

Grundlagen Programmierung

(Praxis Programmierung mit C++)

# **Inhalte Teil I**

* Datentypen & Variablen
* Kontrollstrukturen
  + Verzweigungen ( if … else … )
  + Schleifen ( while … do … )
* Komplexe Datentypen
* Design
* Programmablaufplan
* Nassi-Shneiderman Diagramm
* Pseudo Code

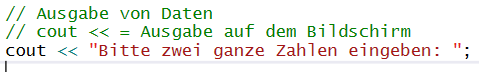
# **Inhalte Teil II**

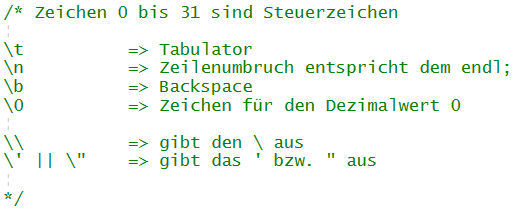
* Funktionen
* Bibliotheken
* Fehlerbehandlung

# **Klausur**

Termin: 04.12.2020  
Dauer: ca. 3 Std.  
Themen: Inhalt Teil I & Teil II

# **Kommentare**

**// Zeilen Kommentar**Nach dem Kommentar endet die Zeile & der Kommentar  


**/\* Block Kommentar \*/**  


# **Trennzeichen**

Trennzeichen dienen dazu Zeichenketten zu trennen oder zu verknüpfen.  
Auch schließen diese eine Zeile oder ein Codeabschnitt ab.

* Leerzeichen
* Semikolon ; | Kommando Ende
* Kommentare
* Operatoren
  + Z.b. + - \* / (Mathematische Operatoren)
* Komma

# **Bezeichner**

Die Bezeichner sind die Namen von Klassen, Funktionen oder Variablen.  
Sie können Alphanumerisch sein, müssen aber bestimme Regeln einhalten.

* Ziffern, Buchstaben & Unterstriche sind erlaubt  
  Sonderzeichen, Umlaute sind nicht erlaubt
* Das erste Zeichen darf keine Nummer sein   
  int 0Null ist nicht erlaubt  
  int Null0 ist erlaubt  
  int \_0Null ist erlaubt
* Groß- & Kleinschreibung wird unterschieden  
  int Null und int null sind zwei verschiedene Variablen
* Bezeichner dürfen keine Schlüsselwörter sein

# **Programmiersprachen**

Es gibt viele verschiedene Programmiersprachen. Jede Programmiersprache hat seine eigenen Schlüsselwörter und Syntax. Die Schlüsselwörter und Syntax sind teilweise in verschiedenen Programmiersprachen ähnlich.

|  |  |
| --- | --- |
| **Übersetzte Programmiersprachen** | **Interpretierte Programmiersprachen** |
| C / C++, C#, Fortan, Cobol, Visual Basic | PHP, JavaScript, PowerShell, Python,  Visual Basic Script (VBS) |
| Explizierte Datentypen   * Datentypen müssen FEST zugeordnet werden   Int, char, float, long, string, …. | Implizierte Datentypen   * Datentypen werden anhand des Wertes zugeordnet   123 = int 12.3 = double “123“ = text |

# **Zusammensetzung eines Programms**

**\*.h**  
**\*.cpp**  
Header Datei   
Quellcode / Programmcode

**COMPILER  
Verwandelt Header & Quellcode  
in ein Objekt um**

**\*.obj**

**BINDER  
(Deutsch: Linker)**

**\*.lib**    
Bibliotheken

**\*exe**  
Ausführbare Datei

# **Datei Aufbau**

Eine Datei ist immer in mehreren “Abschnitte“ unterteilt.

Compiler Anweisungen

\*cpp  
Quellcode / Programmcode

Definitionen  
Deklarationen

Klassen  
Hauptfunktionen

**Compiler Anweisung**

Eine Compiler-Anweisung ist eine in den Quelltext eingefügte Steueranweisungen für den Compiler. Der Umfang und die Syntax von Compiler-Anweisungen sind abhängig von der Programmiersprache und teilweise auch vom Compiler

**Definition / Deklarationen**

Deklaration ist eine Festlegung von Dimension, Bezeichner, Datentyp und weiteren Aspekten einer Variablen oder eines Unterprogramms

**Klassen / Hauptfunktionen**

Eine Funktion (englisch function) ist in verschiedenen höheren Programmiersprachen die Bezeichnung eines Programmkonstrukts, mit dem der Programm-Quellcode strukturiert werden kann, so dass Teile der Funktionalität des Programms wiederverwendbar sind.  
  
