Operatoren in Variablen-/Typdeklarationen

Es gelten die gleichen Vorrangregeln:

```
int (*table[10])(double);
```

`table` ist ein 10-elementiges Feld von Zeigern auf Funktionen, die ein `double`-Argument erwarten und `int` zurückliefern.

Slide 33

```
int (*f)(double)[10];
```

`f` ist ein Zeiger auf eine Funktion, die ein `double`-Argument erwartet und einen Zeiger auf ein 10-elementiges `int`-Feld zurückliefert. Da Feldgrenzen nicht überprüft werden, ist folgende Deklaration identisch:

```
int * (*f)(double);
```

⇒ „von innen nach außen“

8 Anweisungen

8.1 Leere Anweisung

Slide 34

```
;
```

8.2 Einfache Anweisung

`ausdruck` `;`

8.3 Blöcke

Slide 35

```
{
[deklarationen]

anweisungen...
}
```

8.4 Auswahl

Slide 36

```
if (ausdruck) anweisung
[ else anweisung ]

switch (ausdruck)
{
case int-konstante: anweisungen... [ break; ]
...
[ default: anweisungen... [ break; ] ]
}
```

Achtung: Standardmäßig fall-through!

Siehe auch: `?:`-Operator

8.5 Schleifen

Slide 37

```
while (ausdruck) anweisung
```

```
for (init-ausdruck ; bedingungsausdruck ; iter-ausdruck)
anweisung
```

```
do anweisung
while (ausdruck) ;
```

Schleifenabbruch mit `break`

Nächste Iteration mit `continue`

8.6 Go to

Slide 38

Nur innerhalb einer Funktion. Markendefinition mit:

```
marke :
```

```
goto marke ;
```

Beispiel

Slide 39

```
int i;

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

  if (i > 78) break;

  printf("%d\n", i);
}
```

9 Funktionen

Definition:

rückgabetyp name (*parametervar-dekl*, ...)
block

Deklaration extern definierter Funktionen:

Slide 40

[`extern`] *rückgabetyp name* (*parametervar-dekl*, ...) ;

Typ `void` für „keine Parameter“ oder „kein Rückgabewert“.

Rückkehr aus Funktion mit

`return` [*rückgabewert*] ;

(kann mehrmals auftreten; kann in `void`-Funktionen weggelassen werden)

9.1 Parameterübergabe

Parameter werden auf dem Stack übergeben.

`main()` erhält drei Parameter, kann diese aber ignorieren:

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

Slide 41

`argv` Feld von Zeigern auf Parameter-Strings;

`argv[0]` enthält Name des aufgerufenen Programms

`argv[n]` enthält Parameter n; letztes Feldelement enthält Nullzeiger (`NULL`)

`argc` Anzahl gültiger Elemente von `argv[]`

`argc == 1` wenn kein Parameter übergeben wurde

`envp` Feld von Zeigern auf Umgebungsvariablen, z.B.

`PATH=/usr/bin:/bin:.`

letztes Feldelement enthält Nullzeiger (`NULL`)

Slide 42

Nur „call by value", kein „call by reference"!

Um Referenzparameter (Rückgabeparameter) zu übergeben, müssen die Referenzen von Hand mit dem Referenzoperator „`&`" erzeugt werden (= Zeiger-Bildung).

Beispiel

Slide 43

```
#include <stdio.h>

int main(void)
{
  int v1, v2;

  scanf("%d %d", &v1, &v2);

  printf("zahl1 = %d\nzahl2 = %d\n", v1, v2);

  return 0;
}
```

9.2 Automatische Variablen

werden auf dem Stack angelegt (kein Aufruf von Laufzeitsystem oder Bibliotheksfunktionen! ⇒ schnell)

Der Stack ist „ausreichend“ groß, aber begrenzt (implementationsabhängig und meist veränderbar):

Slide 44

```
% limit stacksize
stacksize       8192 kbytes
```

```
$ ulimit -s
8192
```

Unter Unix wird die Grenze grundsätzlich überwacht; Überschreitungen führen zu „Segmentation fault“.

