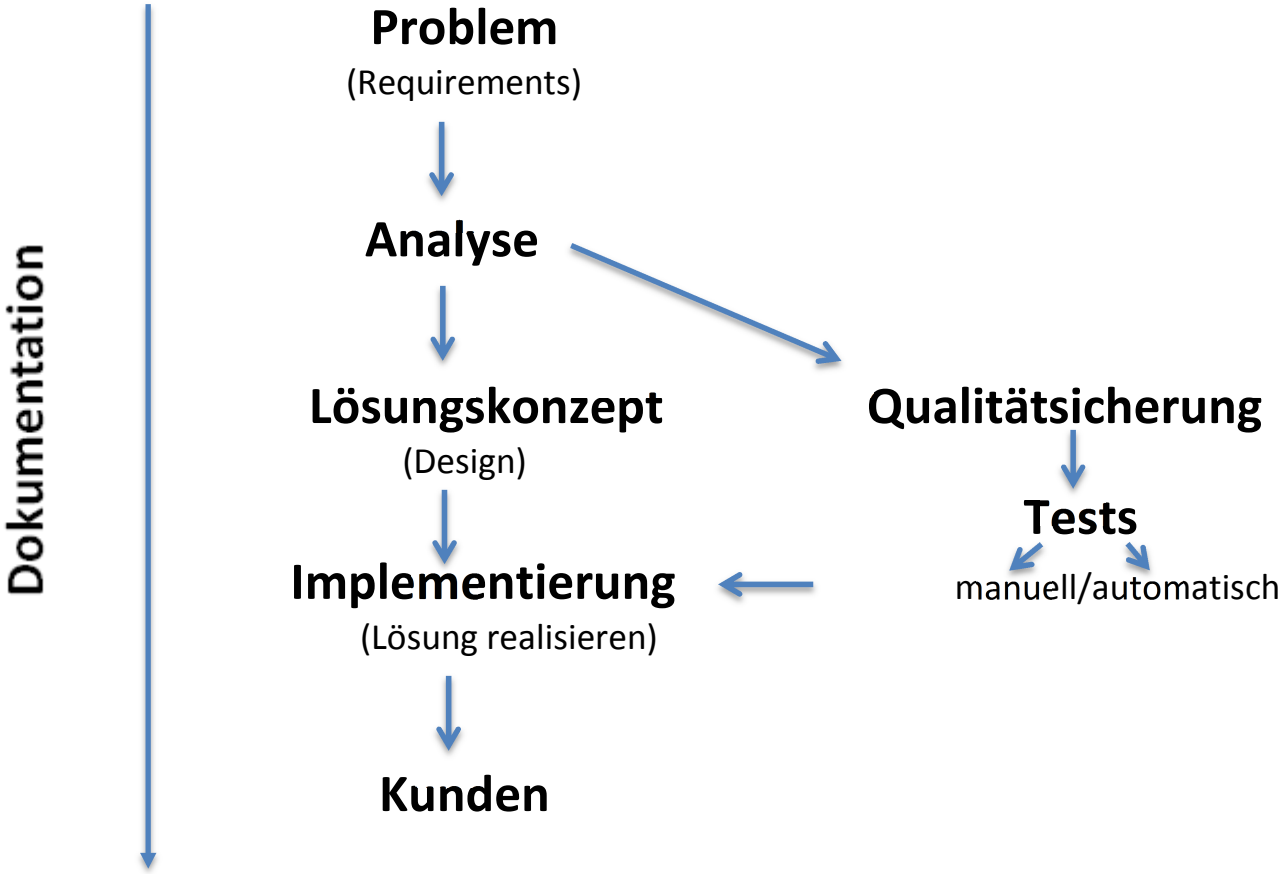
Programmieren in C

Zusammenfassung

Inhaltsverzeichnis



Compiler übersetzt den C-Source Code in eine Sprache (Binärcode) die der Computer direkt ausführen kann.

Syntax ist die Grammatik der Programmiersprache. Diese beinhaltet die Regeln, wie Sprachkonstrukte gebildet werden dürfen.

Semantik gibt die Bedeutung der Sprachkonstrukte wieder, somit den Sinn.

stdlib.h

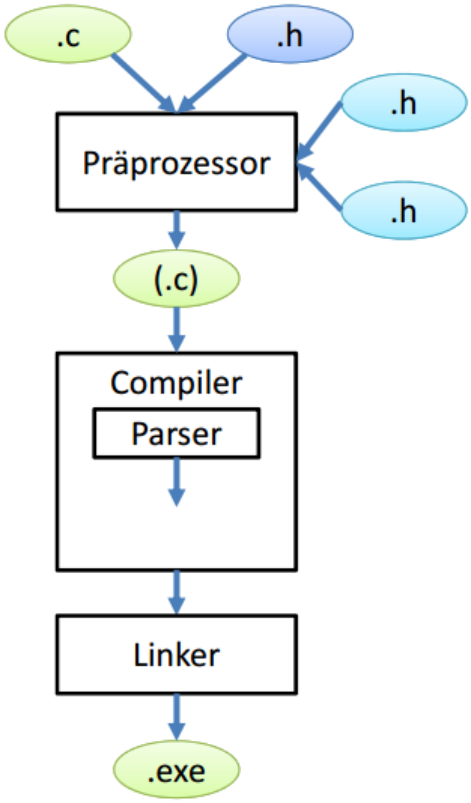
stdio.h

math.h

string.h

Präprozessor

Der Präprozessor wird vor der eigentlichen Übersetzung (Kompilierung) ausgeführt. Der Präprozessor führt eine Text-zu-Text Transformation durch. Dazu werden u.a. die eingebetteten Präprozessor-Direktiven ausgewertet.



Hinweis:
Durch Setzen der Compiler-Option `-E` wird der gesamte Übersetzungsvorgang nach dem Präprozessor abgebrochen.

