

PR: Programmiersprache C (Linux) Einheit 6: Pointer (II), Präprozessor

Roland A. Eggetsberger
Inst. f. Informationsverarbeitung und Mikroprozessortechnik (FIM)
Johannes Kepler Univ. Linz, Altenbergerstr. 69, A-4040 Linz, Austria

Pointer (II)

WH: Call by Reference Pointerarithmetik Pointer und Arrays WH: Pointer und Strukturen Häufige Fehler Dynamische Strukturen Pointer auf Pointer	Kommandozeilen- eingabe Pointer auf Funktionen Variable Parameterlisten
---	--

WH: Call by Reference

- Beispiel

```
void zaehle(int *zahl_ptr) {
    (*zahl_ptr)++;
}
```

Hier wird der Inhalt der Speicheradresse erhöht.

Pointerarithmetik (I)

- Man kann mit Pointern rechnen

```
int *int_ptr;
:
++int_ptr;
```

- Es werden 4 Bytes addiert

Pointerarithmetik (II)

- Keine Adressarithmetik ist hingegen auf void-Pointern möglich
- Möglich ist hingegen eine Zeiger-Subtraktion (falls sie definiert ist) zum Auffinden von Indizes

Pointer und Arrays (I)

- Beispiel (I)

```
int vec[10], elm, i;
int *vec_ptr;
:
vec_ptr = &vec[0];
elm = *vec_ptr; (elm = vec[0];)
```

- Allgemein

```
*(vec_ptr + i) ... vec[i]
```

Pointer und Arrays (II)

- Beispiel (II)

```
int vec[10], elm, i;
int *vec_ptr;

vec_ptr = vec;
```

- Allgemein

```
vec_ptr + i ..... &vec[i]
*(vec_ptr + i) ... vec[i]
```

Pointer und Arrays (III)

- Unterschied

Pointer ist eine Variable

```
vec_ptr = vec
++vec_ptr
```

Array ist keine - nicht möglich daher sind

```
vec = vec_ptr
++vec
```

Pointer und Arrays (IV)

- Funktionsdeklarationen

Äquivalent sind

```
int strlen(char s[]) ...
int strlen(char *s)
```

- Funktionsaufrufe

Äquivalent sind

```
strlen(s) ... strlen(&s[0])
```

Pointer und Arrays (V)

- Arrays von Pointern (I)

Beispiel - Zeilen eines Textes

Ein einziges char-Array

Pointer in eigenem Array

Vergleiche mit strcmp()

Vertauschen als Pointer

Pointer und Arrays (VI)

- Arrays von Pointern (II)

Verhindert

Komplizierte Speicherverwaltung

Aufwand zur Zeilenvertauschung

Pointer und Arrays (VII)

- Arrays von Pointern (III)

Static Initialisierer

```
static char *tage = {
    "Montag",
    "Dienstag",
    ...
}
```

Pointer und Arrays (VIII)

- Mehrdimensionale Arrays (I)
Sind Arrays die Zeile für Zeile abgespeichert sind
Daher ist bei der Übergabe als Parameter die
Anzahl der Spalten relevant, die der Zeilen nicht

```
int mat[2][3];  
f(int mat[][3]);    (f(int (*mat)[3]);)
```

Pointer und Arrays (IX)

- Mehrdimensionale Arrays (II)
Man unterscheide

```
int (*a)[5];
```

Pointer zu einem Feld von 5 Integers

```
int *a[5];
```

Feld von 5 Pointern zu Integers

Pointer und Arrays (X)

- Mehrdimensionale Arrays (III)

Bei

```
int *a[5]; bzw.  
int a[4][5];
```

ist `a[1][2]` möglich, hängt aber im ersten Fall
davon ab wieviel Speicher angelegt wird

Pointer und Arrays (XI)

- Strings aufspalten

```
void *strchr(...);
```

Teilt nach dem ersten String in zwei Strings auf
und liefert einen Pointer auf den zweiten String

