Zahlendarstellung ... **3**
Zahlensysteme mit festem Stellenwert ... 3
Binär -> Hex ... 3
Dezimal ->Binär ... 3
Einerkomplement ... 3
Hex -> Binär ... 3
Binär -> Dezimal ... 3
Binär -> Oktal ... 3
Oktal -> Binär ... 3
Zweierkomplement ... 3
Gültigkeit einer Addition / Subtraktion im Zweierkomplement ... 3
Wert einer Zahl im Zweierkomplement ... 3
Einführung und Generelles ... **3**
Eigenschaften von C ... 3
Struktur Quelldatei ... 3
Vom Problem zum Programm ... 3
Grundkonzepte von imperativ prozeduralen Programmiersprachen ... 3
Einige Begriffe ... 3
Datentypen und Variablen ... **3**
Datentyp ... 3
Floating Point Standard ... 3
Deklaration und Definition ... 3
Attribut *volatile* ... 4
Klassifikation von Datentypen ... 4
Blöcke ... **4**
Generell ... 4
Gültigkeit einer Variable ... 4
Funktionen ... **4**
Deklaration ... 4
Definition ... 4
Syntaktisch und semantische Korrektheit ... 4
Aufruf ... 4
inline ... 4
Pointer ... **4**
Generell ... 4
Pointerarithmetik ... 5
Beispiel ... 5
Beispiel ... 5
Beispiel ... 5
Beispiel: Pointer auf ein Array ... 5
Pointer auf Pointer ... 5
Pointer auf Funktion ... 5
Funktion mit Rückgabewert "Pointer auf ein Array von char-Pointer" ... 5
Leseweise ... 5
Arrays (und Standardfunktionen für den Umgang mit Speicher) ... **6**
Generell ... 6
Arraygrösse (oder Arraylänge) ... 6
String ... 6
Vergleiche von Arrays ... 6
Mehrdimensionale Arrays ... 6
Übergabe von mehrdimensionalem Array ... 6
Standardfunktionen für Strings ... 6
Standardfunktionen für Speicher ... 6
Lexikalische Konventionen ... **7**
Bestandteile des Source-Codes ... 7
Zeichenvorrat von C ... 7
Namen ... 7
Styleguide: Namen ... 7
Symbolische und literale Konstanten ... 7
Enumeration (Aufzählungstyp) ... 7
const ... 8
Anweisungen, Ausdrücke und Operatoren ... **8**
Operatoren und Operanden ... 8
Postfix- und Präfixoperatoren ... 8
Ausdrücke und Anweisungen ... 8
Komma-Operator (Schleifen) ... 8
Auswertungsreihenfolge (operator precedence) ... 8

L-Werte und R-Werte ... 9
Bitwise Operators ... 9
Schiebe- (Shift-) Operator ... 9
Bedingungsoperator (Ternärer Operator) ... 9
Explizite Typumwandlung ... 9
Implizite Typumwandlung ... 10
Syntaktisch erlaubte Typumwandlungen ... 10
Sequenzpunkte ... 10
Strukturen ... 10
Generell ... 10
Syntax ... 10
Operationen auf Strukturen ... 10
Zugriff auf Strukturmember ... 11
Lage im Speicher ... 11
Initialisierung einzelner Strukturmember ... 11
Unions ... 11
Generell ... 11
Typedef ... 11
Generell ... 11
Beispiel ... 12
Portabilität ... 12
Namensräume ... 12
Generell ... 12
Rekursion ... 12
Generell ... 12
Anwendung von Rekursion ... 12
Beispiel: Fakultät ... 12
Adressraum eines Programms ... 13
Speicherbereich eines C-Programms ... 13
Speicherklassen ... 14
Interne vs. externe Bindung ... 14
Speicherklasse extern ... 14
Speicherklasse static ... 14
Speicherklassen bei lokalen Variablen ... 14
Scope ... 14
Preprocessor-Direktiven und Bedingungsanweisungen ... 14
Generell ... 14
Direktiven ... 15
Bedingungsanweisungen ... 15
#define ... 15
#undef ... 15
#include ... 15
#line ... 15
#error ... 15
#pragma ... 15
Aufbau von Bedingungsanweisungen ... 15
Input und Output ... 15
Datei (File) ... 15
Dateizugriffsfunktionen ... 15
Streamkonzept von UNIX ... 15
I/O-Konzept von C ... 16
Standardeingabe- und Ausgabe ... 16
I/O-Funktionen in C ... 16
High-Level-Dateizugriffsfunktionen ... 18
Kommandozeilenparameter ... 18
Kompilierungsvorgang ... 19
Generell ... 19
Preprocessor ... 19
Erste Phase des Compilers ... 19
Feste vs. relokative Adressen ... 19
clang ... 19
Good Practice ... 19
Anforderungen in grösseren Projekten ... 19
Information hiding ... 19
Zusatz ... 20
Beispiel: Zeiger auf literale Konstante als Rückgabewert einer Funktion (C -> ASM) ... 20
Beispiel: Zeiger auf lokale Variable als Rückgabewert einer Funktion (C -> ASM) ... 20
Beispiel: Zeiger auf lokale Variable als Argument einer Funktion (C -> ASM) ... 20

Zahlendarstellung

Zahlensysteme mit festem Stellenwert

$$z = \sum_{k=-m}^{n-1} a_k \cdot B^k$$

z: Wert der Zahl (im Dezimalsystem)

a: Nennwert der Ziffer

B: Basis des Zahlsystems, $B \geq 2$

k: Stellenwert

n: Stellenzahl

m: Anzahl Nachkommastellen

Hex -> Binär

Jede Ziffer der HEX Zahl wird durch zugehöriges binäres Nibble ersetzt.

Binär -> Dezimal

1. Jede Ziffer mit Stellenwert multiplizieren
2. Produkt addieren

-> Formel (Zahlensysteme mit festem Stellenwert)

Binär -> Oktal

1. Bilde Gruppen von 3 Bits, beginnend beim Komma. Fehlende Bits mit 0 ergänzen.
2. Ersetze gruppenweise durch Oktal-Ziffer.

Oktal -> Binär

Jede Ziffer der Oktal-Zahl wird durch zugehörige 3-stellige Binärzahl ersetzt.

Binär -> Hex

1. Bilde Gruppen von 4 Bits (=Nibble), beginnend beim Komma. Fehlende Bits mit 0 ergänzen.
2. Ersetze gruppenweise Nibbles durch Hex-Ziffer.

Dezimal ->Binär

Die Dezimalzahl 109, $78125_{(10)}$ ist in eine Dualzahl umzuwandeln.

Lösung für ganze Zahl:

109	:	2 = 54	Rest	1	
	:	2 = 27	Rest	0	
	:	2 = 13	Rest	1	
	:	2 = 6	Rest	1	$109_{(10)} = 110'1101_{(2)}$
	:	2 = 3	Rest	0	
	:	2 = 1	Rest	1	
	:	2 = 0	Rest	1	Leserichtung

0,78125	*	2 = 0,5625	+	1	Leserichtung
	*	2 = 0.125	+	1	
	*	2 = 0.25	+	0	$0,78125_{(10)} = 11001_{(2)}$
	*	2 = 0.5	+	0	
	*	2 = 0	+	1	

Resultat: $109,78125_{(10)} = 110'1101,1100'1_{(2)}$

Einerkomplement

Negative Zahl = Positive Zahl invertiert

Vorteil: Vorzeichen am ersten Bit erkennbar

Nachteil: ± 0

Zweierkomplement

Negative Zahl = Positive Zahl invertiert + 1

Vorteil: Vorzeichen am ersten Bit erkennbar; 0 existiert nur einmal; Addition und Subtraktion mit gleichem Algorithmus.

Gültigkeit einer Addition / Subtraktion im Zweierkomplement

	Richtiges Ergebnis	Überlauf
A+B	$c_n = 0$, $c_{n-1} = 0$	$c_n = 0$, $c_{n-1} = 1$
A-B	$c_n = c_{n-1}$	Nicht möglich
-A-B	$c_n = 1$, $c_{n-1} = 1$	$c_n = 1$, $c_{n-1} = 0$

Wann ist Resultat einer Addition oder Subtraktion gültig?

Wert einer Zahl im Zweierkomplement

$$z = -(2^{n-1}) \cdot a_{n-1} + \sum_{k=0}^{n-2} a_k \cdot 2^k$$

z: Wert der Zahl (im Dezimalsystem)

a: Nennwert der Ziffer

k: Stellenwert

n: Stellenzahl

Einführung und Generelles

Eigenschaften von C

> Imperativ (Folge von Befehlen in definierter Reihenfolge)

> Prozedural

> Case-sensitive

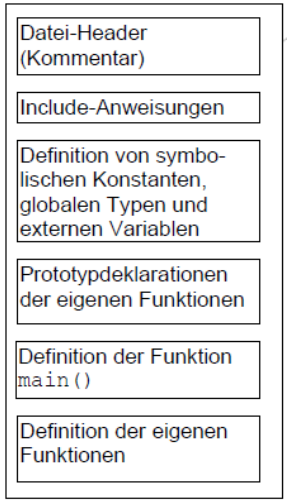
Vom Problem zum Programm

1. Erfassen der Problemstellung
2. Lösungsansatz formulieren
3. UML (Aktivitätsdiagramm), Struktogramm, Pseudocode
4. Implementierung in Programmiersprache

Grundkonzepte von imperativ prozeduralen Programmiersprachen

- Sequenz (Folge von Anweisungen)
- Selektion (Auswahl)
- Iteration (Schleife)
- Abstraktion (Unterprogramme)

Struktur Quelldatei



Einige Begriffe

Toolchain: alle Werkzeuge, die zur Erstellung und Installation einer Software benötigt werden, insbesondere Compiler, Linker, etc.

Host System: System, auf dem entwickelt wird: Editor, Compiler, Linker (häufig ein PC mit Linux oder Windows)

Target System: System, auf dem die entwickelte Software ausgeführt wird, z.B. Linux, Windows, bei Embedded Systems häufig ein Microcontroller basiertes System

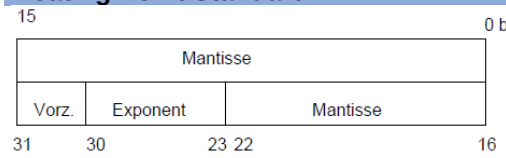
Cross Compiler: Ein Compiler der auf einem vom Target System verschiedenen Host System ausgeführt wird und Code für das Target System produziert.