Unter einer Klasse (auch Objekttyp genannt) versteht man in der objektorientierten Programmierung ein abstraktes Modell bzw. einen Bauplan für eine Reihe von ähnlichen Objekten.

# **Nassi-Shneiderman Diagramm**

Ein Nassi-Shneiderman-Diagramm ist ein Diagrammtyp zur Darstellung von Programmentwürfen im Rahmen der Methode der strukturierten Programmierung.

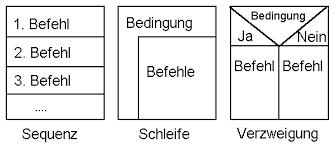


Abbildung 1 | Nassi Shneiderman Blöcke

# **Programmablaufplan (PAP)**

Ein Programmablaufplan ist ein Ablaufdiagramm für ein Computerprogramm, das auch als Flussdiagramm oder Programmstrukturplan bezeichnet wird. Es ist eine grafische Darstellung zur Umsetzung eines Algorithmus in einem Programm und beschreibt die Folge von Operationen zur Lösung einer Aufgabe.

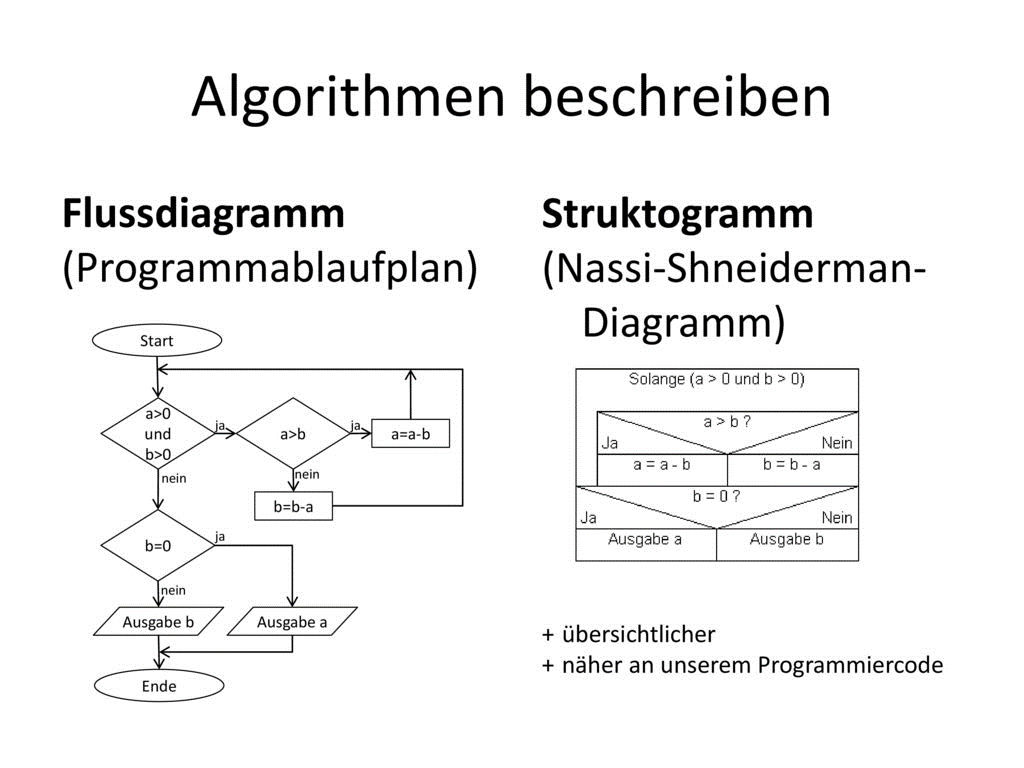


Abbildung 2 | Links PAP - Rechts Nassi-Shneidman Diagramm

# **Datentypen**

Datentypen definieren den Typ von Daten (Werten) der gespeichert werden kann.

