# Kurze Einführung in C

Florian Hilger, Franz Och

Lehrstuhl für Informatik VI

- Allgemeines
- Datentypen
- Programmaufbau
- Kontrollstrukturen
- Zeiger
- Ein– und Ausgabe
- Besonderheiten von C

www-i6.informatik.rwth-aachen.de → Course Material

# C allgemein

- C wurde unter UNIX und für UNIX entwickelt.
- C ist sehr machinen–nah und dadurch relativ schnell.
- C hat kein strenges Typkonzept.
- C kann wegen des begrenzten Sprachumfangs als sehr kleines System laufen.
- C ist nicht immer bequem aber sehr vielseitig.
- C ist sehr gut durch Funktionsbibliotheken erweiterbar.
- Nachfolger sind C++ und Objective C.

Vorsicht:

- In C lässt sich fast alles compilieren!
- Der Programmierer trägt die Verantwortung!

# Datentypen:
# Vergleich zwischen Modula und C:

| MODULA | C |
|---|---|
| VAR | wird durch die Position impliziert |
| i: INTEGER; | int i; |
| ui: CARDINAL; | unsigned int ui; |
| r: REAL; | float r; double r; |
| c: CHAR; | char c; |
| b: BOOLEAN; | gibt es so nicht |
| vektor: ARRAY[0..99] OF CHAR; | char vektor[100]; |
| verbund = RECORD OF member: INTEGER; c: CHAR; END; | struct verbund{ int member; char c; } |

# Typdefinitionen:

MODULA

```
TYPE person = RECORD OF
  name: ARRAY[0..79] OF CHAR;
  age: INTEGER;
  END;

VAR p: person;
```

C

```
typedef struct{
  char name[80];
  int age;
} Person;

Person p;
```

**In C muß bei rekursiven Strukturen ein zusätzlicher Name angegeben werden, der in der Struktur verwandt werden kann:**

```
typedef struct _NODE{
  int member;
  struct _NODE *nextode;
} NODE;

NODE node;
```

**Der Stern * im Beispiel bezeichnet einen Zeiger.**

# Zugriff auf Strukturen und Arrays:

**In C wie in Modula mit Punkt . für Elemente von Strukturen und eckigen Klammern [ ] bei Arrays:**

```
typedef struct{
  char name[80];
  int age;
} Person;

Person person;

person.age = 42;

person.name[0] = 'M';
person.name[1] = 'e';
person.name[2] = 'y';
person.name[3] = 'e';
person.name[4] = 'r';
```

**In C ist beim Initialisieren auch folgende Deklaration gültig:**

```
char name[] = "Meyer" ;
```

# Programmaufbau:

**Der Programmaufbau ist in C weniger stark festgelegt als unter Modula. Ein ähnicher Aufbau ist aber möglich:**

```
MODULE test                          /* Programmname wird
                                          nicht angegeben */

FROM ... IMPORT ...;                 #include <...>

(* Typdefinitionen *)                /* Typdefinitionen */
TYPE ...                             typedef ...

(* Globale Variablen *)              /* Globale Variablen */
VAR ...                              int ...

(*Prozeduren und Funktionen *)       /* Funktionen */

(* Hauptprogramm *)                  /* Hauptprogramm */
BEGIN                                void main (void)
                                     {
...                                  ...

END.                                 }
```

# Beispiel:

```
#include <stdio.h>

float Celsius_to_Kelvin(float celsius)
{
  return(273.16 + celsius);
}

void main(void)
{
  float eingabe, kelvin;

  printf ("Temperatur in Grad Celsius? ");
  scanf ("%f", &eingabe);

  kelvin = Celsius_to_Kelvin(eingabe);
  printf("\n %f Grad Celsius entsprechen
  %f Kelvin.",eingabe,kelvin);
}
```

# Kontrollstrukturen:

**Die in Modula vorhandenen Kontrollstrukturen gibt es auch in C:**

```
IF Bedingung THEN
  Anweisungen
ELSE
 Anweisungen
END;
```

```
if (Bedingung){
  Anweisungen
}else{
  Anweisungen
}
```

```
CASE Variable OF
  Ausdruck: Anweisungen
  ...
  ...
  Ausdruck: Anweisungen
  ...
ELSE
  Anweisungen
END;
```

```
switch (Variable){
   case Ausdruck: Anweisungen
   break;
   ...
   case Ausdruck: Anweisungen
   break;
   default: Anweisungen

 }
```

```
FOR i:=1 to N DO
  Anweisungen
END;
```

```
for(i=1;i<=N;i=i+1){
  Anweisungen
}
```