Datentypen und Variablen

Datentyp

- legt fest, wie ein Bitmuster umgewandelt, bzw. interpretiert werden soll
- legt den Speicherbedarf einer Variablen fest
- definiert den Wertebereich einer Variablen
- definiert die Genauigkeit bei Gleitpunktzahlen (Floating point numbers)

Floating Point Standard



float	double
1 Vorzeichenbit	1 Vorzeichenbit
8 Bits Exponent (Bias)	11 Bits Exponent (Bias)
23 Bits Mantisse	52 Bits Mantisse

Deklaration und Definition

Deklaration

legt nur die Art und den Typ der Variable, bzw. die Schnittstelle der Funktion fest ohne

Definition

legt die Art und den Typ der Variablen bzw. Funktionen fest und reserviert Speicherplatz dafür

Definition = Deklaration + Reservierung des Speicherplatzes

Speicherplatz zu reservieren

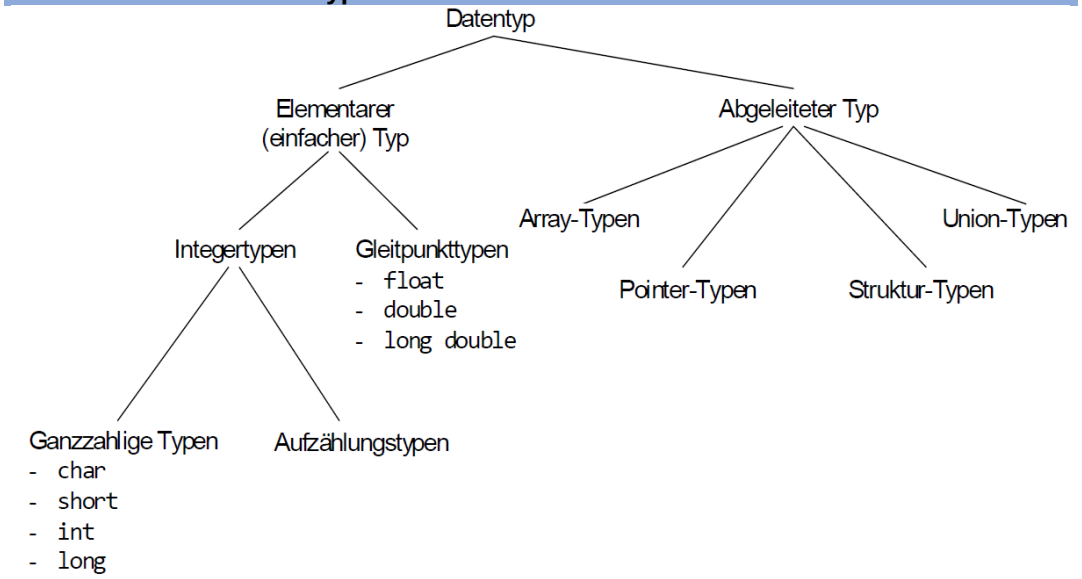
Attribut *volatile*

- Eine mit volatile gekennzeichnete Variable wird nicht (weg-)optimiert durch den Compiler, Adressen der Variablen werden nicht geändert
- Variablen können auch ausserhalb des Programmkontextes ihren Wert ändern, z.B. eine Variable, welche den Zustand von digitalen Eingängen repräsentiert.
- Eine Kopie in einem Prozessorregister wäre oft nicht aktuell. volatile sorgt dafür, dass der wirklicheWert bei jedem Zugriff gelesen wird.
- volatile verhindert, dass eine Variable in ein Prozessorregister gelegt wird

Welche Variablen müssen mit *volatile* gekennzeichnet werden?

- Alle Variablen, die ausserhalb des Programmkontextes geändert werden können
  - Variablen, die speziell bei Embedded Systems ein Hardwareregister darstellen
  - Globale Variablen, auf die in mehreren Tasks (concurrency, gleichzeitige Ausführungseinheiten) zugegriffen wird
  - Globale Variablen, auf die in Interruptserviceroutinen (ISR) zugegriffen wird

Klassifikation von Datentypen



Blöcke

Generell

- Der Rumpf einer Funktion ist ein Block (mehr dazu heute)
- Anweisungsfolgen können mit Hilfe von Blocks gegliedert werden
- Alle in einem Block getätigten Vereinbarungen (z.B. Variablendefinitionen) sind genau innerhalb dieses Blocks gültig und sichtbar, ausserhalb nicht

Gültigkeit einer Variable

- Eine Variable ist an einer bestimmten Stelle gültig, wenn ihr Name an dieser Stelle dem Compiler durcheine Vereinbarung bekannt ist
- Gültige Variablen können für den Programmierer unsichtbar sein, wenn sie durch eine andere Variabledesselben Namens verdeckt werden

Funktionen

Deklaration

```
Rückgabetyp Funktionsname(Parameterliste...);
```

Definition

```
Rückgabetyp Funktionsname(Parameterliste...) { Funktionsrumpf }
```

Syntaktisch und semantische Korrektheit

Syntaktisch korrekt

```
double sin(double arg);
```

Nicht klar ob *arg* in Radiant oder Grad

Syntaktisch und semantisch korrekt

```
double sin(double rad);   oder   double sin(double deg);
```

Aufruf

```
Funktionsname(Argumentliste...);
```

Funktion muss bekannt sein beim Aufruf!

inline

```
static inline int max(int a, int b);
```

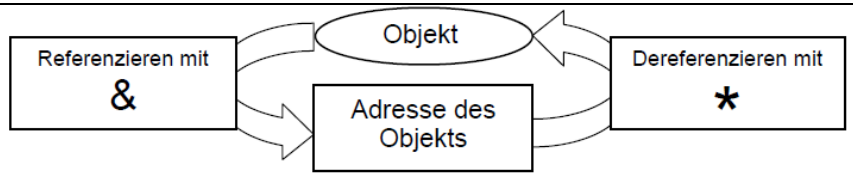
- Lösen das Overhead-Problem
  - Code wird direkt eingefügt, kein Funktionsaufruf
- Typenprüfung findet statt
- Einsetzen wenn der Codeumfang der Funktion sehr klein ist und die Funktion häufig aufgerufen wird (z.B.
- in Schleifen)
- Achtung: Rekursive Funktionen und Funktionen, auf die mit einem Funktionspointer gezeigt wird, werden nicht inlined.
- Die Inline-Funktionen müssen in Headerfiles definiert sein, damit der Compiler auch Inlining machen kann
- Wenn die Funktionen `static` deklariert werden, wird garantiert, dass die Funktionen nicht auch noch im Objectfile als Funktion vorhanden sind.
- Inlining wird nur gemacht, wenn dem Compiler auch eine Optimierungsstufe mitgegeben wird, z.B.`clang -c -O3 foo.c`

Pointer

Generell

- Der Speicherbedarf einer Pointervariablen ist unabhängig vom Pointer-Typ. Er ist so gross, dass die maximale Adresse Platz findet (z.B. 32 Bit)
- Pointer unterschiedlicher Datentypen dürfen einander nicht zugewiesen werden (Schutzmechanismus)
- Einem Pointer eines bestimmten Typs dürfen Pointer dieses Typs oder void-Pointer zugewiesen werden
- Einem void-Pointer dürfen beliebige Pointer zugewiesen werden (nützlich aber gefährlich)

```
Typ* Zeigername;  // Zeigername ist ein Pointer auf Typ
```

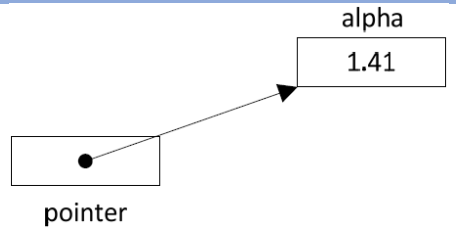


Pointerarithmetik

- Zu einem Pointer darf eine ganze Zahl oder ein anderer Pointer desselben Typs addiert werden
- Von einem Pointer kann eine ganze Zahl oder ein anderer Pointer desselben Typs subtrahiert werden
- Wenn eine ganze Zahl n addiert / subtrahiert wird, so bewegt sich der Pointer auf das nächste Element des Pointertyps. Die Zahl n wird also nicht als Byte interpretiert, der Pointer bewegt sich um n * sizeof(Typ) Bytes

Beispiel

```
float alpha;
float* pointer;
alpha = 1.41f;
pointer = &alpha;
```



Beispiel

```
int wert;          // Variable wert vom Typ int wird definiert
int* ptr;          // Pointer ptr auf den Typ int wird definiert
                   // ptr zeigt auf eine nicht definierte Adresse
ptr = &wert;       // ptr zeigt nun auf die Variable wert, d.h. ptr enthält die Adresse
                   // der  Variablen wert
*ptr = 23;         // in die Speicherzelle, auf welche ptr zeigt (hier: auf die Variable
                   // wert), wird 23 geschrieben. Äquivalent: wert = 23;
```

Beispiel

```
int a = 1,
b = 2,
result = 0;

int *pA = &a,
*pB = &b,
*pResult = &result;

int *pResult2 = pResult;
int **ppResult = &pResult;

result = a + b;
```

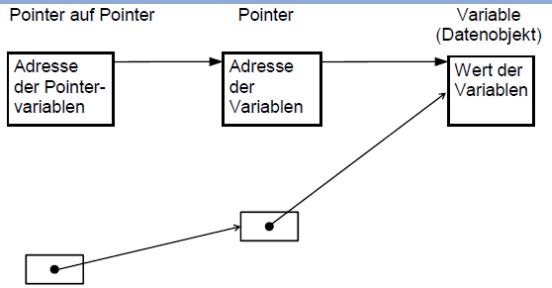
1. Variablen vom Typ Integer werden definiert und initialisiert
2. Zu jeder Variable wird ein Zeiger definiert und mit den Adressen der Variablen initialisiert
3. Wird...
   a. ...der Zeiger `pResult2` definiert und mit der Adresse von `result` (Wert von `pResult`) initialisiert; Es wird also der Zeiger `pResult` kopiert
   b. ...der *Zeiger auf einen Zeiger* `ppResult` definiert und mit der Adresse des Zeigers `pResult` initialisiert
4. Nun gibt es für `a` und `b` je einen Zeiger und für `result` gibt es zwei direkte Zeiger und einen Zeiger auf den ersten Zeiger

Beispiel: Pointer auf ein Array

```
int arr[5];
int (*alpha)[5] = &arr;
printf("Adresse von alpha : %p\n", alpha);
printf("Adresse von (alpha+1): %p\n", alpha+1);
```

- **alpha ist ein Pointer auf einen Array von 5 ints, d.h. alpha zeigt auf einen Bereich, der 5 · 4 = 20 Bytes gross ist.(alpha+1) liegt demnach 20 Bytes höher.**

Pointer auf Pointer



Pointer auf Funktion

- Der Name der Funktion kann als Adresse auf den ersten Befehl der Funktion verwendet werden (analog Array)
- Auf einen Funktionspointer kann nur lesend zugegriffen werden! Sonst löst eine Speicherverletzung aus.
- Interruptvektortabelle: Tabelle von Funktionspointer
  - ISR = Interrupt Service Routine

Pointer auf ISR n
...
...
Pointer auf ISR 2
Pointer auf ISR 1

```
/* pFunc ist ein Pointer auf eine Funktion, die ein int
zurückgibt */
int (*pFunc)(....);
/* pFoo ist ein Pointer auf eine Funktion, die einen
int-Pointer zurückgibt */
int* (*pFoo)(....);
```

Funktion mit Rückgabewert "Pointer auf ein Array von char-Pointer"

( ) hat immer Vorrang, [ ] hat Vorrang vor *

```
/* delta ist eine Funktion mit der Parameterliste (....) und
gibt einen Pointer auf ein Array von 10 Pointern auf char
zurück */
char* (*delta(....))[10];
```

Leseweise

Von rechts nach links lesen!