⇒ keine großen Felder auf den Stack anlegen, sondern besser als `static`-Variablen.

10 C-Präprozessor

10.1 Präprozessor-Anweisungen

Slide 45

`#include "`*datei*`"`	Quelldatei einbinden (aktuelles Verzeichnis)
`#include <`*datei*`>`	Quelldatei einbinden (Systemverzeichnis)
`#define` *NAME*	Präprozessor-Symbol definieren
`#define` *NAME text...*	Präprozessor-Konstante definieren
`#define` *NAME*`(`*parameter...*`) \` *text...*	Präprozessor-Funktionsmakro definieren
`#undef` *NAME*	Präprozessor-Symbol löschen

Slide 46

`#if` *bedingung*	bedingte Compilierung
`#ifdef` *NAME*	compiliere Abschnitt, wenn Präprozessorsymbol *NAME* definiert
`#ifndef` *NAME*	compiliere Abschnitt, wenn Präprozessorsymbol *NAME* *nicht* definiert
`#elif` *bedingung*	andere Bedingung, wenn `#if` fehlschlägt
`#else`	compiliere Abschnitt, wenn `#if` fehlschlägt
`#endif`	Ende eines bedingt compilierten Abschnitts
`#warn` *text...*	Compilerwarnung ausgeben
`#error` *text...*	Compilierung mit Fehler abbrechen

`#pragma ...`	implementationsabhängige Compilerfunktion

Slide 47

10.2 Präprozessor-Bedingungsausdrücke

Verwendet werden können konstante Integer-Ausdrücke, alle logischen Operatoren und die Konstruktion `defined(`*`NAME`*`)`.

Beispiel

Slide 48

```
#define DEBUG
#define VERSION 15
#define PATCHLEVEL "BETA"

#ifdef DEBUG
# if VERSION > 20 && !defined(PATCHLEVEL)
#  warn Achtung Achtung!
# endif
#else
# error DEBUG nicht definiert!
#endif

printf(PATCHLEVEL);
```

10.3 Funktionen-Makros

`#define` *NAME(parameter...) text...*

Beim Aufruf des Makros werden die Parameter in den Text eingesetzt und der Text anstelle des Makroaufrufs eingesetzt:

```
#define QUADRAT(a) (a*a)
printf("Quadrat von 10 = %d\n", QUADRAT(10));
```

Slide 49

Für Makro-Argumente gibt es spezielle Makro-Operatoren:

Operator	Bedeutung	Beispiel
`#`	bilde String	`#define STR(x) #x`
`##`	lexikalische Bindung	`#define CONCAT(a,b) a##b`

Beispiel

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] =
{
  COMMAND (quit),
  COMMAND (help)
};
```

Slide 50

wird zu

```
struct command commands[] =
{
  { "quit", quit_command },
  { "help", help_command }
};
```

11 Häufige Anfängerfehler in C

Pascal-/Modula-Programmierer:

- `==` statt `=` in Vergleichen
- C unterscheidet zwischen Groß- und Kleinschreibung

Slide 51

- Rückgabeparameter müssen als Zeiger übergeben werden: `&`
- auch parameterlose Funktionen werden mit `()` aufgerufen; Funktionsname ohne `()` liefert Zeiger auf Funktion
- jede Anweisung endet mit „`;`“, auch vor Blockende „}“ und vor `else`
- `[ ][ ]` statt `[ , ]` für mehrdimensionale Felder

Allgemein:

- Pointer-Initialisierung nicht vergessen!
- Arrays werden stets bei 0 beginnend indiziert
- Werden Makro-Parameter mehr als einmal verwendet, wird der übergebene Ausdruck auch mehr als einmal ausgewertet!

Slide 52

```
#define QUADRAT(a) (a*a)
y = QUADRAT(x++);
```

- Makroparameter und Makroausdrücke als ganzes sollten geklammert werden:

```
#define PRODUKT(a,b) a * b
y = - PRODUKT(x1, x2 + x3);
```

12 Standardbibliothek

Schnittstelle zum Betriebssystem

Dienste für:

Slide 53

`string.h` Stringfunktionen („`man 3 string`“)

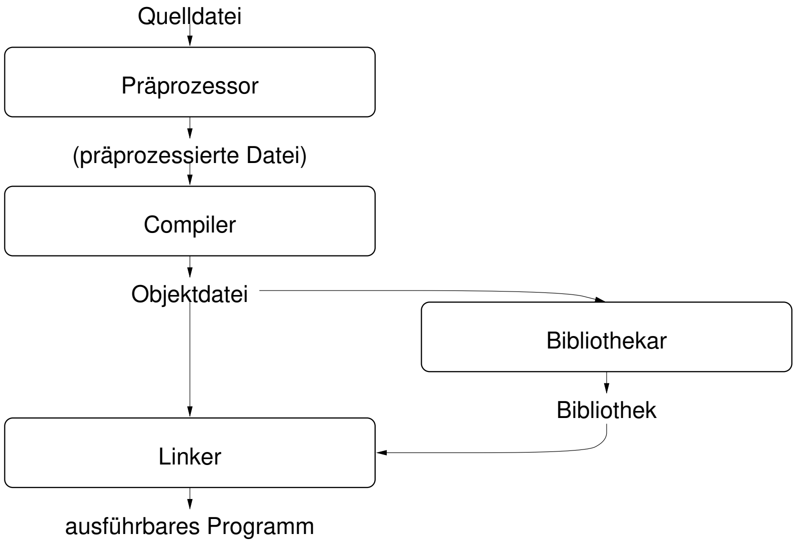
`stdio.h` Ein-/Ausgabe („`stdio`“)

`stdlib.h` dynamische Speicherverwaltung („`malloc`“), Typumwandlung („`strtol`“), Zufallszahlen („`rand`“), Sortieren („`qsort`“), Suche („`bsearch`“), Ausführung externer Programme („`system`“)

`stdarg.h` Unterstützung von Funktionen mit variabler Parameteranzahl („`stdarg`“)

13 Module

Slide 54



Slide 55

Erzeugen eines ausführbaren Programms: (Compilation + Linken in einem Schritt)

```
gcc -o foo foo.c
```

Erzeugen einer Objektdatei:

```
gcc -c -o bar.o bar.c
```

Erzeugen einer Bibliotek aus mehreren Objekten:

```
ar -r libfrob.a gnar.o fnurg.o
ranlib libfrob.a
```

Linken von Objektdateien und Bibliotheken:

```
gcc -o program bar.o baz.o -lfrob
```

Slide 56

Datei `nachfolger.h`:

```
#ifndef NACHFOLGER_H
#define NACHFOLGER_H

int nachfolger(int x);

#endif /* NACHFOLGER_H */
```

Datei `nachfolger.c`:

```
#include "nachfolger.h"

int nachfolger(int x)
{
  return x + 1;
}
```

Datei `main.c`:

```
#include "nachfolger.h"

int main(void)
{
  int i;

  for (i = 0; i < 100; i++)
    {
      printf("Nachfolger(%d)" = %d\n",
             i, nachfolger(i));
    }

  return 0;
}
```

Übersetzung des Programms:

Slide 57

```
gcc -c -o nachfolger.o nachfolger.c
gcc -c -o main.o main.c
gcc -o main main.o nachfolger.o
```

14 Fortgeschrittene Programmierung

14.1 Programmieren mit Zeigern

Zeiger-Arithmetik

Slide 58

```
long *zeiger, feld[20];
zeiger = feld;
```

Feld und Zeiger können nun äquivalent verwendet werden: `feld[1]` entspricht `zeiger[1]`.

Zeiger können in Einheiten der Größe ihres Typs inkrementiert werden:

```
zeiger++;
```

Zeiger zeigt nun auf `feld[1]` (d.h. `zeiger == &feld[1]`).