# Kontrollstrukturen:

```
WHILE Bedingung
  Anweisungen
END;
```

```
while (Bedingung){
  Anweisungen
}
```

```
REPEAT
  Anweisungen
UNTIL Bedingung
```

```
do {
  Anweisungen
} while (Bedingung)
```

```
LOOP
  Anweisungen
  IF Bedingung THEN
    EXIT END;
END;
```

```
while(1){
  Anweisungen
  if (Bedingung)
    break;
}
```

## Zuweisung:

| MODULA | C |
|---|---|
| := | = |

## Bedingungen:

| | | |
|---|---|---|
| = | == | Beliebter Fehler !!! |
| <> bzw # | != | |
| < | < | |
| <= | <= | |
| > | > | |
| >= | >= | |
| | | |
| AND | && | |
| OR | \|\| | |

**Beachte den wichtigen Unterschied bei der Zuweisung und beim Vergleich auf Übereinstimmung!**

# Zeiger:

**Wichtig sind der Adress–Operator & und der Inhalts–Operator *:**

```
int x = 1, y = 2;
int *ip;          /* ip ist ein Pointer auf int */
...
ip = &x;          /* ip zeigt auf x  */
y = *ip;          /* y ist jetzt 1   */
*ip = 0;          /* x ist jetzt 0   */
```

**Wichtige Besonderheit Komponenten von Strukturen:**

```
typedef struct{
  char name[80];
  int age;
} Person;

Person person;
Person *person_pointer;      /* Zeiger auf Person */
...
person.age = 30;
*person_pointer.age = 30;    /* FEHLER, SO NICHT! */
(*person_pointer).age = 30;  /* geht              */
person_pointer -> age = 30;  /* besser            */
```

## Parameterübergabe – Call by Value:

```
PROCEDURE test (i: integer)
BEGIN
 i:= 42;
END;
...
test (x);
```

```
void test(int i);
{
  i = 42;
}
...
test (x);
```

## Variablenübergabe – Call by Reference:

```
PROCEDURE test (VAR i: integer)
BEGIN
 i:= 42;
END;
...
test (x);
```

```
void test(int *i);
{
  *i = 42;
}
...
test (&x);
```

# Zeiger und Arrays:

**Array Variablen sind Zeiger auf das erste Element.**

```
int number[100];
int i, *num_pointer:

num_pointer = &number[0];  /* Diese Zeilen     */
num_pointer = number;      /* sind aequivalent */

for (i=0;i<100;i=i+1){

  number[i] = 0;

  /* genau so geht: */

  *num_pointer = 0;
  num_pointer = num_pointer + 1;

}
```

# Zeichenketten – Strings:

- Strings sind Zeiger auf char Arrays:
  char string[256];
- Das letzte Element ist '\0'
- Benutze einfache ' bei einzelnen Zeichen
- und doppelte " bei Zeichenketten
- Es gibt viele Bibliotheksfunktionen:
  strcpy(), strcmp(), strcat()

```
char name1[] = "Rolf";
char name2[] = "Hans";

name2 = name1;
name2[1] = 'a';           /* ergibt Ralf */
```

# Ausgabe:

**Die Ausgabe erfolgt mit der printf Funktion:**

```
printf (kontrollstring, arg1, arg2, ...);
```

**Der Kontrollstring enthält den Ausgabetext und mit % gekennzeichnete Platzhalter (Umwandlungsangaben) für die Argumente → Tabellen in der Literatur.**

**Beispiele:**

```
/* Hallo */
printf ("Hallo\n");

/* ASCII 65 entspricht A */
printf ("ASCII %d entspricht %c\n", 65,65);

/* Hallo       Welt */
printf("Hallo %10s\n","Welt");
```

