Slide 1

Einführung in die Programmiersprache C
für Hörer der Betriebssysteme-Vorlesung
URL: `http://os.inf.tu-dresden.de/~hohmuth/ckurs/`

Michael Hohmuth <*hohmuth@inf.tu-dresden.de*>

19. Oktober 2000

Slide 2

1 Überblick über diesen Kurs

3 Doppelstunden:

1. Einführung: Warum C? — Hello World! — Aufruf des Compilers unter Unix — Online-Hilfe unter Unix — Typen — Variablen — Ausdrücke
2. Steueranweisungen — Präprozessor/Makros — Standard-Bibliothek — Module, Bibliotheken, Linker
3. Programmieren mit Zeigern — UNIX-Systemrufe — Objektorientiertes Programmieren in C

Übungsaufgaben: zu 1. und 2. jeweils $\frac{1}{2}$h

Slide 3

Voraussetzungen (wird *nicht* beprochen):

- Algorithmus; Programmierung allgemein
- Einloggen, Ausloggen
- Unix-Editor

2 Einführung

Slide 4

Warum C für Betriebssysteme?

- portabel — Compiler für alle Plattformen verfügbar
- ausgereift — lange Historie als Implementationssprache für Betriebssysteme; Stärken/Schwächen wohlbekannt
- minimalistisch, hardwarenah und effizient — Programmierer verliert nicht den Überblick über Umsetzung in Maschinencode; kein Laufzeitsystem erforderlich
- Syntax unterstützt „faule“ Programmierer (wenig zu tippen)

Schwächen von C

Slide 5

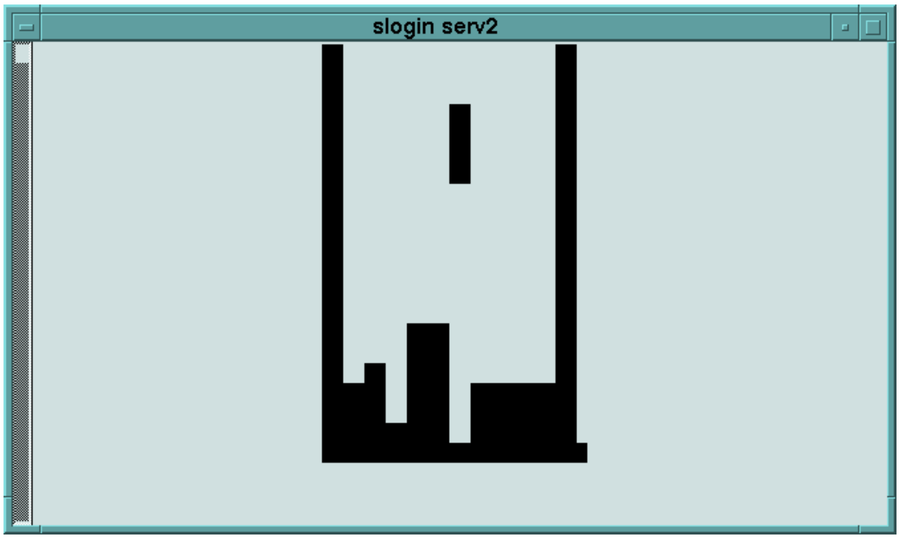
- einige Fallstricke für Anfänger
- erfordert diszipliniertes Programmieren, damit der Compiler Fehler erkennen kann und damit der Code für Fremde lesbar wird
- nur begrenzte Unterstützung für Modularisierung — Programmierer großer Systeme müssen eigene Konventionen erfinden