Direktive #include

An der Stelle wird die angegebene Datei eingefügt. Je nach Einklammerung des Dateinamens wird an unterschiedlichen Stellen gesucht:

`#include <test.h>` (Suche im Systemverzeichnis)

`#include „test.h“` (Suche im lokalen Verzeichnis)

Symbol

```
#define <IDENTIFIER>
```

Definiert ein Symbol mit dem angegebenen Namen (Identifier). Dieses Symbol kann später zur Steuerung der Kompilierung verwendet werden.

Bsp:

```
#define DEBUG
#define RELEASE
```

Konvention: Identifier vollständig in **GROßBUCHSTABEN**

Geltungsbereich bis Dateiende oder bis `#undef` …

Makro

```
#define <Identifier> <Ersatztext>
```

An allen Stellen im Nachfolgenden Quellcode wird der Identifier (Makro-Name) gegen den Ersatztext ausgetauscht.

Parametrisierte Präprozessor-Makros

```
#define <Identifier> (<Parameter>) <Ersatztext mit (<Parameter>)>
```

An allen Stellen im nachfolgenden Quellcode wird der Identifier gegen den Ersatztext ausgetauscht.

Bsp:

`#define SQUARE(x) ((x)*(x))` -> An allen Stellen wird z.B. `SQUARE(x)` durch `(x*x)` ersetzt.

Achtung: Auf korrekte Klammerung achten!

Bedingtes Kompilieren

Mittels der Direktiven `#if, #ifdef, #inndef, #else, #elif` und `#endif` lassen sich Bereiche ein- und ausblenden.

Ein `#if` muss immer mit einem `#endif` abgeschlossen werden.

Direktive	Bedeutung
`#if`	Wenn nachfolgende Bedingung erfüllt, dann …
`#ifdef`	Wenn nachfolgendes Symbol definiert ist, dann …
`#ifndef`	Wenn nachfolgendes Symbol nicht definiert ist, dann …
`#else`	… alternativ …
`#elif`	… alternativ wenn …
`#endif`	… bis hier

Beispiele:

Gesteuerte Ausgabe von Debug-Informationen (Beispiel nicht vollständig!):

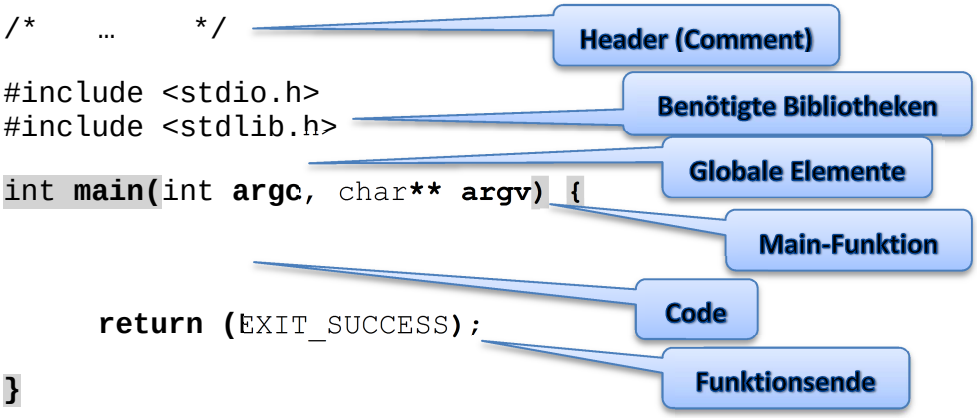
```
#define DEBUG

#ifdef DEBUG
  #define dprintf(x) printf((x))
#else
  #define dprintf(x)
#endif
```

Header-File nur ein mal einlesen:

```
#ifndef ASSERT_H
  #define ASSERT_H
  /* Header file content ... */
#endif
```

Hinweis: Ein Symbol kann bei den Compiler-Optionen mittels Parameter –D gesetzt werden, zum Beispiel „-D DEBUG“

Aufbau einer C-Source-Datei



Begrenzer und Kommentare

;	Semikolon schließt eine Anweisung ab
,	Komma trennt Argumente
{ }	Geschweifte Klammern fassen einen Anweisungsblock zusammen
/* … */	Kommentar über mehrere Zeilen
//…	Einzeiliger Kommentar

Operatoren

Operator	Bedeutung	Bemerkung
=	Zuweisung	
+	Addition	
-	Subtraktion	
*	Multiplikation	
/	Division	Ganzzahldivision bei ganzen Zahlen
%	Modulodivision	Rest der Division

Bsp: `((a/2) * (b+3)) % 7`

Assertions (Zusicherungen)

Verarbeitet man Daten, so erwartet man bei konkreten Eingaben konkrete Ergebnisse (Test Case). Assertions stellen logische Bedingungen dar, die immer wahr sein sollten. Ist das nicht der Fall, so wird dieses als Fehler angesehen und das Programm (zur Sicherheit) beendet.

Bsp.

```
ASSERT( 0.005 > fabs( <Sollwert> - <Ergebnis> ) );
ASSERT( 7 == z );
```

Variablen

Definition: <Typ> <Bezeichner> [= <Wert>];
Typ: char, short, int, long, (long long)
(Datentypen mit 2er-Komplement: negative und positive)

- Variablen **immer initialisieren** (ersten Wert zuweisen, Bsp. = 0)
- **kleinsten Datentyp** mit erforderlichen Wertebereich wählen
- werden **keine negativen Zahlen** benötigt, wird **unsigned** verwendet
- Variablendefinitionen stehen **am Beginn des Blocks** (der Funktion)
- Variablen werden mit **geringstem scope** (Gültigkeitsbereich) eingesetzt

Bsp:

```
unsigned char index = 0;
int value = -12399822;
```

Bezeichner (Variablennamen)

- guter, sinngebender Name
- bestehend aus Buchstaben, Ziffern, Unterstrich
- erstes Zeichen immer Buchstabe (Unterstrich bei Systemfunktionen)
- Kein Schlüsselwort
- Die ersten 31 Zeichen werden ausgewertet
- Englisch!
- Zusammengesetzte Wörter:
    - Camel Case totalAmount

Typ-Umwandlung

Bei arithmetischen Operationen wird immer der Typ der Variablen als Berechnungsgrundlage verwendet. Ergebnisse werden an den Zieltyp angepasst. Um Datenverlust zu vermeiden gibt es Cast-Operatoren. Dadurch kann der Datentyp explizit angepasst werden (casting).