|  |  |  |
| --- | --- | --- |
| Name | Wert | Größe |
| bool | true / false | 1 Byte |
| char | -128 bis 127 | 1 Byte |
| Ganze Zahlen z.B. 123 | | |
| short | −32.768 bis 32.767 | 2 Byte |
| long | –2.147.483.648 bis 2.147.483.647 | 4 Byte |
| long long | –9.223.372.036.854.755.808 bis 9.223.372.036.854.755.807 | 8 Byte |
| Komma Zahlen z.B. 1.23 | | |
| float | 1.2E-38 bis 3.4E+38 | 4 Byte |
| double | 2.3E-308 1.7E+308 | 8 Byte |
| C++ Only | | |
| long double | 3.4E-4932 1.1E+4932 | 12 Byte |

# **Modifizierer**

Dienen dazu Datentypen zu bearbeiten

|  |  |
| --- | --- |
| Name | Beschreibung |
| const | Konstante | der Wert des Datentyps kann nicht verändert werden |
| signed | Datentyp mit Vorzeichen | Standard |
| unsigned | Datentyp ohne Vorzeichen |

# **Variablen**

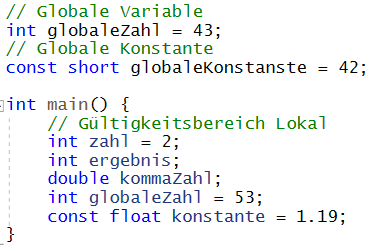
Syntax:

*Modifizierer + Datentyp + Bezeichner = Wert;*

Modifizierer und Wert sind Optional. Sollte der Modifizierer **“const“** angeben werden, so ist der Wert zwingend erforderlich.

**int** *zahl* = 0; **unsigned char** *zeichen* = 255;  
**long** *longzahl* = 0; **bool** *logisch* = true  
**short** *shortzahl* = 0; **float** *kommazahl* = 3.14;  
  
**const** **double** *pi* = 3.14;  
Der Wert *pi* muss direkt angegeben werden.

# **Gültigkeitsbereich**

Der Gültigkeitsbereich gibt einen Bereich an wo eine Variable sichtbar ist.  
Es gibt den Globalen und den Lokalen Gültigkeitsbereich.  
  
**Globale Sichtbarkeit**Eine Variable die außerhalb von { } definiert werden sind Global und von überall in der Datei zur Verwendung frei gegeben.  
  
**Lokale Sichtbarkeit**  
Wird eine Variable innerhalb von { } definiert, so ist diese nur in den vorher bestimmten teil des Quellcodes gültig und verwendbar.  
  


**Hierarchie**Variablen folgen einer bestimmten Hierarchie.  
 **Global -> Lokal**Sollte eine Globale Variable Lokal nochmal neu definiert werden, so zählt die Lokale Variable für den Programm abschnitt.

# **Implizierte / Explizierte Umwandlung**

Bei Umwandlung von Datentypen geht es nur von dem aktuellen Datentyp auf den nächst kleineren (Implizierte Umwandlung) oder auf den nächst größeren (Explizierte Umwandlung) Datentyps  
  
**Impliziert**von klein auf großbool -> char -> short -> …

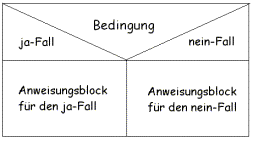
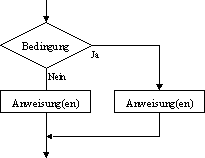
**Expliziert**von groß auf kleinlong double -> double -> …

long long -> long -> … -> bool

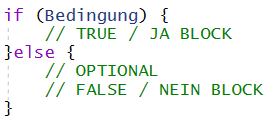
# **Kontrollstrukturen**

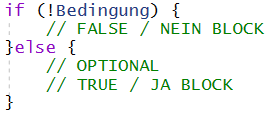
Mit Kontrollstrukturen lassen sich Variablen mit einer Bedingung prüfen. So kann ein Codeblock erst durchgeführt werden, wenn die Überprüfung TRUE oder FALSE ist.  
Eine weitere Kontrollstruktur sind die Schleifen mit denen wir ein Codeblock solange durchlaufen lassen können bis die Bedingung erfüllt wurde.  
  