„C combines the speed of assembler with the readability of assembler.“

Slide 6

```
long h[4];t(){h[3]-=h[3]/3000;setitimer(0,h,0);}c,d,l,v[]={(int)t,0,2},w,s,I,K
=0,i=276,j,k,q[276],Q[276],*n=q,*m,x=17,f[]={7,-13,-12,1,8,-11,-12,-1,9,-1,1,
12,3,-13,-12,-1,12,-1,11,1,15,-1,13,1,18,-1,1,2,0,-12,-1,11,1,-12,1,13,10,-12,
1,12,11,-12,-1,1,2,-12,-1,12,13,-12,12,13,14,-11,-1,1,4,-13,-12,12,16,-11,-12,
12,17,-13,1,-1,5,-12,12,11,6,-12,12,24};u(){for(i=11;++i<264;)if((k=q[i])-Q[i]
){Q[i]=k;if(i-++I||i%12<1)printf("\033[%d;%dH",(I=i)/12,i%12*2+28);printf(
"\033[%dm  "+(K-k?0:5),k);K=k;}Q[263]=c=getchar();}G(b){for(i=4;i--;)if(q[i?b+
n[i]:b])return 0;return 1;}g(b){for(i=4;i--;q[i?x+n[i]:x]=b);}main(C,V,a)char*
*V,*a;{h[3]=1000000/(l=C>1?atoi(V[1]):2);for(a=C>2?V[2]:"als pq";i;i--)*n++=i<
25||i%12<2?7:0;srand(getpid());system("stty cbreak -echo stop u");sigvec(14,v,
0);t();puts("\033[H\033[J");for(n=f+rand()%7*4;;g(7),u(),g(0)){if(c<0){if(G(x+
12))x+=12;else{g(7);++w;for(j=0;j<252;j=12*(j/12+1))for(;q[++j];)if(j%12==10){
for(;j%12;q[j--]=0);u();for(;--j;q[j+12]=q[j]);u();}n=f+rand()%7*4;G(x=17)||(c
=a[5]);}}if(c==*a)G(--x)||++x;if(c==a[1])n=f+4**(m=n),G(x)||(n=m);if(c==a[2])G
(++x)||--x;if(c==a[3])for(;G(x+12);++w)x+=12;if(c==a[4]||c==a[5]){s=sigblock(
8192);printf("\033[H\033[J\033[0m%d\n",w);if(c==a[5])break;for(j=264;j--;Q[j]=
0);while(getchar()-a[4]);puts("\033[H\033[J\033[7m");sigsetmask(s);}}d=popen(
"stty -cbreak echo stop \023;sort -mnr -o HI - HI;cat HI","w");fprintf(d,
"%4d from level %1d by %s\n",w,l,getlogin());pclose(d);}
```

Slide 7



Slide 8

Warum nicht C++ für Betriebssysteme?

- C++ enthält alle Fallstricke von C plus weitere Fallstricke ⇒ C++ schwerer zu erlernen
- schwerer zu überschauen ⇒ durch Einsatz falscher Mittel werden die compilierten Programme oft dick und lahm
- C++ erfordert umfangreiches Laufzeitsystem, das ins Betriebssystem integriert werden müßte

Warum nicht Modula-2 oder Modula-3 für Betriebssysteme?

zwar leichter zu erlernen, Modulunterstützung und weniger Fallen, aber

Slide 9

- nicht so verbreitet
- nicht so portabel
- M-2 nicht ausdrucksstark genug

Geschichte der Programmiersprache C

1971–1973 C entsteht als Implementationssprache für UNIX als Nachfolger für B, dessen Vorgänger BCPL war.

1973–1982 C und UNIX tragen jeweils zum Erfolg des anderen bei und werden weltweit verwendet

Slide 10

1983 Kernighan/Ritchie veröffentlichen „The C Programming Language“, die Definition und Referenz für „K&R C“, dem klassischen Dialekt.

1983–1989 Standardisierungsbestrebungen führen zu „ANSI C“, einem modernisierten Dialekt

1990 ANSI C mit kleinen Veränderungen wird durch ISO zum internationalen Standard

Slide 11 $\Rightarrow$ 2 Dialekte. Wir beschäftigen uns ausschließlich mit ANSI C.

3 Überblick: Hello, World!

