[Prg.Struktur] Übersicht

- Deklaration , Definition
- Global, Lokal
- Modularisierung
- make
- statische Variablen
- Präprozessor

[Prg.Struktur] Deklarationen

- Funktionsprototypen deklarieren
  - am besten vor der main-Funktion
    (sonst Warnung vor Mischung von Code und Deklaration)
  - zumindest vor erstem Funktionsaufruf
    (sonst Warnung vor impliziter Deklaration)

- Variablen deklarieren
  Gültigkeit innerhalb des jeweiligen Programmblocks

```
int main()
{
...
}

int i;
void einefunktion()
{
    i++;
}
```

```
int j;
int main()
{
    j++;
    einefunktion();
}
void einefunktion()
{
    j++;
}
```

[Prg.Struktur] Definitionen: intern und extern

- Variablen:
  Variablen, deren Definition innerhalb eines Blocks stehen, heißen *intern*. Sonst *extern*.
- Funktionen: immer extern
  (die Definition einer Funktion in einer Funktion ist verboten)

- Sinnvolle Verwendung externer Größen
  - Wenn mehrere Funktionen auf die gleichen Variablen zugreifen (stack - Beispiel)
  - mit Vorsicht, da schnell nicht mehr nachvollziehbar ist, wann/wo eine (externe) Variable verändert wird
    (insbesondere bei banalen Variablennamen).

[Prg.Struktur] Verschattung

- Verschattung: Neudeklaration einer Variablen in einem Block

```
int i;

for (i=0;i< LAENGE;i++)
{
    float i;
    ....
}
```

Ergo: Erlaubt, aber selten sinnvoll.

[Prg.Struktur] Modularisierung

- Wiederverwendung von Programmteilen: Modulkonzept
  - Modul: unabhängig übersetzbare Einheit
  - Headerdatei (*name.h*)
    enthält nur die Deklarationen, welche
    der Modul exportiert:
    - Funktionsprototypen
    - Typdeklarationen
    - z.T. Variablen, Konstanten
  - Programmdatei (*name.c*)
    - Enthält Definitionen
    - bindet eigene Deklarationen aus *.h mit *include* ein
    - u.U. zusätzlich lokale Funktionen

[Prg.Struktur] Modularisierung Beispiel

```
/* Prototypen */
int full(void);
int empty (void);
void push(char zeichen);
char pop (void);

/*----- stapel.h -------*/
```

```
#include "stapel.h"

#define HOEHE 10

char stapel[HOEHE];
int spitze = 0;

int full(void)
{
    return spitze>=HOEHE;
}
int empty(void)
{
    return spitze <= 0;
}
void push(char zeichen)
{
    stapel[spitze++] = zeichen;
}
char pop(void)
{
    return stapel[--spitze];
}

/*----- stapel.c -------*/
```

Modularisierung Beispiel

```
#include <stdio.h>
#include "stapel.h"

int main(void)
{
    ...
    if (full())
        printf("Stapel ist voll\n");
    else
        printf("Stapel noch nicht voll\n");
    return 0;
}


/*---- Hauptprogramm: Stapeltest.c ----*/
```

- Compilierung

```
gcc stapeltest.c stapel.c

# oder in einzelnen Schritten:
gcc -c stapeltest.c stapel.c
gcc -o stapeltest stapeltest.c stapel.c
```

Modularisierung, *make*

```
# Es folgen GCC Compiler-Optionen als Variablendefinition
GCCFLAGS = -ansi -pedantic -Wall

# Kommandos
stapeltest: stapeltest.o stapel.o
    gcc -o stapeltest stapeltest.o stapel.o

stapeltest.o : stapeltest.c stapel.h
    gcc $(GCCFLAGS) -c stapeltest.c

stapel.o : stapel.c stapel.h
    gcc $(GCCFLAGS) -c stapel.c

clean:
    rm -f *~ stapel*.o stapeltest

#/*---- Makefile ----*/
```

```
make
make stapeltest
make stapel.o
```