  
**Verzweigung**

* Einfache Verzweigungen
  + Ja / Nein Entscheidung
  + Operatoren Entscheidung

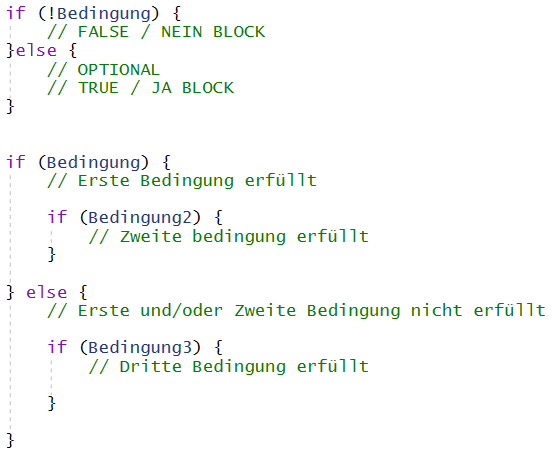
**Syntax:**

  
Um nur den **FALSE / NEIN** Block zu erhalten, muss die Bedingung mit einem **!** negiert werden.

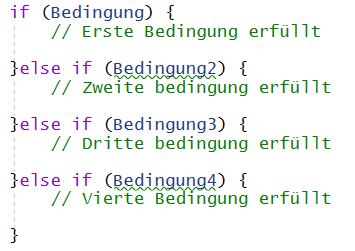


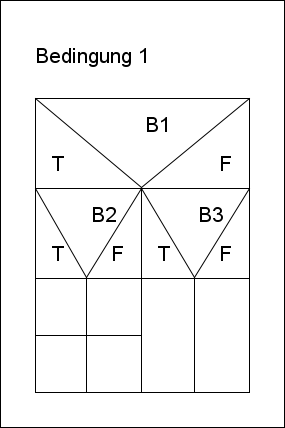
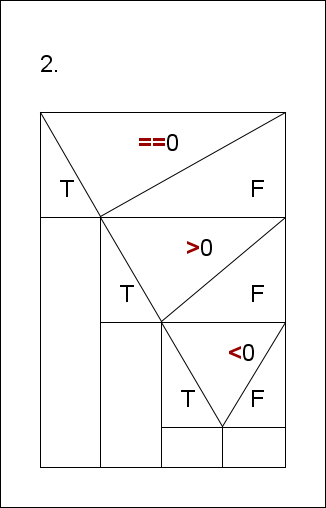
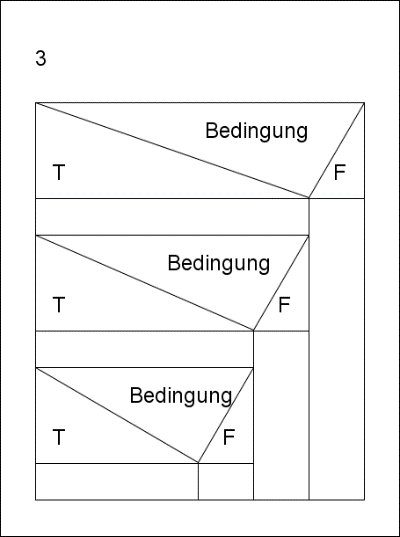
* Komplexe / Mehrfache Verzweigungen

