SWO: C

printf - Formatting

Zeichen	Format
%d	int
%u	unsigned int
%h	short int
%o / %x	octal / hex
%p	pointer
%f	float
%e	Exponent Darstellung
%s	String
%c	char

Modifikatoren

%6d auf 6 Stellen auffüllen

%-6s Pad-Right 6 Stellen

%ld longint

**%d* variable Länge

%62f 6 Stellen insgesamt, 2 Nachkomma daher 3 Vorkommastellen

%% wird zu %

Beispiel

```c
#include <stdio.h>

int main(){
    int i = 24;
    float x = 2.781;
    char *s = "addition";
    printf("%-12s: %03d + %f = %10.2f\n",s,i,x,i+x);
    printf("%0*.*f\n", 8,3,12.34);
    return 0;
}
```

Ausgabe

```
addition    : 024 + 2.781000 =      26.78
0012.340
```

Args

Kommandozeilen Argumente werden an die Main übergeben mit zwei Werten übergeben, **argc** ist die Anzahl der übergebenen Argumente und **argv** sind die übergebenen Argumente, wobei argv[0] immer den Pfad zur Applikation beinhaltet.

Beispiel

```c
#include <stdio.h>

int main(int argc, char * argv[]){ /* argv[0] is programm name! */
    if(argc != 2){
        printf("Wrong number of arumgents!\n");
        printf("Usage: %s <args> \n", argv[0]);
        return -1;
    }else{
        printf("args = \"%s\"\n",argv[1]);
```

```
    }
    return 0;
}
```

Argumente in Integer konvertieren

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

int main(int argc, char *argv[]){
    int sum = 0;
    for(int i = 1; i < argc; i++){
        int num;
        sscanf(argv[i], "%d", &num);
        sum += num;
    }
    printf("sum = %d\n",sum);
    return EXIT_SUCCESS;
}
```

Debuggen mit gdb

Kompilieren mit

```
gcc -g -gdb -o <filename>
```

danach gdb ausführen und mit file debugging starten

String Funkionen

Standardlib

strlen(s) gibt die Länge des Strings zurück

strcpy(dest,src) kopiert einen String - Achtung füllt ganze Größe auf

strcat(dest,src) konkatiniert einen String

strcmp(a,b) vergleicht String a mit String b

string.h

strncpy(dest,src,n) kopieren mit Zeichenanzahl vorgegeben

Typecasts

```
int i = 0;
double d = 2;
i = (int)d;
```

Multidimensionale Arrays

sind jagged Arrays - Pointer auf anderes Array in Index

```
type a[SIZE1][SIZE2][SIZEU];
//speicher benötigt sizeof(a) * SIZE1 * SIZE2 * SIZEU

//init
int a[3] = { 1 , 2 ,3 };
int ma[3][2] = {
  {1,2},
  {3,4},
  {5,6}
};
```

Speicheranordnung

Code / Text ReadOnly!

> [f()][f2()]... [main()][strcopy()]

Data

> readonly constants
>
> R/W data global vars
>
> BSS uninit data

Heap

> Heap (wächst nach unten)

< -- brk

shared libs

> Shared Library Code

Stack

> [f()] [f2()]
>
> [main][ret][p] [ln]
>
> (Funktion + lokale Variablen + Parameter)

Größen der Speicherbereiche

```
[jan@starshine 30.09]$ gcc -o segment-sizes segment-sizes.c
[jan@starshine 30.09]$ size segment-sizes
   text    data     bss     dec     hex filename
   1499     592       8    2099     833 segment-sizes
```

Call by value vs call by reference

Call by reference lässt sich nur über Pointer realisieren in C.

Call by Value Implementierung von Swap: **Achtung** falsch!

```
#include <stdio.h>

void swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}

int main(){
    int x = 10;
    int y = 20;

    printf("x = %d, y = %d\n",x,y);
    swap(x, y);
    printf("x = %d, y = %d\n",x,y);

    return 0;
}
```

Ausgabe:

```
x = 10, y = 20
```

```
x = 10, y = 20
```

die **Call bei Reference** Methoden realisiert mit Pointern

```c
#include <stdio.h>

void swap(int *x, int *y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(){
    int x = 10;
    int y = 20;

    printf("x = %d, y = %d\n",x,y);
    swap(&x, &y);
    printf("x = %d, y = %d\n",x,y);

    return 0;
}
```

Ausgabe:

```
x = 10, y = 20
x = 20, y = 10
```

Achtung Arrays halten immer Pointer

Variablen Keywords

static erzeugt lokale Variable

Beispiel

```c
#include <stdio.h>

void f(){
    static int cnt = 0;
    cnt++;
    printf("f() was called %d times\n",cnt);
}

int main(){
    for(int i = 0;i < 10; i++){
        f();
    }

    return 0;
}
```

Ausgabe

```
f() was called 1 times
f() was called 2 times
f() was called 3 times
f() was called 4 times
f() was called 5 times
f() was called 6 times
f() was called 7 times
f() was called 8 times
f() was called 9 times
f() was called 10 times
```

extern definiert in .h Files dass die Variable in der Implementierung existiert (nur Deklaration).

Make Files

make

- -f Makefile
- -d
- --trace
- -r leave Default-Regeln

Aufbau

taret : depdencies

(tab) command args

Beispiel makefile

main.c benötigt geo.c und weight.c, weight.c benötigt geo.c

einfaches Beispiel

```
CC = gcc # compiler
CFLAGS = -pedantic -Wall -Wextra -std=c11 -ggdb # compiler flags

LD = gcc # linker
LDFLAGS = -lm # not really needed (links math lib m)

.c.o: # sufix  rule (*.c -> *.o)
        $(CC) $(CFLAGS) -c $<
# $< name of first dependecy / prerequisite

#main application
geo: main.o geo.o weight.o
        $(LD) -o $@ main.o geo.o weight.o $(LDFLAGS)
# @a targetname

# dependecies
geo.o: geo.c geo.h # first dep
weight.o: weight.c weight.h geo.h
main.o: main.c geo.h weight.h
```

erweitertes Beispiel

```
CC = gcc # compiler
CFLAGS = -pedantic -Wall -Wextra -std=c11 -ggdb # compiler flags

LD = gcc # linker
LDFLAGS = -lm # not really needed (links math lib m)

OFILES = main.o geo.o weight.o # Defines .o files
PROGRAM = geo # Programname

%.o: %.c # pattern rule
        $(CC) $(CFLAGS) -c $<
# $< name of first dependecy / prerequisite

all: clean $(PROGRAM)

#main application
$(PROGRAM): $(OFILES)
        $(LD) -o $@ $(OFILES) $(LDFLAGS)
# @a targetname

.PHONY: all clean # all and clean are not files!
clean:
        rm -f $(OFILES) $(PROGRAM) # delete o files and executable

# dependecies
geo.o: geo.c geo.h # first dep
weight.o: weight.c weight.h geo.h
main.o: main.c geo.h weight.h
```

Autodiscover makefile

erkennt automatisch Dependencies und führt nach dem Build die Anwendung aus

```
PROGRAM = geo # Programname
ARGUMENTS = # Arguments

CC = gcc # compiler
CFLAGS = -pedantic -Wall -Wextra -std=c11 -ggdb # compiler flags

LD = gcc # linker
LDFLAGS = -lm # not really needed (links math lib m)

CFILES = $(wildcard *.c)
OFILES = $(CFILES:.c=.o)

all: depend $(PROGRAM) run

#main application
$(PROGRAM): $(OFILES)
        $(LD) -o $@ $(OFILES) $(LDFLAGS)
# @a targetname

.PHONY: all clean depend run debug # all and clean are not files!
clean:
        rm -f $(OFILES) $(PROGRAM) # delete o files and executable

DEPENDFILE = .depend

run: $(PROGRAM) # dependecy is program
        ./$(PROGRAM) $(ARGUMENTS)

debug:
        gdb $(PROGRAM) $(ARGUMENTS)

depend:
        $(CC) $(CFLAGS) -MM $(CFILES) > $(DEPENDFILE)
        cat $(DEPENDFILE)

-include $(DEPENDFILE)
```

Pointer arithm.

++, -- springt immer die Länge des Datentypes, mit der Differenz von Pointer - Pointer kann man zB Array länge ermitteln.

malloc calloc realloc

malloc (size);

calloc(s,ds)

realloc ! **Achtung, gibt Null zurück wenn kein Speicher mehr verfügbar ist**

Beispiel für malloc

mit Integer

```
int *a = (int*)malloc(sizeof(int)*n);

if(a == NULL)
  //sonderfall null
```

mit Char

```
char *s = (char*)malloc(sizeof(char)*(n+1)); // +1 wegen \0
s[2] = 'x';
*(s+2) = 'x'; //selbes wie s[2]
```