`const int alpha[];`	"alpha ist ein Array von int-Konstanten" (konstantes Array)
`const int arr[] = {12, 56};`	"arr ist ein Array von int-Konstanten"
`const char* text = str;`	"text ist ein Pointer auf eine char-Konstante"
`char* const text = str;`	"text ist ein konstanter Pointer auf ein char"
`const char* const text = str;`	"text ist ein konstanter Pointer auf eine char-Konstante"

`char* pointer[];`	"pointer ist ein Array von Pointer auf char"
`struct* const cpStruct;`	"cpStruct ist ein konstanter Pointer auf ein struct"
`int (*alpha)[8];`	"alpha ist ein Pointer auf ein Array von acht int-Werten"

Arrays (und Standardfunktionen für den Umgang mit Speicher)

Generell

```
Typ Name[n];  // Name ist ein Array (Pointer auf das erste Element)
              // vom Typ Typ und beinhaltet n Elemente
```

- Globale Arrays werden automatisch mit 0 initialisiert
  - globale Arrays sollten aber nur ausnahmsweise verwendet werden
- Lokale Arrays werden nicht automatisch initialisiert
  - der Inhalt eines lokalen Arrays ist bei der Definition undefiniert

Arraygrösse (oder Arraylänge)

```
sizeof(arr)/sizeof(arr[0])
```

String

- Array von chars mit letztem char = 0
- `char str[5] = { 'h', 'a', 'l', 'l', 'o', '\0' };`

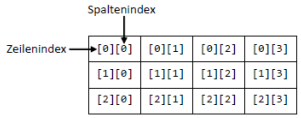
Vergleiche von Arrays

- In C gibt es keinen Operator ==, der zwei Arrays miteinander vergleicht
- Arrayvergleiche müssen explizit Element um Element durchgeführt werden
- oder: der Inhalt der beiden Speicherbereiche wird mit Hilfe der Funktion `memcmp()` byteweise verglichen

Mehrdimensionale Arrays

```
Typ Name[n][m];
```

- Matrix



Übergabe von mehrdimensionalem Array

> Der erste Array kann offen sein, die weiteren müssen explizit angegeben werden.

> Der Aufruf erfolgt mit

```
printMat(matA, sizeof(matA)/sizeof(matA[0]), cols);
```

```
void printMat(const double mat[][cols],     // Matrix
              int m,                        // Anzahl Zeilen
              int n);                       // Anzahl Spalten
```

Standardfunktionen für Strings

- Funktionen für Strings und Speicher sind prinzipiell dasselbe
- Werden in <string.h> angeboten
- Funktionen, die mit str beginnen:
  Dienen der Stringverarbeitung, erkennen das '\0'-Zeichen
- Funktionen, die mit mem beginnen:
  Dienen der Speicherverarbeitung, erkennen das '\0'-Zeichen nicht, deshalb muss die Bufferlänge in Byteebenfalls als Parameter übergeben werden

Strings kopieren

```
#include <string.h>
char* strcpy(char* dest, const char* src);
char* strncpy(char* dest, const char* src, size_t n);
```

- **string copy**
- kopiert (bei `strncpy()` maximal `n` chars) von `src` nach `dest`, inklusive '\0'
- return: dest
- `dest` muss bei `strcpy()` auf einen Bereich zeigen, der genügend gross ist
- Achtung: Ist der zu kopierende Buffer bei `strcpy()` grösser als der Zielbuffer, dann werden nachfolgende Speicherbereiche überschrieben (Buffer overflow)
- Deshalb sollte `strncpy()` bevorzugt werden

Strings zusammenfügen

```
#include <string.h>
char* strcat(char* dest, const char* src);
char* strncat(char* dest, const char* src, size_t n);
```

- **string concatenate**
- hängt (bei `strncat()` maximal `n` chars) von `src` an `dest` an, inklusive '\0'
- Das ursprüngliche '\0' von `dest` wird überschrieben
- return: `dest`
- `dest` muss auf einen Bereich zeigen, der genügend gross ist
- Achtung: Ist der zu kopierende Buffer grösser als der Zielbuffer, dann werden nachfolgende Speicherbereiche überschrieben (Buffer overflow)
- Deshalb sollte `strncat()` bevorzugt werden

Strings vergleichen

```
#include <string.h>
int strcmp(const char* s1, const char* s2);
int strncmp(const char* s1, const char* s2, size_t n);
```

- **string compare**
- vergleicht die beiden Strings, auf die `s1` und `s2` zeigen, bei `strncmp()` nur die ersten `n` char's
- return:
  - < 0 : *s1 ist lexikographisch kleiner als *s2
  - == 0 : *s1 und *s2 sind gleich
  - > 0 : *s1 ist lexikographisch grösser als *s2

Stringlänge bestimmen

```
#include <string.h>
size_t strlen(const char* s);
```

- string length
- bestimmt die Länge des Strings s, d.h. die Anzahl char's. Das Stringende-Zeichen '\0') wird dabei nichtmitgezählt!
- return: Länge des Strings

Standardfunktionen für Speicher

- Formelle Parameter sind vom Typ void* statt char*
- Die mem-Funktionen arbeiten byteweise
- Im Gegensatz zu den str-Funktionen wird das '\0'-Zeichen nicht speziell behandelt
- Die Bufferlänge muss als Parameter übergeben werden

Funktionen zur Speicherbearbeitung

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

- Speicherbereich kopieren

```
void* memcpy(void* dest, const void* src, size_t n);
```

- Speicherbereich verschieben

```
void* memmove(void* dest, const void* src, size_t n);
```

- Speicherbereiche vergleichen

```
int memcmp(const void* s1, const void* s2, size_t n);
```

- "Zeichen" (Byte mit ASCII-Code c) in Speicherbereich suchen

```
void* memchr(const void* s, int c, size_t n);
```

- Speicherbereich mit Wert belegen

```
void* memset(const void* s, int c, size_t n);
```

Bei memcpy() dürfen sich die Buffer nicht überlappen, memmove() kann auch mit überlappenden Buffernumgehen

Lexikalische Konventionen

Bestandteile des Source-Codes

> Lexikalische Elemente ("Text"):

Befehle, Variablen, Deklarationen, ...

> Trennzeichen:

Leerzeichen, Tabulatoren, Newline

> Preprocessor-Direktiven

Zeichenvorrat von C

> Grosse und kleine Buchstaben, keine Umlaute:	**A – Z, a – z**
> Underscore:	**_**
> Ziffern:	**0 – 9**
> Leerzeichen (Blank, Space), Tabulator:	
> Strichpunkt (Semikolon):	**;**
> Punkt (Dezimaltrennung, Selektionsoperator):	**.**
> Sonderzeichen für Operatoren:	**()[]<>+- */%^~&|=!?:,**
> Hochkomma, Doppelhochkomma:	**' "**
> Geschweifte Klammern (curly braces):	**{ }**
> (Hash), Sharp, "Gartenhag":	**#**
> Backslash:	****

Namen

- Namen können in C bestehen aus
  - Buchstaben **a-z, A-Z**
  - Ziffern **0-9**
  - Underscore _
- **Das erste Zeichen eines Namens darf keine Ziffer sein**

Styleguide: Namen

- Variablen
  - mit Kleinbuchstaben beginnen
  - erster Buchstabe von zusammengesetzten Wörtern ist gross (camelCase)
  - keine Underscores
  - Beispiele: `counter, maxSpeed`
- Funktionen
  - mit Kleinbuchstaben beginnen
  - erster Buchstabe von zusammengesetzten Wörtern ist gross
  - keine Underscores
  - Beispiele: `getCount(), init(), setMaxSpeed()`

Symbolische und literale Konstanten

Symbolische Konstanten

- Haben Namen, der ihren Wert repräsentiert
- #define

```
#define PI 3.14159
#define LIST_LENGTH 40
```

- Verhindern von "magic numbers"
- Vereinfacht Änderungen
- Der Preprocessor ersetzt überall im Code den definierten Namen (stur 1:1) durch den entsprechenden Wert
- Symbolische #define-Konstanten haben keinen Typ und vor allem auch keinen Scope. Deshalb sollen sie wenn immer möglich durch enums oder consts ersetzt werden!

```
enum {listLength = 40,
      commandLength = 8,
      dateLength = 128};
```

Literale Konstante

- Haben keinen Namen
- Werden durch ihren Wert repräsentiert

Ganzzahlige Konstanten (default: int)

```
254    035    0x3f    -34    14L    14U    14UL
dezimal okt    hex            long   unsigned unsigned long
```

Zeichenkonstanten

```
'c'    '\n'    '\x4a'    '\14'    '\\'    L'a'
               ASCII hex ASCII okt        Double Byte
```

Gleitpunktkonstanten (default: double)

```
254.89   -13.0   3.45e23      4.65f      3.14159L
                 Exp-Schreibweise float-Konstante long double
```

- Aufzählungskonstante (enum)

Enumeration (Aufzählungstyp)

```
enum State {idle, // 0
            standby = 34, // 34
            startingUp, // 35
            running, // 36
```

```
            blocked = 897, // 897
            shuttingDown}; // 898
enum State s = idle;
```

- Aufzählungskonstanten haben einen konstanten ganzzahligen Wert
- Die erste Konstante erhält den Wert 0, die zweite 1, etc.
- Die jeweils nächste Konstante wird automatisch um 1 erhöht
- Werte können auch explizit zugewiesen werden

const

- "Variablen", die einen festen Wert annehmen und nicht mehr geändert werden sollen, können mit const gekennzeichnet werden.
- Sie sind dann keine Variablen, sondern Konstanten
- Konstanten müssen bei der Definition zwingend initialisiert werden

```
const int listLength = 40; // enum wäre auch möglich
const double pi = 3.14159; // enum nicht möglich
```

Anweisungen, Ausdrücke und Operatoren

Operatoren und Operanden

- unär (monadisch): hat einen einzigen Operanden, z.B. Minusoperator (Vorzeichenwechsel)
- binär (dyadisch) : hat zwei Operanden (z.B. `3+4`)

Postfix- und Präfixoperatoren

- Postfix-Operatoren sind unäre Operatoren, die nach (post) ihrem Operanden stehen.

