

C-Syntax II

Themenübersicht

1. Enumerationen
2. Arrays und Zeiger
3. Strukturen und Unions
4. undefiniertes Verhalten
5. Der Präprozessor
6. dynamische Speicherallokation
7. Speicherklassen und -qualifizierer
8. Typqualifizierer

Enumerationen

- ▶ Syntax: `enum [Bezeichner] { Bezeichner [= Wert] [, ...] };`
- ▶ Definiert symbolische Ganzzahlkonstanten vom Typ `int`
- ▶ Kann auch als Typ verwendet werden
- ▶ Wird *Wert* weggelassen, so wird die nächste Zahl verwendet

Enumerationen

```
1 enum {
2     FOO,           // Wert: 0
3     BAR,           // Wert: 1
4     BAZ,           // Wert: 2
5     QUUX = 42,     // Wert: 42
6     LOL,           // Wert: 43
7     NOPE = -1,    // Wert: -1
8     YES,           // Wert: 0
9 };
```

Zeiger

- ▶ *Zeiger* zeigen auf Objekte
- ▶ Deklaration: *Typ * Bezeichner [= Wert] ;*
- ▶ Der *-Operator *dereferenziert* einen Zeiger
- ▶ Der &-Operator *referenziert* ein Objekt
- ▶ * und & löschen einander aus, &*E ist immer äquivalent zu E
- ▶ Der NULL-Zeiger zeigt nirgendwo hin

Zeiger

```
1 int x = 0, y = 1, *p;  
2 p = &x;           // p zeigt auf x  
3 printf("%d\n", *p); // gibt 0 aus  
4 *p = 42;         // weise 42 indirekt zu  
5 printf("%d\n", x); // gibt 42 aus  
6 p = &y;           // p zeigt auf y  
7 printf("%d\n", *p); // gibt 1 aus  
8 *p = 23;         // weise 23 indirekt zu  
9 printf("%d\n", y); // gibt 23 aus
```

Arrays

- ▶ *Arrays* sind stetige Regionen gleichartiger Objekte
- ▶ Deklaration: *Typ Bezeichner* [[*Ausdruck*]] ;
- ▶ Ausdruck in eckigen Klammern bezeichnet die Arraylänge
- ▶ Zugriff auf Array-Einträge mit dem []-Operator
- ▶ `array[index]` ist äquivalent zu `*(array + index)`
- ▶ Arrays verhalten sich fast überall wie Zeiger auf ihr erstes Element.
- ▶ Arraygröße muss meistens eine Konstante sein

Arrays

```
1 int foo[5] = { 1, 2, 3, 4, 5 }, *bar;
2 printf("%d\n", foo[3]); // gibt 4 zurück
3 bar = &foo[4]; // Zeiger auf 4. Element
4 bar = foo + 4; // dito
5
6 char test[] = "soso" // Strings sind Arrays
7 printf("%d\n", test[4]); // gibt 0 zurück
```


hilfreiche Funktionen

```
1 char buf[20];  
2 memset(buf, '\0', 20);           // Puffer mit 0 initialisieren  
3 strcpy(buf, "Hallo_");          // Zeichenkette kopieren  
4 strcat(buf, "Welt!\n");         // Zeichenkette anhängen  
5 printf("%s", buf);              // Ausgabe: "Hallo Welt!\n"
```

Strukturen

- ▶ eine *Struktur* sammelt benannte Felder verschiedener Typen in einem Objekt
- ▶ Syntax: `struct [Bezeichner] { Deklaration ; ... }`
- ▶ auf Strukturfelder kann mit dem Operator `.` zugegriffen werde
- ▶ `a->b` ist Abkürzung für `(*a).b`

Strukturen

```
1 /* Deklaration der Struktur */
2 struct mensch {
3     char *vorname, *nachname;
4     int  groesse, alter;
5 };
6
7 struct mensch hans, *kunde = &hans;
8
9 hans.vorname = "Hans";
10 hans.nachname = "Meier";
11 kunde->alter = 42;
```

Unions

- ▶ eine *Union* ist wie eine Struktur, hält aber nur eines seiner Felder gleichzeitig
- ▶ alle Felder einer Union liegen auf dem gleichen Stück Speicher
- ▶ ansonsten funktionieren sie exakt wie Strukturen
- ▶ wir gehen nicht näher darauf ein

Unzulässiges Zeugs

Was passiert...

- ▶ wenn man durch Null teilt?
- ▶ wenn man eine uninitialisierte Variable ließt?
- ▶ wenn man hinter das Ende eines Arrays schreibt?
- ▶ wenn man auf freigegebenen Speicher zugreift?
- ▶ ...