Modularisierung, *make*

```
# Dies ist ein Kommentar

# Es folgen GCC Compiler-Optionen als Variablendefinition
GCCFLAGS = -ansi -pedantic -Wall
WCGFLAGS = -l

# Dateiname: Name von Objektdateien durch Leerzeichen getrennt
# (Beschreibung der Abhängigkeiten)
stapeltest: stapeltest.o stapel.o
# Hier einrücken mit <TAB> !!!
# (sonst: Makefile:17: *** missing separator.  Stop.)
# dann eine Befehlsfolge, welche durch die shell ausgewertet wird
    gcc -o stapeltest stapeltest.o stapel.o

# Aufräumen:
clean:
    rm -f *~ stapel*.o stapeltest

# Man kann auch ganz andere Dinge machen:
stats:
    echo "Quelltextzeilen: ";cat stapel.c stapeltest.c |wc $(WCFLAGS)
```

[Prg.Struktur] lokale / globale Größen

- Variablen, welche extern (ausserhalb einer Funktion) definiert werden, sind auch ausserhalb eines Moduls bekannt, d.h. *global*
  (müssen aber an geeigneter Stelle deklariert, also bekanntgemacht werden; z.B. im Header-File)

- Interne Variablen sind lokal

- Konstanten, welche in der Headerdatei definiert werden (mittels #define) sind ebenso lokal

[Prg.Struktur] Verstecken von Variablen/Funktionen

- Um zu verhindern, dass externe Variablen global werden, können sie mit dem Speicherklassenattribut *static* versehen werden.

```
#define HOEHE 10

static char stapel[HOEHE];
static int spitze = 0;
```

- In gleicher Weise können Funktionen versteckt werden.
  Sie sind dann ausserhalb des Moduls nicht mehr sichtbar.

[Prg.Struktur] extern vs. *extern*

- Eine Variable kann extern sein, also ausserhalb einer Funktion definiert und zugänglich

- Eine Variable kann das Attribut *extern* besitzen, womit eine Speicherklasse gemeint ist.

  (sie wurde damit lediglich deklariert, nicht definitiert)

Deklaration von Variablen: Speicherklassenattribut *extern*

```
extern int klarasalter;

/* --- programm.h --- */

klarasalter = 10;         /* das geht nicht! Variable nicht definiert */

/* --- programm.c --- */
```

```
extern int klarasalter = 10;   /* das geht nicht! Deklaration und
                                   Initialisierung */

/* --- programm.c --- */
```

```
extern int klarasalter;

/* --- programm.h --- */


static int klarasalter;       /*  das geht nicht! Widerspruch zwischen
                                  lokaler definition und globaler
                                  Deklaration */

/* --- programm.c --- */
```

[Prg.Struktur] Deklaration und Definition

- Die Deklaration einer Funktion kann in einem Gesamtprogramm mehrmals erfolgen.
  Sinn ist, den Aufruf einer Funktion zu ermöglichen.

- Die Definition kann genau einmal in einer Funktion erfolgen. Sie legt die Operationen im Falle eines Aufrufs der Funktion fest.

- Entsprechend gibt es auch die Möglichkeit, Variablen zu deklarieren:
  Speicherklassen-Attribut *extern*

[Prg.Struktur] Deklaration und Definition

- Eine Definition impliziert gegebenenfalls eine Deklaration.
- Auf Deklarationen kann verzichtet werden. Die Definition muss dann aber vor der ersten Verwendung erfolgt sein (implizite Deklaration)

```
int main()
{
    testfunktion();
}

void testfunktion()
{
    printf(“Ruf kommt nicht an!“);
}
```

```
void testfunktion()
{
    printf(“Funktion aufgerufen!“);
}

int main()
{
    testfunktion();
}
```

[Prg.Struktur] Statische und automatische Variablen

- statische Variablen
  - haben einen festen Speicherplatz und eine feste Speichergröße.
  - und können daher mit konstanten Werten und konstanten Ausdrücken (also zur Compilierzeit bekannt) initialisiert werden (default: 0) und behalten diesen Wert.

- automatische Variablen
  - werden immer wieder neu angelegt, wenn der Block durchlaufen wird, in welchem sie definiert sind.
  - werden dynamisch erzeugt. Zur Compilierzeit muß noch nicht bekannt sein, welche Ausdrücke diesen Variablen zugewiesen werden.

