

Systemprogrammierung

Teil 4: ANSI-C Programme

Funktionen / Übersetzungseinheiten / Bibliotheken

ANSI-C Programme: Aufbau

Ein C-Programm ist **technisch** eine Sammlung von **Funktionen** und globalen Daten.

- genau eine der Funktionen muss **main** heissen

*Bei **main** beginnt die Ausführung des Programms.*

Ein C-Programm ist **organisatorisch** eine Sammlung von **Übersetzungseinheiten**.

- eine Übersetzungseinheit enthält Definitionen einiger logisch zusammengehöriger Funktionen und Daten
- Übersetzungseinheiten dienen dazu, Programme überschaubar zu gliedern
- logisch zusammengehörige Übersetzungseinheiten werden wiederum in **Bibliotheken** zusammengefasst.

größere Programme enthalten leicht hunderte Übersetzungseinheiten und tausende Funktionen, von denen viele nicht selbst implementiert sind, sondern aus (gekauften) Bibliotheken stammen.

ANSI-C Funktionen: Eigenschaften

Funktionen (Unterprogramme, Prozeduren)

fassen Folgen von Anweisungen zusammen, die immer wieder gebraucht werden.

- eine Funktion hat einen **Namen**:
*Namenskonvention wie bei Variablen,
Name muss im Gültigkeitsbereich eindeutig sein (kein Überladen)*
- eine Funktion kann **Parameter** und einen **Rückgabewert** haben:
dienen der Übergabe von zu verarbeitenden Werten und zu liefernden Ergebnissen
- eine Funktion hat einen **Typ**:
legt Anzahl und Typen der Parameter sowie den Typ des Rückgabe-Werts fest
- eine Funktion hat einen **Rumpf**:
enthält Variablen-Definitionen und Anweisungen.
- eine Funktion hat eine **Adresse**:
die Anfangsadresse ihres ausführbaren Codes im Hauptspeicher

ANSI-C Funktionen: Syntax (1)

- Funktions-Prototyp (Funktions-Deklaration):

```
Typ Name( Parameterliste );
```

```
void Name( Parameterliste ); /* ohne Rückgabewert */
```

```
Typ Name(void); /* ohne Parameter */
```

Erst nach ihrer Deklaration ist eine Funktion benutzbar

(der Compiler braucht den Prototyp, um Funktionsaufrufe auf Korrektheit prüfen zu können).

Die Parameterliste besteht aus durch Komma getrennten Typnamen.

Zusätzlich können (und sollten) Parameternamen angegeben werden.

- Beispiel:

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

*Parameter-Namen
(ruhig weglassen, wenn die Namen wie hier
nichts über die Bedeutung der Parameter aussagen)*

Rückgabebetyp

Funktions-Name

Parameter-Typen

ANSI-C Funktionen: Syntax (2)

- Funktions-Definition:

```
Rückgabetyp Name( Parameterliste) } Kopf
{                                     }
    Anweisungen                      } Rumpf
}
```

Der Kopf muss genau dem Prototyp entsprechen, aber Parameternamen sind Pflicht.

Der Rumpf enthält mindestens eine **return**-Anweisung:

```
return Rückgabewert; /* Typ des Werts muss zum Rückgabetyp passen */
return; /* bei void-Funktionen (darf am Ende des Rumpfs auch fehlen) */
```

- Beispiel:

```
int max(int a, int b)
{
    if (a > b) return a;
    return b;
}
```

Funktion mit Wert a verlassen (pointing to `return a;`)

Funktion mit Wert b verlassen (pointing to `return b;`)

ANSI-C Funktionen: main

Jedes Programm muss genau eine Funktion mit dem Namen **main** enthalten.

- es gibt eine Variante mit und eine ohne Parameter:

<pre>int main(int argc, char *argv[]) { ... }</pre>		<pre>int main(void) { ... }</pre>
---	--	---------------------------------------

Erklärung der Parameter:

<code>argv</code>	Feld von String-Pointern (argument vector).
<code>argc + 1</code>	Feldgröße (argument count).
<code>argv[0]</code>	Programm-Name (Kommando)
<code>argv[1]</code>	erstes Kommandozeilen-Argument
<code>argv[argc - 1]</code>	letztes Kommandozeilen-Argument
<code>argv[argc]</code>	0 (NULL-Pointer)

