C - Programmierung

Übung zur Betriebssystemarchitektur
WS 2004/05

Dipl.-Inf. Bernhard Rabe
Betriebssysteme & Middleware

Literatur

- The C Programming Language, 2nd Edition, Brain Kernighan & Dennis Ritchie, 1988 Prentice Hall
- http://www.physik.uni-regensburg.de/studium/edverg/ckurs/script, 2004

Die Sprache C

C-Sprache

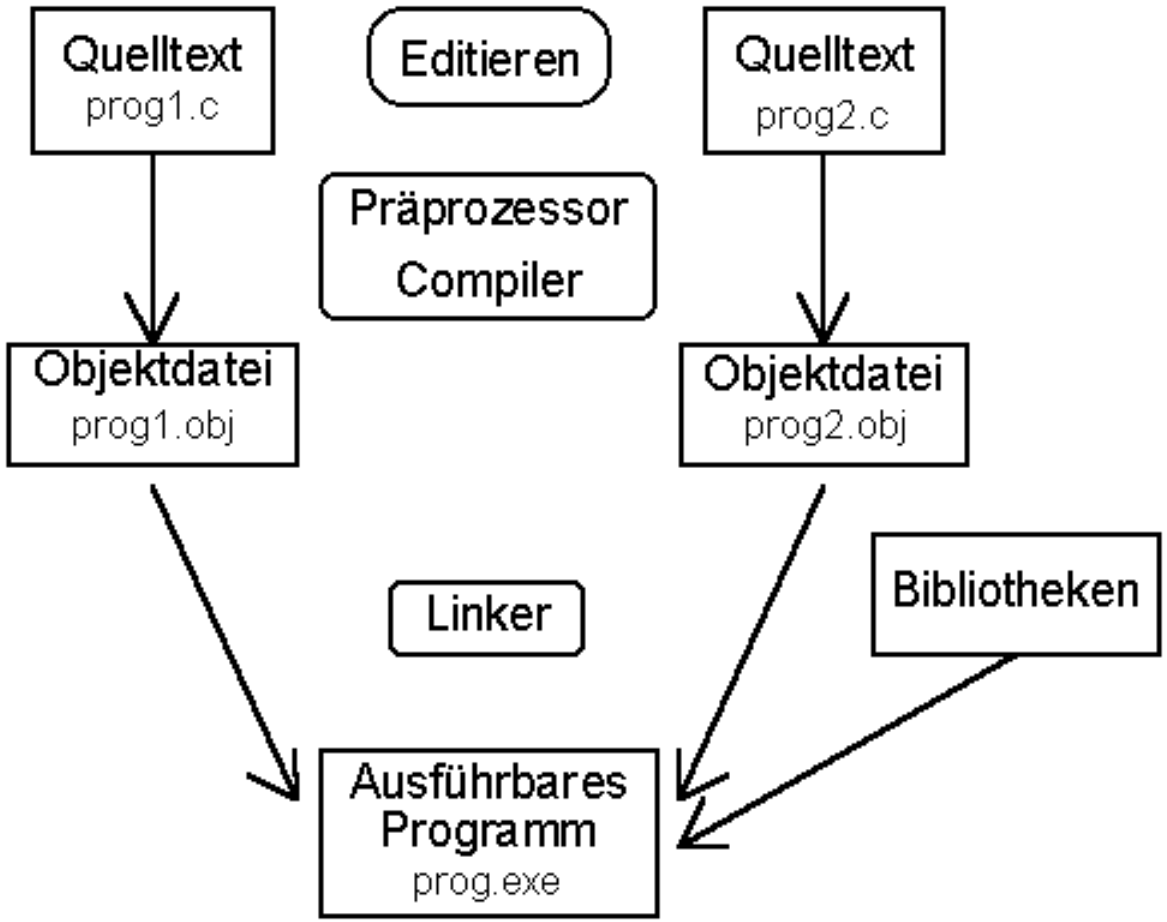
Typen, Operatoren,
Kontrollstrukturen,
Funktionen

Bibliotheken

Standard C,
Mathematik,
Netzwerk

- “General Purpose” Programmiersprache
- für UNIX auf PDP-11 von Dennis Ritchie

Entwicklungsprozess



Hello, World

```
#include <stdio.h>

main()
{
    printf("Hello, World\n");

}   /* Kommentar*/
```