Slide 12

```
#include <stdio.h>             /* Präprozessoranweisung: lade
                                  Standard-Deklarationen
                                  für I/O */
int main(void)                 /* Beginn Definition Funktion
                                  main() */
{
  printf("Hello, World!\n");   /* rufe Funktion printf() mit
                                  Stringkonstante als Parameter
                                  auf */
  return 0;                    /* Liefere Ergebis 0 (= OK) an
                                  Vaterprozeß zurück */
}
```

Erkennbare Eigenschaften:

- Präprozessor kann Include-Dateien mit Deklarationen einbinden
- Jedes C-Programm enthält eine Funktion `main()`, die als Hauptprogramm beim Programmstart angesprungen wird.

  Ergebnis von `main()` wird an Vaterprozeß zurückgeliefert.

  („Funktion" synonym für „Prozedur")
- Zeichenkettenkonstanten werden in doppelte Hochkommas eingeschlossen und können Escape-Zeichen enthalten („`\n`" = Newline o. Linefeed = LF = ASCII 0xa).
- `printf()` ist kein in C eingebauter „Befehl", sondern eine Funktion in der Standard-Bibliothek, die zu allen Programmen hinzugebunden wird.
- `return` ist keine Funktion (kein Aufrufoperator „`()`" notwendig), sondern eine C-Anweisung.
- Kommentare eingeschlossen in `/* ... */`.

Aufruf des GNU C-Compilers

Warum GNU C Compiler?

- hohe Verfügbarkeit
- unterstützt Anfänger durch viele Warnungen

Slide 13 Aufruf im Terminal, wenn obiges Programm in Datei `hello.c` vorliegt:

```
$ gcc -Wall -o hello hello.c
```

Aufruf des compilierten Programms:

```
$ ./hello
Hello, World!
$
```

Online-Hilfe unter Unix

Das Unix-Referenzmanual ist elektronisch online verfügbar.

Hilfe abrufbar z.B. mit

Slide 14

```
$ man gcc
$ man 3 printf
```

oder mit fensterorientiertem Programm XMan:

```
$ xman
```

4 Struktur eines C-Programms

Ein C-Programm enthält:

- Präprozessoranweisungen (z.B. `#include`)
- Typdefinitionen
- Funktionsdeklarationen (Prototypen): deklarieren, welche Parameter eine Funktion erwartet und welchen Typs ihr Ergebnis ist)
- Variablendeklarationen: deklarieren den Typ von Variablen
- Funktionsdefinitionen: lokale Deklarationen, Steueranweisungen und Ausdrucksberechnungen

Slide 15

Jedes C-Programm muß eine Funktion `main()` enthalten.

5 Datentypen

Variablen sind typisiert und müssen vor Gebrauch deklariert werden. I.A.:

```
int zahl;
```

Slide 16

5.1 Einfache Datentypen

Datentyp	Typschlüsselwort	Bsp. f. Konstanten

Zeichen	`char`
	`signed char`
	(`unsigned char`)
		`'a'`, `'\n'`, `15`

Slide 17

Datentyp	Typschlüsselwort	Bsp. f. Konstanten

Integer-Zahl	`int`
	`(signed int)`
	`unsigned int`
	`long int`
	`signed long int`
	`unsigned long int`
		`10`, `-15314`, `0731`, `0xdeadbeef`

Real-Zahl	`float` `double`	`1.3542e-23`, `-2345.2211`

Kein Boolean-Typ! Wahrheitswerte werden durch Integer-Variablen gespeichert: `0` = falsch, `!=0` = wahr.

5.2 Strukturierte Typen

Slide 18

Datentyp	Syntax	Beispiele f. Variablen-deklaration
Feld	*typ ID*[*dimension*]	`int feld[100]`

String	`char` *ID*[*dimension*]	`char string[100]`
		`"das ist ein String"`
		`"Zeile 1\nZeile 2"`

Zeiger	*typ* *	`int *p_int` `char *s`

Felder werden stets mit 0 beginnend indiziert.

String-Konstanten werden als `char`-Felder mit anhängter Null (0x0, `'\0'`) im Speicher abgelegt.