Beispiel-Programm main

- Quellcode:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    for (i = 0; i <= argc; ++i)
    {
        printf("%d: %s\n", i, argv[i]);
    }

    return 0;
}
```

- Aufruf:

```
main arg1 arg2
```

- Ausgabe:

```
0: main
1: arg1
2: arg2
3: (null)
```

ANSI-C Funktionen: Adresse und Zeiger

- Die Adresse einer Funktion erhält man durch Angabe ihres Namens:

Name

Anfangsadresse im Code-Segment des Hauptspeichers, wo die Maschinenbefehle liegen, die der Compiler aus dem Rumpf der Funktion erzeugt hat

- es gibt Variablen vom Typ Funktions-Zeiger, die als Wert die Adresse einer Funktion haben:

Typ (*Zeigername) (Parameterliste) = Name;

Rückgabetyt und Parameterliste des Zeigertyps und der zugewiesenen Funktion müssen übereinstimmen.

- Beispiel:

```
int (*maximum)(int , int ) = max;
```

Zeigername →

← *Name der Funktion*

ANSI-C Funktionen: Aufruf

- der Aufruf-Operator **()** veranlasst die Ausführung einer Funktion und liefert den Rückgabe-Wert der Funktion:

Name(Argumentliste)

Zeigernname(Argumentliste)

Die Argumentliste besteht aus durch Komma getrennten Ausdrücken.

Die Parameter der Funktion werden mit den Werten der Argumentliste initialisiert.

- Beispiel:

Aufruf über Funktions-Name

```
int z = max(7, 8); /* setzt z auf 8 */
```

Aufruf über Funktions-Zeiger

```
int (*maximum)(int , int) = max;  
int z = maximum(7, 8); /* setzt z auf 8 */
```

ANSI-C Funktionen: globale und lokale Variablen

- **Globale Variablen**

Definition außerhalb der Funktionsrumpfe (*dann in vielen Funktionen benutzbar*)
oder **static**-Definition innerhalb eines Funktionsrumpfs (*dann funktions-privat*)

Lebensdauer: Programmlauf
Speicherort: Daten-Segment

- **Lokale Variablen** (*automatische Variablen*)

Definition innerhalb eines Funktionsrumpfs am Anfang eines Anweisungsblock {},
dadurch nur innerhalb dieses Anweisungsblocks benutzbar

Lebensdauer: Ausführung des Anweisungsblocks
Speicherort: Stack-Segment

- **Parameter**

spezielle lokale Variablen, Definition im Funktionskopf,
Initialisierung mit den Argumenten des Funktionsaufrufs

Lebensdauer: Ausführung der Funktion
Speicherort: Stack-Segment

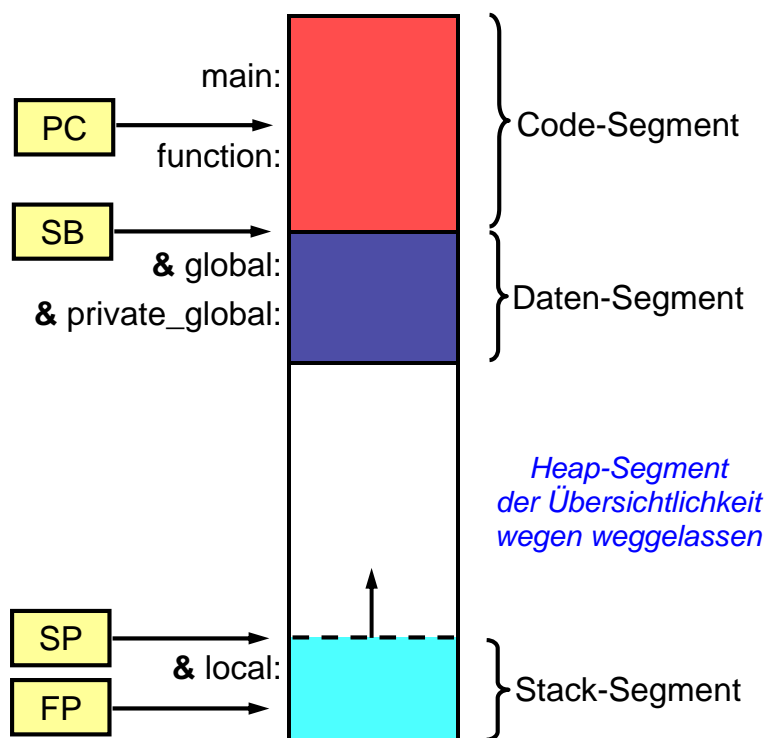
Beispiel-Programm globale und lokale Variablen

- Quellcode:

