

Systemprogrammierung

Teil 4: C Programme

Funktionen / Übersetzungseinheiten / Bibliotheken

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 zugelieferten Bibliotheken stammen.

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

C Funktionen: Syntax (1)

- Funktions-Prototyp (Funktions-Deklaration):

Typ Name (Parameterliste) ;

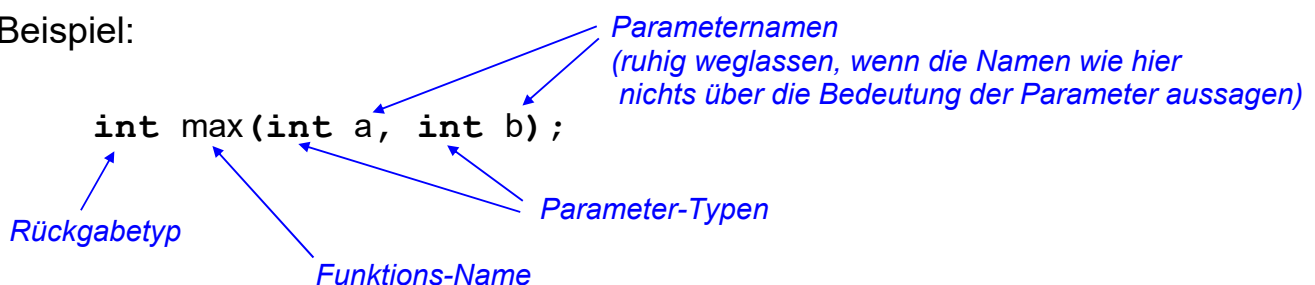
void Name (Parameterliste) ; // ohne Rückgabewert

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 Parameternamen angegeben werden).*

- Beispiel:



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;`)

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 (<u>a</u> rgument <u>v</u> ector).
<code>argc + 1</code>	Feldgröße (<u>a</u> rgument <u>c</u> ount).
<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)

Beispielprogramm main

- Quellcode:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    for (int 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)
```

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 Funktionszeiger, 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*

C Funktionen: Aufruf

- der Aufrufoperator **()** 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 Funktionszeiger

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

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

Beispielprogramm 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++;
```

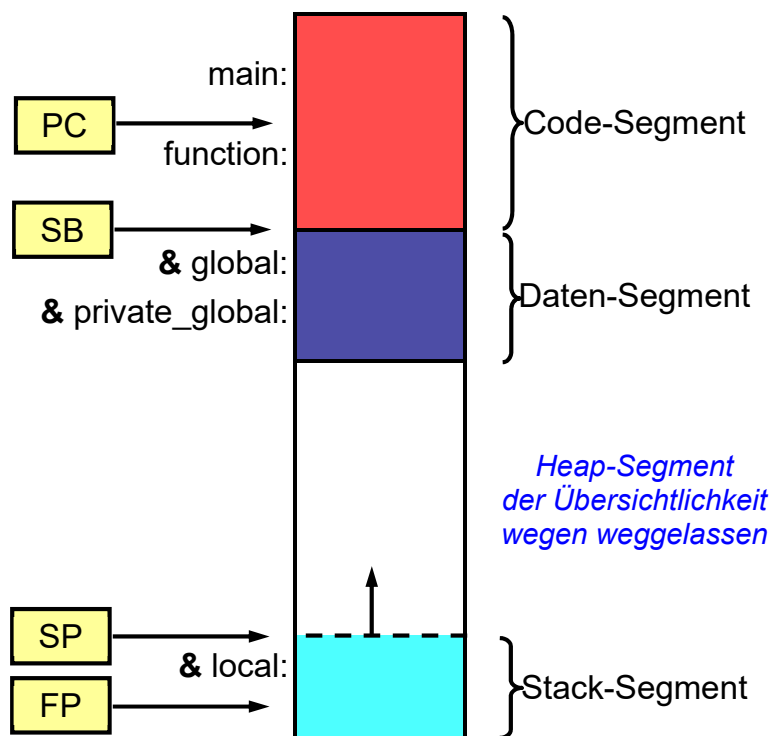
```
    global = param + 2;
```

```
    return local + private_global;
```

```
}
```

} Funktions-Definition (Implementierung)

C Funktionen: Hauptspeicherbelegung (1)