Undefiniertes Verhalten

Antwort: *Das Verhalten ist undefiniert.*

- ▶ die Sprache C trifft keine Aussagen darüber, was dann passiert
- ▶ alles ist erlaubt, von »garnichts« über »Festplatte formatieren« bis »kleine Teufelchen fliegen aus deiner Nase«
- ▶ korrekte C-Programme enthalten kein undefiniertes Verhalten
- ▶ passt also beim Programmieren auf!

sonstiges uneindeutiges Zeugs

Es gibt noch zwei weitere Formen uneindeutigen Verhaltens:

- ▶ bei *implementationsabhängigem Verhalten* (z. B. Größe von Typen, negative Zahlen nach rechts schieben) muss die Implementierung der Sprache C eine Entscheidung treffen und dokumentieren. Das Verhalten ist dann immer so wie dokumentiert.
- ▶ bei *unspezifiziertem Verhalten* (z. B. Reihenfolge der Termauswertung und Variableninitialisierung) gibt es mehrere erlaubte Verhaltensweisen, der Compiler darf sich jedes mal eine Verhaltensweise beliebig auswählen.

Der Präprozessor

- ▶ Der Präprozessor läuft vor dem Compiler und bereitet den Quelltext für den Compiler auf
- ▶ Die Arbeit des Präprozessors kann mit `gcc -E quelltext.c` verfolgt werden
- ▶ Aufgaben: Kommentare entfernen, Direktiven auswerten, Makros ersetzen
- ▶ Das `#` einer Präprozessordirektive muss erstes Zeichen der Zeile sein

Präprozessordirektiven

```
#include "header.h"
```

```
#include <header.h>
```

- ▶ kopiert den Inhalt von `header.h` in das aktuelle Modul
- ▶ `<header.h>` für Systemheader, `"header.h"` für eigene Header

Präprozessordirektiven

```
#define MACRO definition ...  
#define MACRO(x,y,z) definition ...  
#undef MACRO
```

- ▶ Definiert ein Makro. Das Makro wird überall durch seine Definition ersetzt
- ▶ Funktionsartige Makros haben Argumente, die mit ersetzt werden
- ▶ Vorsicht bei der Klammerung in funktionsartigen Makros
- ▶ Makros können mit `#undef` aufgehoben werden

Präprozessordirektiven

```
#ifdef MACRO  
#if ausdruck  
#else  
#elif ausdruck  
#endif
```

- ▶ Wie `if`-Anweisung, wird aber im Präprozessor ausgeführt
- ▶ spezieller Operator `defined()` prüft, ob Makro definiert ist
- ▶ `elif` ist wie `else if`
- ▶ nützlich, um das Programm beim kompilieren anpassen zu können

Präprozessordirektiven

```
1 #if defined(__linux)
2 # define OS "Linux"
3 #elif defined(__FreeBSD__)
4 # define OS "FreeBSD"
5 #elif defined(_WIN32) || defined(_WIN64)
6 # define OS "Windows"
7 #else
8 # error Unbekanntes Betriebssystem!
9 #endif
```

Präprozessordirektiven

```
#error Nachricht ...  
#pragma ...  
#line 12 "foo.c"  
#
```

- ▶ `#error` bricht die Kompilation mit Fehlermeldung ab
- ▶ `#pragma` stellt kompilerspezifische Erweiterungen bereit, z. B. `#pragma omp` für OpenMP oder `#pragma STDC` für spezielle C-Features
- ▶ `#line` setzt Zeilennummer und Dateiname für Warnungen und Fehlermeldungen
- ▶ `#` ist die leere Direktive und macht nichts

dynamische Speicherallokation

- ▶ Was, wenn wir nicht vorher wissen, wie viel Speicher wir brauchen?
- ▶ dynamische Speicherallokation mit `malloc()` aus `<stdlib.h>`
- ▶ der Speicher wird mit `free()` wieder frei gegeben
- ▶ der `sizeof`-Operator ist ein Präfixoperator der die Größe eines Objektes bestimmt
- ▶ ideal zur Bestimmung der benötigten Speichergröße für `malloc()`
- ▶ für Arrays muss die Speichergröße mit der Anzahl der Elemente multipliziert werden

dynamische Speicherallokation

```
1 int *foo = malloc(sizeof *foo); // alloziert einen int
2 if (foo == NULL) // Anforderung erfolgreich?
3     perror("malloc"); // Fehlerbehandlung!
4 *foo = 1337;
5 free(foo); // Speicher freigeben
6
7 foo = malloc(16 * sizeof *foo); // alloziere 16 int
8 if (foo == NULL) /* ... */ ;
9 foo[14] = 4711;
10 free(foo);
```

hilfreiche Funktionen

Aus `<string.h>`

`memset(p, c, s)` Setzt die ersten `s` Bytes in `p` auf `c`

`memcpy(b, a, s)` kopiert Objekt `a` der Länge `s` nach `b`

`strlen(str)` gibt die Länge von `str` zurück.