Präprozessoranweisung
zur Benutzung von Bibliotheks-
funktionen

Hello, World

```
#include <stdio.h>

main()
{
    printf("Hello, World\n");

}   /* Kommentar*/
```

Funktionsname:
main ist Standard-Eintrittspunkt

optionale Übergabe-paramter in den Klammern

Hello, World

```
#include <stdio.h>

main()
{
    printf("Hello, World\n");

}   /* Kommentar*/
```

Begrenzung von Blöcken
mit geschweiften Klammern

Hello, World

```
#include <stdio.h>

main()
{
    printf("Hello, World\n");

}   /* Kommentar*/
```

Funktionaufruf
man 3 printf

Hello, World

```
#include <stdio.h>

 main()
{
     printf("Hello, World\n");

}    /* Kommentar*/
```

Begrenzung von Anweisungen mit einem Semikolon

Kompilieren

- <Compiler> <C-Datei> -o <Ziel>
  - `cc hello.c -o hello`
  - `cl hello.c -o hello.exe (cl hello.c)`
- Make
  - <Make> <Ziel>
  - nutzt interne Regeln (make –p, make –n Ziel)
  - `make hello`
  - `nmake hello.exe`

Demo

C-Datentypen*

- **char** 8 bit, –128..127
- **int** 32 bit 2^{31}-1..2^{31}
- **short** 16 bit 2^{15}..2^{15}-1
- **float** 32 bit (7 Stellen Fließkomma)
- **double** 64 bit (19 Stellen Fließkomma)
- **long** 32 bit
- **void** “leer”
- **const** <Typ> nur lesbar
- *unsigned* vorzeichenlos, *signed* vorzeichenbehafted
- Größen sind Plattformabhängig: limits.h *32Bit x86
- **`sizeof`**`(Datentyp)` liefert Größe in Byte

Beispiel 2

```
1.  void main()
2.  {
3.      int celsius, fahr;
4.      int lower, upper, step;
5.      lower=0;
6.      upper=300;
7.      step=20;
8.      fahr=lower;
9.      while(fahr <= upper)
10.     {
11.         celsius=5 * (fahr - 32) / 9; /*  Celsius = 5/9*(Fahrenheit-32)*/
12.         printf("%d\t%d\n", fahr, celsius);
13.         fahr=fahr+step;
14.     }
15. }
```

Demo

Beispiel 2 v2

```
1.    void main()
2.    {
3.        float celsius, fahr;
4.        int lower, upper, step;
5.        lower=0;
6.        upper=300;
7.        step=20;
8.        fahr=lower;
9.        for( fahr=0 ; fahr <= upper ; fahr = fahr + step )
10.       {
11.           celsius=(5.0/9.0) * (fahr - 32); //    Celsius = 5/9*(Fahrenheit-32)
12.           printf("%3.0f\t%6.1f\n", fahr, celsius);
13.
14.       }
15.   }
```

Demo

Präprozessor

- Anweisungen beginnen mit #
- Ersetzung vor dem Compiler-Lauf !
- Bedingte Kompilierung möglich
  - `#if, #ifdef, #else, #endif, ...`
- Symbolische Konstanten
  - `#define <NAME> <Wert>`

Demo

Arrays

- Deklaration durch [] an einer Variable
- Zählung beginnt bei 0
  - `int a[10];  /*Array von 10 int's*/`
  - `int b[3]={1,2,3}; /*Initialisierung*/`
  - Zugriff: `a[0...9]=..;`
- Mehrdimensionale Arrays:
  - `int b[10][10];`
  - `b[2][3]` ⇔ `b[0][2*10+3]`
  - sind intern eindimensional
- Es erfolgt keine Überprüfung auf gültigen Speicher!

char/Strings

- **char** c='A', d=65, e=0x41, f=\101;

```
printf("%c\n",c); /*A*/
printf("%x\n",c); /*0x41*/
```

h	e	l	l	o	\n	\0

- **char** ca[10], da[]="hello\n",

```
ca[0]='h';ca[1]='e';ca[2]='l';...
printf("%s",ca); /*hello*/
printf("%s",da); /*hello*/
```