WH: Pointer und Strukturen

- Aufpassen auf Typen und Tags
- Zugriff auf Komponenten
- Listen, ...
- Bei einem Feld von Strukturen besser Pointer
sortieren als Strukturen

Häufige Fehler (I)

- Keine Adresse

```
int *ptr;  
*ptr = 5;
```

- Abhilfe

```
int *ptr, i;  
ptr = &i;  
*ptr = 5;
```

Häufige Fehler (II)

- Funktion `malloc` liefert Pointer

```
*ptr = (char *)malloc(100);
*ptr = 'y';
```

- Abhilfe

```
ptr = (char *)malloc(100);
*ptr = 'y';
```

Häufige Fehler (III)

- Vergessen auf Null-Pointer

```
ptr = (char *)malloc(100);
*ptr = 'y';
```

- Abhilfe

```
ptr = (char *)malloc(100);
if (ptr != NULL)
    *ptr = 'y';
```

Häufige Fehler (IV)

- Bemerkung

Möglich ist

```
char *s;
char *p = s;
```

Möglich ist auch

```
char feld[3];
char *f_ptr = &feld[0];
```

Häufige Fehler (V)

- Achtung

Keine Pointer auf lokale Variablen zurückgeben

Dynamische Strukturen (I)

- Speicher initialisieren

```
void *calloc(typ anzahl, typ groesse);
```

- Speicher vergrößern

```
void *realloc(void *ptr, typ groesse);
```

Dynamische Strukturen (II)

- Bemerkung

Größenangaben sind vor allem für Strukturen interessant (händische Berechnung umständlich)
Nach dem Freigeben von Speicher Pointer auf `NULL` setzen

Dynamische Strukturen (III)

- Günstig ist folgende Vorgehensweise
Speicher mit `malloc` (o.ä. Funktionen) anlegen
bzw. erweitern
Durcharbeiten mit
[] ... Bei wahlfreiem Zugriff
*() ... Bei sequenziellem Zugriff
Speicher freigeben

Pointer auf Pointer (I)

- Modell



Pointer auf Pointer (II)

- Deklaration

```
int **int_ptr;
```

- Alternativ

```
int *int_ptr[];
```

- Zugriff daher möglich via

```
int_ptr[0];
```

Kommandozeileingabe (I)

- Aufruf von `main`

```
main(int argc, char **argv);
```

`argc` ... Anzahl (inklusive Programmname)
`argv` ... Argumente selbst

Kommandozeileingabe (II)

- Leerzeichen als Trennzeichen
- Leerzeichen in Argumenten unter Hochkommata
- Standards beachten
`command options file1 file2 ...`
- Bemerkung
Wildcards werden schon vorher expandiert

Kommandozeileingabe (III)

- Beispiel - Optionen abarbeiten

```
while ((argc > 1) && (argv[1][0] == '-')){
    switch (argv[1][1]){
        :
    }
    :
    --argc;
    ++argv;
}
```

Pointer auf Funktionen (I)

- Verwendung

```
int (*f_ptr)();
int f();
f_ptr = &f;
```

- Parameterdeklaration

```
int (*f_ptr)(int);
int f(int);
f_ptr = &f;
```

Pointer auf Funktionen (II)

- Aufruf

Möglich sind

```
int i;
i = f(1);
i = f_ptr(1);
```

Variable Parameterlisten (I)

- Form

```
typ func(parameter, ...) {
:
}
```

Variable Parameterlisten (II)

- Deklaration

```
#include <stdarg.h>
```

- Datentyp für Parameterliste

```
va_list arg_ptr;
```

Variable Parameterlisten (III)

- Makros (I)

Initialisierung der Argumentliste

```
void va_start(va_list argptr, ...);
```

Nächsten Parameter lesen

```
typ va_arg(va_list argptr, typ);
```

Variable Parameterlisten (III)

- Makros (II)