`strcpy(b, a)` kopiert String `a` in Puffer `b`. Puffer muss lang genug sein!

`strcat(b, a)` hängt String `a` an String `b` an. Genug Platz muss da sein!

Deklarationen und Definitionen

Wir müssen Deklarationen und Definitionen unterscheiden:

- ▶ Eine *Deklaration* macht einen Bezeichner dem Compiler bekannt. Die selbe Variable oder Funktion kann beliebig oft deklariert werden
- ▶ Eine *Definition* erzeugt gleichzeitig eine Variable oder definiert eine Funktion. Jede Variable oder Funktion muss genau einmal definiert werden; jede Definition ist zugleich eine Deklaration

Speicherklassen

Speicherklassen bestimmen, wie lange ein Objekt am Leben ist.

- ▶ Ein *automatisches* Objekt wird alloziert und dealloziert, wenn der es umgebende Block betreten bzw. verlassen wird (wie Stackvariable in ASM)
- ▶ Ein *statisches* Objekt wird beim Programmstart alloziert und beim Programmende dealloziert (wie *DD* Variable in ASM)
- ▶ Ein *dynamisches* Objekt wird mit `malloc()` alloziert und mit `free()` dealloziert
- ▶ Ein *Thread-lokales* Objekt ist wie ein statisches Objekt, aber jeder Thread hat seine eigene Kopie. Hier nicht weiter betrachtet.

Verlinkung

Besteht ein Programm aus mehreren Modulen, so bestimmt die *Verlinkung* eines Bezeichners, wie dieser über Modulgrenzen gehandhabt wird.

- ▶ Alle Bezeichner gleichen Namens mit *externer Verlinkung* bezeichnen das selbe Ding innerhalb eines Programmes, wie *global* in ASM
- ▶ Alle Bezeichner gleichen Namens mit *interner Verlinkung* bezeichnen das selbe Ding innerhalb eines Moduls
- ▶ Alle Bezeichner gleichen Namens *ohne Verlinkung* bezeichnen verschiedene Dinge

Speicherklassenspezifizierer

Speicherklasse und Verlinkung eines Bezeichners kann durch einen *Speicherklassenspezifizierer* angegeben werden. Dieser wird in einer Deklaration dem Typ vorangestellt.

`static` bestimmt statische Speicherklasse und interne Verlinkung

`extern` bestimmt statische Speicherklasse und externe Verlinkung (Standard für globale Variablen). Unterscheidet zwischen Variablendefinition und -deklaration

`auto` bestimmt automatische Speicherklasse ohne Verlinkung, Standard für lokale Variablen, fast nie explizit angegeben

`register` wie `auto` mit Hinweis, Variable in Register zu halten, obsolet

`_Thread_local` bestimmt Thread-lokale Speicherklasse und externe Verlinkung

`typedef` erzeugt einen Typ-Alias anstatt einer Variable

Deklarationen und Definitionen

```
1 int a;           // def. statische Variable, externe Verl.
2 extern int a;   // dekl. statische Variable, externe Verl.
3 static int a;   // def. statische Variable, interne Verl.
4
5 int foo(void);  // dekl. Funktion, externe Verl.
6 extern int foo(void); // dito
7 static int foo(void); // dekl. Funktion, interne Verl.
8
9 int foo(void) { } // def. Funktion, externe Verl.
10 extern int foo(void) {} // dito
11 static int foo(void) {} // def. Funktion, interne Verl.
```

Typqualifizierer

Typqualifizierer verändern Typen. Sie werden je nachdem, was sie qualifizieren, an verschiedene Stellen geschrieben.

`const` Variable/Objekt darf ggf. nach Initialisierung nicht beschrieben werden

`restrict` auf Objekt wird nur durch diesen Zeiger zugegriffen

`volatile` der Compiler darf Zugriffe auf das Objekt nicht optimieren

`_Atomic` Variable oder Objekt ist atomar

Typqualifizierer

```
1 const int x;           // Konstanter int
2 const int *x;         // Zeiger auf konstanten int
3 int const *x;         // dito
4 int *const x;         // Konstanter Zeiger auf int
5 int *restrict x;      // restrict-Zeiger auf int
6 int x[restrict 12];   // restrict-Array von 12 int
```