**if else**

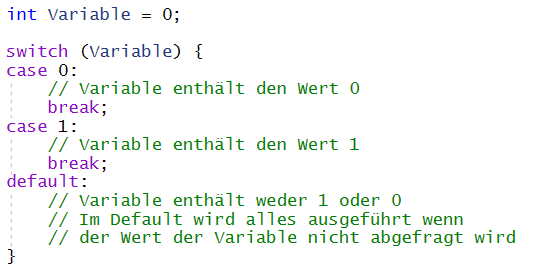


**If else if**

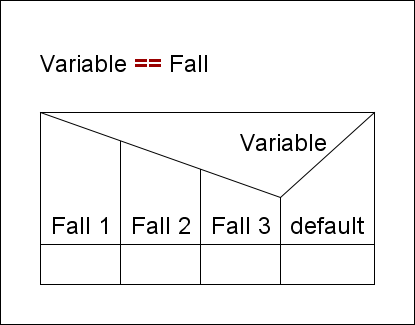


**Switch**

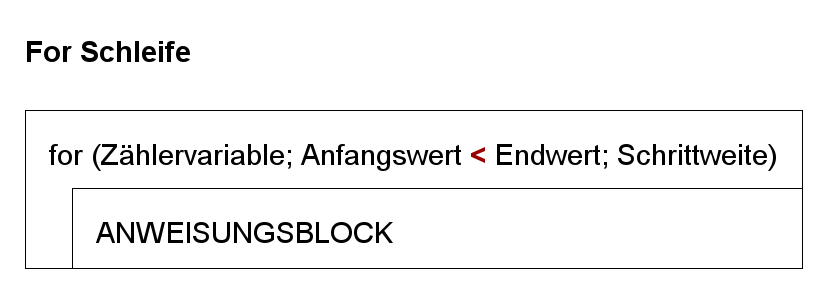


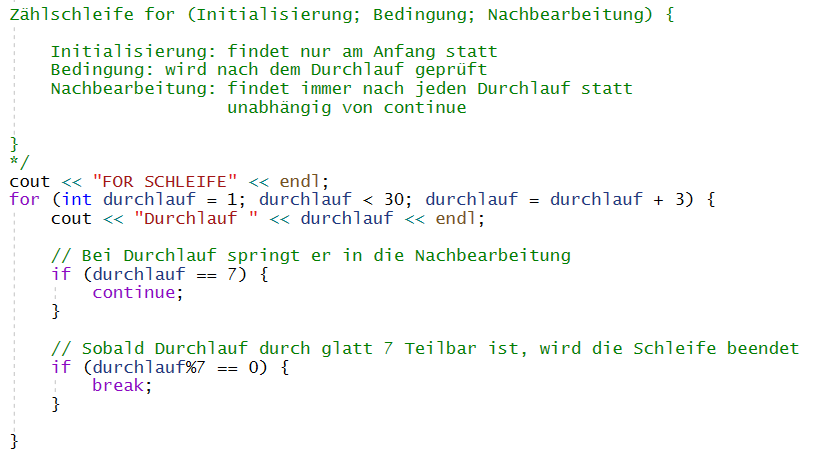
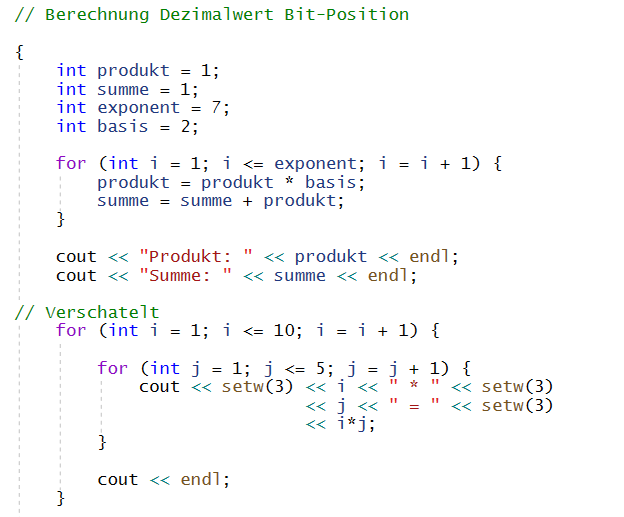
Die Variable zur Überprüfung muss Nummerisch sein, damit diese im **CASE** (Fall) abgefragt werden kann. Sollte die Variable mit dem des **CASE** übereinstimmen, wird der Codeblock bis zum **BREAK**; ausgeführt.  
Es wird immer bis zum nächsten **BREAK**; abgearbeitet, auch wenn dieser erst im nächsten **CASE** steht.

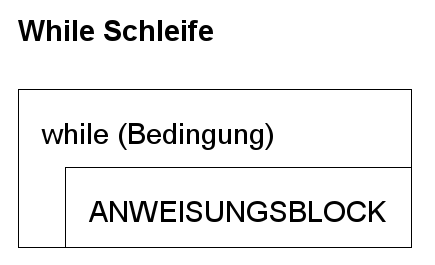
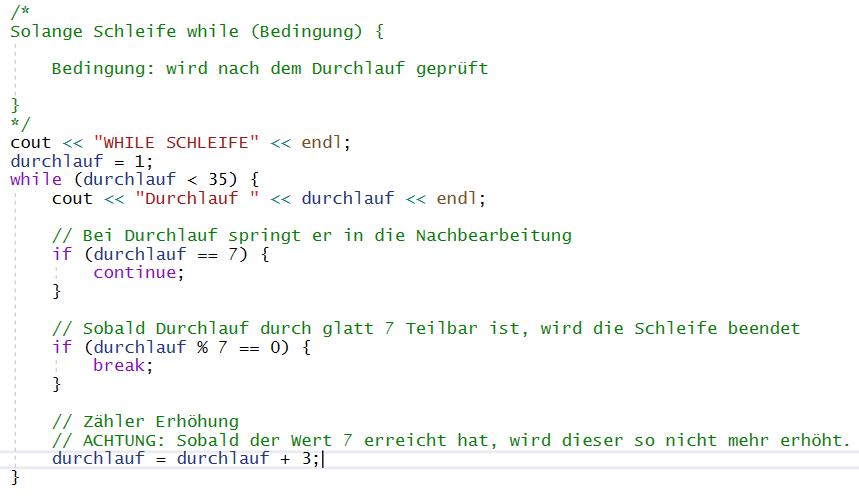