Cast-Operator: (<data type>)<variable identifier>

Bsp.

```
unsigned char x = 13;
unsigned char y = 14;
float z = 0;
z = x / y;                          // Ergebnis: 0.0
z = (float)x / (float)y;            // Ergebnis: 0.92857…
```

Zahlen

Ganze Zahlen

Bit 0/1
Byte 8 Bits (256 mögliche Permutationen)

Bit Wertigkeit	7 128	6 64	5 32	4 16	3 8	2 4	1 2	0 1		2er
	0	0	1	1	0	0	0	1	49	49
	0	1	1	1	1	0	0	0	120	120
	1	0	0	0	0	1	1	1	135	-121
	1	1	1	1	1	1	1	1	255	-1
	1	0	0	0	0	0	0	0	128	-128

Rechentrick für negative Zahlen 2er-Komplement

1. Binärdarstellung invertieren (0→1, 1→0)
2. Wert bestimmen
3. 1 drauf addieren
4. Vorzeichen umkehren

Gleitkommazahlen

IEEE 754:
Zerlegung der Zahl x in

- Ein Vorzeichen v (1/-1)
- Eine Mantisse m (1≤m<2)
- Einen Exponenten exp zur Basis 2

$$x = v \cdot m \cdot 2^{exp}$$

Bsp:

$$-4{,}5 = -1 \cdot 1{,}125 \cdot 2^2$$

→ Speicherplatz der Mantisse bestimmt die Genauigkeit
→ Speicherplatz des Exponenten bestimmt den Wertebereich

Bits von	v	m	exp
float (4Byte)	1	23	8
double (8 Byte)	1	52	11

Spezielle Codierungen

- kleine Zahlen werden auf Null gerundet. Es gibt -0.0
- Es gibt „keine Zahl“ (NaN) und Unendlich

Datentypen

C ist eine typisierte Sprache, d.h. alle Variablen haben einen im Programm definierten Datentyp. Der Datentyp gibt den Wertebereich und den Platzbedarf im Speicher an.

Datentyp	Byte	von	bis
`char`	1	-128	127
`unsigned char`	1	0	255
`short`	2	-32768	32767
`int`	4	-2,1 Mrd. (-2^{31})	2,1 Mrd. (2^{31}-1)
`unsigned int`	4	0	4,3 Mrd. (2^{32})
`long`	4	wie int	wie int
`float`	4	≈5,877*10^{-39}	≈3,403*10^{38}
`double`	8	≈1,1125*10^{-308}	≈1,798*10^{308}

Gültig für Intel 32-Bit Prozessor!

Wertebereich

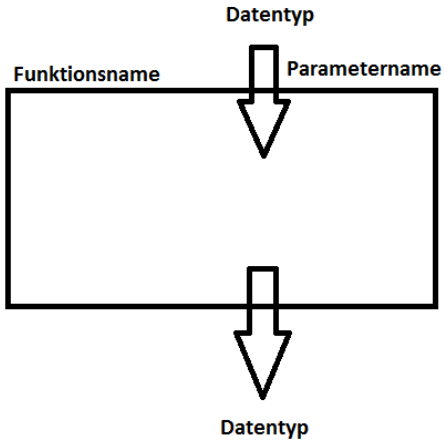
Der Wertebereich ist systemabhängig(!) und in der Header-Datei `limits.h` definiert.
Grundsätzlich gilt: `char` ≤ `short` ≤ `int` ≤ `long` ≤ `long long`

Funktionen

Um mehr Übersichtlichkeit zu gewinnen kann man Programmteile auslagern, z.B. mathematische Funktionen oder Dateioperationen usw. Auch Identische Code-Abschnitte werden in Funktionen ausgelagert.

Funktionen aus Standardbibliothek

Funktionen müssen dem Compiler und dem Linker bekannt sein. Dem Linker sind sie durch die Projekteinstellungen bekannt, dem Compiler muss die Verwendung in der C-Datei durch `#include` angezeigt werden.



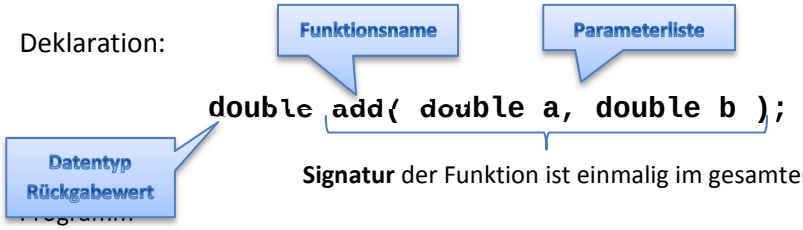
Schnittstellen/ Funktions-Deklaration

Die Funktionalität einer Funktion (Prozedur) wird über eine Schnittstelle zugänglich.

Daten rein, Parameter	(0…viele)
Daten raus, Ergebnis	(0/1)

(über Call-by-Reference können auch mehr Werte zurückgegeben werden)

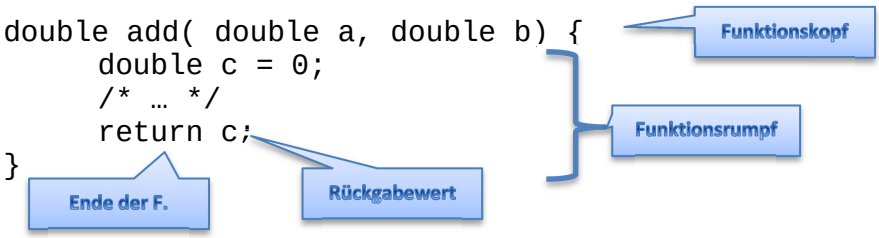
Deklaration:



Signatur der Funktion ist einmalig im gesamten

Funktions-Definition

Die Funktionalität einer Funktion (Prozedur) muss mittels C-Code realisiert werden. Die Funktion kann wiederum Funktionen aufrufen.



