Grundlagen der Programmiersprache C++

Um den Studierenden den Einstieg in die FE-Programmierung zu erleichtern werden die wesentlichen Elemente eines C-Programmes beschrieben, soweit sie in den Lehrprogrammen zur FE-Methode umgesetzt werden. Für Detailinformationen zur Programmiersprache C sei auf die Literatur verwiesen.

1 Elementare Grundlagen

1.1 Programmaufbau und grundlegende Begriffe

Die Lehrprogramme zur FE-Methode setzen sich aus mehreren Teilen zusammen.

Compileranweisungen (z.B. #include ...)
Deklaration globaler Objekte und Funktionen

Hauptprogramm
int main()
{
Anweisungen;
return 0;
}

Funktionsdefinitionen

Über die Compileranweisungen werden weitere Dateien in das Programm eingefügt. In den Dateien können Standardfunktionen (z.B. cin und cout zur Ein- und Ausgabe) deklariert und definiert sein.
Im Hauptprogramm main() wird die Programmaufgabe durch Anweisungen umgesetzt. Teilaufgaben können in Unterprogramme (Funktionen) ausgegliedert werden. Die Funktionen werden aus dem Hauptprogramm aufgerufen.
Vor dem Programmdurchlauf müssen eigene Funktionen deklariert werden. Hinter dem Hauptprogramm befinden sich die Funktionsdefinitionen, in denen der Source-Code zur Bearbeitung der Teilaufgaben zu finden ist.

1.2 Einfache Datentypen und Programmelemente

Sollen in einem Programm Berechnungen durchgeführt werden, ist für ganze Zahlen der **Datentyp** int und für Gleitkommazahlen der Datentyp double vorhanden. Für Zeichen wie Buchstaben oder Ziffern gibt es den Datentyp char.
Beispiel für eine Deklaration: int a,b,c; a,b,c können veränderliche Werte annehmen, sie heißen **Variablen**.

Mit `const double x=1.5;` wird eine **Konstante** deklariert und definiert. Der Wert einer Konstanten kann während des Programmdurchlaufes nicht mehr verändert werden.
Ein **Ausdruck** kann sich aus einer oder mehreren Konstanten, Variablen oder Funktionen zusammensetzen. Beispiele: `i++` oder `c = a + b`
In **Anweisungen** werden die Programmieraufgaben umgesetzt. Einfache Beispiele sind die Initialisierung einer Variablen `double x=4;` oder die Summierung zweier Zahlen `int a,b; int c = a + b;`. Anweisungen enden mit einem Semikolon.
Anweisungen können in **Anweisungsblöcken** { } zusammengefasst werden.

1.3 Operatoren

Werden Daten bearbeitet, erfolgt die Beschreibung der Operation mittels eines Operators. Je nach Datentyp sind unterschiedliche Operationen zulässig.
Für Daten des Dateityps Integer und Double seien folgende Operatoren vorgestellt:

Operator	Beispiel	Bedeutung
arithmetisch		
+	+i	unäres Plus
-	-i	unäres Minus
+	i + 1	binäres Plus
-	i - 2	binäres Minus
*	i * 3	Multiplikation
/	i / 4	Division
=	i = 5	Zuweisung
+=	i += 6	i = i + 6
-=	i -= 7	i = i - 7
*=	i *= 8	i = i * 8
/=	i /= 9	i = i / 9
vergleichend		
<	i < 3	kleiner als
>	i > 3	größer als
<=	i <= 3	kleiner gleich
>=	i >= 3	größer gleich
==	i == 3	gleich
!=	i != 3	ungleich

Bei Daten vom Dateityp Integer sind auch das Autoinkrement und Autodekrement zulässig:

++	i++	vorherige Inkremetierung um eins
	++i	nachfolgende Inkremetierung um eins
- -	i- -	vorherige Dekrementierung um eins
	- -i	nachfolgende Dekrementierung um eins

Zudem gibt es die logischen Operatoren && und **||**:

&&	if(x < 3 && y > 4)	wenn x < 3 UND y > 4, ...
||	if(x < 3 **||** y > 4)	wenn x < 3 ODER y > 4, ...

Wichtig ist die Abstufung der Priorität, die die einzelnen Operatoren untereinander aufweisen. Bei Ausdrücken, in denen mehrere Operatoren aufeinander folgen ist damit definiert, in welcher Reihenfolge die Operatoren ausgeführt werden. An dieser Stelle sei auf die weiterführende Literatur verwiesen.