Beispiel:

```
printf("%d", i++);
```

i wird auf den Bildschirm geschrieben, **anschliessend** inkrementiert.

- Präfix-Operatoren sind unäre Operatoren, die vor (prä) ihrem Operanden stehen.

Beispiel:

```
printf("%d", ++i);
```

i wird **zuerst** inkrementiert, dann auf den Bildschirm geschrieben.

Ausdrücke und Anweisungen

- Ausdruck
  - hat immer einen Rückgabewert (Typ ist durch Operanden bestimmt)
  - kann Teil eines grösseren Ausdrucks sein
  - kann eine Anweisung werden
  - Beispiel: `3 + 4.5`
- Anweisung
  - kann keinen Rückgabewert haben
  - kann nicht Teil eines grösseren Ausdrucks sein
  - Selektions-, Iterations-, Sprung-, Ausdrucksanweisung
  - Beispiel: `while (a > 14)`
- Nebeneffkte
  - Bei Nebeneffekten wird nebst der eigentlichen Aufgabe noch eine weitere Operation in dieselbe Anweisung gepackt (häufig mit Präfix- und Postfix-Operatoren)
  - zurückhaltend einsetzen
  - Beispiel:

```
j = i++;
```

ist äquivalent zu:

```
j = i;
i = i+1; // i++
```

Komma-Operator (Schleifen)

```
int j = 0;
int i = 0;
for(; i < 10; i++, j--) {
      //...
}
```

- In einem Loop kann mittels Komma-Operator eine zweite Aktion nach jedem Durchlauf angefügt werden.
- So kann zum Beispiel gleichzeitig `i` hoch und `j` runter gezählt werden. Dies funktioniert dann auch wenn der Loop `continue` beinhaltet.

Auswertungsreihenfolge (operator precedence)

1. Wie in der Mathematik werden als erstes Teilausdrücke in Klammern ausgewertet.
2. Dann werden die Operatoren der Ausdrücke entsprechend festgelegter Prioritäten abgearbeitet.
3. Wenn zwei Operatoren die gleiche Priorität besitzen, so wird mit der sogenannten Assoziativität eine Abarbeitungsrichtung festgelegt, sprich, ob die Operatoren von rechts nach links oder von links nach rechts abgearbeitet werden.

- Für jeden Operator ist die Priorität und die Assoziativität definiert.

Achtung: Assoziativität legt die *Reihenfolge der Operatoren* fest, sie sagt aber *nichts über die Reihenfolge der Auswertung der Operanden* aus (die Reihenfolge der Auswertung legt C nicht fest!)

Beispiel

```
*p++
```

Zuerst wird der Pointer p inkrementiert, der Rückgabewert von p++ ist das noch nicht inkrementierte p! Dieser Rückgabewert (das noch nicht inkrementierte p) wird nun dereferenziert.

Beispiel

```
*++p
```

Zuerst wird der Pointer inkrementiert, der Rückgabewert von ++p ist der inkrementierte Pointer, dann wird (das inkrementierte) p dereferenziert.

Beispiel

```
(*p)++
```

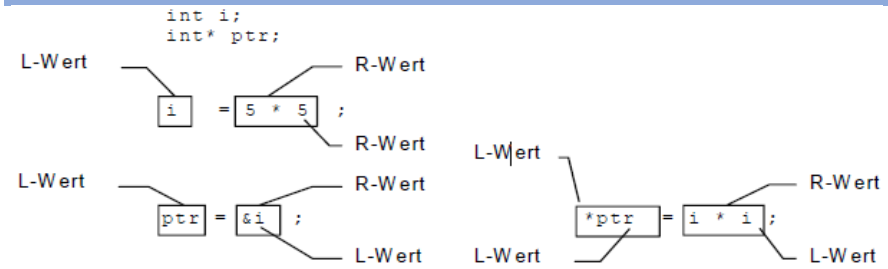
Zuerst wird wegen der Klammer der Pointer dereferenziert, anschliessend wird *p, d.h. der Inhalt der Speicherzelle, auf welche p zeigt, inkrementiert.

Beispiel

- Der Ausdruck **n++ - n** ist nicht definiert!
  - entweder `n - (n+1) = -1`, falls zuerst `n++` ausgewertet wird
  - oder `n - n = 0` , falls zuerst `n` ausgewertet wird
  - in jedem Fall ist aber `n` am Schluss um `1` inkrementiert

Priorität	Operatoren		Assoziativität
Priorität 1	() [ ] -> . ++ -- (Typname) {}	Funktionsaufruf Array-Index Komponentenzugriff Inkrement, Dekrement als Postfix compound literal[96]	links links links links links
Priorität 2	! ~ ++ -- sizeof + - (Typname) * &	Negation (logisch, bitweise) Inkrement, Dekrement als Präfix Vorzeichen (unär) cast Dereferenzierung, Adresse	rechts rechts rechts rechts rechts rechts
Priorität 3	* / %	Multiplikation, Division modulo	links links
Priorität 4	+ -	Summe, Differenz (binär)	links
Priorität 5	<< >>	bitweises Schieben	links
Priorität 6	< <= > >=	Vergleich kleiner, kleiner gleich Vergleich größer, größer gleich	links links
Priorität 7	== !=	Gleichheit, Ungleichheit	links
Priorität 8	&	bitweises UND	links
Priorität 9	^	bitweises Exklusives-ODER	links
Priorität 10	|	bitweises ODER	links
Priorität 11	&&	logisches UND	links
Priorität 12	||	logisches ODER	links
Priorität 13	?:	bedingte Auswertung	rechts
Priorität 14	= +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=	einfache Wertzuweisung kombinierte Zuweisungs-operatoren	rechts rechts
Priorität 15	,	Komma-Operator	links

L-Werte und R-Werte




```
a = 5 * a;          // ok
6 = a;              // nicht zulässig, 6 ist kein lvalue
a * b = c;          // nicht zulässig, (a*b) ist kein lvalue
b = 3++;            // nicht zulässig, 3 ist kein lvalue
```

Zugriff auf L- und R-Werte

- Ein lvalue erfordert immer Schreibzugriff
- Auf einen rvalue wird nur lesend zugegriffen

Bitwise Operators

- Bitweises AND `A & B`
- Bitweises OR `A | B`
- Bitweises NOT (Inverter) `~A`
- Bitweises XOR `A ^ B`

```
int a = 11, b = 9;
c = a & b;
a  0000 1011
b  0000 1001
-------------
c  0000 1001
```

Schiebe- (Shift-) Operator

- Rechts-Shift um n Bits `A >> n`
- Links-Shift um n Bits `A << n`

Bedingungsoperator (Ternärer Operator)

`A ? B : C`

Ist eine verkürzte Schreibweise für

```
if (A)
      "return" B; // Returnwert des Ausdrucks ist B
else
      "return" C;
```

Beispiel: Maximum von zwei Zahlen a, b ermitteln

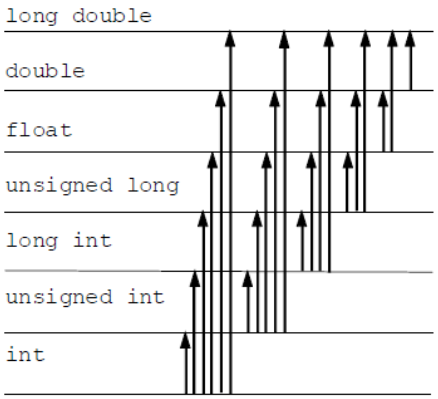
```
int maximum = a > b ? a : b;
```

Explizite Typumwandlung

- Nebst den impliziten (automatischen) Typumwandlungen kann mit Hilfe des cast-Operators eine explizite Typumwandlung bewirkt werden.
- Der Programmierer ist bei einer expliziten Typumwandlung verantwortlich, dass die Umwandlung keine Probleme ergibt. (z.B. Umwandlung von grosser Zahl in kleineren Typ)
- Syntax: `(Zieltyp)Ausdruck`
- Beispiel:

```
int a = (int)4.6; // a ← 4
double d = (double)a; // d ← 4.0
```

Implizite Typumwandlung



Syntaktisch erlaubte Typumwandlungen

- Zwischen skalaren Typen (Integer, Floating Points, Pointer)
- Von skalarem Typ in `void`

Sequenzpunkte

- Der ISO-Standard für C gibt Punkte in der Ausführungssequenz eines Programms an, an denen die Nebeneffekte vorangegangener Ausdrücke alle eingetreten sein müssen.
- Diese Punkte werden als **Sequenzpunkte** (sequence points) bezeichnet
- An einem solchen Sequenzpunkt dürfen Nebeneffekte von Ausdrücken, die im Programm nach einem Sequenzpunkt stehen, auf keinen Fall stattgefunden haben.
- Bei arithmetischen Operatoren liegen z.B. nie Sequenzpunkte vor, d.h. der Compiler ist frei, zuerst den linken oder den rechten Operanden zu berechnen

Wo liegen Sequenzpunkte vor?

- Bei Funktionsaufrufen: nach der Berechnung der Argumente und des Funktionsbezeichners, bevor die Funktion aufgerufen wird. (Für die Berechnung der Argumente ist jedoch keine Reihenfolge definiert.)
- Bei den folgenden Operatoren jeweils nach der Auswertung des ersten Operanden und vor der Auswertung des zweiten Operanden (gemäss der Abarbeitungsrichtung des Operators):
  - logisches AND &&, logische OR ||
  - Bedingungsoperators, d. h. der Bedingung A in A ? B : C
  - Komma-Operator
- Am Ende der Auswertung der folgenden Ausdrücke:
  - manuelle Initialisierung
  - eines Ausdrucks in einer Ausdrucksanweisung
  - der Bedingung in einer if-Anweisung
  - des Selektionsausdrucks in einer switch-Anweisung
  - der Bedingung einer while- oder do while-Schleife
  - eines jeden der drei Ausdrücke einer for-Anweisung
  - des Ausdrucks einer return-Anweisung

Strukturen

Generell

- Daten, welche logisch zusammengehören, können zusammengenommen werden
- Die Struktur ist ein zusammengesetzter Datentyp, sie setzt sich aus den Feldern zusammen
- Die einzelnen Felder der Struktur können (aber müssen nicht) unterschiedliche Typen haben
- Jedes Feld wird mit einem innerhalb der Struktur eindeutigen Namen versehen
- Strukturen können verschachtelt werden

Syntax

```
struct S {
    int x;
    int y;
};

struct S myS = {1,
2};
```

- Der Bezeichner *S* wird im *struct* Namensbereich definiert, bzw. der Typ *struct S* wird definiert
- Die Variable `myS` vom Typ *struct S* wird definiert und initialisiert

```
struct S {
    int x;
    int y;
} myS;

myS = {1, 2};
```

- Der Bezeichner *S* wird im *struct* Namensbereich definiert, bzw. der Typ *struct S* wird definiert und weiter wird die Variable `myS` von diesem Typ definiert
- Die Variable `myS` wird initialisiert

```
typedef struct S {
    int x;
    int y;
} S;

S myS = {1, 2};
```

- Der Typ *struct S* wird definiert und mit ihm der Typ *S* der den Typ *struct S* hat, also ein Alias auf *struct S* ist
- Die Variable `myS` vom Typ *S* (*struct S*) wird definiert und initialisiert

Operationen auf Strukturen

- Zuweisung
  - liegen zwei Strukturvariablen a und b vom gleichen Strukturtyp vor, so kann der Wert der einen Variablen der anderen zugewiesen werden

    ```
    a = b;
    ```
- Selektion eines Feldes
  - Direkt: `s.member1`
  - Pointer: `s->member1`
- Ermittlung der Grösse der Struktur
  - mit sizeof-Operator
- Ermittlung der Speicheranordnung der Struktur (ab C11)
  - mit _Alignof-Operator
- Ermittlung der Adresse der Strukturvariablen
  - mit Adressoperator &

Zugriff auf Strukturmember

Beispiel: Punktoperator

```
struct S {
    char *name;
    int age;
} s;

s.name = "Fabian";
s.age = 21;

printf("Name: %s, Age: %d",
      s.name,
      s.age
      );
```

Auf Strukturmember kann mit dem Punkt-Operator (.) zugegriffen werden.