Projektorganisation

Funktionen werden in eigenen Dateien mit zugehörigen Header-Dateien angelegt. Dateinamen sind gleich (Konvention), Dateiendung .c bzw .h

Header-File <.h> (im include-Pfad)

```
double add ( double a, double b );
```

Main-File (vor die main-Funktion)

```
#include „add.h“
```

Source-File <.c> (im Projekt/Linker-Pfad)

Funktions-Definition (siehe oben)

Funktionen ohne Rückgabewert (**Prozeduren**) haben den Datentyp **void** und es gibt dann keine return-Anweisung. Bei Funktionen ohne Parameter ist die Parameterliste leer oder void. Funktionen ohne Deklaration müssen im Quellcode vor dem Ort der Verwendung stehen. Rückgabewerte sollten immer ausgewertet werden und die Funktion muss getestet werden (Test Cases)

Sinnvolle Funktionsnamen verwenden!

printf( ) - Ausgabe auf Konsole

#include <stdio.h>

`printf()` sendet an **`stdout`** eine Sequenz von Daten nach den Angaben im Format-String.

```
int printf( const char * fomat, … );
```

Escape Sequenzen von printf()

`\n`	newline (Systemabhängig)
`\r`	carriage return
`\t`	tabulator
`\“`	Hochkommata
`\\`	\
`\0`	Null-Terminierung (-> Zero Terminated String)

Format-String

Im Format-String werden an der Ausgabeposition Formatierungscode eingefügt. Diese enthalten Angaben über die Position und die Formatierung der Daten. Formatierungscodes sind z.B. %i, %d

```
printf(„Wert: %i\n“, value);
```

Formatierungspezifizierung

Im Format-String werden Daten an der Position des Format Tags im angegebenen Format eingefügt. Struktur:

```
%[flags][width][.pecision][length]specifier
```

specifier	Bedeutung	Beispiel
`i oder d`	Ganze Zahl mit Vorzeichen	`396`
`a oder A`	Hexadezimalzahl (ohne Vorzeichen)	`7AF`
`e oder E`	Wiss. Schreibweise (Mantisse, Exponent)	`3.9265e+2`
`f`	Gleitkommazahl	`347.21`
`c`	Zeichen	`a`
`s`	String	`sample`
`p`	Pointer Adresse	`b8000000`

length	
`(None)`	Int
`hh`	Signed char
`h`	Short int
`l`	Long int
`ll`	Long long int

flags	Bedeutung
`+`	mit Vorzeichen
`-`	Innerhalb der vorgegebenen Feldbreite linksbündig
`#`	mit Zahlensystemzeichen
`0`	Leere Felder der vorgegebenen Feldbreite werden mit Nullen ausgefüllt

precision	
`.<Zahl>`	Anzahl der Nachkommastellen
`*`	Die *Genauigkeit* wird nicht in dem *Format-String* angegeben, sondern als zusätzlicher Integer-Wert vor dem Argument, das formatiert werden soll

width	
`<Zahl>`	Minimale Anzahl der Zeichen die ausgegeben werden sollen
`*`	`printf ("Width trick: %0*d \n", 5, 10)` Ausgabe: `Width trick:00010`

Bsp:

```
printf („pi: %+010.3“, 3.141526);     // pi: +00003.142
```

scanf() – Zeichenströme einlesen

#include <stdio.h>

Der im Konsolenfenster eingegebene Zeichenstrom wird von `stdin` entgegengenommen und dann `scanf()` übergeben und in die Pointer-Adresse geschrieben. Der Zeichenstrom muss den Regeln der Grammatik entsprechen ansonsten erfolgt keine Übernahme.

Problem: Eine Funktion liefert nur ein Rückgabewert.

Lösung: Funktion muss gezielt auf „äußere“ Variablen Zugriff erhalten. Dies geschieht durch die Verwendung von Referenzen (Referenzoperator: &) → **call by reference.** Dabei wird der Funktion die (interne) Adresse der Variable übergeben so erhält die Funktion indirekten Zugriff (→Pointer)

```
int scanf ( const char * format, … );
```

Format

```
% [*][width][modifiers]type
```

Formatierungszeichen

Empfehlung:

Formatierungzeichen	Verwendeter Datentyp für den Parameter	
%c	char	Buchstabe
%d	int	Ganzzahl
%f	float	Kommazahl
%lf	double	genaue Kommazahl
%s	char*	String

Bsp:

```
printf(„Enter value:“);
scanf(„%d“, &x);
printf(„\nEntered: %d\n“, x);
```

if-then-else Anweisung - Kontrollstrukturen

Software muss auf Basis von Bedingungen verschiedene Aktionen durchführen. Die Basis für die Entscheidungen sind immer strenge logische Aussagen, die wahr (`true`) oder falsch (`false`) sind. Wenn die Bedingung erfüllt ist (`true`) wird der Code innerhalb des if-Blocks ausgeführt.

```
if ( <condition> ) <block>
else <block>
```

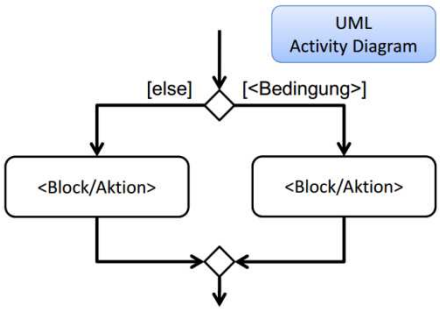
In C gibt es keinen eigenen Datentyp für **Bool´sche Werte**, meist wird **`unsigned char`** verwendet. **`true`** und **`false`** sind in **`<stdbool.h>`** definiert.