```
int function(int param); } Funktions-Prototyp
int global = 1;
int main(void)
{
    int local = 1;
    local = function(local); /* local wird 4 */
    local = function(global); /* local wird 7 */
    return 0;
}
int function(int param)
{
    static int private_global = 1;
    int local = param + 1;
    private_global++;
    global = param + 2;
    return local + private_global;
}
```

Funktions-Definition (Implementierung)

ANSI-C Funktionen: Hauptspeicherbelegung (1)



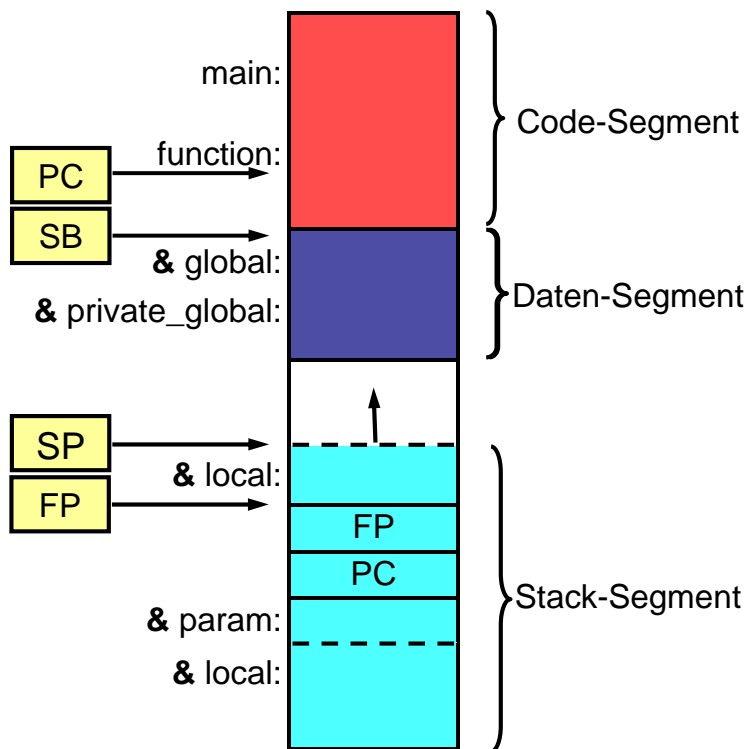
Der Prozessor merkt sich

- die Adresse des nächsten Befehls im Program Counter **PC**
- das aktuelle Ende des Stack-Segments im Stack Pointer **SP**

Der Prozessor adressiert

- globale Daten relativ zur Static Base **SB**
- lokale Daten relativ zum Frame Pointer **FP**

ANSI-C Funktionen: Hauptspeicherbelegung (2)



Ein Funktions-Aufruf benutzt den Stack:

- zum Speichern von Argumenten (hier: *param*)
- zum Speichern der Rückkehrinformation (alter *PC* und *FP*)
der Rückgabewert wird in einem Prozessorregister oder auf dem Stack geliefert (hier nicht gezeigt)
- zum Reservieren von Platz für die lokalen Variablen (hier: *local*)

ANSI-C Funktionen: Eingabe-Parameter

Mit **Eingabe-Parametern** übergibt ein Aufrufer Werte an eine Funktion:

- bei Grundtyp-Parametern Übergabe per Wertkopie

```
void funktion(int zahl);
```

- bei Zeiger- und Feld-Parametern Übergabe per Adresskopie

```
void funktion(const int *zeiger);
```

```
void funktion(int feldlaenge, const int feld[]);
```

- bei Strukturtyp-Parametern Übergabe per Adresskopie

Wertkopie ist bei großen Strukturen zu ineffizient

```
struct struktur { ... };
```

```
void funktion(const struct struktur *s);
```

const verwenden, damit die Funktion nur Lesezugriff auf den Speicher des Aufrufers hat

ANSI-C Funktionen: Ausgabe-Parameter

Mit **Ausgabe-Parametern** übergibt eine Funktion Werte an ihren Aufrufer:

- **Adresskopie** verwenden