Slide 19

Felder und Zeiger eines Typs sind zuweisungskompatibel:

```
p_int = feld;                    /* p_int zeigt auf
                                    0. Element von feld */
s = "das ist ein String";       /* s zeigt auf 'd' */
s = string;
```

Beispiel

Slide 20

```
#include <stdio.h>

char name[] = "Zweites Programm.\n";
char *ausgabe;

int main(void)
{
  ausgabe = name;

  printf(ausgabe);

  return 0;
}
```

- Deklaration zweier globaler Variablen
- Feld `name` wird mit String initialisiert; da dem Compiler die Größe des Initializers (String-Konstante) bekannt ist, muß keine Felddimensionierung angegeben werden
- Zuweisung Feld zu Zeiger
- Aufruf `printf()` mit Zeiger-Argument

Slide 21

Datentyp	**Syntax**	**Beispiele f. Variablen-deklaration**
Aufzähl-Typ	`enum` [*ID*] { *ID* [= *val*], ... }	`enum { GELB = 0, GRUEN }`
Struktur	`struct` [*ID*] { *var.-dekl.*; ... }	`struct foo { int a; char *s; }`
Union	`union` [*ID*] { *var.-dekl.*; ... }	`union bar { int a; char *s; }`

5.3 Benutzerdefinierte Typen

`typedef` *var-dekl* ;

Beispiele:

Slide 22

```
typedef int feldtyp[100];        /* Typdeklaration */
feldtyp feld;                    /* Variablendeklaration */

typedef char *string;
string name;
```

6 Variablendeklaration

globale Variablen: werden außerhalb von Funktionen deklariert

lokale Variablen: können zu Beginn eines jeden Blocks ($\{\ldots\}$) deklariert werden

Parametervariablen: aktuelle Parameter einer Funktion

Slide 23

Variablen kann durch eine Speicherklasse zugeordnet werden, die Gültigkeitsbereich und Lebensdauer beeinflussen:

Slide 24

<table>
<tr><th>Schlüsselwort</th><th>globale Variablen</th><th>lokale Variablen</th><th>Parametervariablen</th></tr>
<tr><td>(ohne)</td><td>programm-globale Gültigkeit; „common“</td><td>= <code>auto</code></td><td>lebt bis Verlassen der Funktion</td></tr>
<tr><td><code>auto</code></td><td>—</td><td>lebt bis Verlassen des Blocks</td><td>—</td></tr>
<tr><td><code>static</code></td><td>nur in diesem Modul gültig</td><td>Wert bleibt nach Verlassen bestehen</td><td>—</td></tr>
<tr><td><code>extern</code></td><td colspan="2">Variable in anderem Modul programm-global definiert</td><td>—</td></tr>
<tr><td><code>register</code></td><td>—</td><td colspan="2">wenn möglich, in Registern ablegen</td></tr>
</table>

Beispiel

Slide 25

```
static int wie_oft = 0;

void init(void)
{
  static int wurde_initialisiert = 0;

  if (! wurde_initialisiert)
    {
      /* führe Initialisierung durch... */
      wie_oft += 1;

      wurde_initialisiert = 1;
    }
}
```

7 Ausdrücke und Operatoren

Slide 26

Außer Deklarations- und Steueranweisungen enthalten C-Programme nur noch Ausdrucksberechnungen. Alle Ausdrucksanweisungen sind mit einem Semikolon abgeschlossen. Beispiele:

```
variable1;
a = b + c;
funktion();
```

Operatoren in der Reihenfolge ihrer Ausführung (Operatoren einer Gruppe haben die gleiche Priorität und werden von rechts nach links ausgeführt):

Slide 27

Operator	**Bedeutung**	**Beispiel**
Klammernoperator		
`( )`		`(a + b) * c`
Aufruf- und Selektionsoperatoren		
`()`	Funktionsaufruf	`sin(x)`
`[]`	Feldindizierung	`stunde[0]`
`.`	Strukturauswahl	`person.name`
`->`	Strukturauswahl (Zeiger) (`a->b` entspricht `(*a).b`)	`person->name`