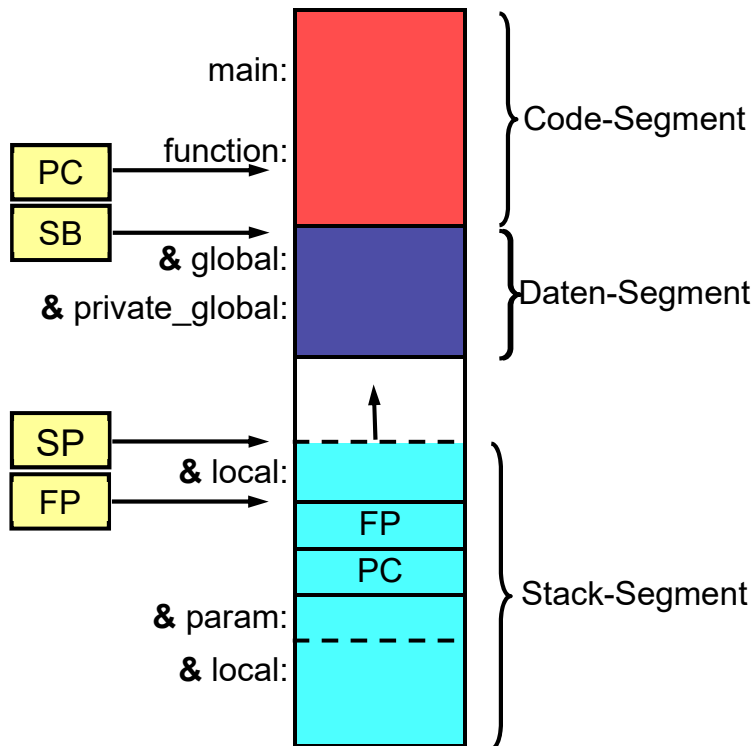
Der Prozessor merkt sich

- 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**

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)

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

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).*

C Funktionen: **Vergleich mit Java**

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

- C Funktionen entsprechen im Prinzip den Java Klassenmethoden und globale C 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

- 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

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

C Übersetzungseinheiten: Header-Dateien

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

- #include-Anweisungen**, falls innerhalb der Header-Datei Namen aus weiteren Übersetzungseinheiten verwendet werden, z.B. als Parametertypen

- benutzerdefinierte Typen:**

```
struct date { ... };  
typedef struct date date;
```

*nur die Deklaration
typedef struct date date;
wenn die Strukturkomponenten privat sein sollen
(dann in anderen Übersetzungseinheiten nur noch
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);
```

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*

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*

Beispielprogramm 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*

Beispielprogramm 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;
}
```

Beispielprogramm 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*

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)*

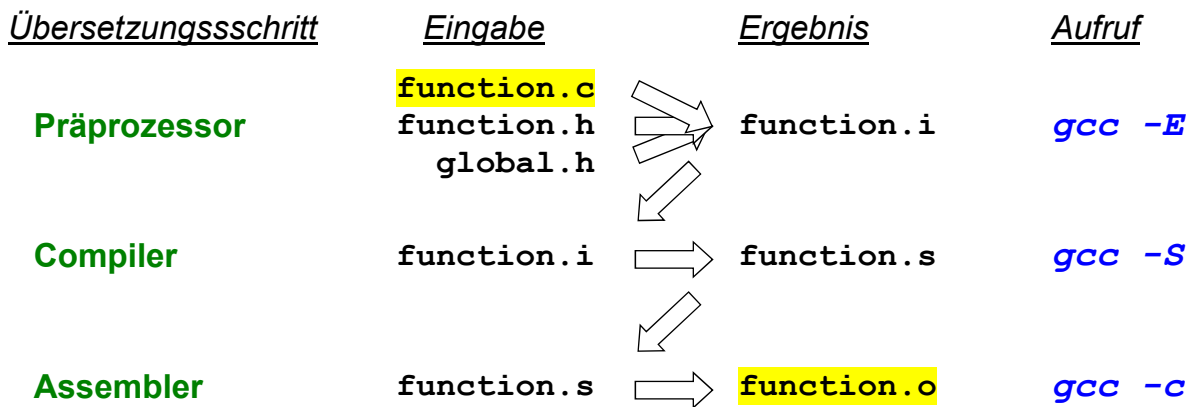
- dann den Objektcode der Übersetzungseinheiten (Endung .o)
zu einem ausführbaren Programm binden:

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

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

C Übersetzungseinheiten: Compiler und Linker-Aufrufe (2)

Die Übersetzungsschritte im Einzelnen:



- üblicherweise alle Übersetzungsschritte mit einem Aufruf:

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

die obigen Zwischenschritte sind aber manchmal bei der Fehlersuche hilfreich

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).*

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

C Übersetzungseinheiten: **Vergleich mit Java** (1)

Bei den Übersetzungseinheiten unterscheiden sich 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 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 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*

C Übersetzungseinheiten: Vergleich mit Java (2)