`0 → false`

`ungleich 0 → true`

Bsp:

```
if (x>5) {
      y = 5;
} else {
      y = x;
}
```



Vergleichsoperatoren	
Operator	Bedeutung
<	Kleiner als
<=	Kleiner gleich
>	Größer
>=	Größer gleich
==	Gleich
!=	ungleich

Logische Operatoren	
Zeichen	Bedeutung
!	Negation
&&	UND
||	ODER

Bei && wird der zweite Teil nicht mehr ausgewertet wenn der erste `false` ist. Bei || wird der zweite Teil nicht mehr ausgewertet wenn der erste `true` ist.

DeMorgansche Regeln

a && b = !(!a || !b) !(a && b) = !a || !b

a || b = !(!a && !b) !(a || b) = !a && !b

Bedingte Zuweisung

Soll einer Variablen im true- und im false-Zweig ein Wert aufgrund der Bedingung zugewiesen werden, so kann man diese bedingte Zuweisung zusammenfassen.

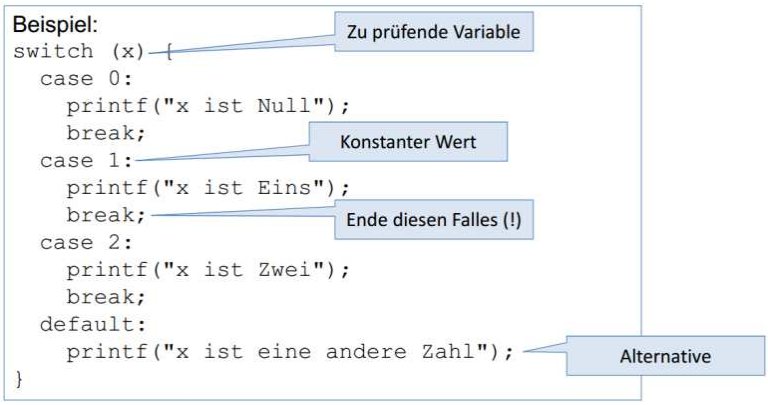
```
<variable> = ( <condition> ) ? <true-value> : <false-value>
```

Bsp:

```
y = (x>5) ? 5 : x;
```
(identisch zum Beispiel oben)

switch-case-Anweisung

Oftmals muss eine Auswahl aus mehreren festen Möglichkeiten getroffen werden. Hier wird die gleiche Variable immer gegen eine Konstante geprüft.



Emumerations (Aufzählungen)

Um besser mit eingeschränkten, symbolischen Werten (z.B. Wochentagen) arbeiten zu können, kann man sich eigene Aufzählungstypen deklarieren.

```
enum <Bezeichner> {<Element0>, < Element1> , … , <Elementn>};
```

Bsp:

```
enum Wochentag {Mo, Di, Mi, Do, Fr, Sa, So};

enum Wochentag heute = So;
enum Wochentag morgen = Mo;

if (heute == So)
    printf(„frei!“);
```

Enumerations mit swith-case

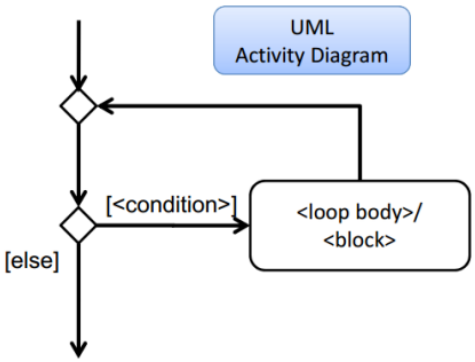
Bsp:

```
enum liste {Null, EINS; Zwei, Drei} x = Eins;
switch (x) {
    case Null:
        printf(„x ist Null“);
        break;
    case Eins:
        printf(„x ist Eins“);
        break;
    case Zwei:
        printf(„x ist Zwei“);
        break;
    default:
        printf(„x ist eine andere Zahl“);
}
```

Schleifen

while-Schleife (Kopfgesteuert)

Eine Schleife wird so oft ausgeführt, wie die Bedingung zutrifft. Trifft die Bedingung direkt beim ersten Mal nicht zu, so wird die Schleife direkt übersprungen.



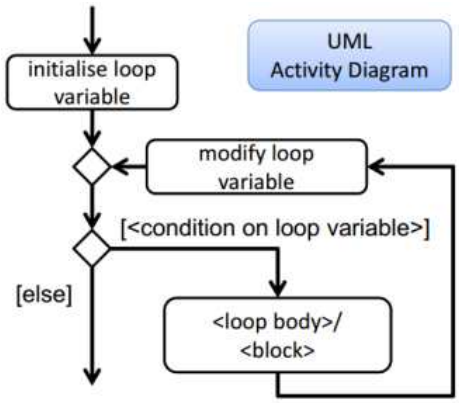

```
while ( <condition> ) <block>
```

Bsp:

```
while (x<5) {
      x = x + 1;
}
```

for-Schleife (Kopfgesteuert)

Eine Schleife wird so oft ausgeführt, wie die Bedingung zutrifft. Trifft die Bedingung direkt beim ersten Mal nicht zu, so wird die Schleife direkt übersprungen.




```
 for ( <init> ; <condition> ; <modify> ) <block>
```

Bsp:

```
for(int x=0 ; x<5 ; x++) {       //läuft 5x durch
      printf(„x:%i\n“,x);
}
```

Modifier

Operator	Bedeutung	Bsp
++	Increment	x++
--	Decrement	x--

Die Operatoren können als Postin-/decrement oder als Prein-/decrement eingesetzt werden.

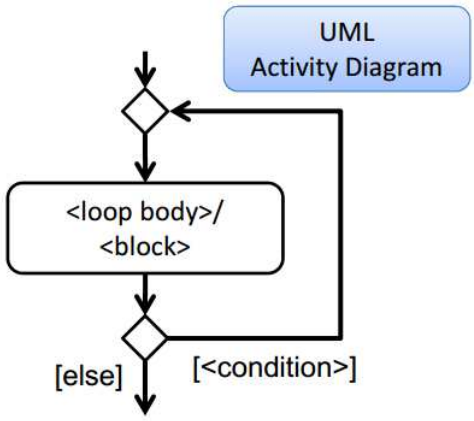
Ausgangssituation jeweils x = 42.