Beispiel: Pfeiloperator

```
struct S {
    char* name;
    int age;
} *s;

s = (struct S*) malloc(sizeof(struct S));
s->name = "Fabian";
s->age = 21;

printf("Name: %s, Age: %d",
      s->name,
      s->age
      );
```

Wenn ein Zeiger auf eine Struktur vorliegt, kann mit dem Pfeil-Operator (->) auf Strukturmember zugegriffen werden. Der Pfeil-Operator ist äquivalent zu `(*s).member`

Beispiel: Zugriffsverletzung

```
struct S{
      char* letter;
} s;

*(s.letter) = 'a';
// Zugriffsverletzung! letter zeigt auf undefinierte Adresse!
```

```
struct S{
      char* letter;
} s;

char a = 'a';
s.letter = &a;    // Korrekt
```

Lage im Speicher

- Die Felder einer Strukturvariablen werden nacheinander gemäss der Definition in den Speicher plaziert.
- Lesen und Schrieben aus/in den Speicher geht am effizientesten, wenn die Daten ausgerichtet sind (aligned). Die Ausrichtung der Daten orientiert sich an der Wortgrösse des Systems.
- Durch das Alignment kann es vorkommen, dass einzelne Bytes nicht verwendet werden, d.h. im Speicher ausgelassen werden.
- Die exakte Grösse einer Strukturvariablen kann nicht durch Addieren der Grössen der Felder ermittelt werden, nur sizeof() liefert den genauen Wert
- Die Speicheranordnung der Struktur kann ab C11 mit dem _Alignof-Operator ermittelt werden

11	zahlLoLo
10	zahlLoHi
9	zahlHiLo
8	zahlHiHi
7	Not Used
6	text[2]
5	text[1]
4	text[0]
3	wertLoLo
2	wertLoHi
1	wertHiLo
0	wertHiHi

Damit das Feld `zahl` auf einer 4 Byte Grenze liegt, wurde die siebte Adresse ausgelassen. `zahl` wurde ausgerichtet (aligned).

(Zahlendarstellung in Big endian.)

```
struct B
{
    int wert;
    char text[3];
    int zahl;
};
```

Initialisierung einzelner Strukturmember

```
struct Angestellter maier =
{
      .personalnummer = 56321,
      .name = "Maier"
};
```

- Die nicht explizit initialisierten Felder werden automatisch auf Null gesetzt

Unions

Generell

- ähnlich der Struktur
- beinhaltet auch mehrere Felder unterschiedlichen Typs
- **im Gegensatz zur Struktur ist aber nur ein einziges Feld jeweils aktiv (abhängig vom Typ)**
- **Die Grösse einer Union ist so gross wie das grösste Feld in der Union**
- Bei der Union sind dieselben Operationen wie bei einer Struktur definiert (siehe dort)

Typedef

Generell

- Vor allem bei komplexeren Typen (z.B. bei `structs`) macht es Sinn, einen eigenen (Alias-)Namen für den Typ zu definieren. Das Schlüsselwort `struct` kann anschliessend bei der Definition von Strukturvariablen weggelassen werden.

- Eigene Typen können mit dem Befehl `typedef` definiert werden.
- Zwischen dem Schlüsselwort `struct` und `{` wird üblicherweise kein zusätzlicher Name definiert.
- `typedefs` werden im Normalfall global, d.h. ausserhalb einer Funktion definiert.
- Konvention: eigene Typen werden mit einem Grossbuchstaben begonnen
- Ein `typedef` ist eine reine Textersetzung

Beispiel

```
typedef enum {black, red, green, yellow, blue} Color;

typedef struct
{
      int x;
      int y;
} Point;

typedef struct
{
      Point p1;
      Point p2;
      Color col;
} Line;

int main (void)
{
      Line myLine = {{12, -34}, {783, 12}, yellow};
      printf ("Anfangspunkt: (%d, %d)\n", myLine.p1.x,
      myLine.p1.y);
      printf ("Endpunkt: (%d, %d)\n", myLine.p2.x,
      myLine.p2.y);
      printf ("Farbe: %d\n", myLine.col);
      return 0;
}
```

Portabilität

- Oft muss z.B. in ein Register ein 16 Bit breiter Wert geschrieben werden. Welcher Typ ist nun 16 Bit breit?
- Das ist implementationsabhängig
  - (vielleicht `unsigned short, unsigned int, …`)
- Um die Portabilität (Umschreiben auf ein anderes System) zu vereinfachen, wird ein 16 Bit breiter Datentyp (Word) definiert und dann ausschliesslich verwendet. Auf einem anderen System ist dann nur noch dieser typedef zu ändern.

```
typedef unsigned short uint16_t;
```

- Ab C99 werden im Headerfile `<inttypes.h>`, bzw. `<stdint.h>` verschiedene Typen mit genauer Breite definiert. Es ist deshalb sinnvoll, entweder genau diese Typen zu verwenden oder allenfalls diese mittels typedef zu definieren.
- Die Konventionen lauten wie folgt:
  - intN_t, wobei N = 8, 16, 32 für signed int-Typen
    Beispiele: `int8_t, int16_t, int32_t`
  - uintN_t, wobei N = 8, 16, 32 für unsigned int-Typen
    Beispiele: `uint8_t, uint16_t, uint32_t`

Namensräume

Generell

Namen, die zu **verschiedenen** Namensräumen gehören, dürfen auch innerhalb desselben Gültigkeitsbereichs gleich sein.

- In C gibt es die folgenden Namensräume:
  - Marken
  - Namen von Strukturen, Unions und enums
  - Jede Struktur und Union für ihre Feldnamen
  - Bezeichner von Variablen, Funktionen, typedef-Namen, enum-Konstanten

Rekursion

Generell

- Rekursion
  - Funktion enthält Abschnitte, in der sie selbst direkt oder indirekt wieder aufgerufen wird
- Iteration
  - Algorithmus enthält Abschnitte, die innerhalb einer Ausführung mehrfach durchlaufen werden (Schleife)
- Jeder rekursive Algorithmus kann auch iterativ formuliert werden
- Die rekursive Form kann eleganter sein, ist aber praktisch immer ineffizienter als die iterative Form
- Das Abbruchkriterium ist bei beiden Formen zentral

Anwendung von Rekursion

- Wachstums-Vorgänge (siehe später)
- Backtracking-Algorithmen
  - z.B. Finden eines Weges durch ein Labyrinth (zurück aus Sackgasse und neuen Weg prüfen)
- Traversierungen von Suchbäumen
- In Mathematik rekursiv definierte Algorithmen
- **Achtung: der Stackbedarf ist bei Rekursion schwer abzuschätzen. Rekursion ist deshalb bei Embedded Systems verpönt.**
- Indirekte Rekursion ist unbedingt zu vermeiden
- Diese kommt z.B. zustande, wenn sich zwei oder mehrere Funktionen wechselseitig oder im Kreis aufrufen

Beispiel: Fakultät

Mit Iteration

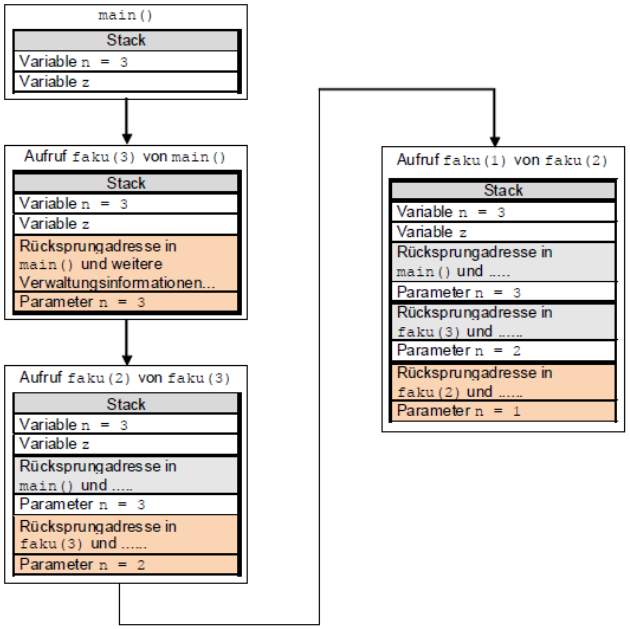
```
unsigned long faku(unsigned int n)
{
      unsigned long fak = 1UL;
      for (unsigned int i = 2; i <= n; ++i)
      {
```

```
            fak = fak * i;
      }
      return fak;
}
```

Mit Rekursion

```
unsigned long faku(unsigned int n)
{
      if (n > 1)
      {
            return n * faku(n-1);
      }
      else
      {
            return 1UL;
      }
}
```

Aufbau des Stacks



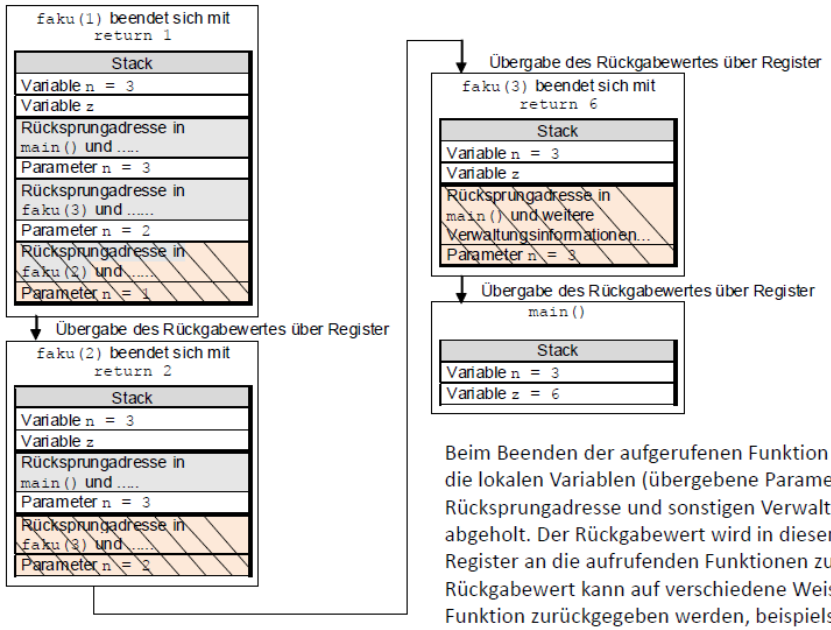
Bei jedem Aufruf von `faku()` werden die Rücksprungadresse und weitere Verwaltungsinformationen auf einem Stack abgelegt, der durch das Laufzeitsystem verwaltet wird.
Auch die übergebenen Parameter (hier nur einer) werden auf diesem Stack abgelegt.