**Ausgabe in Dateien erfolgt mit mit fprintf:**

```
fprintf (FILE *, kontrollstring, arg1, arg2, ...);
```

# Eingabe:

**Völlig analog zur Ausgabe mit scanf bzw. fscanf:**

```
scanf (kontrollstring, arg1, arg2, ...);
```

**Wichtig: Die Argumente müssen Zeiger sein!**

**Beispiel:**

```
scanf("%f", &eingabe);    /* ist korrekt */

scanf("%f", eingabe);     /* GEHT NICHT! */
```

# Besonderheiten von C:

**C bietet einige nützliche Besonderheiten, die es in anderen Programmiersprechen nicht gibt.**

**Vorsicht: Bei unvernünftiger Anwendung kann man damit völlig unverständliche Programme schreiben!**

- **Die Operatoren ++ und −−:**

```
n++ entspricht n = n + 1 (Inkr. nach der Ausfuehrung)
++n entspricht n = n + 1 (Inkr. vor der Ausfuehrung)
-- analog
```

- **Abkürzende Schreibweise:**

```
n += 5 entspricht n = n + 5
n *= 5 entspricht n = n * 5
usw.
```

- **Mehrfachzuweisungen:**

```
a = b = c = d = 0;    ist erlaubt und funktioniert
int a, b = 0;         a wird nicht initialisiert!
```

- **Rechnen mit Zeigern:**
  - Ist ptr ein Zeiger, dann ist ptr++ das naechste Element.
  - Vergleiche == < > sind auch moeglich.
  - Aber Vorsicht: Es gibt keine Ueberpruefung der Grenzen!

- **Casting:**
  - C hat kein strenges Typkonzept. Es erfolgt eine weitgehend automatische Typanpassung.
  - Es kann auch explizit mit dem Cast-Operator angepasst werden:

```
int n;
float inv_n;
...
inv_n = 1 / n;          /* ergibt 0 */
inv_n = (float)(1/n);   /* ergibt 0 */

inv_n = (float) 1 / (float) n; /* ergibt 0.2 */
inv_n = (float) 1 / n;         /* ergibt 0.2 */
inv_n =  1 / (float) n;        /* ergibt 0.2 */
inv_n =  1. / n;               /* ergibt 0.2 */
```

## Vier Versionen der Implementierung von String–Copy strcpy:

1. Ganz einfach, mit Arrays

```
void strcpy(char dest[],char src[])
{
  int i;
  i = 0;
  while (src[i] != '\0'){
    dest[i] = src[i];
    i = i + 1;
  }
  dest[i] = '\0';
}
```

2. Kuerzer, mit Arrays

```
void strcpy(char dest[],char src[])
{
  int i = 0;
  while ((dest[i]=src[i]) != '\0')
    i = i + 1;
}
```

## Versionen von strcpy():

3. Rechnen mit Pointern und ++

```
void strcpy(char *dest,char *src)
{
  while (src != '\0'){
    *dest = *src;
    dest++;
    src++;
  }
  *dest = '\0';
}
```

4. Ganz kurz ...

```
void strcpy(char *dest,char *src)
{
  while (*dest++ = *src++);
}
```

## Modularisierung:

Besteht ein Programm aus mehreren Teilen (Modulen) werden diese in C erst unabhängig compiliert und dann später zusammen gelinkt.

## Prototypen:

Welche Funktion aus einem anderen Modul gelinkt werden muss nicht, sollte aber mit sog. Prototypen angegeben werden.

Die Deklaration einer Funktion ist ein Prototyp:

```
void strcpy (char *, char *);
```

Üblicherweise werden die Prototypen eines Moduls in einer Header–Datei (.h) zusammengefasst.

Mit #include können sie dann zur Compile–Zeit geladen werden.

# Wichtige Header Dateien:

**Es gibt einige Header–Dateien für Funktionen aus der Standard–Library, die eigentlich immer gebraucht werden:**

`stdio.h` Funktionen zur Ein-- und Ausgabe:
`printf, scanf, fopen, fclose, ...`

`string.h` Funktionen zum Bearbeiten von Strings:
`strcpy, strlen, strdup, strchr, ...`

`math.h` Mathematische Funktionen:
`sin, cos, sqrt, exp, ...`

# Literatur:

**B. W. Kernighan, D. M. Ritchie**
**“The C Programming Language, Second Edition ANSI C”**
**Prentice–Hall International, New Jersey, 1988**

**UNIX man–Pages**