Organisatorisches

Heute

- Rest von Foliensatz 6
- Teil 7: Spezielle Operatoren, Bitoperationen, Zeigerarithmetik
- Teil 8: struct und typedef

- Teil 9: Präprozessor, malloc, Wiederholungsaufgaben → nett, zur Klausurvorbereitung, neue Inhalte nicht klausurrelevant
- Teil 10: Dateiein-/ausgabe, fprintf → nur zur Info

Nächste Woche

- Klausurvorbereitung → **vorbereiten!**

13.6.7 break und continue

break 1. dient dazu eine Schleife (while, do, for) vorzeitig zu verlassen.

Typischer Anwendungsfall ist der Abbruch einer Schleife, wenn besondere Bedingungen vorliegen.

I.allg. sollte eine strukturierte Lösung dem *break* vorgezogen werden. Gelegentlich kann der Code durch *break* auch übersichtlicher werden.

2. Darüberhinaus beendet *break* ebenfalls switch-Anweisungen.
Hier ist *break* sinnvoll und üblich.

continue dient in Schleifen (while, do, for) dazu, die nächste Wiederholung der umgebenden Schleife sofort zu beginnen.

Beispiel:

```
/* druckt alle Zahlen von 0 ... 99, mit Ausnahme */
/* der durch 5 teilbaren Zahlen.                 */
for(i=0; i<100; i++){
    if(i%5 == 0) continue;
    printf("%2d \n",i);
}
```

Frage: Wie könnte man das Programm besser (ohne *continue*) schreiben?

ÜBUNG: Wahl der richtigen Kontrollstruktur

1. Vereinfachen Sie:

```
int i=0;
while(i<100) {
   if((x=x/2) > 1) {
      i++;
      continue;
   }
   break;
}
```

2. Vereinfachen Sie:

```
int i,total=0,count=0;
/* Zufallszahl von 0..100 */
while((i=random(100)) != 0) {
   total++;
   if(i==50)   break;
   if(i%3==0)  continue;
   if(i%5==0)  count++;
}
```

13.7 Funktionsdefinition und –deklaration

Funktionen sind die Träger von Algorithmen bzw. Berechnungen.
Funktionen werden nach folgender Syntax vereinbart.

SYNTAX:

funktionsdefinition : fkt-kopf fkt-rumpf

fkt-kopf : ret-*type-specifier fkt-bezeichner* '(' *formale-argumente* ')'
fkt-rumpf : *verbund-anweisung*

ret-type-specifier : 'void' | 'int' | 'float' | ...usw... Anm.: unvollständig !
formale-argumente : 'void' | *argumentenliste*

argumentenliste : *type-specifier variablen-bezeichner*
{ ',' *variablen-bezeichner* }
{ ',' *type-specifier variablen-bezeichner*
{ ',' *variablen-bezeichner* } }

Beispiele:

```
void wenig ( void ) { .... }
int inkrementiere ( int zahl, int incr ) { .... }
void update ( float druck, temp, int wert ) { .. }
```

13.7 Funktionsdefinition und –deklaration (Fortsetzung)

Deklaration

= Bekanntgabe von Name und Typ eines Objektes an den Compiler.

Definition

= Detailbeschreibung eines Objektes.

```
/* Deklaration von Variablen*/
int  i, j, k;
char txt;

/* Deklaration von Funktionen */
int QuadSum (int x, int y);


int main(){
   ...
   k=QuadSum(m,n);   /*Aufruf*/
   ...
}
```

```
/* Definition einer Funktion */


int QuadSum (int x, int y)
{
    int res;
    res = x * x + y * y;
    return res;

}
```

13.8 Return-Statement

Das Return - Statement beendet die Ausführung einer Funktion und gibt, sofern für die Funktion ein Ergebnistyp vereinbart wurde, das erarbeitete Ergebnis an seine Aufrufumgebung zurück.

Syntax:

return [*expression*] ;

Der Wert des optionalen Ausdrucks liefert den Return-Wert der Funktion und muss zum Ergebnistyp der Funktion kompatibel sein.

Beispiele:

```
return ( a + b )/2 ;
return 23 ;
```

ÜBUNG Funktionsdefinition / Funktionsdeklaration

Schreiben Sie eine C-Funktion `XNorm()` zur Berechnung von:

$$x_{Norm} = \left| \frac{x}{\sqrt{x^2 + y^2}} \right|$$

x und y sind vom Typ `int`, der Rückgabewert sei vom Typ `double`.
Im Fall x=y=0 soll der Rückgabewert 0.0 sein.

Es können folgende Bibliotheksfunktionen verwendet werden:

- `double sqrt(double x)`
- `double fabs(double x)`

Schreiben Sie ein Testprogramm, welches die Funktion aufruft.
Das Testprogramm und die Funktion `XNorm()` sollen in verschiedenen Dateien realisiert werden (`Test.c`, `Norms.c`).