Dabei wächst der Stack mit der Rekursionstiefe der Funktion.

Der letzte Aufruf von `faku()` mit dem Parameter `n = 1` bewirkt keine weitere Rekursion, da ja die Abbruch-bedingung erfüllt ist.

Der Abbau des Stacks geschieht in umgekehrter Reihenfolge, wie aus dem folgenden Bild ersichtlich wird.

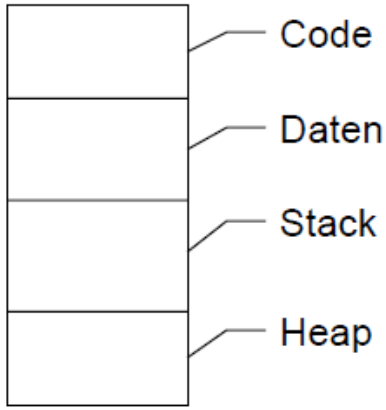
Abbau des Stacks



Beim Beenden der aufgerufenen Funktion werden auf dem Stack die lokalen Variablen (übergebene Parameter) freigegeben und die Rücksprungadresse und sonstigen Verwaltungsinformationen abgeholt. Der Rückgabewert wird in diesem Beispiel über ein Register an die aufrufenden Funktionen zurückgegeben. Der Rückgabewert kann auf verschiedene Weise an die aufrufende Funktion zurückgegeben werden, beispielsweise auch über den Stack. Dies ist vom Compiler abhängig.

Adressraum eines Programms

Speicherbereich eines C-Programms



- **Code**
  - Programm in Maschinencode
  - evtl. Konstanten, insbesondere Texte und grosse konstante Tabellen (Arrays)
- **Daten**
  - Globale Variablen, static Variablen
- **Stack**
  - Lokale Variablen
  - Parameter einer Funktion
  - Rücksprungadressen
- **Heap**
  - Dynamische Variablen (Speicherplatz wird erst zur Laufzeit angefordert)

Zugriff

- **Codesegment:**
  - Zugriff: lesen, ausführen im Flash EPROM oder im RAM (z.B. beim PC)
- **Datensegment, Stack, Heap:**
  - Zugriff: lesen und schreiben
  - im RAM, evtl. in Registern

Speicherklassen

Interne vs. externe Bindung

- Symbole mit interner Bindung liegen vollständig in einer Übersetzungseinheit (.c) vor und werden vollständig vom Compiler behandelt. Dazu gehören:
  - Lokale Variablen
  - static Variablen
  - Lokale Funktionen
- Symbole mit externer Bindung werden in mehreren Übersetzungseinheiten (.c) genutzt. Sie müssen überall bekannt sein, die Definition geschieht in genau einer Übersetzungseinheit. Die Auflösung nimmt der Linker vor. Dazu gehören:
  - Globale Variablen
  - Externe Funktionen
- Bei Symbolen mit externer Bindung muss geachtet werden, dass keine Namenskonflikte auftreten. Deshalb werden für diese Symbole Unit- (Modul-)kürzel verwendet

Speicherklasse extern

ext1.c

```
int zahl = 6;
```

ext2.c

```
extern int zahl;
```

- Eine externe Variable kann nur in einer einzigen Datei definiert werden (ohne extern)
- In den anderen Dateien wird sie mit extern deklariert (bekannt gemacht)
- Eine manuelle Initialisierung ist nur bei der Definition möglich
- Globale Variablen, welche nicht manuell initialisiert werden, werden automatisch mit 0 initialisiert
- extern-Deklarationen werden üblicherweise in einer Headerdatei deklariert. Am Beginn der .c - Datei wird die Headerdatei mit #include eingefügt

Speicherklasse static

Variablen

- `static` Variablen sind im Datenbereich, nicht auf dem Stack
- Gültigkeitsbereich ist der Block, in dem die Variable definiert ist
- `static` Variablen, welche ausserhalb einer Funktion definiert sind (globale Variablen), sind nur in der Datei gültig, in der sie definiert werden
- `static` Variablen sind nur einmal vorhanden (auch in Multithreading-Umgebungen), d.h. ihr Wert wird erhalten, auch wenn die Funktion beendet ist. Beim nächsten Aufruf der Funktion geht es mit dem alten Wert weiter.

Funktionen

- `static` Funktionen sind im Codebereich (wie alle Funktionen), sie sind bloss während der Compilezeit nicht sichtbar für andere Dateien.
- static Funktionen sind nur in der Datei, in welcher sie definiert sind, sichtbar
- Alle Funktionen, welche nicht aussen (für andere Units) sichtbar sein sollen, sollten deshalb als static definiert werden
  - Mit anderen Worten: alle Funktionen static definieren, ausser jene, welche die Schnittstelle nach aussen bilden
- Ein versuchter Zugriff auf statische Elemente von einer weiteren Datei ergibt einen Linkfehler

Speicherklassen bei lokalen Variablen

- Externe und statische lokale Variablen werden automatisch mit 0 initialisiert. Es ist guter Programmierstil, auch diese Variablen explizit zu initialisieren.

Auto

- Variablen der Speicherklasse `auto` werden auf dem Stack angelegt.
- Wenn bei einer Variablen keine Speicherklasse angegeben wird, so hat sie "automatisch" dieSpeicherklasse `auto`, d.h. auto ist der Default.
- Die Speicherklasse `auto` wird praktisch nie explizit angegeben.
- Automatische Variablen werden nicht durch das System initialisiert. Ihr Wert ist deshalb vor der manuellen Initialisierung undefiniert.

Register

- Bei Angabe der Speicherklasse register wird dem Compiler empfohlen, für diese Variable ein Register (schnellste Speicherzelle) zu verwenden.
- Der Compiler wird dies versuchen, eine Garantie besteht aber nicht, da dem Compiler z.B. u.U. gar nicht genügend Register zur Verfügung stehen.

```
void beispiel(register long a)
{
      register int b;
      ...
}
```

- Bei formalen Parametern ist das die einzig mögliche explizite Speicherklasse.
- Der Compiler optimiert meist gut, register soll deshalb üblicherweise nicht angegeben werden.

Static

- siehe Speicherklasse static
- einziger Unterschied: Scope ist nur auf Block beschränkt

Scope

- Compiler arbeitet dateiweise
- Namen in einer anderen Datei sind dem Compiler nicht bekannt
- (Globale) Variablen, welche in einer anderen Datei definiert werden, können mit Hilfe des extern-Statements bekannt gemacht werden
- Symbolen mit externer Bindung soll ein Modul- (Unit-)kürzel vorangestellt werden, um Namenskonflikte zu vermeiden.

```
extern int foo_globalVariable;
```

- Durch das extern-Statement wird kein Speicherplatz reserviert.
- Funktionsprototypen und Definitionen, die von anderen Modulen genutzt werden können (Schnittstellen), werden in einer Headerdatei definiert
- Durch `#include` der Headerdatei wird der Header geladen und die Namen somit bekannt gemacht.

Preprocessor-Direktiven und Bedingungsanweisungen

Generell

- Preprocessoranweisungen sollten zurückhaltend eingesetzt werden, da der Code durch zu viele Preprocessoranweisungen sehr schnell unübersichtlich werden kann.
- Auf den ersten Blick sind Preprocessoranweisungen oft nicht nachvollziehbar.
- Häufig gibt es Alternativen, die ebenso effizient und zudem viel sicherer sind.
- Richtig eingesetzt kann der Preprocessor sehr nützlich sein.

Direktiven	Bedingungsanweisungen
#define	#if
#undef	#ifdef
#include	#ifndef
#line	#elif
#error	#else
#pragma	#endif

#define

- Definition von Makros (Gefahren beachten!)

```
#define MAX(a,b) ((a)>(b)?(a):(b))
```

- Definition von symbolischen Konstanten (Gefahren beachten!)

```
#define PI 3.14159265358
```

- Es erfolgt eine reine Textersetzung ohne Typenprüfung. Diese kann erst der Compiler vornehmen
- `#defines` haben keinen Scope
- Simple Definition eines Symbols (z.B. bei Include-Guard)

```
#define FOO_H_
```

#undef

- Definition eines Symbols rückgängig machen

```
#undef SYMBOLNAME
```

#include

- Vollständiges Einfügen einer Datei

```
/* Datei stdio.h wird in den definierten Include-Verzeichnissen gesucht und eingefügt */
#include <stdio.h>
/* Datei foo.h wird im aktuellen Verzeichnis gesucht und eingefügt */
#include "foo.h"
/* Pfadangaben müssen immer relativ zum aktuellen Verzeichnis sein. Niemals absolute Pfade verwenden! */
#include "../drivers/doo.h"
```

#line

- Direktes Setzen der Nummerierung von Sourcecode-Zeilen. Durch die optionale Angabe eines Dateinamens lässt sich der Compiler einen neuen Dateinamen unterschieben. Das kann z.B. bei vom Compiler erstellten Dateien nützlich sein.

```
#line 67 "main.c"
// die nächste Zeile im Sourcecode erhält die Nummer 67
// die Sourcedatei erhält den Namen "main.c"
```

#error

- Sofortiger Abbruch des Compile-Vorgangs und Ausgabe einer Fehlermeldung

```
#ifndef MODEL
#error MODEL ist nicht definiert
#endif
```

#pragma

- #pragma – Direktiven erlauben die Verwendung von implementierungsspezifischen proprietären Direktiven. Entwicklungsumgebungen können somit ihre eigenen Anweisungen definieren. **Diese Direktive ist damit per Definition nicht portabel.**
- Konflikte entstehen keine, da ein Compiler unbekannte #pragma-Direktiven ignoriert
- Die Portabilität ist damit aber nicht gewährleistet. Ein anderer Compiler versteht die Direktive u.U. nichtund erzeugt deshalb nicht den gewünschten Code.