```
void funktion(int *zeiger);
```

```
void funktion(int feldlaenge, int feld[]);
```

```
void funktion(struct struktur *s);
```

*kein **const**,
damit die Funktion
Schreibzugriff
auf den Speicher
des Aufrufers hat*

- Funktionen mit Ausgabe-Parametern bezeichnet man auch als **Funktionen mit Seiteneffekten**.

*Nach der reinen Lehre sollten Funktionen als Ergebnis nur einen Rückgabe-Wert liefern (**return**-Anweisung).*

ANSI-C Funktionen: **Vergleich mit Java**

Bei den Funktionen gibt es deutliche Unterschiede zwischen ANSI-C und Java:

- die ANSI-C Funktionen entsprechen im Prinzip den Java Klassenmethoden und ANSI-C globale Variablen entsprechen den Java Klassenvariablen

weil es keine umschließenden Klassen gibt, liegen anders als bei Java die Namen aller Funktionen und globalen Variablen in einem gemeinsamen Namensraum

- ANSI-C Funktionen können nicht überladen werden

der Funktionsname muss ohne Betrachtung der Parametertypen eindeutig sein

- Ausgabeparameter gibt es in Java nicht

es gibt lediglich Seiteneffekte auf per Eingabeparameter übergebene Objekte

ANSI-C Übersetzungseinheiten: Aufbau

Ein C-Programm wird in Module gegliedert, die sich getrennt übersetzen lassen (Übersetzungseinheiten).

- jede Übersetzungseinheit besteht aus einer **.h**-Datei und einer **.c**-Datei:

Name.h

Header-Datei

Name.c

Implementierungs-Datei

*beim Hauptprogramm (Übersetzungseinheit mit nur der Funktion **main**)
entfällt die Header-Datei*

- die Header-Dateien werden mit Präprozessor-Anweisungen in Implementierungs-Dateien oder andere Header-Dateien hineinkopiert, immer wenn Deklarationen daraus verwendet werden:

```
#include "Name.h"
```

- nur die Implementierungs-Dateien werden mit dem Compiler übersetzt

ANSI-C Übersetzungseinheiten: Header-Dateien

Eine Header-Datei enthält die Schnittstelle der Übersetzungseinheit, d.h. Deklarationen von Namen, die in anderen Übersetzungseinheiten sichtbar sein sollen

- benutzerdefinierte Typen:**

```
struct date
{
    int day, month, year;
};
typedef struct date date;
```

*wenn die Strukturkomponenten
privat sein sollen:*

***typedef struct date date;**
dann in anderen Übersetzungseinheiten
nur date-Zeiger möglich*

- symbolische Konstanten** und **Makros:**

```
#define MAXDAY 31
```

- globale Variablen** (**extern**-Deklaration ohne Initialisierung):

```
extern date epoch; /* wegen extern keine Speicherreservierung */
```

- Funktions-Prototypen:**

```
void print_date(const date *d);
```

ANSI-C Übersetzungseinheiten: Implementierungs-Dateien

Eine **Implementierungs-Datei** enthält die Implementierung der Schnittstelle.

- **Kopie der eigenen Schnittstelle** (und ggf. anderer verwendeter Schnittstellen):

```
#include "date.h"
```

- **Definition der Schnittstellenvariablen** (ohne **extern** und mit Initialisierung):

```
date epoch = {1, 1, 1970}; /* Start der Unix Zeitrechnung */
```

- **Definition der Schnittstellen-Funktionen:**

```
void print_date(const date *d)
{
    printf("%d.%d.%d\n", d->day, d->month, d->year);
}
```

- **private Deklarationen und Definitionen**

Typen, symbolische Konstanten, Makros, globale Variablen, Funktionen, die nur innerhalb der Übersetzungseinheit verwendet werden

*private globale Variablen und Funktionen werden mit **static** markiert*

ANSI-C Übersetzungseinheiten: Präprozessor-Anweisungen

Der **Präprozessor** des C-Compilers nimmt vor der eigentlichen Übersetzung Textersetzen vor.

- Präprozessor-Anweisungen stehen in Zeilen, die mit **#** beginnen
- eine **#include-Anweisung** muss immer verwendet werden, wenn in einer Datei (egal, ob Header- oder Implementierungs-Datei) ein Name aus einer anderen Übersetzungseinheit verwendet wird

```
#include "Name.h"
```

ersetzt der Präprozessor rekursiv durch den Inhalt von Name.h

- **#ifndef/#define/#endif-Anweisungen** in allen Header-Dateien sorgen dafür, dass deren Inhalte auch bei verschachtelten **#include**-Anweisungen höchstens einmal in jede Implementierungs-Datei kopiert werden