Code		x	y
`y = (x++);`	Postinkrement	43	42
`y = (x--);`	Postdecrement	41	42
`y = (++x);`	Preinkrement	43	43
`y = (--x);`	Predekrement	41	41

Abkürzungen: x += 5; -> x = x + 5; (klappt mit allen Operatoren)

do-while-Schleife (Fußgesteuert)

Eine Schleife wird so oft ausgeführt, wie die Bedingung zutrifft. Die Bedingung wird allerdings am Ende geprüft und somit wird die Schleife (Loop-Body) mindestens einmal ausgeführt.



do <block> while (<condition>);

Bsp:

```
int x = 0
do {
      printf(„Bitte Zahl:“);
      scanf(„%d“, &x);
} while (x < 0);
```

Magic Numbers und Konstanten

In Programmen tauchen immer wieder feste Werte auf, insbesondere im Zusammenhang mit Schleifen. Diese Zahlenwerte werden oft als „Magic Numbers“ bezeichnet. Die Bedeutung des Wertes ist nicht offensichtlich.

Besser: Verwendung von Makros oder Variablen mit „selbstsprechenden“ Bezeichnern.

Bsp. for-Loop:

```
for(int x=0 ; x<5 ; x++) {
      printf(„x:%i\n“,x);
}
```

Bsp. Makro

```
#define MAX_VALUE 5
for(int x=0 ; x< MAX_VALUE ; x++) {
      printf(„x:%i\n“,x);
}
```

Bsp. Konstante wenn Datentyp wichtig!

```
const int maxValue = 5;
for(int x=0 ; x<maxValue ; x++) {
      printf(„x:%i\n“,x);
}
```

Bearbeitung von (Text-)Dateien

#include <stdio.h>

Dateien sind aus Sicht des Computers eine Folge von Bytes, die nacheinander gelesen oder geschrieben werden können (vergleiche Tonband). Typischerweise sind Dateien persistent (nicht flüchtig) und werden in einem Dateisystem hierarchisch organisiert und über einen Pfad + Name identifiziert.

Textdateien verwenden nur Zeichen der vom Menschen lesbaren Menge des ASCII-Zeichensatzes. Für den Computer ergibt sich die Schwierigkeit, dass dieser die Struktur analysieren und für interne Datentypen aufbereiten muss.

Ablauf ist immer **öffnen**, **bearbeiten**, **schließen**.

Öffnen/Schießen

Funktionen für den Zugriff auf Dateien sind in C in der Header-Datei **`stdio.h`** deklariert. Beim Öffnen der Datei werden Eigenschaften des Zugriffs festgelegt (lesen, schreiben, …). Als Ergebnis erhält man einen Pointer auf eine Verwaltungsstruktur des Betriebssystems. Über diese Struktur wird die Datei bei nachfolgenden Operationen identifiziert. Dateiname unter Windows müssen einem doppelten Backslash „\\“ angegeben werden.

```
FILE * fopen ( const char * filename, const char * mode );
```

Zugriffseigenschaften (mode)

`r`	nur lesen
`w`	nur schreiben (existierende Datei wird überschrieben)
`a`	dranhängen (schreibt nur wenn Datei schon existiert)
`r+ / w+`	lesen und schreiben
`a+`	append (lesen und schreiben)

Für binäre Dateien muss ein „b“ ergänzt werden. (-> API-Doku)

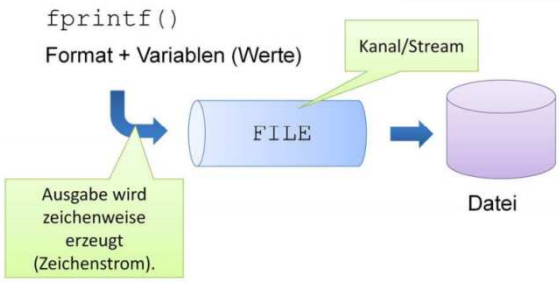
Schließen der Datei:

```
int fcole (FILE* stream );
```

#include <stdio.h>

Schreiben

Daten können über einen Kanal, analog zu `stdout` für die Ausgabe auf Konsole, Zeichen für Zeichen ausgegeben werden. Die Zeichenfolge wird von `fprintf()` genauso erzeugt, wie mittels `printf()`. Im Gegensatz zu `printf()` muss der Kanal mit angegeben werden. Man kann auch sagen, `printf()` ist eine Sondervariante von `fprintf()` die automatisch `stdout` als Kanal verwendet.



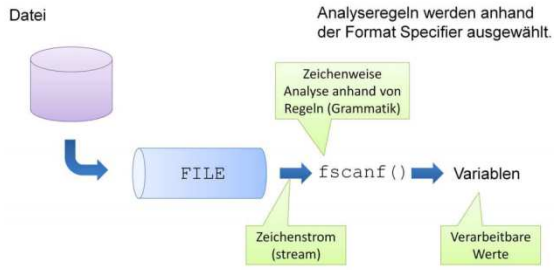
Hinweis: Der Kanal zu einer bestimmten Datei muss zuvor geöffnet werden.

```
int fprintf ( FILE * stream, const char * format, … );
```

Bsp:

```
FILE *exampleFile = 0;
exampleFile = fopen(„c:\\temp\\textdatei.txt", „w");

if (exampleFile != 0) {
    fprintf(exampleFile, „%s", „Zeichenkette");
    fclose(exampleFile);
}
```

Lesen



Hinweis: Der Kanal zu einer bestimmten Datei muss zuvor geöffnet werden.

```
int fscanf ( FILE * stream, const char * format, … );
```

Dateiende wird mittels `feof()` erkannt.

