

Lehreinheiten

zu

**Programmieren II**

## Vorlesung Programmieren II

### Inhalte der Lehreinheiten

Lehreinheit	Inhalt	Folie
<a href="#">c1</a>	<ul style="list-style-type: none"> <li>■ C-Grundlagen 1</li> <li> <ul style="list-style-type: none"> <li>■ Handwerkszeug für einfache C-Programme 1</li> <li>■ Besonderheiten in C 2</li> <li>■ Ein-/Ausgabe mit Dateien 2</li> <li>■ Bibliotheken 2</li> <li>■ Einfache Typen 3</li> <li>■ Einfache Ein-/Ausgabe 3</li> <li>■ Einfache Ausdrücke und Zuweisungen 4</li> <li>■ Kontrollstrukturen 5-7</li> <li>■ Funktionen 8</li> </ul> </li> </ul>	
<a href="#">c2</a>	<ul style="list-style-type: none"> <li>■ Präprozessor 1-3</li> </ul>	
<a href="#">c3</a>	<ul style="list-style-type: none"> <li>■ Typen 1           <ul style="list-style-type: none"> <li>■ Aufzählungstypen 1</li> <li>■ Zeiger und Felder 2-4</li> </ul> </li> </ul>	
<a href="#">c4</a>	<ul style="list-style-type: none"> <li>■ Strukturen und Unions 1</li> <li>■ Bitfelder 2</li> <li>■ Komplizierte Typen 3-4</li> </ul>	
<a href="#">c5</a>	<ul style="list-style-type: none"> <li>■ Speicherklassen und Gültigkeitsbereiche 1           <ul style="list-style-type: none"> <li>■ Arten von Deklarationen 2</li> <li>■ Gültigkeitsbereiche 2</li> <li>■ Vorwärtsreferenzen 3</li> <li>■ Lebensdauer von Objekten 3</li> <li>■ Bindung von Bezeichnern 4</li> <li>■ Speicherklassen 5-6</li> </ul> </li> </ul>	
<a href="#">c6</a>	<ul style="list-style-type: none"> <li>■ Bibliotheken 1           <ul style="list-style-type: none"> <li>■ Verwendung von Bibliotheksfunktionen 2</li> <li>■ Beispiele für Bibliotheksfunktionen 3-4</li> <li>■ Erstellung eigener Bibliotheken 5-6</li> </ul> </li> </ul>	

C für Javanesen

- Ein einfaches Programm
- Handwerkszeug für einfache C-Programme
  - einfache Typen
  - einfache Ein-/Ausgabe
  - einfache Ausdrücke und Zuweisungen
  - Kontrollstrukturen
  - Funktionen

- Besonderheiten in C
  - Präprozessor
  - Typen
  - Zeiger - Feld
  - Bitfeldtypen
  - Funktionstypen
  - Deklarationen und Speicherklassen
- Ein-/Ausgabe mit Dateien
- Bibliotheken

- Einfache Typen
  - int short int long int signed/unsigned
  - char signed/unsigned
  - float double long double
  - Zeiger auf TYP: TYP \* Variablenname;
- Einfache Ein- / Ausgabe
  - printf() und scanf()
  - Formatangaben(%): d,o,x,u,c,s,f,e, ...

- Einfache Ausdrücke und Zuweisungen
  - x = a + b; a += b; b++; - b;
  - x = i / j; k = i%j;
  - x = (float) i;
  - (k == 3) (i == j) && (lauf < max) a || b

- Kontrollstrukturen
  - for (expr1; expr2; expr3) statement
  - if (expr) statement1 else statement2

*continue;  
break;  
Klammern*

**C-Regel:** Zu Kontrollstrukturen wie for, if, while gehörige Statements grundsätzlich mit Klammern versehen.

- if (expr1 statement1 else if (expr2) statement2 else if (expr3) statement3 else if (expr4) statement4 else statement5
- switch (expr)
  - {
    - case const1: statement1
    - case const2: statement2
    - ....
    - default: statement

*; nicht vergessen.  
break;  
optional*

- while (expr) statement
- do statement while (expr);
- goto error; error: statement

*kein "do"  
continue;  
break;*

- Funktionen
  - Deklaration int maximum(int, int);
  - Definition
    - int maximum(int k, int l)
      - {
        - return ( k > l ? k : l );

Parameterübergabe grundsätzlich per value!