```
#ifndef NAME_H
#define NAME_H
... /* Inhalt der Header-Datei Name.h */
#endif
```

*if not defined :
beim ersten #include ist
NAME_H noch nicht definiert,
ab dem zweiten #include wird
der #ifndef-Block übersprungen*

Beispiel-Programm Übersetzungseinheiten (1)

Globale Variable als getrennte Übersetzungseinheit (siehe zum Vergleich Folie 4-10)

- Header-Datei:

```
/* global.h */  
#ifndef GLOBAL_H  
#define GLOBAL_H  
extern int global;  
#endif
```

*#ifndef / #define / #endif
verhindert mehrfaches Kopieren
in dieselbe Implementierungs-Datei*

- Implementierungs-Datei:

```
/* global.c */  
#include "global.h"  
int global = 1;
```

*die Implementierungs-Datei
einer Übersetzungseinheit enthält
immer die eigene Header-Datei*

Beispiel-Programm Übersetzungseinheiten (2)

Funktion als getrennte Übersetzungseinheit (siehe zum Vergleich Folie 4-10)

- Header-Datei:

```
/* function.h */  
#ifndef FUNCTION_H  
#define FUNCTION_H  
int function(int param);  
#endif
```

*die Funktion benutzt einen Namen
aus der Übersetzungseinheit global
(deshalb #include "global.h")*

- Implementierungs-Datei:

```
/* function.c */  
#include "function.h"  
#include "global.h"  
int function(int param)  
{  
    static int private_global = 1;  
    int local = param + 1;  
    private_global++;  
    global = param + 2;  
    return local + private_global;  
}
```

Beispiel-Programm Übersetzungseinheiten (3)

Hauptprogramm als getrennte Übersetzungseinheit (siehe zum Vergleich Folie 4-10)

- Header-Datei:

entfällt

- Implementierungs-Datei:

```
/* localglobalvar.c */  
#include "function.h"  
#include "global.h"  
int main(void)  
{  
    int local = 1;  
    local = function(local); /* local wird 4 */  
    local = function(global); /* local wird 7 */  
    return 0;  
}
```

das Hauptprogramm
benutzt Namen aus den
Übersetzungseinheiten
function und global

ANSI-C Übersetzungseinheiten: Compiler und Linker-Aufrufe (1)

Compiler/Linker-Aufrufe bei Programmen mit mehreren Übersetzungseinheiten:

- jede Übersetzungseinheit getrennt übersetzen:

```
gcc -c -I. function.c
```

```
gcc -c -I. global.c
```

```
gcc -c -I. localglobalvar.c
```

die Option `-I.` gibt an, dass die Header-Dateien
im lokalen Verzeichnis zu suchen sind
(mehrere Optionen `-I` Verzeichnisname möglich)




- dann den Objektcode der Übersetzungseinheiten (Endung `.o`)
zu einem ausführbaren Programm binden:

```
gcc localglobalvar.o function.o global.o -o localglobalvar
```

das ausführbare Programm nennt man üblicherweise so
wie die Übersetzungseinheit mit dem Hauptprogramm `main`

ANSI-C Übersetzungseinheiten: Compiler und Linker-Aufrufe (2)

Die Übersetzungsschritte im Einzelnen:

<u>Übersetzungsschritt</u>	<u>Eingabe</u>		<u>Ergebnis</u>	<u>Aufruf</u>
Präprozessor	function.c function.h global.h		function.i	gcc -E
Compiler	function.i		function.s	gcc -S
Assembler	function.s		function.o	gcc -c

- üblicherweise alle Übersetzungsschritte mit einem Aufruf:

```
gcc -c function.c
```

die obigen Zwischenschritte sind aber manchmal bei der Fehlersuche hilfreich

ANSI-C Übersetzungseinheiten: Bibliotheken

Bibliotheken fassen mehrere Übersetzungseinheiten in einer Datei zusammen:

- **Statische Bibliothek** (unter Linux Präfix **lib** und Endung **.a** für Archiv)

```
ar rs libbeispiel.a function.o global.o
```

Eine statische Bibliothek ist eine Sammlung von Objekt-Dateien.
- **Dynamische Bibliothek** (unter Linux Präfix **lib** und Endung **.so** für Shared Object)