Zusammengesetzte Datentypen

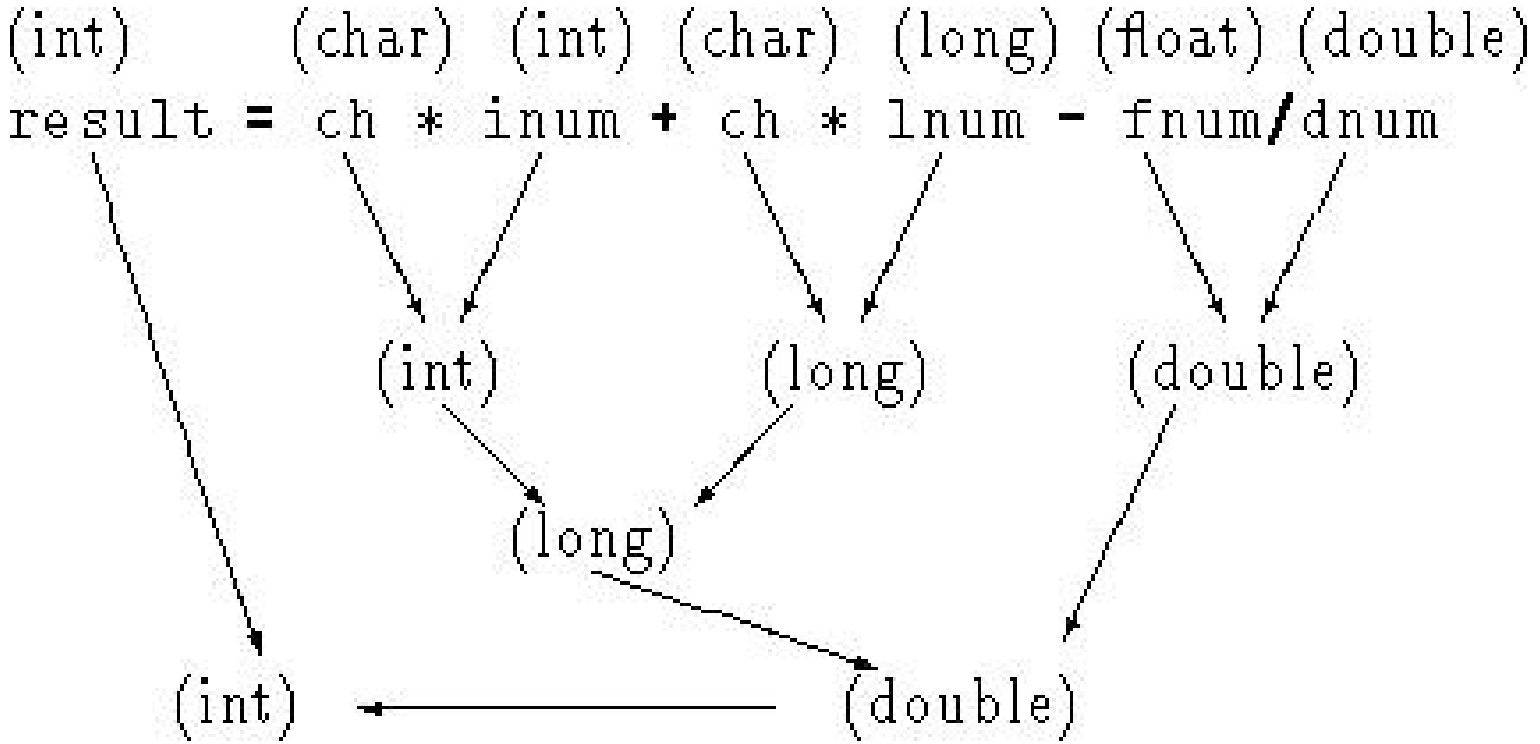
```
struct _item{                    /*Strukturname*/
    int ival;                    /*int Element*/
    double dval;      /*double Elelment*/
    char sval[10];    /*char Array*/
} item;               /*deklarierte Variable*/

struct _item a,
 b={1,3.141,"Hello"}; /*Initialisierung*/
```

Operatoren

- = Zuweisung a=1;
- +, −, *, /
- ++ Inkrement −−Dekrement
  - Post-Inkrement a++, Pre-Inkrement ++a
- ==, !=, <, >, >=, <= Vergleichsoperatoren
- && logisches Und, || logischers Oder
- +=, −=, *=, /= Operation und Zuweisung
  - a+=12; <=> a=a+12;

Typumwandlung

- implizit:



- explizit: (Datentyp)Ausdruck
  - `int celsius=((float)5/9)*(fahr-32)`

Funktionen

- <Rückgabetyp> name (<parameter>)