[Prg.Struktur] Initialisierung statischer Variablen

```
#include <stdio.h>

int variable1;

int main()
{
	int variable2;
	printf("%d\n%d\n",variable1,variable2);

	return 0;
}
```

Anwendung statischer Variablen

```
int main()
{
    int i;
    for (i = 0 ; i<=3 ; i++)
        printf("Aufruf %d: %d - %d", i, f1(),f2());
}

int f1 ()
{
    int z =1;
    return z++;
}
int f2 ()
{
    static int z = 1;
    return z++;
}
```

```
Aufruf 1: 1 - 1
Aufruf 2: 1 - 2
Aufruf 3: 1 - 3
```

[Prg.Struktur] Präprozessor

- Prä(compilations)(textersetzungs)Prozessor
- Direktiven zur Einbindung / Ersetzung
  - include , define
- Direktiven zur bedingten Kompilation
  - if / elif / else / endif
  - ifndef / undef
- Direktiven beginnen mit # als erstes Zeichen in einer Zeile
- Die Direktive geht bis zum Zeilenende; eine zusätzliche Zeile kann mit '\' direkt am Ende angeschlossen werden.

[Prg.Struktur] Präprozessor: Ein Beispiel

```
#define DIT     (
#define DAH     )
#define __DAH   ++
#define DITDAH  *
#define DAHDIT  for
#define DIT_DAH malloc
#define DAH_DIT gets
#define _DAHDIT char
_DAHDIT _DAH_[]="ETIANMSURWDKGOHVFaLaPJBXCYZQb54a3d2f16g7c8a90l?e'b.s;i,d:"
;main                   DIT                         DAH{_DAHDIT
DITDAH                  _DIT,DITDAH                 DAH_,DITDAH DIT_,
DITDAH                  _DIT_,DITDAH                DIT_DAH DIT
DAH,DITDAH              DAH_DIT DIT                 DAH;DAHDIT
DIT _DIT=DIT_DAH        DIT 81                      DAH,DIT_=_DIT
__DAH;_DIT==DAH_DIT     DIT _DIT                    DAH;__DIT
DIT'\n'DAH DAH          DAHDIT DIT                  DAH_=_DIT;DITDAH
DAH_;__DIT              DIT                         DITDAH
_DIT_?_DAH DIT          DITDAH                      DIT_ DAH:'?'DAH,__DIT
DIT' 'DAH,DAH_ __DAH    DAH DAHDIT                  DIT
DITDAH                  DIT_=2,_DIT_=_DAH_;         DITDAH _DIT_&&DIT
DITDAH _DIT_!=DIT       DITDAH DAH_>='a'?           DITDAH
DAH_&223:DITDAH         DAH_ DAH DAH;               DIT
DITDAH                  DIT_ DAH __DAH,_DIT_        __DAH DAH
DITDAH DIT_+=           DIT DITDAH _DIT_>='a'?      DITDAH _DIT_-'a':0
DAH;}_DAH DIT DIT_      DAH{                        __DIT DIT
DIT_>3?_DAH             DIT                          DIT_>>1 DAH:'\0'DAH;return
DIT_&1?'-':'.';}__DIT DIT                           DIT_ DAH _DAHDIT
DIT_;{DIT void DAH write DIT                        1,&DIT_,1 DAH;}
```

[Prg.Struktur] Präprozessor: Makros

- Makros: Textbausteine ersetzen
- Eine Makroexpandierung darf auch zuvor definierte Makros enthalten (der Makroname hingegen ist selbst nicht einer Ersetzung unterworfen)

```
#define BREITE 10
#define HOEHE (BREITE + 5)
```

- Der Makroersetzer ist gierig:

```
#include <stdio.h>
#define abc 123
#define abc4 567

int main()
{
	printf("%d\n",abc);
	printf("%d\n",abc4);
	printf("%d\n",1234);
	return 0;
}
```

[Prg.Struktur] Präprozessor: Makros mit Parametern

```
#define QUADRAT(x)   x * x

x = QUADRAT(4.5);    /* ergibt:  x = 4.5 * 4.5 */
y = QUADRAT(x+1);    /* ergibt:  y = x+1 * x+1 */
                     /* richtig wäre (x+1)*(x+1) */

i = QUADRAT(j++)     /* i= j++ * j++ */
                     /* nicht unbedingt so erwünscht */
```