Slide 28

Operator	**Bedeutung**	**Beispiel**
Unäre Operatoren		
`!`	logisches NICHT	`! initialized`
`~`	bitweises NICHT	`~0xffff`
`++`	Inkrementieren	`a++, ++b`
`--`	Dekrementieren	`a--, --b`
`+`	positives Vorzeichen	`+a`
`-`	negatives Vorzeichen	`-b`
`*`	Zeiger-Dereferenzierung	`*zeiger`
`&`	Adreßoperator	`&var`
`sizeof`	Größe	`sizeof int`
`(`*typ*`)`	Typumwandlung	`(long *) 0x1000`

Slide 29

Operator	Bedeutung	Beispiel
Binäre Punkt-Operatoren		
*	Multiplikation	a * b
/	Division	a / b
%	Divisionsrest (modulo)	a % b
Binäre Strich-Operatoren		
+	Addition	a + b
-	Subtraktion	a - b
Bitschiebeoperationen		
<<	bitweises Schieben links	a << 2
>>	bitweises Schieben rechts	a >> 2

Slide 30

Operator	Bedeutung	Beispiel
Relationale Operationen		
<	kleiner als	a < b
>	größer als	a > b
<=		
>=		
==	Gleichheit	a == b
!=	Ungleichheit	a != b
Bitoperationen		
&	bitweises UND	a & b
^	bitweises EXKLUSIV-ODER	a ^ b

Slide 31

Operator	**Bedeutung**	**Beispiel**
`|`	bitweises ODER	`a | b`
Logische Operationen		
`&&`	logisches UND	`this && that`
`||`	logisches ODER	`this || that`
Bedingungsausdruck		
`? :`		`(a > b) ? a : b`
Zuweisungsoperatoren		
`=`	einfache Zuweisung	`a = b`
`+=`	`a += b` entspricht `a = a + b`	`a += b`
`-=`		

Slide 32

Operator	**Bedeutung**	**Beispiel**
`*=`		
`/=`		
`%=`		
`&=`		
`^=`		
`|=`		
`<<=`		
`>>=`		
Kommaoperator		
`,`	erst a, dann b	`a, b`

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

Slide 40

Definition:

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

Deklaration extern definierter Funktionen:

[`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`*`)`.

Slide 48

Beispiel

```
#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:

Slide 49

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

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

Slide 50

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

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

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:

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

Slide 53

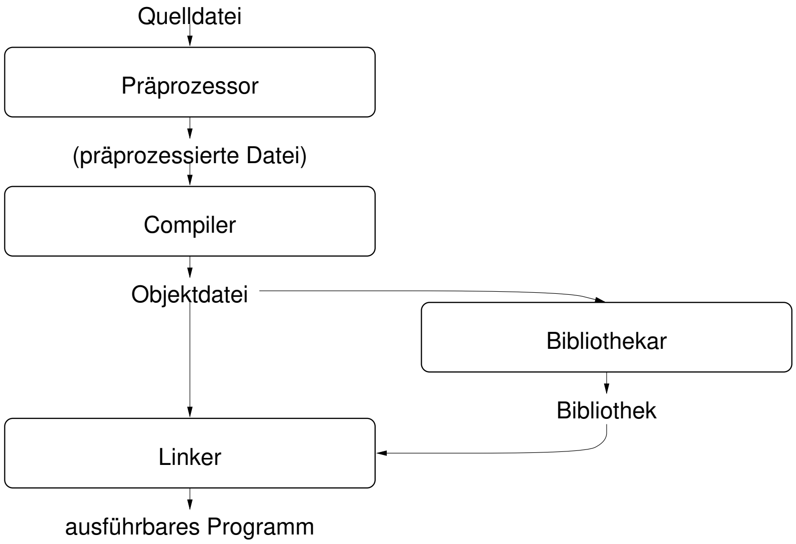
`stdio.h` Ein-/Ausgabe („`stdio`")

`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]`).

Feld-Addressierung	Zeiger-Adressierung
`feld[0]`	`*feld`
`&feld[1]`	`feld + 1`
`feld[1]`	`*(feld + 1)`

Iterieren mit Zeigern benötigt keine Schleifenvariable:

Slide 59