1.4 Kommentare

```
// ...        Kommentar bis Zeilenende
/* ... */     Kommentar über mehrere Zeilen möglich
```

1.5 Einfache Ein- und Ausgabe

Folgende Ein- und Ausgabefunktionen werden über die Datei *iostream.h* eingebunden:

```
cin     Standardeingabe über Tastatur
cout    Standardausgabe auf Bildschirm
cerr    Standardfehlerausgabe auf Bildschirm
```

Beispiele für Ein- und Ausgaben:

```
cin » x;          // Einlesen einer Zahl und Belegung mit der Variablen x
cout « y;         // Ausgabe des Wertes der Variablen y
cout « "\n";      // Ausgabe eines Zeilenumbruches
```

Zu beachten ist die Pfeilrichtung.

2 Einfache Sprachstrukturen

2.1 Auswahlanweisungen

Hier muß eine Bedingung erfüllt sein, sodass die Anweisung ausgeführt werden.
Sollen mehrere Anweisungen durchgeführt werden, wenn die Bedingung erfüllt ist, werden die Anweisungen mit { } zu einem Anweisungsblock zusammengefasst. Der Block wird anstelle einer Einzelanweisung hinter die Bedingung gesetzt. Gleiches gilt auch für Schleifenanweisungen (s. Kapitel 2.2).

if-Anweisung

```
if(Bedingung)
    Anweisung;
```

if-else-Anweisung

Ist die Bedingung nicht erfüllt, kann mit else eine Alternative angewiesen werden.

```
if(Bedingung)
    Anweisung1;
else
    Anweisung2;
```

2.2 Schleifen

Schleifen werden eingesetzt, wenn Programmieraufgaben wiederholt ausgeführt werden müssen. Alle Schleifenanweisungen weisen eine Bedingung auf. Ist die Bedingung nicht erfüllt, wird der Schleifendurchlauf beendet.
Die for-Schleife beinhaltet im Gegensatz zu den anderen Schleifenanweisungen eine Initialisierung und Inkrementierung, die sich üblicherweise auf die Laufvariable beziehen.

for-Schleife

```
for(Initialisierung; Bedingung; Inkrementierung)
    Anweisung;
```

while-Schleife

```
while(Bedingung)
    Anweisung;
```

do-while-Schleife

```
do
    Anweisung;
while(Bedingung)
```

2.3 Weitere Anweisungen

return-Anweisung
Mit einer return-Anweisung kann eine Funktion augenblicklich verlassen werden.
Der Anweisung kann ein Rückgabewert zugewiesen werden.
Beispiel: Beenden des Hauptprogramms mit der Rückgabe des Wertes *0*

```
int main ()
{
    :
    return 0;
}
```

Die Bedeutung des Rückgabewertes wird detaillierter in Kapitel 4 erläutert.
Soll kein Wert mit der return-Anweisung zurückgegeben werden, schreibt man einfach:

```
{
    :
    return;
}
```

3 Erweiterte Datentypen

3.1 Zeiger

Wird eine Integer-Variable deklariert und initialisiert (z.B. int x=4;) ist der Variablen auch eine Speicherplatzadresse zugewiesen. Siehe folgende Abbildung:

Inhalt	Adresse	Name
⋮	⋮	
23	4125	
4	4126	x
4	4127	
4126	4128	yy
⋮	⋮	

Über die Variable vom Datentyp Zeiger kann auf Adressen von Speicherbereichen zugegriffen werden. Mit int *yy; wird beispielsweise auf eine Adresse einer Integer-Variablen *gezeigt*. Die Definition variabler Zeiger erfolgt durch den Operator *.
Über den Operator & werden Adressen angesprochen.
Beispiel: int *yy=&x; // Der Zeiger *yy zeigt auf die Adresse der Variablen x (s. Abbildung).
Auch Zeiger haben eine Adresse. Der Zugriff erfolgt über den Operator &.

Die Initialisierung variabler Zeiger kann auch mit den Befehlen new und delete erfolgen:
Datentyp * *Name* = new *Datentyp*; // Initialisierung
Datentyp * *Name* = delete *Datentyp*; // Freigabe

3.2 Felder

Mit **eindimensionalen** Feldern können Vektoren gespeichert werden.
Die Definition erfolgt mit

Datentyp Feldname [*Anzahl der Feldelemente*];.

