h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

C++ Übersicht

von

Marco Münch B.Sc.

1 Aufbau eines C++-Programmes

```
1 #include <iostream>          //Includes werden gebunden
2
3 using namespace std;         //std-Namensbereich benutzen.
4
5 int main()                   //Es muss genau !eine! Funktion main geben, hier beginnt das Programm
6 {                            //Beginn der Funktion bzw. eines Blocks.
7       cout << "Hello World"; //Ausgabe von Hello World
8       return 0;              //Rücksprung
9 }                            //Ende der Funktion bzw. des Blocks.
```

2 Variablen

```
1 char                         //Ein Zeichen (Byte) (z.B. 'a','B')
2 short, int, long             //ganze Zahl, dabei gilt short<=int<=long
3 float, double                //Flieskommazahl (z.B. 2.345)
```

Nicht im std-Namensbereich integriert:

```
1 #include <string>
2
3 string                       //Eine Zeichenkette (z.B. "Hallo")
```

3 Definition von Variablen

```
1 int i;                       //Definition einer Variablen
2 int k,l,m;                   //oder mehrere Variablen.
3 int j=10;                    //Definition und Initialisierung.
4 i=28;                        //Zuweisung eines Wertes zu einer Variablen
```

4 Operatoren

```
1 k=i+j;                       //Addition
2 k=i-j;                       //Subtraktion
3 k=i/j;                       //Division
4 k=i*j;                       //Multiplikation
5 k=27 % 10;                   //Modulo-Rechnung: Ergibt den Rest einer Division:
6                              //hier ist k=7 da 27 /10 = 2 Rest 7.
7 i=1;                         //Zuweisung eines Wertes zu einer Variablen
8 i++;                         //Inkrement-Operator, erhöht i um 1
```

Für manche Operatoren existieren Kürzel:

```
k+=i;                          //k = k + i
k-=i;                          //k = k - i
k*=i;                          //k = k * i
k/=i;                          //k = k / i
```

5 Bedingungen

Bedingungen setzen sich aus Logische Operatoren (== gleich, != ungleich, < kleiner, > größer, <= kleiner gleich, >= größer gleich) und Logische Verknüpfungen (&& und, || oder, ! nicht) zusammen.

5.1 if Bedingung

```
if (k == 1)                    //k==1 = Bedingung, "wenn k gleich 1 dann führe aus"
{                              //Beginn des Anweisungsblocks
     cout << "k ist eins ";    //Beachte: Bei einer Anweisung können die  {} weggelassen werden.
}
```

5.2 if-else Bedingung

```
if  (k<0)                       //k==0 = Bedingung, "wenn k gleich 0 dann ..."
      cout << "k ist negativ";
else                            // "... wenn nicht dann"
      cout << " k ist positiv oder 0";
```

5.3 verkettete if-else Bedingung

```
if  (k<0)                       //k<0 = Bedingung, "wenn k kleiner 0 dann ..."
      cout << "k ist negativ";
else if (k==0)                  // "... wenn nicht dann schaue ob k gleich 0 ist dann ..."
      cout << " k ist 0";
else                            // "... wenn nicht dann"
      cout << " k ist positiv";
```

5.4 switch-case Anweisung

```
char a  = '1';
cin >> a;
switch (a)                     //beginn der switch-case, a = Variable nach der geschaut wird
{
    case '1': ...              //erster Fall, erster Anweisungsblock
              break;           //break beendet den Anweisungsblock
    case '2': {                //zweiter Fall, zweiter Anweisungsblock
              ...              //{ möglich wenn neue Variablen deklariert werden sollen
              break;           //break beendet den Anweisungsblock
              }
    case 'a':                  //dritter Fall, dritter Anweisungsblock
    case 'A': ...              //Eingabe 'a' und 'A' führen gleichen Anweisungsblock aus
              break;
    default:                   //default-Anweisung, wenn kein case Fall gefunden wird
            cout << "Falsche Eingabe";
}
```

```
int a  = 1;
cin >> a;
switch (a)                     //beginn der switch-case, a = Variable nach der geschaut wird
{
    case 1: ...                //erster Fall, erster Anweisungsblock
              break;           //break beendet den Anweisungsblock
    case 2: ...                //zweiter Fall, zweiter Anweisungsblock
              break;           //break beendet den Anweisungsblock
    default:                   //default-Anweisung, wenn kein case Fall gefunden wird
            cout << "Falsche Eingabe";
}
```