  {

  return Wert; /*wenn nicht void*/

  }

- "Call by Value" Übergabe

```
int add(int a, int b)
{
  return a+b;
}
```

Funktionen 2

- Funktionen müssen vor Nutzung bekannt sein

```
int add(int a, int b){..} /*Definition*/
main(){ int=add(1,3); }
```

- oder

```
int add(int , int ); /*Deklaration*/
main(){ int c=add(1,3); }
int add(int a, int b){...} /*Definition*/
```

Kontrollstrukturen

- **`if`**(<Bedingung>) <Anweisung> [**`else`** <Anweisung>]
- Bedingte Zuweisung: var=(<Bedingung>)? val1:val2
- **`switch`**(<Ausdruck>) {

  `case` <konst. Ausdruck>:

  Anweisungen

  `case` <konst. Ausdruck>:

  Anweisungen

  `default`:

  Anweisungen

  }

Wiederholungen

- **`for`**(<start>;<Abbruchbedingung>;<Schrittweite>) <Anweisung>
- **`while`**(<Bedingung>) Anweisung
- **`do`** Anweisungen **`while`**(<Bedingung>);

Pointer & Adressen

- Adressoperator: & liefert die Adresse einer Variablen
  - `p=&c`
  - `p` wird die (Speicher-) Adresse von `c` zugewiesen
- Inhaltsoperator: * liefert den Inhalt an einer bestimmten Adresse
  - `*p` der Wert an der Adresse auf die p zeigt

```
int x = 1, y = 2, z[10];
int *ip; /* ip ist ein Zeiger auf einen int */
ip = &x;         /* ip zeigt auf x */
y = *ip;         /* y=1 */
*ip = 2;         /* x=2 */
ip = &z[0];      /* ip zeigt auf z[0] */
```

Pointer & Adressen

- NULL Pointer zeigt auf nichts
- Zugriffsergebnisse sind Compiler/OS abhängig

```
int *ip;  /* Ziel ? */
*ip=5; /*Fehler ?*/
printf("%d\n",*ip); /* Fehler ?! */
ip=NULL; /*Ok*/
if(NULL==ip)
   printf("No target\n");
```

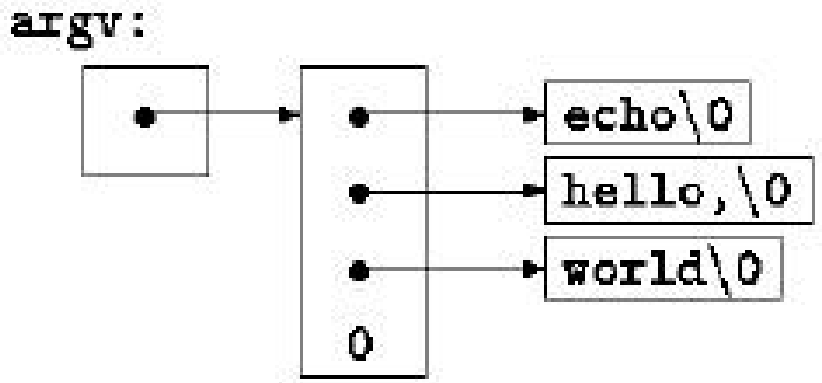
“Call by Reference”

- Mit Pointer’n ist Call by Reference Parameterübergabe möglich
- `void swap(int a, int b) //geht nicht`
-
```
void swap(int *pa, int *pb)
{     int c=*pa;
        *pa=*pb;
        *pb=c;
}
```
-
```
int a=1,b=2;
swap(&a,&b);
```

Kommandozeilen-Argumente

```
#include <stdio.h>
int main(int argc, char *argv[])
{   int i;
    for(i=0;i<argc;i++)
      printf("%s\n",argv[i]);/* *(argv+i) */
    return 0;
}
```

- > echo hello, world
- 1. Argument ist Programm



Kommandozeilen-Argumente (2)

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        while(*argv)
           printf("%d %s\n",strlen(*argv)
                   *argv++);
         return 0;
}
```

Verkette Listen

- kompilieren Linux / Windows
- strlen bzw *argv[i]
- if Zuweisungsfehler
- shellscript “” leere strings und Test, unix rechte
- einfach verkettete Listen
- doppelte
- ldd depends
- *Präprozessor falle
- Make
- binutils