```
int main(int argc, char *argv[], char *envp[])
{
  while (*envp)
    {
      printf("%s\n", *envp++);
    }
  return 0;
}
```

14.2 Dynamische Speicherverwaltung

Vier Bibliotheksfunktionen verwalten den Heap:

```
#include <stdlib.h>
```

Slide 60

`void *malloc(size_t size)`

fordert `size` Bytes Hauptspeicher an.

`void *calloc(size_t nmemb, size_t size)`

fordert null-initialisierten Speicher für `nmemb` Feldelemente der Größe `size` an.

`void free(void *ptr)`

gibt mit `malloc()` oder `calloc()` angeforderten Speicherblock an Adresse `ptr` wieder frei.

Slide 61

`void *realloc(void *ptr, size_t size)`

verändert die Größe des mit `malloc()` oder `calloc()` angeforderten Speicherblocks an Adresse `ptr` auf `size` Bytes und liefert die Adresse des neuen (evtl. verschobenen) Blocks zurück. Es ist sowohl ein Verkleinern als auch ein Vergrößern des Speicherblocks möglich.

Slide 62

```
#include <stdlib.h>

struct elem
{
  char name[100];
  int zensur;
  struct elem *next, *prev;
};
typedef struct elem *elem_p;

struct list
{
  elem_p first, last;
};
typedef struct list *list_p;

void list_init(list_p l)
{
  l->first = l->last = NULL;
}
```

```
void list_insert(list_p l, elem_p e)
{
  e->next = l->first;
  e->prev = NULL;

  if (l->first) l->first->prev = e;
  else l->last = e;

  l->first = e;
}

void list_delete(list_p l, elem_p e)
{
  if (e->prev) e->prev->next = e->next;
  else l->first = e->next;

  if (e->next) e->next->prev = e->prev;
  else l->last = e->prev;
}
```

```
elem_p schueler;
list_p klasse9a;

klasse9a = (list_p) malloc(sizeof(struct list));
list_init(klasse9a);

schueler = (elem_p) malloc(sizeof(struct elem));
strcpy(schueler->name, "Heinz");
schueler->note = 4;

list_insert(klasse9a, schueler);
```

Slide 63

14.3 Unix-Systemrufe

Zugriff auf Unix-Kernobjekte:

`#include <unistd.h>`

Anhaltspunkte:

Slide 64

- Dateien und Verzeichnisse: `open()`, `opendir()`, `close()`, `closedir()`, `read()`, `write()`, `lseek()`, `unlink()`, `rmdir()` („`man 2 open`“ usw.)
- Pipes: `pipe()`
- Signale: `kill()`, `signal()`, `alarm()`, `pause()`
- Nutzer und Gruppen: `chown()`, `fchown()`, `getuid()`, `getgid()`

Slide 65

- Aktuelles Verzeichnis: `chdir()`, `fchdir()`, `getcwd()`
- Dateideskriptoren: `dup()`, `dup2()`, `select()`
- Prozesse: `fork()`, `execve()`, `wait()`, `waitpid()`

Beispiel

Slide 66

```
pid_t pid;
int status;

if ((pid = fork()) == 0)
  {
    /* Kindprozeß */
    execl("/bin/sh", "sh", "-c", "ls -l | more");
    /* kehrt nie zurück */
  }
/* Elternprozeß */
if (waitpid(pid, &status, 0) == -1)
  printf("Error!\n");
else
  printf("Subprozeß-Rückgabewert: %d\n", WEXITSTATUS(status));
```

14.4 Objektorientiertes Programmieren

Ausgehend vom OO-Paradigma: Objekt = Daten + Methoden

Slide 67

```
struct figur {
  int x, y;

  int (*draw)(struct figur *f);
}
```

Slide 68

```
struct figur *new_circle(void)
{
  struct figur *f = malloc(sizeof(struct figur));
  if (f)
    {
      f->draw = circle_draw;
      f->x = f->y = 0;
    }
  return f;
}

struct figur *k;
k = new_circle();
k->x = 15;  k->y = 76;
k->draw(k);                /* entspricht (*k->draw)(k); */
```