6 Schleifen

6.1 for Schleife

```
for (int i=0 ; i<10 ;  i++ )   //int i=0 : Start, wird nur einmal ausgeführt
                               //i<10 : Bedingung, wird bei jedem Schleifendurchlauf überprüft.
                               //i++ : Schrittweite, wird am Ende eines Durchlaufes ausgeführt.
{
     cout <<  i;
}
```

6.2 while Schleife

```
char a='n';
while (a != 'j')               //a != 'j' :  Bedingung, wird bei jedem Schleifendurchlauf überprüft.
{
      ... Anweisungsblock ...
}
```

6.3 do-while Schleife

```
char a='n';
do{                            //Im Gegensatz zur while-Schleife wird die do-while-Schleife mindestens
                               //einmal durchlaufen. Die Schleifenbedingung a != 'j'  wird immer
                               //am Ende eines Durchlaufes überprüft.
      ... Anweisungsblock ...
}while  (a != 'j') ;           //Semikolon am Ende!
```

6.4 Zusatzbefehle

```
int i=1;
while (1) {
     cout << i;
     i = i + 1;
     if (i >=10 )  break;      //Schleife wird sofort abgebrochen.
}
```

```
for (i=1 ; i<100; i++ )
{
     if (i % 2 != 0)
          continue;            //fängt den nächsten Schleifendurchlauf
     cout << i;
}
```

7 Arrays (Reihen)

7.1 1D Array

```
int feld[10];                  //Ein Integerfeld mit 10 (n) Elementen
feld[0] = 101;                 //In C++ wird bei 0 angefangen zu zählen
feld[9] = 22;                  //das erste Element ist feld[0], damit das letzte feld[n-1]
                               //Felder können mit jedem Variablentyp angelegt werden.
```

7.2 2D Array

```
int feld [4][5];               //Zweidimensionales Feld mit 4 (n) * 5 (m)=20 Elementen
feld[0][0] = 9;                //das erste Element ist feld[0][0]
feld[3][4] = 32;               //das letzte Element ist feld[n-1][m-1]
```

8 Zeiger

```
int i=5;
int *p;                    //Definition eines Zeigers auf einen Integer
p=&i;                      //Die Adresse von i wird p zugewiesen
                           //& ist der Adressoperator: gibt die Adresse der Variable
cout <<  *p;               //Der Inhalt, wo p hinzeigt wird ausgegeben
                           //* ist der Dereferenzierungsoperator: gibt den Inhalt der Adresse
```

8.1 Zusammenhang Array + Pointer

```
int feld[3] = {1,2,3};     //Der Feldname ist die Adresse des ersten Elementes des Feldes
int *p;
p=feld;                    //Feld wird einem Zeiger zugewiesen.
cout << *p;                //Der Inhalt des Zeigers wird ausgegeben: 1
p=p+1;                     //Der Zeiger wird um eins erhöht
cout << *p;                //Der Inhalt des Zeigers ist nun: 2
                           // feld=feld+1 geht nicht, da feld ein Array ist!
```

9 Funktionen

```
int summe (int a, int b)    //Beginn der Funktion: int = Funktionstyp -> erwarteter Rückgabetyp
                            //summe = Funktionsname, damit wird im Programm die Funktion aufgerufen
                            //int a, int b = Parameter vom Typ int, werden in der Funktion verwendet
{
     int c;
     c=a+b;
     return (c);            //Rückgabe eines Wertes vom Typ der Funktion
}

int main()
{
...
cout << summe(25,10);       //Aufruf der Funktion
...
}
```

Funktionen können auch hinter der Main-Funktion stehen werden, müssen aber vorher deklariert werden:

```
int summe (int a, int b);   //Deklaration

int main()
{
...
cout << summe(25,10);       //Aufruf der Funktion
...
}

int summe (int a, int b)    //Beginn der Funktion
{
     ...
}
```

9.1 Überladene Funktionen

```
1 int summe (int a, int b);          //Überladene Funktionen können
2 float summe (float a, float b);    //unterschiedliche Parameter besitzen
3 int summe (int a, int b, int c);   //oder mehr Parameter
4                                    //Der Funktionsname muss immer identisch sein
5                                    //Auswahl der richtigen Funktion läuft über die Parameter und deren
                                         Typen
```

9.2 Übergabe von Array an Funktionen

```
1 void fkt (int *feld);          //Parameter muss ein Pointer sein, da die Übergabe als Adresse geschieht
2
3 int main()
4 {
5     int  arr[3]={1,2,3};
6     fkt (arr);                 //Array wird ohne Besonderheiten übergeben
7 }
```