Aufbau von Bedingungsanweisungen

- Bedingungsanweisungen sind nach folgendem Schema aufgebaut

```
[Bedingungsprüfung]
Direktiven
beliebig viele #elif-Gruppen mit Direktiven
optional ein #else mit Direktiven
#endif
```

- Mögliche Bedingungsprüfungen sind

`#if` gibt true, falls Bedingung true ist

`#ifdef SYMBOL` gibt true, falls SYMBOL definiert ist

`#ifndef SYMBOL` gibt true, falls SYMBOL nicht definiert ist

Beispiel

```
#if INT_MAX > 32767
      int i;
#else
      long i;
#endif
```

Beispiel

```
#ifndef CALCULATOR_H_
#define CALCULATOR_H_
// ...
#endif
```

Beispiel

```
#if 0
// Auskommentieren des gesamten hier stehenden Codes.
// Ist sehr nützlich während der Entwicklung.
#endif
```

Input und Output

Datei (File)

Eine Datei ist eine Zusammenstellung von logisch zusammengehörenden Daten, die als eine Einheit behandelt werden und unter einem dem Betriebssystem bekannten Dateinamen (file name) abgespeichert werden. Dateien werden normalerweise auf einer Harddisk gespeichert. Auch Geräte oder Schnittstellen (z.B. serielle Schnittstelle) können als Dateien betrachtet werden.

Dateizugriffsfunktionen

Dateizugriffsfunktionen abstrahieren den Zugriff auf die einzelnen Dateisysteme

Vorteil: auf der Ebene der Programmiersprache muss man sich nicht um Details der Dateisysteme kümmern.

Streamkonzept von UNIX

- Eine Datei unter UNIX ist ein einziger mit einem Namen versehener serieller Datensatz beliebiger Länge, der aus Bytes besteht. Ein solcher Datensatz aus beliebig vielen Bytes wird als Datenstrom (stream) bezeichnet.
- Die Interpretation dieses Bytestroms (int, double, eigene struct, …) muss durch den Anwendungsprogrammierer erfolgen.
- Programme, die über Datenströme kommunizieren, haben sich über das Format zu verständigen.

I/O-Konzept von C

- Die Sprache C selbst bietet keine Sprachmittel für den Zugriff auf Dateien
- Funktionen für den Zugriff auf Dateien sind in der Bibliothek `<stdio.h>` enthalten
- Eine Datei muss über ihren Namen geöffnet werden, anschliessend erfolgt der Zugriff über einen eindeutigen Filepointer. Dadurch muss nicht bei jeder Funktion der ganze Dateiname mitgegeben werden.

Dateizugriff über Filepointer

- Der Programmierer muss in seinem Programm beim Öffnen einer Datei festlegen, welche Dateivariable (Filepointer) er mit welcher Datei des Dateisystems verknüpfen möchte.

```
FILE* fp = fopen("test.dat", "w");
```

- fp stellt die Verbindung (Kanal) vom Programm zur Datei dar

Standardeingabe- und Ausgabe

- Es gibt drei vordefinierte Standardkanäle (Filepointer)
  - Standardeingabe `stdin`, meist die Tastatur
  - Standardausgabe `stdout`, meist die Konsole
  - Standardfehlerausgabe `stderr`, oft die Konsole

Umlenkung von Standardkanälen über die Konsole

- Die Umlenkung der Standardausgabe erfolgt in der Konsole mit dem Operator >

```
./myprog > test.out
```

- Die Umlenkung der Standardeingabe erfolgt in der Konsole mit dem Operator <

```
./myprog < test.in
```

- Die Umlenkungen können in einem Befehl kombiniert werden

```
./myprog < test.in > test.out
```

I/O-Funktionen in C

- High-Level-Funktionen
  - standardisiert
  - portabel
  - z.B. `printf(), scanf(), fprintf(), fscanf() , fopen()`
  - bevorzugt diese verwenden, wir behandeln nur diese
- Low-Level-Funktionen
  - nicht standardisiert
  - nicht portabel
  - z.B. `open(), creat(), write(), read(), lseek()`
  - "schnell"
  - Spezialitäten des jeweiligen Betriebssystems können genutzt werden
  - wenn möglich nicht verwenden

printf()

```
#include <stdio.h>
int printf ( const char * format, ... );
```

- format

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

specifier	Output	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: `%e` or `%f`	392.65
G	Use the shortest representation: `%E` or `%F`	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.	
%	A `%` followed by another `%` character will write a single `%` to the stream.	%

flags	description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or −) even for positive numbers. By default, only negative numbers are preceded with a − sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero.

	Used with `a`, `A`, `e`, `E`, `f`, `F`, `g` or `G` it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
`0`	Left-pads the number with zeroes (`0`) instead of spaces when padding is specified (see width sub-specifier).

width	description
`(number)`	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
`*`	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	description
.number	For integer specifiers (`d`, `i`, `o`, `u`, `x`, `X`): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of `0` means that no character is written for the value `0`. For `a`, `A`, `e`, `E`, `f` and `F` specifiers: this is the number of digits to be printed after the decimal point (by default, this is 6). For `g` and `G` specifiers: This is the maximum number of significant digits to be printed. For `s`: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for precision, `0` is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Beispiele

```
printf ("Characters: %c %c \n", 'a', 65);
printf ("Decimals: %d %ld\n", 1977, 650000L);
printf ("Preceding with blanks: %10d \n", 1977);
printf ("Preceding with zeros: %010d \n", 1977);
printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
printf ("Width trick: %*d \n", 5, 10);
printf ("%s \n", "A string");
```

Output:

```
Characters: a A
Decimals: 1977 650000
Preceding with blanks:       1977
Preceding with zeros: 0000001977
floats: 3.14 +3e+000 3.141600E+000
Width trick:    10
A string
```

putchar()

```
#include <stdio.h>
int putchar(int c);
```

- `putchar(c)` entspricht dem Befehl `putc(c, stdout)`
- `putchar()` und `putc()` sind üblicherweise als Makro implementiert. Nebeneffekte in den Argumenten sind deshalb zu unterlassen

puts()

```
#include <stdio.h>
int puts(const char* s);
```

- `puts()` schreibt den übergebenen String `s` auf `stdout`

scanf()

```
#include <stdio.h>
int scanf(const char* format, ...);
```

- Sobald der Zeichenstrom in stdin und der Formatstring nicht mehr zusammenpassen, bricht `scanf()` ab
- Die Standardeingabe ist zeilengepuffert. Sobald ein `<RETURN>` eingegeben wird, wird der Tastaturpuffer geleert und als Dateipuffer von stdin an `scanf()` gegeben.

getchar()

```
#include <stdio.h>
int getchar(void);
```

- `getchar()` entspricht dem Befehl `getc(stdin)`
- `getchar()` und `getc()` sind üblicherweise als Makro implementiert. Nebeneffekte in den Argumenten (bei `getc()`) sind deshalb zu unterlassen

gets()

```
#include <stdio.h>
char* gets(char* s);
```

- Mit `gets()` können im Gegensatz zu `scanf()` auch Leerzeichen eingelesen werden.
- `gets()` prüft nicht, ob der Buffer, auf den s zeigt, gross genug ist. Wenn der Buffer zu klein ist, wird der folgende Speicher überschrieben.
- `gets()` wurde wegen diesem Verhalten oft von Virenprogrammierern eingesetzt und ist ein Sicherheitsrisiko. Statt `gets()` sollte deshalb `fgets()` verwendet werden.

fgets()

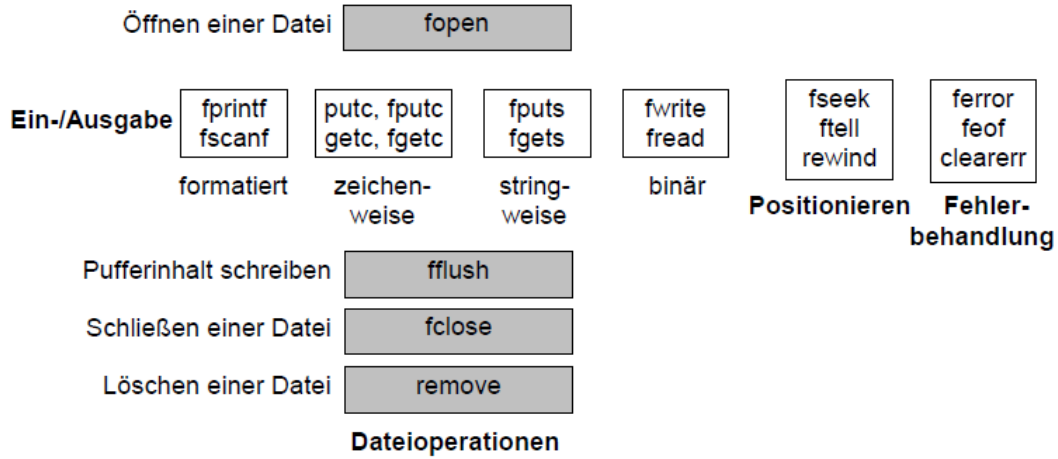
```
#include <stdio.h>
char* fgets(char* s, int n, FILE* fp);
```

- `fgets()` liest höchstens `(n-1)` Buchstaben von `fp` ein. `fp` muss auf `stdin` gesetzt werden, wenn von der Tastatur eingelesen werden soll.
- Beispiel:

```
fgets(buf, sizeof(buf) - 1, stdin);
```

- Nachteil: mit `fgets()` wird immer auch noch ein `'\n'` in den Buffer kopiert

High-Level-Dateizugriffsfunktionen



Öffnen einer Datei

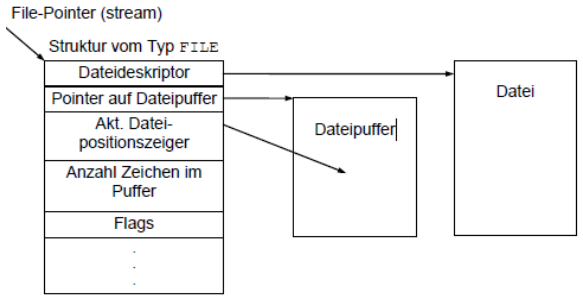
```
#include <stdio.h>
FILE* fopen(const char* filename, const char* mode);
```

- Bevor auf eine Datei zugegriffen werden kann, muss sie mit `fopen()` geöffnet werden