Bsp:

```
FILE *exampleFile = 0;
int x = 0;
int y = 0;
exampleFile = fopen(„c:\\temp\\textdatei.txt", „r");
if (exampleFile != 0) {
        while (!feof(exampleFile)) {
            fscanf(exampleFile, „%i:%i\n", &x, &y);
            printf(„%i: %i\n", x, y);
}
fclose(exampleFile);
}
```

Pointer

Variablen bestehen aus: Bezeichner, Datentyp, Wert, Platzbedarf, Speicherort und Scope.

Pointer sind Variablen, die als Wert eine Adresse (also Speicherort) einer anderen Variablen speichern.

<type>* <pointer name>;

&x -> Adresse von x

***x -> Wert von x**

Bsp:
```
char* letter = 0;     // Datentyp Pointer vom Typ char mit Namen letter
```

Intern haben Pointer immer die gleiche Größe, z.B. 4Byte, egal welchem Typ sie referenzieren.

Pointer zuweisen

Die Adresse einer Variablen kann mittels Referenzoperator „&“ bestimmt werden.

Bsp:
```
int x = 5;
int* value = &x;      // value verweist nun auf x
```

Zugriff auf Wert der Speicherstelle

Auf den Wert der Speicherstelle, auf die ein Pointer zeigt, kann über den Operator „*“ zugegriffen werden.

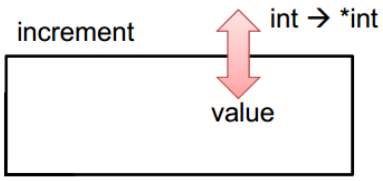
Bsp:
```
int x = 5;
int* value = &x;      // value verweist nun auf x
int y = *value;
*value = 23;
```

Arbeiten mit Pointern

Pointer sind „normale“ Variablen, die einen Wert beinhalten und somit auch zugewiesen werden können. Auch die Interpretation des Ziels kann mittels Casting geändert werden, mit allen (schädlichen) Konsequenzen!

Bsp:
```
int x = 5;
    int* value = &x;        // value verweist nun auf x
    void* other = 0;        // Ziel hat kein Typ!
    other = value;
    *((int*)other) = 40;
```

Pointer als Parameter



Ist die Adresse eines Wertes (Variablen) bekannt, kann dieser unabhängig vom Scope manipuliert werden. ->Werden Adressen an Funktionen übergeben, können sie den referenzierten Bereich verändern (mit allen schädlichen Konsequenzen). ->Funktion erwartet eine zu manipulierende Adresse (**Call by Reference**)

Bsp:
```
void increment( int* value) {
     (*value) +=1;
}
increment(&x);

int* refernez = &x;
increment(refernez);
```

Arrays (=Pointer)

Definition

„Tabellen“ mit fester Größe und gleichem Datentyp.

```
<type> <name>[<total>];
```

Bsp:
```
int values[6];                          <- 6 Elemente
int values[] = {23, 42, 2073423};       <- 3 Elemente
int values[10] = {23, 42, 2073423};     <- 10 Elemente
int values[10] = {0};                   <- 10 Elemente alle „0“
```

Größe des Arrays in Bytes: **`sizeof(Array)`**

Größe eines Feldes in Bytes: **`sizeof(Array[0])`**

Anahl der Felder: **`sizeof(Array)/ sizeof(Array[0])`**

Zugriff

```
int values[6];              <- Array mit 6 Elementen

values[0] = 23;             <- Zuweisung des ersten Elements mit „0“

int x = values[1];          <- Zugriff aufs zweite Element mit „1“

int i = 2;
int y = values[i - 1];      <- Dynamischer Zugriff mit Index-Variablen
```

Achtung: Der Zugriff außerhalb der Grenzen wird nicht verhindert!

Übergabe an Funktionen

Bei der Übergabe wird (leider) nur die Position im Speicher übergeben (->Referenz). Die Länge geht dabei verloren.

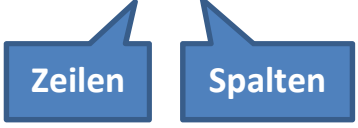
Die Größe muss extra übergeben werden!

```
int sum(int values[], unsigned int size) {
    int summe = 0;
    unsigned int i = 0;

for (i = 0; i < size; i++) {
        summe = summe + values[i];      // summiert alle Werte des Arrays
}
return summe;
}
```

Mehrdimensionale Arrays

Arrays können auch mehr als eine Dimension haben. Die Anzahl der Elemente je Dimension wird in den eckigen Klammers angegeben.

Bsp: 4x2-Matrix: `int matrix[4][2];`



Initialisierung
int matrix[][3]= {{1, 0, 0}, {0, 1, 0,}, {0, 0, 1}};

Zugriff erfolgt mit vollständiger Angabe der Indizes: `int x = matrix[2][0];`

Strings (Arrays vom Typ char)

#include <string.h>

Texte (Strings) werden in C einfach in Arrays vom Typ `char` oder `unsigned char` gespeichert.

Bsp: `char name[14] = "Prof. Lehmann";` `// 13 Zeichen -> 14 Array Elemente`

Bei Strings muss die Länge bekannt sein (-> Array Overflow) oder der String muss **Null-Terminiert '\0'** sein.
Konstante Strings werden automatisch Null-Terminiert

Strings sollten immer Null-Terminiert sein. `name[13] = 0;`
Zusätzlichen Platz bei der Deklaration beachten.

Ausgabe von Strings

Die Ausgabe von Strings mittels `printf()` wird durch das Formatierungssymbol %s signalisiert.

Achtung: printf() gibt den String bis zum Erreichen der Null-Terminierung aus!

