automatisch (gibt Warnung aus).

Besser: Typumwandlungen per Hand vornehmen!

```
int i, j;
double d;
i = 3;
j = 4;
d = i / j;                  /* d = ? */
                            /* 0.0 */
d = ((double) i) / j;       /* d = ? */
                            /* 0.75 */
```