## Der Präprozessor

Der Präprozessor ist ein Programm, das der eigentlichen Übersetzung vorgeschaltet ist.

Anweisungen für den Präprozessor werden zwischen den C-Code geschrieben und bewirken

- Ergänzungen
- Ersetzungen oder
- Auslassungen

im C-Code des Programmes.

**cc -E Quelldatei** liefert Ergebnis des Präprozessors.

---

C C.2 1

---

Totzauer April 2001

```
#include < file name >
```

```
#include " file name "
```

```
#define identifier token-sequence
```

```
#define identifier(identifier-list) token-sequence
```

```
#undef identifier
```

Zeilen verbinden mit "\

Besonderheiten: "#" oder "##" in der Parameterliste

---

Präprozessor C.2 2

```
#if const-expression ( #ifdef id #ifndef id )
```

```
Code
```

```
#elif const-expression ( defined id )
```

```
Code
```

```
#else
```

```
Code
```

```
#endif
```

In den Code-Teilen dürfen Anweisungen an den Präprozessor enthalten sein.

---

Präprozessor C.2 3

Typen

Aufzählungstyp

Den Elementen werden von 0 beginnend aufsteigend int-Werte zugeordnet.

```
enum Tage { montag, dienstag, mittwoch, donnerstag,
           freitag, samstag, sonntag };
```

```
enum Monate { jan=1, feb, mar, apr, mai, jun, jul,
             aug, sep, okt, nov, dez };
```

Zeiger und Felder

int a[10] definiert ein Feld mit 10 int-Elementen  
a ist ein konstanter Zeiger auf das Element a[0]

```
int d[] = { 225, 289, 324 }; /*definiert einen
                             3-elementigen Vektor */
```

```
int *b, *c; /* definiert Zeiger auf int */
```

mögliche Operationen

```
b = a; c = b + 3; i = b[3]; /* wirklich b */
```

```
j = a - b; b++; j = *(b+3);
```

"Dies ist eine konstante Zeichenkette"

Sie wird intern mit dem Nullzeichen '\0' beendet.

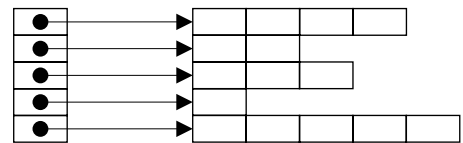
2 unterschiedliche Definitionen

```
char feld[] = "nasowas"; /* initialisiertes Feld */
char *zeig = "nasowas"; /* initialisierter Zeiger */
```

Mehrdimensionale Felder

```
int mdim[2][3][4];
int zeigmdim[3][4];
```

2-dimensionale Strukturen mit unterschiedlich langen Zeilen



Argumente aus der Kommandozeile

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

Umwandlung durch atoi(argv[i]) und atof(argv[j]);

Strukturen und Unions

```

struct Bezeichner
{
  Komponenten
};
struct Bezeichner variable;

```

*union wie struct,  
nur Elemente beginnen  
bei gleicher Adresse*

**Zugriff:** Strukturvariable.Komponentenname  
[\*Zeigervariable].Komponentenname  
Zeigervariable->Komponentenname

```

in <stdlib>
void *calloc(size_t anz, size_t groesse);
void *malloc(size_t groesse);
void free( *zeiger);

```

Bitfelder

```

struct bitgrp
{
  unsigned int grp1: 3;
  unsigned int grp2: 5;
  unsigned int grp3: 4;
};

```

Alternative  
Verwendung von Integers und Maskenvariablen

Komplizierte Typen

```

Typ von v
int v;           integer
int (v);         integer
int *v;          Zeiger auf int
int v[];         Vektor von int
int v(Param);   Funktion(Param) mit Ergebnis int

```

Verschachtelung durch Definition neuer Typen

```

typedef int length, *zeigint;
typedef zeigint feldzeigint[10];
typedef zeigint fktzeigint(int);
Hinschreiben als Wahnsinnsausdruck

```

\* hat geringere Priorität als [] und ()  
Auswertung von innen nach außen  
Aufschreibung von links nach rechts vor Basistyp

Beispiele:

```

int *v(int, int);   Fkt mit Erg Zeiger auf int
int [*v](int, int); Zeiger auf Fkt mit Erg int
int *v[10];         Vektor[10] von Zeiger auf int
int [*v][10]        Zeiger auf Vektor[10] von int

```

Speicherklassen und Gültigkeitsbereiche

In C kann derselbe Name für unterschiedliche Objekte benutzt werden.