Bsp:

```
char name[] = "Prof. Lehmann";
    printf("Name: %s", name);
    puts (name);                    //Alternative
```

Strings sind „normale" Arrays:

```
char name[] = "Prof. Lehmann";
```

aus Sicht des Computers:

```
char name[] = {'P', 'r', 'o', 'f', '.' ,' ' ,'L'
               'e' ,'h' ,'m' ,'a' ,'n' ,'n' ,0};
```

oder auch:

```
char name[] = {80, 114, 111, 102, 46, 32, 76, 101,
               104, 109, 97, 110, 110, 0};
```

Zugriff auf Strings

Strings sind intern Arrays und somit kann auf die Buchstaben wie in einem Array zugegriffen werden.

```
char name[] = "Prof. Lehmann";
printf("Name: %s/n", name);

int i = 0;
pintf("Einzeln:");
for (i = 0; name[i] != '\0'; i++) {
    printf(„%c", name[i]);
}
printf("\n");
```

Komplexere String-Verarbeitung

Stringgröße beachten! ‚0\' nicht verlieren

Funktion	Aufgabe/Wirkung	Return Value
strlen	Gibt die Länge des Strings zurück (bis und ohne Null-Terminierung '\0') `unsigned int x = strlen(<String>);`	String-Länge
strcpy	Kopiert einen String in einen anderen. `strcpy(<Ziel>, <Quelle>);`	Ziel
strcmp	Vergleicht zwei Strings miteinander int x = strcmp(str1,str2); **0** -> gleich **(>0)** -> erster versch. Buchstabe von str1 > als der von str2 **(<0)** -> umgekehrt	0 , -1 , 1
strcat	Verknüpft Strings strcat (str1, str2); oder strcat (str1, "drangehängter Text"	Ziel (hier: str1)
strncpy	Kopiert die ersten n Buchstaben strncpy (<Ziel>, <Quelle>, n); (evtl '0\' hinzu)	Ziel
strncat	Verknüpft n Buchstaben mit String strncat (<Ziel>, <Quelle>, n);	Ziel

Listen

Strings können auch wieder in Listen (Arrays) organisiert werden.

Bsp:

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

int main() {
    char str[3][5] = {"R2D2", "C3PO", "R2A6"};
    int n;
    printf("Looking for R2 astromech droids...\n");
    for (n = 0; n < 3; n++)
        if (strncmp(str[n], "R2xx", 2) == 0) {
            printf("found %s\n", str[n]);
        }
    return 0;
}
```

Strukturen

Strukturen (Records) in C fassen mehrere Daten zusammen.

```
struct <struc_name> {
    <type> <element name>;
    <type> <element name>;
    <type> <element name>;
};
```

Bsp:

```
struct point {
    int x;
    int y;
    int z;
};
```

Variablen-Definition

```
struct <struct_name> <variablen_name> [ = {<initial>} ];
```

Bsp:

```
struct point start = {1, 1, 1};
```

Denkweise

- Welche Daten (ggf. unterschiedlichen Typs) bilden inhaltlich wieder eine Einheit?
- Aufbau von größeren, inhaltlich zusammenhängenden Strukturen

Zugriff auf Komponenten

Auf die Elemente/Komponenten wird über den **Punkt-Operator** zugegriffen.

```
<variablen name>.<element name> = <wert>;
```

Bsp:

```
struct point start = {1, 1, 1};
start.x = 5;
u = start.y;
```

Strukturen und Pointer

Wird eine Struktur über einen Pointer referenziert, so muss der **"->"-Operator** verwendet werden.

```
<poiner_name> -> <element name> = <wert>;
```

Bsp:

```
struct lookupTable {char ID[4]; int value; };

// Pointer to struct
struct point *pointReference = &start;
pointReference->x = 10;

// Pointer in struct
struct lookupTable hash = {"HAW", 0};
hash.value = 1;
strncpy(hash.ID, "UAS", 3);
```

Strukturen und Arrays

Strukturen bilden einen Datentyp und könne auch in Arrays verwendet werden.

Bsp:

```
struct point {
    int x;
    int y;
    int z;
};
struct point points[5] = {{0,0,0},{1, 1, 1}};

points[0].x = 5;
```

Typedef

Oftmals möchte man Strukturen als “echten“ Datentypen haben (ohne struct). Eigene Datentypen kann man mittels dem **Schlüsselwort „`typedef`“** deklarieren/definieren.

```
typedef <type-information> <type name> [ = {<initial>} ];
```

Bsp:

```
typedef struct {     // <- Anonyme Struktur
    int x;
    int y;
    int z;
} Points_t;          // <- Datentyp-Bezeichner beginnt
                     // mit Großbuchstaben und Suffix "_t"

Points_t points[5] = {
    {0, 0, 0},
    {1, 1, 1}
};

points[0].x = 5;
```

Umbenennung von Standard-Datentypen

Typdefinitionen können auch für die Umbenennung von Standard-Datentypen verwendet werden, wenn sich dadurch die Lesbarkeit erhöht.

```
typedef unsigned char Boolean;
typedef unsigned char Matrikelnummer[7];
...
Matrikelnummer mat = {1,2,3,4,5,6};
```

Bedingung	
Signatur	Name, Datentype des Rückgabewertes und Parameter einer Funktion
Funktion	Festgelegte Aufgabe die durch Parameter variieren kann
Syntax	Umsetzung der Programmiersprache
Assertion	Selbsttest eines Programms
Semantik	Sinn hinter der Syntax
Prozedur	Folge von Anweisungen
Definition	Einer Variablen einen Datentype zuweisen
Datentype	Größe des Speicherplatzes
Compiler	Übersetzt das Programm in Computersprache
Variable	Name eines Speicherbereiches
Initialisierung	Variablen einen Wert zuweisen
Parameter	Ein Wert der Übergeben wird
Linker	einzelne Programmmodule zu einem ausführbaren Programm zusammenstellt
Debugger	Werkzeug zum auffinden von Fehlern
Deklaration	Variable festlegen
Editor	Bearbeitungsprogramm
Analyse	
Wertebereich	Größer eines Datentyps
Anweisung	Codezeile die etwas ausführt
Argument	Funktionsparameter
Operatoren	Mathematische Anweisungen (+,-,*,/)
Konsole	Ausgabe auf dem Bildschirm
Header-Datei	Funktionsbibliothek
Implementierung	Umsetzen eines Algorithmus oder Softwareentwurfs in ein Computerprogramm
IDE	Schnittstelle
ASCII	Zeichentabelle
Arithmetik	Mathematische Operatoren
Source-Datei	Quellcode