Die Anzahl der Feldelemente darf sich im Programmdurchlauf nicht ändern.
Der Zugriff auf die Adressen und die im Feld abgespeicherten Werte ist über mehrere Befehle möglich:

double rlae[10];	Initialisierung des Feldes rlae[10]
rlae[0]	Zugriff auf den ersten Wert des Feldes
*rlae	Zugriff auf den ersten Wert des Feldes
rlae[i]	Zugriff auf den (i+1)-ten Wert des Feldes
*(rlae+i)	Zugriff auf den (i+1)-ten Wert des Feldes

rlae	Adresse des ersten Elementes
&rlae	Adresse des ersten Elementes
&rlae [0]	Adresse des ersten Elementes
rlae + i	Adresse des (i+1)-ten Elementes
&rlae [i]	Adresse des (i+1)-ten Elementes

Zweidimensionale Felder eignen sich zum Speichern von Matrizen.
Definition eines 2D-Feldes:

Datentyp Feldname [*Anzahl der Zeilen*] [*Anzahl der Spalten*];.

Ein 2D-Feld lässt sich als Zeilenvektor deuten, in dem jedes Element wiederum ein Vektor ist.
Die Syntax für den Zugriff auf Werte und Adressen zweidimensionaler Felder ist im folgenden nur soweit angegeben, wie sie in den Programmen zur FE-Methode eingesetzt wird:

double iinz[10][2];	Initialisierung des Feldes iinz[10][2]
iinz[0][0]	Zugriff auf den Wert der ersten Zeile und ersten Spalte des Feldes
iinz[i][j]	Zugriff auf den Wert der (i+1)ten Zeile und (j+1)ten Spalte des Feldes
**iinz	Zugriff auf den Wert der ersten Zeile und ersten Spalte des Feldes
iinz	Adresse des ersten Elementes
iinz[0]	Adresse des ersten Elementes
iinz[i]+j	Adresse des Elementes der (i+1)ten Zeile und (j+1)ten Spalte
*iinz	Adresse des ersten Elementes

Darüber hinaus gibt es weitere Möglichkeiten für den Zugriff auf Inhalte und Adressen.

3.3 Dynamische Felder

Mit Hilfe dynamischer Felder ist es möglich, den Speicherplatz für Felder während des Programmdurchlaufes zu reservieren und wieder freizugeben.
Das Anfordern des Speicherplatzes kann mit dem Operator new erfolgen, das Freigeben mit dem Operator delete. Beispiel für **eindimensionale Felder**:

*Datentyp *Feldname* = new *Datentyp* [*Anzahl der Feldelemente*];. //Anfordern
delete [ ] *Feldname*;. //Freigeben

Beispiel für **zweidimensionale Felder**:

```
Datentyp **Feldname = new Datentyp* [ Anzahl der Zeilen ];          //Anfordern
for (i = 0; i < Anzahl der Zeilen ; i++)
    Feldname[i] = new Datentyp [ Anzahl der Spalten ];

for (i = 0; i <  Anzahl der Zeilen ; i++)                          //Freigeben
    delete [ ] Feldname[i];
delete [ ] Feldname;
```

3.4 Referenzen

Eine Referenz verweist auf ein anderes Objekt. Sie wird mit dem Operator & gekennzeichet. Referenzen müssen deklariert und durch ein anderes Objekt, z.B. eine Variable initialisiert werden. Es ist unzulässig Referenzen einen Wert zuzuweisen.
Beispiel:

```
int a;          // Deklaration der Variablen a
int &b = a;     // Referenz auf Variablen a
a = 2;          // Die Integer-Variable a hat den Wert 2
                // → Damit hat auch b den Wert 2
```

4 Funktionen

Wie schon erwähnt, werden viele Teilaufgaben in Funktionen ausgelagert. Die Folge von Anweisungen, die diese Teilaufgabe umsetzt, ist in der Funktionsdefinition zu finden.
Um ausgeführt zu werden, muss die Funktion aufgerufen werden. Dies kann im Hauptprogramm main () oder aus anderen Funktionen heraus geschehen.
Vor dem Hauptprogramm main () muss die Funktion *deklariert* werden.