- **`filename`**: Dateiname, evtl. inklusive Pfad
- **`mode`**: Festlegung der Zugriffsrechte auf die Datei

r	read
w	write
a	Öffnen zum Anfügen (append)
r+	Zum schreiben und lesen öffnen, Datei muss existieren
w+	Zum schreiben und lesen erzeugen
a+	Zum anfügen und lesen öffnen
b	binäre Daten

- Rückgabe: Bei fehlerfreier Ausführung liefert `fopen()` einen Filepointer, sonst den Wert `NULL`



Beispiel: fopen() und fclose()

```
#include <stdio.h>
FILE* fp;
fp = fopen("test.dat", "rw");
     /* Datei öffnen mit Lese- und Schreibzugriff;

     hier können nun über fp Operationen auf der Datei
     ausgeführt werden */
fclose(fp);
```

Weitere Dateifunktionen

```
#include <stdio.h>
int remove(const char* filename); // Datei löschen
int fflush(FILE* stream); // noch im Ausgabepuffer liegende
Daten auf Harddisk schreiben
int fprintf(FILE* stream, const char* format, ...); // printf()
auf stream
int fscanf(FILE* stream, const char* format, ...); // scanf()
von stream
int fputc(int c, FILE* stream); // putc() auf stream
int fgetc(FILE* stream); // getc() von stream
int ungetc(int c, FILE* stream); // Zeichen c für erneute
Leseoperation auf stream zurückstellen
int fputs(const char* s, FILE* stream); // puts() auf stream
char* fgets(char* s, int n, FILE* stream); // gets() von stream
```

Weitere Funktionen

• `fseek()`:	Positionieren in Dateien
• `ftell()`:	Liefert die Position in der Datei
• `rewind()`:	Positioniert an Dateianfang
• `feof()`:	Überprüft das Dateiende-Flag
• `ferror()`:	Überprüft das Fehler-Flag
• `clearerr()`:	Setzt Dateiende- und Fehler-Flag zurück

Kommandozeilenparameter

Das Hauptprogramm `main()` hat zwei Varianten:

```
int main(void);
```

und:

```
int main(int argc, char* argv[]);
// argc: argument counter
// argv: argument vector
```

- Wenn Parameter übergeben werden sollen, muss die zweite Variante genommen werden.
- In `argc` sind die Anzahl der übergebenen Parameter gespeichert (automatisch durch den Aufruf).
- Im Array `argv` sind die einzelnen Parameter als String gespeichert, wobei in `argv[0]` immer der Name des Programms gespeichert ist.

Kompilierungsvorgang

Generell

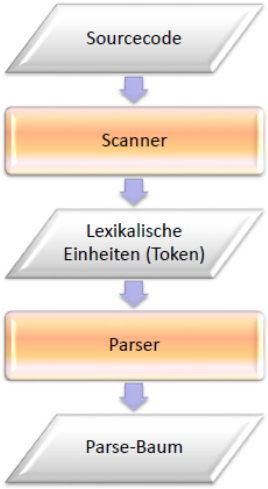
- Quelldateien (.c) werden getrennt übersetzt
- Der Compiler erzeugt aus jeder Quelldatei eine Objektdatei (.o) mit Maschinencode
- Eine Objektdatei ist nicht ausführbar, da die Adressen der Funktionen (und globalen Variablen), die in einer anderen .c - Datei definiert sind, noch nicht bekannt sind.
- Der Linker löst diese noch offenen Referenzen auf, indem er alle zum ausführbaren Programm gehörenden Objektdateien linked (deutsch: bindet).
- Der Linker meldet genau dann einen Fehler, wenn er eine noch offene Referenz nirgends gefunden hat, bzw. nicht auflösen konnte

Preprocessor

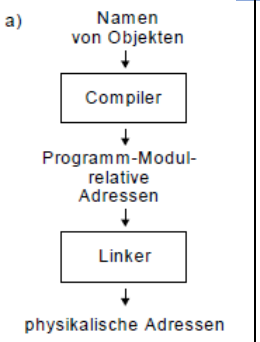
- Der Preprocessor bearbeitet den Sourcecode vor der eigentlichen Übersetzung
- Die Direktiven des Preprocessors beginnen immer mit # (z.B. `#include`)
- Führt textuelle Manipulationen von Source-Dateien durch
  - Makro-Substitution
  - Bedingte Übersetzung
  - Einfügen von Dateien
- Der Output des Preprocessors wird dem Compiler übergeben
- Der Preprocessor ist zeilenorientiert

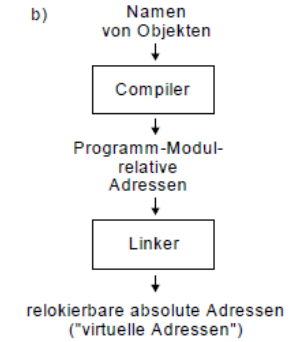
Erste Phase des Compilers

- Scanner
  - Zeichengruppen finden.
  - Alle zusammengehörenden Zeichengruppen, welche nicht durch einen Trenner (z.B. Operator oder Trennzeichen) unterbrochen werden, gelten als lexikalische Einheiten (Token)
- Parser
  - lexikalische Einheiten werden auf korrekte Syntax überprüft



Feste vs. relokative Adressen





- Der Linker kann den Code auf feste physikalische (fixe) Adressen legen,
- <u>oder</u> auf virtuelle Adressen, welche erst beim Programmstart auf eine physikalische Adresse gelegt werden

clang

Kompilierungsschritte

C-Dateien kompilieren ohne linken

```
clang -c [FILENAMES, e.g. "hello.c"]
```

Object Code (O-Dateien) linken (bzw. Ausführbare Datei erstellen)

```
clang -o hello [FILENAMES, e.g. "hello.o"]
```

C-Dateien direkt kompilieren UND linken

```
clang -o hello hello.c
```

Um bei grösseren Projekten diese Befehle nicht immer mühsam einzeln eintippen zu müssen, werden **make-Files** verwendet.

Optionen

`-Wall`
- Alle Warnungen aktivieren

`-lm`
- Mit Math-Library linken

Good Practice

Anforderungen in grösseren Projekten

- Der gesamte Code soll in mehrere `.c` - Dateien aufgeteilt werden können
- Der Gültigkeitsbereich von Namen (Konstanten, Variablen, Typen, Funktionen) soll möglichst klein sein, die Namen sollen nicht im ganzen Projekt abgestimmt werden müssen.
- Die Schnittstellen zwischen den einzelnen Teilen (Dateien) sollen möglichst klein sein -> stark entkoppeln

Information hiding

- Grundsatz: nur das gegen aussen preisgeben, das auch notwendig ist
- Interne Implementationsdetails nicht veröffentlichen
- Funktionen, die in keiner weiteren Unit verwendet werden, sollen nur lokal bekannt sein
- Schnittstelle möglichst klein halten

Zusatz

Beispiel: Zeiger auf literale Konstante als Rückgabewert einer Funktion (C -> ASM)

```c
char* getStr() {
    return "abc";
}

int main(void) {
    printf("str: %s", getStr());
    return 0;
}
```

```asm
.LC0:
    .string "abc"
getStr:
    push    rbp
    mov     rbp, rsp
    mov     eax, OFFSET FLAT:.LC0
    pop     rbp
    ret
.LC1:
    .string "str: %s"
main:
    push    rbp
    mov     rbp, rsp
    mov     eax, 0
    call    getStr
    mov     rsi, rax
    mov     edi, OFFSET FLAT:.LC1
    mov     eax, 0
    call    printf
    mov     eax, 0
    pop     rbp
    ret
```

1. Der String-Literal "abc" ist statisch gespeichert
2. Aktuelle Adresse auf Stapel ablegen und nach getStr springen
3. Aktueller Rahmenzeiger (frame pointer) wird auf dem Stapel abgelegt und Rahmenzeiger zeigt nun an die Spitze des Stapels
4. Die Adresse vom String-Literal "abc" wird in *eax* geladen (Rückgabewert der Funktion)
5. Rahmenzeiger wieder zurücksetzen und Ausführung beim Aufrufer (main) fortsetzen

`getStr` gibt also einen Zeiger auf den Typ *char* zurück, welcher auf den statischen String "abc" zeigt

Beispiel: Zeiger auf lokale Variable als Rückgabewert einer Funktion (C -> ASM)

```c
char* getStr() {
    char str[] = {'a','b', 0};
    return str;
}

int main(void) {
    printf("cStr: %s", getStr());
    return 0;
}
```

```asm
getStr:
    push    rbp
    mov     rbp, rsp
    mov     WORD PTR [rbp-3], 25185
    mov     BYTE PTR [rbp-1], 0
    mov     eax, 0
    pop     rbp
    ret
.LC0:
    .string "str: %s"
main:
    push    rbp
    mov     rbp, rsp
    mov     eax, 0
    call    getStr
    mov     rsi, rax
    mov     edi, OFFSET FLAT:.LC0
    mov     eax, 0
    call    printf
    mov     eax, 0
    pop     rbp
    ret
```

1. Der String wird lokal auf dem Stapel erstellt

   bin(25185) = 0110001001100001

   97, 98 = 'a', 'b' (ASCII-Kodierung)
2. Es kann kein Zeiger auf eine lokale Variable an den Aufrufer zurückgegeben werden! Deshalb wird ***0*** zurückgegeben. Das Programm funktioniert nicht (in der Ausgabe steht Unsinn oder das Programm stürzt ab).

Beispiel: Zeiger auf lokale Variable als Argument einer Funktion (C -> ASM)

```c
void printStr(char* cpStr) {
  printf("str: %s", cpStr);
}

int main(void) {
  char str[] = "abc";
  printStr(str);
  return 0;
}
```

Merke: **Code ist optimiert** (*GCC -O1*)

```asm
.LC0:
    .string "str: %s"
printStr:
    sub     rsp, 8
    mov     rsi, rdi
    mov     edi, OFFSET FLAT:.LC0
    mov     eax, 0
    call    printf
    add     rsp, 8
    ret
main:
    sub     rsp, 24
    mov     DWORD PTR [rsp+12], 6513249
    lea     rdi, [rsp+12]
    call    printStr
    mov     eax, 0
    add     rsp, 24
    ret
```

1. `"abc"` wird auf dem Stapel erstellt (für Details siehe hier)
2. `str[0]` (Zeiger zum ersten Array-Element) wird ins Register rdi kopiert (erstes Argument gemäss Standard-Aufrufkonvention) und `printStr` aufgerufen
3. Jetzt werden die Argumente für `printf` vorbereitet. Dafür wird `cpStr` (*rdi*) nach *rsi* (Register für zweites Funktionsargument) kopiert. Dann wird das erste Argument, der String-Literal, nach *rdi* kopiert. Nun kann `printf` aufgerufen werden.
4. Hier wird der Rückgabewert von `printStr` gesetzt. Weil die Funktion vom Typ *void* ist, hat sie natürlich keinen Rückgabewert. Um das zu garantieren, wird *eax* auf **0** gesetzt.