- als Präprozessorname
- als Marke
- als Name eines Strukturtypen
- als Komponente innerhalb einer Struktur
- als anderer Bezeichner

Vermeiden, aber beruhigend bei Komponentennamen.

Arten von Deklarationen

- extern oder global, außerhalb von Funktionen
- Parameterdeklaration in der Funktionskopfzeile
- lokale Deklaration innerhalb eines Funktionsrumpfes oder Blockes
- Funktionen gelten immer als global deklariert, selbst wenn sie innerhalb einer anderen Funktion geschrieben sind.

Gültigkeitsbereiche

ab Auftreten bis Block, Funktions- oder Dateiende Sichtbarkeit kann verdeckt sein.

Vorwärtsreferenzen

sind nicht erforderlich bei Typbezeichnern von Strukturtypen, die in typedefs oder in Verbindung mit einem Zeigertypen benutzt werden.

Lebensdauer von Objekten

- *automatisch*, Anlegen bei Betreten eines Blockes und Freigeben bei Verlassen eines Blockes
- *dynamisch*, mit malloc() und free()
- *statisch*, Objekte existieren während der gesamten Programmausführung, sind jedoch evtl. nur in einer Funktion oder innerhalb eines Blockes sichtbar.

Bindung von Bezeichnern

Wird ein Programm auf mehrere Dateien aufgeteilt, so kann es erwünscht sein, daß eine einzige Variable aus allen Dateien angesprochen werden kann. Auf der anderen Seite sollen nicht alle Daten und Funktionen innerhalb einer Datei von außen sichtbar sein.

```
extern int i; /* bezieht sich auf i aus anderer Datei */
static f(int, int); /* außen nicht sichtbare Funktion */
static int j; /* außen nicht sichtbar */
```

Speicherklassen

beeinflussen **Bindung**, **Lebensdauer** und **Gültigkeit** eines Objektes.

- auto
  - nur am Blockanfang erlaubt und dort Standard, wird daher i.d.R. nicht geschrieben
  - Erzeugung bei jedem Blockeintritt,
  - Freigeben bei Verlassen
  - Objekt bis zum Ende des umschließenden Blocks bekannt

- register

- wie auto, soll aber in Register gelegt werden,
- keine Garantie, sollte man Compiler überlassen

- static

- für Funktionen: nur in Datei bekannt
- für Daten: sie behalten ihre Werte auch nach verlassen des Blocks, in dem sie definiert sind.
- Interessant für lokale Variable in Funktionen.

- extern

- vgl. Bindung, und zusätzlich ....hier uninteressant

- Verwendung von Funktionen existierender Bibliotheken
  - ctype.h
  - math.h
  - string.h
  - stdio.h
  - stdlib.h
  - signal.h .....
- Erstellung und Nutzung eigener Bibliotheken

- vor Gebrauch bekanntmachen durch
    - Angabe des Prototyps
    - include der entsprechenden Datei
- Es kann ohne vorherige Angabe funktionieren, ist aber ein Zufallstreffer.
- zum Binden zugehörige Bibliothek bekanntmachen
    - durch Voreinstellung
    - durch Angabe beim cc-Aufruf

```
<ctype.h> int isalpha(int)
           int isdigit(int)
           int isspace(int)
           int toupper(int)
<string.h> char *strcpy (char *, const char *)
           char *strcat (char *, const char *)
           int *strcmp (const char *, const char *)
           size_t strlen( const char *)
```

```
           int sscanf(const char *, const char *, ...)
<math.h> double sin(double)
           double cos(double)
           double exp(double)
           double log(double)
           double log10(double)
<stdlib.h> double atof( const char * )
           int rand( void )
           void srand( unsigned int seed )
```

Bibliotheken enthalten .o-Dateien

Konvention für Bibliotheksnamen

- enden auf .a
- beginnen mit lib

```
ar -r libf.a d1.o   schreibt d1.o in libf.a
ar -t libf.a       druckt Inhaltsverz. der Bibl.
```

- Angabe der Bibliothek bei cc anstelle der darin enthaltenen .o-Dateien
- Nutzung bestimmter Optionen für cc
  - -ldirname sucht <->-include-Dateien auch in dirname
  - -Ldirname sucht Bibliotheken auch in dirname
  - -lname verwendet die Bibliothek libname.a

Beispiel:

```
cc -o main main.c -I/VERZ/incl -LVERZ/lib -lf
```