4.1 Funktionsdefinition

Syntax der Funktionsdefinition:

Rückgabetyp Funktionsname (*Liste der formalen Parameter*)
{
  Anweisungen;
}

Beispiele:

```
void steif (double** rke, int kunel, double rlae, double rbieg)
{ Anweisungen; }
int* new_int_zeiger_l1 (const int zeilen)
{ Anweisungen; }
```

Über den Rückgabetyp ist der Datentyp des Wertes festgelegt, der mit der return-Anweisung zurückgegeben werden kann. Im zweiten Beispiel (Funktion int* new_int_zeiger_l1) wird mit der return-Anweisung ein Zeiger vom Datentyp Integer zurückgegeben.
Im ersten Beispiel (Funktion steif) wird kein Wert zurückgegeben. Der Rückgabedatentyp ist void.
In der Parameterliste sind die Parameter, die in den Funktionsquellcode übergeben werden sollen, samt Datentyp aufzulisten.

4.2 Funktionsdeklaration

Syntax der Funktionsdeklaration:

Rückgabetyp Funktionsname (*Parameterliste*);

Beispiele:

```
void steif (double**, int, double, double);
int* new_int_zeiger_l1 (const int);
```

In der Parameterliste müssen die Datentypen der übergebenen Parameter angegeben werden. Durch die Deklaration sind dem Compiler die Funktionsnamen und Parameterdatentypen bekannt. Bei einem Funktionsaufruf kann damit der Compiler das Programm auf Syntaxfehler hin überprüfen.

4.3 Funktionsaufruf

Syntax des Funktionsaufrufes:

Funktionsname (*Liste der aktuellen Parameter*);

Beispiele:

```
steif (rke, kunel, rlae[i], rbieg[i]);
int* ivek = new_int_zeiger_l1 (kel);
```

Durch den Aufruf einer Funktion wird der Programmcode der Funktionsdefinition ausgeführt. Die Funktion erhält die Liste der aktuellen Parameter. Damit wird die Liste der formalen Parameter mit den aktuellen Parametern aus dem Funktionsaufruf belegt.

4.4 Parameterübergabe

Bei der **Übergabe per Wert** wird der Wert kopiert und der Funktion übergeben. Innerhalb der Funktion kann der Wert des Parameters verändert werden. Die Änderung bleibt nach dem Verlassen der Funktion aber nicht wirksam.

Die **Übergabe per Zeiger** übergibt einen Zeiger an die Funktion. Die Adresse, auf die der Zeiger zeigt, kann innerhalb der Funktion geändert werden. Die Änderung bleibt nach dem Verlassen der Funktion nicht wirksam.
Bei variablen Zeigern kann der Inhalt der Adresse in der Funktion geändert werden. Diese Änderung bleibt auch nach dem Verlassen der Funktion gültig.

Bei der **Übergabe als Referenz** wird der Funktion keine Kopie des Parameters übergeben. In Funktionsdeklaration und -definition werden Referenzen mit dem Operator & gekennzeichnet.
Werte von Parametern können in der Funktion verändert werden. Diese Änderungen bleiben auch nach dem Verlassen der Funktion erhalten.

Die **Rückgabe mit return** erlaubt die Rückgabe eines Wertes des vorgegebenen Datentypes. Ist der Datentyp void (s. oben Beispiel-Funktion steif), wird kein Wert zurückgegeben. Der Rückgabewert der zweiten Beispiel-Funktion int* ivek (s. oben) ist dann ein Zeiger vom Typ Integer und gibt hier die Adresse einer Integer-Variablen zurück.

5 Ein- und Ausgabe mittels Bibliotheksfunktionen

Sollen Daten aus Ascii-Dateien gelesen und in Dateien geschwieben werden, stehen mit dem Headerfile stdio.h folgende C-Funktionen zur Verfügung:

fopen	Öffnen einer Datei
fclose	Schliessen einer Datei
fscanf	Einlesen von Daten aus einer Datei
fprintf	Schreiben von Daten in eine Datei

Bezüglich der genauen Syntax der genannten Funktionen sei auf die Literatur verwiesen.

Weiterführende Literatur:

- Ulrich Breymann: *C++ Eine Einführung*; Hanser-Verlag, 1999;
- Bernd Möller; *Grundlagen der Bauinformatik, Einführung in C++, Teil 1*; Vorlesungsmanusskript; TU Braunschweig, 1994;
- Bjarne Stroustrup; *Die C++ Programmiersprache*; 4. aktualisierte und erweiterte Auflage; Addison-Wesley Verlag, 2000;