9.3 Übergabe von Pointern an Funktionen

```
1 void fkt (int *p);
2
3 int main()
4 {
5                 int Zahl = 5
6     int *q=&Zahl;
7     fkt (q);
8 }
```

10 Strukturen

```
1 struct  person                 //Eine Struktur ist eine Zusammenfassung von Datentypen
2 {                              //Im Prinzip ist es eine Definition eines neuen Datentyps
3      string name;
4      int  alter;
5 };
```

Benutzung:

```
1 person  p1;
2 p1.name="Otto";                //Mit dem Punktoperator hat man Zugriff auf die
3 p1.alter=27;                   //einzelnen Elemente der Struktur.
4 cout << p1.name << ": " << p1.alter;
```

11 Klassen

```
class  person                   //Klassen entsprechen prinzipiell den Strukturen.
{                               //Eine Klasse ist sozusagen auch ein neuere Variablentyp, die hier
   private:                     //Eigenschaften genannt werden auch Funktionen,die hier
      string name;              //Methoden genannt werden.
      int alter;
   public:                      //Es gibt 2 Aufteilungen:
      void init (string n, int a)
      {                         //private:  Zugriff nur von Methoden der eigenen Klasse
            name = n;           //public:   Zugriff von außen möglich
            alter = a;
      }                         //In der Regel stehen die Eigenschaften in private,
      void anzeige()            //die Methoden in public. Damit wird eine Datenkapselung
      {                         //bewirkt, d.h., ein Zugriff von außen auf die Daten ist
            cout << "Name:" <<  //nicht direkt möglich, sondern nur unter Verwendung der Methoden.
                name << endl;   //"Klassen verwalten sich mit Hilfe der Methoden selbst!"
            cout << "Alter:" << alter;
      }
};
```

Benutzung:

```
person p1;                      //Zugriff ist wieder möglich mit dem Punktoperator.
                                //p1 ist ein Objekt der Klasse person.
p1.init ("Otto", 17);
p1.anzeige();
```

11.1 Überladener Konstruktor

```
//Klasse person von oben wird im Bereich public ergänzt um:

      person (string n, int a) //Konstruktoren werden beim anlegen eines Objektes aufgerufen.
      {                         //Sie können nicht direkt (z.B. über den Punktoperator)
            name = n;           //aufgerufen werden.
            alter = a;
      }                         //Konstruktoren haben den gleichen Namen wie die Klasse und
      person () {  }            //haben keinen Ergebnistyp. Es können beliebig viele
                                //Konstruktoren angelegt werden, sie unterscheiden
                                //sich durch die Parameter.

      ~person()  { }            //Der Destruktor wird beim Löschen eines Objektes aufgerufen
```

Benutzung:

```
person p2("Friedel", 78);       //Beim Erstellen eines Objektes werden die Eigenschaften mit Werten
    versehen.
```

11.2 Vererbung

```
class  women : public person    //Wenn ein Objekt über die gleichen Eigenschaften eines anderen
{                               //Objektes verfügen soll, muss hinter der Klasse ein : public
   .....                        //Klassenname von der geerbt werden soll
}                               //Die Klasse besitzt nun Attribute und Methoden der Hauptklasse

class person
{
   protected:                   //Um auf die privaten Attribute der Hauptklasse zu greifen zu können
      ...                       //müssen sie als protected deklariert werden (nicht mehr private)
   public:
      ...
}
```

12 Dateihandling

```
#inlcude <fstream>              //Bibliothek für Dateihandling

ofstream outClientFile( "oFile.txt", ios::out ); //Output Datei
ifstream inClientFile( "iFile.txt", ios::in );   //Input Datei

if(!inClientFile)               //Überprüft ob Datei geöffnet werden konnte
     cout << "Konnte Datei nicht oeffnen" << endl;
else
{
string dummy;
do
{
getline(inClientFile, dummy, '\n'); //Ließt eine komplette Zeile in dummy ein
outClientFile << dummy << endl;     //Schreibt dummy in eine Datei (ähnlich wie cout)
}while(!inClientFile.eof());        //Überprüft ob die Datei zu Ende ist

outClientFile.close();              //Dateien sollten immer geschlossen werden
inClientFile.close();
}
```

Alternatives Einlesen

```
inClientFile >> zahl1 >> zahl2;    //Wenn die Datentypen bekannt sind und die Datei festen
                                   //Konventionen unterliegt, so können sie direkt in die
                                   //Variable eingelesen werden
```