```
gcc -shared function.o global.o -o libbeispiel.so
```

Eine dynamische Bibliothek ist quasi ein gebundenes Programm ohne Hauptprogramm.

Beim Binden ersparen Bibliotheken das Aufzählen aller Übersetzungseinheiten:

```
gcc localglobalvar.o -L. -lbeispiel -o localglobalvar
```

Mit -L wird das Verzeichnis und mit -l der Name ohne Präfix der Bibliothek angegeben.

Eine statische Bibliothek wird nach dem Binden nicht mehr gebraucht.

*Eine dynamische Bibliothek muss zur Laufzeit des Programms zugreifbar sein
(dazu Verzeichnis in der Umgebungsvariablen LD_LIBRARY_PATH angeben).*

ANSI-C Programme: Deployment

Archive fassen mehrere Dateien in einer Datei zusammen.

Sie erleichtern das **Deployment** (*Distribution von Programmen auf Zielrechner*)

- Linux-Beispiel: **tar**-Kommando

```
tar czf beispiel.tar.gz localglobalvar libbeispiel.so
```

*erzeugt ein mit **gzip** komprimiertes Archiv **beispiel.tar.gz**,
das die Dateien **localglobalvar** und **libbeispiel.so** enthält*

```
tar xzf beispiel.tar.gz
```

extrahiert die im Archiv enthaltenen Dateien wieder

ANSI-C Übersetzungseinheiten: Vergleich mit Java

Bei den Übersetzungseinheiten unterscheiden sich ANSI-C und Java erheblich:

- bei Java sind Klassen zugleich Übersetzungseinheiten
*da Methoden und Variablen immer in Klassen enthalten sind, kann der Compiler
über den qualifizierten Klassennamen immer die richtige Übersetzungseinheit finden*
- bei ANSI-C ist der Name einer Übersetzungseinheit unabhängig vom Inhalt
*weil es weder Klassen noch Pakete gibt, kann der Compiler einem Namen nicht ansehen,
in welcher Übersetzungseinheit er definiert ist*
*deshalb ist eine Aufteilung in Header- und Implementierungs-Dateien notwendig,
und die Header-Dateien müssen explizit in die Implementierungs-Dateien kopiert werden*
- **static** in ANSI-C entspricht eher **private** in Java
***static** markierte Funktionen und globale Variablen
sind nur innerhalb der Implementierungs-Datei zugreifbar, in der sie definiert sind*
***static** markierte Variablen innerhalb einer Funktionen (gibt es in Java nicht)
haben die Lebensdauer einer globalen Variablen,
sind aber nur innerhalb der Funktion zugreifbar*

ANSI-C Standard-Bibliothek: Überblick

Die Schnittstelle der Standard-Bibliothek ist (teilweise etwas beliebig) in diverse Header-Dateien aufgeteilt:

- Grundlegendes zur Sprachunterstützung (*NULL*, *size_t*, *malloc*, *calloc*, *free*, ...)

`<stdarg.h>` `<stddef.h>` `<stdlib.h>`

- Ein-/Ausgabe

`<stdio.h>`

- Umgang mit Zeichen und Zeichenketten

`<ctype.h>` `<string.h>`

- Umgang mit Zahlen:

`<float.h>` `<limits.h>` `<math.h>` `<locale.h>`

- Umgang mit Datum und Zeit

`<time.h>`

- Umgang mit Fehlern und Ausnahmesituationen

`<assert.h>` `<errno.h>` `<setjmp.h>` `<signal.h>`

ANSI-C Standard-Bibliothek: `<stdlib.h>` (1)

- Speicherverwaltung:

Was tun die Funktionen?

```
void* calloc(size_t n, size_t size);   void free(void* p);  
void* malloc(size_t size);   void* realloc(void* p, size_t size);
```

- Programmende und Interaktion mit der Ablaufumgebung:

```
void abort(void);  
int atexit(void (*exit_handler)(void));  
void exit(int status); /* status: EXIT_FAILURE oder EXIT_SUCCESS */  
char* getenv(const char* name); /* Umgebungsvariable abfragen */  
int system(const char* s);   /* Kommando ausführen */
```

- Umwandlung von Zeichenketten in Zahlen, Zufallszahlen, etc.:

```
double atof(const char* s);  
int atoi(const char* s);  
int rand(void); /* Zufallszahlengenerator */  
void srand(unsigned int seed); /* Anfangswert für Zufallszahlen */  
...
```

ANSI-C Standard-Bibliothek: <stdlib.h> (2)

- Suchen und Sortieren:

```
void *bsearch(const void* key, const void* p, size_t n, size_t size,
              int (*cmp)(const void*, const void*));

void qsort(void* p, size_t n, size_t size,
           int (*cmp)(const void*, const void*));
```

- Beispiel:

```
#include <stdlib.h> /* damit bsearch und qsort bekannt sind */
int intcmp(const void *, const void *); /* Vergleichsfunktion */
...
int a[4] = {40, 20, 10, 30};
int n = 50, *p;
qsort(a, 4, sizeof(int), intcmp); /* sortiert a aufsteigend */
p = (int*) bsearch(&n, a, 4, sizeof(int), intcmp); /* sucht 50 in a */
```

Wie sieht die Implementierung von intcmp aus?

ANSI-C Standard-Bibliothek: <ctype.h>

- Prüfen der Zeichenart (Ziffer, Buchstabe, Zwischenraum usw.):

```
int isalnum(int c);      int isprint(int c);
int isalpha(int c);      int ispunct(int c);
int iscntrl(int c);      int isspace(int c);
int isdigit(int c);      int isupper(int c);
int isgraph(int c);      int isxdigit(int c);
int islower(int c);
```

- Wandeln in Klein- / Großbuchstaben:

```
int tolower(int c);
int toupper(int c);
```

- Beispiel:

```
#include <ctype.h> /* damit isdigit bekannt ist */
...
if (isdigit('9')) printf("9 ist eine Ziffer\n");
```


ANSI-C Standard-Bibliothek: <string.h>

- Kopieren, Verarbeiten, Vergleichen von Zeichenketten:

Beispiele siehe Teil 2

- Vergleichen, Kopieren, Initialisieren usw. von Speicherbereichen:

```
int memcmp(const void *cs, const void *ct, size_t n);
void *memcpy(void *p1, const void *p2, size_t n);
void *memset(void *p, int c, size_t n);
...
```

- Beispiel:

```
#include <string.h>  /* damit memset und memcpy bekannt sind */
...
int a[4];
int b[4];
memset(a, 0, sizeof a);  /* initialisiert Speicherbereich von a mit 0 */
memcpy(b, a, sizeof a);  /* kopiert Inhalt von a nach b */
```

ANSI-C Standard-Bibliothek: <float.h> <limits.h> <math.h>

- Symbolische Konstanten für Zahlenbereich und Genauigkeit von Gleitkommazahlen (<float.h>) und ganzen Zahlen (<limits.h>):

DBL_DIG	<i>Genauigkeit von double in Anzahl Dezimalstellen</i>
DBL_EPSILON	<i>kleinste double-Zahl x mit $1.0 + x \neq 1.0$</i>
DBL_MAX	<i>größte double-Zahl</i>
DBL_MIN	<i>kleinste normalisierte double-Zahl</i>
...	
INT_MAX	<i>größte int-Zahl</i>
INT_MIN	<i>kleinste int-Zahl</i>
...	

- Mathematische Funktionen (<math.h>):

```
double sqrt(double x);  Quadratwurzel
double sin(double x);   Sinus
...
```

- Beispiel:

```
#include <math.h>  /* damit sqrt bekannt ist */
double d = sqrt(9.0);  /* setzt d auf 3.0 */
```

ANSI-C Programme: Empfehlungen

- **Getrennte Übersetzungseinheiten** zur Modularisierung verwenden.
Hauptprogramm als eigene Übersetzungseinheit ohne Header-Datei
- **Variablen-Definitionen** immer so lokal wie möglich
globale Variablen vermeiden
- **ANSI-C Standard-Bibliothek** gegenüber Eigenimplementierungen bevorzugen:
z.B. qsort verwenden, statt selbst ein Sortierverfahren zu programmieren

ANSI-C Programme: Lernzettel

#endif #ifndef #include Archiv argc argv Aufrufoperator binden Compiler
dynamische Bibliothek extern frame pointer Funktion Funktionskopf
Funktionsprototyp Funktionsrumpf Funktionszeiger gcc globale Variable
Header-Datei Implementierungsdatei Linker lokale Variable main memcpy
memset Parameter Präprozessor program counter qsort return Rückgabewert
Seiteneffekt stack pointer Standardbibliothek static statische Bibliothek
übersetzen Übersetzungseinheit