Endbehandlung

```
void va_end(va_list argptr);
```

- Ausgabe

```
int vprintf(char *format, va_list argptr);
```

Präprozessor

Eigenschaften
Makrodefinition
Fileeinbindung
Verzweigungen
Diverse
Modularisierung

Eigenschaften (I)

- Erster Schritt beim Compilervorgang
- Eigene Sprache
- Vorteile
 - Lesbarkeit
 - Einfache Erweiterung
 - Einfache Modifizierung
 - Hilfe bei Umgebungsanpassung

Eigenschaften (II)

- Aufgaben im Detail
 - Dateien einbinden
 - Textuelles Ersetzen
 - Bedingtes Compilieren

Makrodefinition (I)

- Form


```
#define MAKRO NAME
```
- Konvention
 - Großbuchstaben

Makrodefinition (II)

- Beispiel


```
#define FALSE 0
#define TRUE !FALSE
```

Makrodefinition (III)

- Dabei sind nicht nur Konstante möglich
- Beispiel


```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

Makrodefinition (IV)

- Klammerung
Runde Klammern für Platzhalter um die Auswertungsreihenfolge nicht durcheinander zu bringen
Geschwungene Klammern für Blöcke

- Beispiel

```
#define SQR(x) (x * x)
SQR(1 + 1); ... 1 + 1 * 1 + 1
```

Makrodefinition (V)

- Definitionen können rückgängig gemacht werden
- Form

```
#undefine makro
```

Makrodefinition (VI)

- Vorteile der Konstantendefinition in C
Teil der Sprache
Wird sofort überprüft
Geltungsbereiche wie üblich
- Defines interessant für bedingtes Compilieren

Fileeinbindung (I)

- Form
- Für Prototypen und Deklarationen

```
#include <file>
#include "file"
```

Fileeinbindung (II)

- Verwendung

```
#include <file>
```

Suche in `/usr/include` bzw. in über `-I` angegebenen Verzeichnissen

```
#include "file"
```

Suche im aktuellen Verzeichnis

Fileeinbindung (III)

- Information über Quelldateien

```
FILE_ ... Name
LINE_ ... Zeile
```

- String-Operator
Liefert Parameter als String (und nicht ausgewertet)

Verzweigungen (I)

- Form

```
#if
#endif
```

- Mehrfach

```
#else
#elif
```

Verzweigungen (II)

- Testen von Flags

```
#ifdef
#endif
```

- Bemerkung

Flags oft wichtig um Mehrfachdefinitionen von Konstanten, Strukturen und Unions zu vermeiden

```
#ifndef __included__
```

Diverse (I)

- Fehler generieren

```
#error Text
```

- Sprungbefehl

```
#line
```

Diverse (II)

- Verlängerungszeile

Normalerweise Direktive bis zum Zeilenende
Verlängerung mit Backslash

- Achtung

Kein Semikolon verwenden
Kein "=" bei defines

Diverse (III)

- Präprozessorausgabe
Via Compileroption

```
gcc -E
```

Diverse (IV)

- Bedingtes Compilieren (I)

```
#define DEBUG
:
#ifdef DEBUG
:
#endif DEBUG /* DEBUG */
```

Diverse (V)

- Bedingtes Compilieren (II)
Über Compileroptionen

```
gcc -DSYMBOL=wert
gcc -DDEBUG
```

Modularisierung (I)

- Modul
Einige Funktionen die ähnliche Dinge tun
- Aufteilung
Public - m.h
Header File mit Datenstrukturen, öffentlichen
Funktionsdeklarationen
Private - m.c (Rest)

Modularisierung (II)

- Extern
Für Variablen und Funktionen, die in anderen
Files definiert wurden
- Static
Für globale Variablen, die privat für das File sind

Modularisierung (III)

- Header
Kommentar
Konstante
Strukturen
Prototypen (public)
Extern (für public Variablen)
Prototypen im Header extern (da sie im Body
definiert werden)

Modularisierung (IV)

- Body
Private Funktionen static