Beispielklasse in Java und entsprechende Übersetzungseinheit in C:

```
// IntegerMethods.java
public final class IntegerMethods {
    private IntegerMethods() { }

    public static
    int max(final int m, final int n) {
        return m > n ? m : n;
    }

    public static
    int min(final int m, final int n) {
        return m < n ? m : n;
    }
}
```

```
// integer_methods.h
#ifndef INTEGER_METHODS_H
#define INTEGER_METHODS_H
int int_max(int, int);
int int_min(int, int);
#endif
```

```
// integer_methods.c
#include "integer_methods.h"
int int_max(int m, int n) {
    return m > n ? m : n;
}
int int_min(int m, int n) {
    return m < n ? m : n;
}
```

C Standardbibliothek: Überblick

Die Schnittstelle der Standardbibliothek ist in diverse historisch gewachsene Header-Dateien aufgeteilt. Die wichtigsten davon:

- Grundlegendes zur Sprachunterstützung `<stdbool.h>` `<stdint.h>` `<inttypes.h>`
`<stdarg.h>` `<stddef.h>` `<stdlib.h>` ...
- Ein-/Ausgabe `<stdio.h>`
- Umgang mit Zeichen und Zeichenketten `<ctype.h>` `<string.h>`
`<uchar.h>` `<wchar.h>` ...
- Umgang mit Zahlen `<float.h>` `<limits.h>` `<math.h>` ...
- Umgang mit Datum und Zeit `<time.h>`
- Umgang mit Fehlern und Ausnahmesituationen `<assert.h>` `<errno.h>`
`<setjmp.h>` `<signal.h>`

C Standardbibliothek: <stdbool.h> <stdint.h> <inttypes.h>

- symbolische Namen als Ersatz für den fehlenden booleschen Grundtyp (<stdbool.h>, seit C99):

```
bool        // als Datentyp verwendbar, wird auf __Bool abgebildet
true        // wird auf das Zahlliteral 1 abgebildet
false       // wird auf das Zahlliteral 0 abgebildet
```

- Typnamen für ganze Zahlen mit festem Platzbedarf und Wertebereich (<stdint.h>, seit C99):

```
int8_t, int16_t, int32_t, int64_t, intmax_t, uint8_t, ... uint64_t, uintmax_t, ...
```

der Compiler bildet die Typnamen plattformabhängig auf die ganzzahligen Grundtypen ab

- Makros für die formatierte Ein-/Ausgabe der Typen aus <stdint.h> (<inttypes.h>, seit C99):

```
PRId8, PRId16, PRId32, PRId64, PRIdMAX, ... // das gleiche auch mit o, x, u
SCNd8, SCNd16, SCNd32, SCNd64, SCNdMAX, ... // das gleiche auch mit o, x, u
```

Beispiel: `int32_t n = 123;`
`printf("%"PRId32"\n", n);`

C Standardbibliothek: <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
...
```


C Standardbibliothek: <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?

C Standardbibliothek: <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");
```

C Standardbibliothek: <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
```

C Standardbibliothek: <float.h> <limits.h> <math.h>

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

```
DBL_DIG           // Genauigkeit von double in Anzahl Dezimalstellen  
DBL_EPSILON       // kleinste double-Zahl x mit 1.0 + x != 1.0  
DBL_MAX           // größte double-Zahl  
DBL_MIN           // kleinste normalisierte double-Zahl  
...  
INT_MAX           // größte int-Zahl  
INT_MIN           // kleinste int-Zahl  
...
```

- Mathematische Funktionen (<math.h>):

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

- Beispiel:

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

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
- **C Standardbibliothek** gegenüber Eigenimplementierungen bevorzugen:
z.B. qsort verwenden, statt selbst ein Sortierverfahren zu programmieren

C Programme: Index

#endif 4-19 bis 4-22	Linker 4-23,4-24
#ifndef 4-19 bis 4-22	lokale Variable 4-9,4-10
#include 4-19 bis 4-22	main 4-1,4-5,4-6
Archiv 4-26	memcpy 4-34
argc 4-5	memset 4-34
argv 4-5	Parameter 4-2,4-9,4-13,4-14
Aufrufoperator 4-8	Präprozessor 4-16,4-19
binden 4-23,4-25	program counter 4-11
Compiler 4-23,4-24	qsort 4-32
dynamische Bibliothek 4-25	return 4-4
extern 4-17,4-20	Rückgabewert 4-2
frame pointer 4-11	Seiteneffekt 4-14
Funktion 4-1 bis 4-5	stack pointer 4-11
Funktionskopf 4-4	Standardbibliothek 4-29
Funktionsprototyp 4-3	static 4-18,4-27
Funktionsrumpf 4-4	statische Bibliothek 4-25
Funktionszeiger 4-7,4-8	übersetzen 4-23
gcc 4-23,4-24	Übersetzungseinheit 4-1,4-16
globale Variable 4-9,4-10,4-15,4-17	
Header-Datei 4-16,4-17	
Implementierungsdatei 4-16,4-18	