- Makroargumente können als Zeichenkette verwendet werden, dabei wird dem Argument ein # vorangestellt.

```
#define TEXTDOPPLER(x)   #x#x

char string[] = TEXTDOPPLER(eintext);
printf("%s",string);                        /* Ausgabe: eintexteintext
```

Präprozessor: Bedingte Compilation

```
#if bedingung
   -- quellcode --
[#elif bedingung
   -- quellcode --
]
[#else
   -- quellcode --
]
#endif
```

```
#define TESTEN 1

void main()
{
    #if TESTEN
        printf('“Testausgabe aktiviert\n“);
    #endif
}
```

```
#define TESTEN

void main()
{
    #if defined(TESTEN)
        printf(“Testausgabe aktiviert\n“);
    #endif

    #ifdef TESTEN
        printf(“oder mit Direktive ifdef“);
    #endif
}
```

Präprozessor: Bedingte Compilation

zusätzliche Direktiven:

ifndef
undef
error
line

Vom C-Standard vorgeschriebene Makros

```
__FILE__ Name der Quelldatei
__LINE__ Nummer Der Zeile im Quellcode
__DATE__ Datum
__TIME__ Zeit
__STDC__ wahr oder falsch: Standardcompiler
```

[Prg.Struktur] Präprozessor: noch ein Beispiel

```
#include <stdio.h>

#define TESTEN

#ifdef TESTEN
    #define REPORT(i,j) printf("DEBUG (%s):%d"\
        " [i]%d [j]%f\n",__FILE__,__LINE__,i,j)
#else
    #define REPORT(i,j)
#endif

int main()
{
    int i;
    double j=10;
    for ( i = 10 ; i > -5 ; i--){
        REPORT(i,j);
        printf("Quotient: %f\n",j/i);
    }
    return 0;
}
```

Präprozessor: Bedingte Compilation

zusätzliche Direktiven:

ifndef
undef
error
line

Vom C-Standard vorgeschriebene Makros

```
__FILE__ Name der Quelldatei
__LINE__ Nummer Der Zeile im Quellcode
__DATE__ Datum
__TIME__ Zeit
__STDC__ wahr oder falsch: Standardcompiler
```

[Prg.Struktur] Einschub: Standardbibliothek

- Der ANSI-Standard schreibt für die Nutzung der Standardbibliothek die Standard-Headerdateien vor, insgesamt gibt es 15 verschiedene davon.

<ctype.h>	<assert.h>	<locale.h>
<float.h>	<errno.h>	<setjmp.h>
<limits.h>	<math.h>	<signal.h>
<stdio.h>	<stddef.h>	<stdarg.h>
<string.h>	<stdlib.h>	
	<time.h>	

Einschub: Standardbibliothek

- z.B. math.h

acos();	asin();	atan();
atan2();	ceil();	cos();
cosh();	exp();	fabs();
floor();	fmod();	frexp();
ldexp();	log();	log10();
modf();	pow();	sin();
sinh();	sqrt();	tan();
tanh();		

[Prg.Struktur] Einschub: Standardbibliothek

hier werden verwendet:

- eine mathematische Funktion zur Berechnung der Quadratwurzel
- errno: Variable, welche von vielen Bibliotheksfunktionen benutzt wird, um Aussagen über aufgetretene Fehler zu machen
- strerror: Ausgabe des Fehlercodes als String

```
gcc beispiel.c -lm
```

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <string.h>

int main(void)
{
	printf("Error-Number: %d\n",errno);
	printf("Error-Number: %s\n",strerror(errno));
	sqrt(-2.0);
	printf("Error-Number: %d\n",errno);
	printf("Error-Number: %s\n",strerror(errno));
}
```

[Prg.Struktur] Einschub: Standardbibliothek

- stdlib.h

EXIT_FAILURE

EXIT_SUCCESS

VOID

RAND_MAX

...

exit()

malloc()

free()

abs()

rand()

srand()

...

Tip: Probieren Sie auch einmal:
man *c-Funktion*

```
/* (Pseudo-) Zufallszahlen */

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

int main(void)
{
	int i=0;
	srand(10);
	for (;i<10;i++)
		printf("%d\n",rand());
	return 0;
}
```