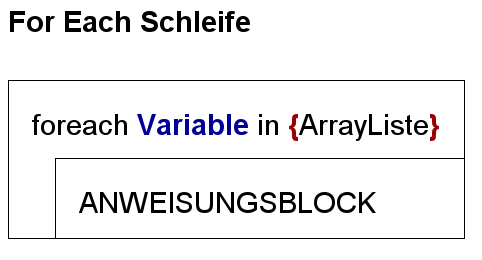
**Schleifen**

* Kopfgesteuert
  + Zählschleife / for
  + so lange bis / while
  + über eine Liste / for each

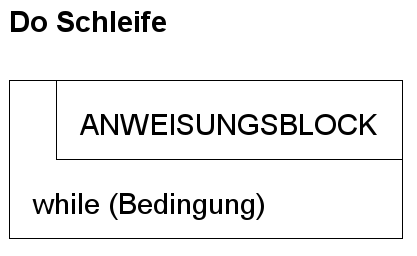


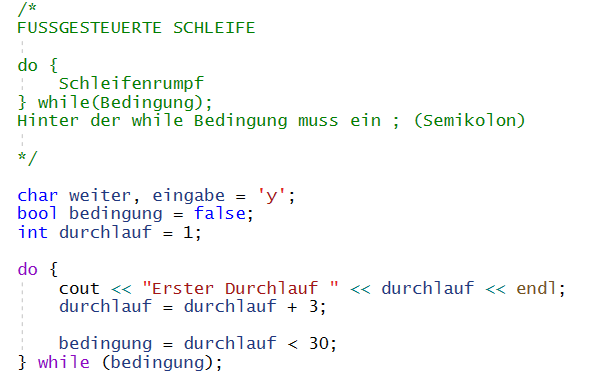
  


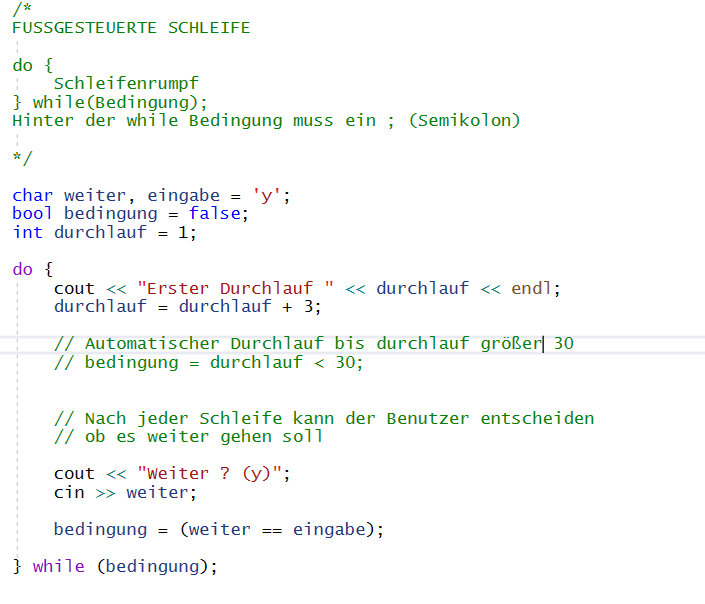
  




* Fussgesteuert
  + so lange bis







**Bedingungen**Die Bedingungen werden genutzt um Entscheidung bei den Verzweigungen & Schleifen fest zu legen.

* Einfache Bedingung
  + Wert Operator Vergleichswert  
    3 < 4 || “Hallo“ == “Guten Tag“  
    Gibt TRUE oder FALSE zurück

**Komplexe Bedingungen**Komplexe Bedingungen werden mit Hilfe eines Operators “verknüpft“. So kann man Prüfen ob die eine Bedingung erfüllt ist oder eine andere oder ob alle Bedingungen erfüllt oder nicht erfüllt wurden.

* Logische Operatoren
  + **||** ODER
  + **&&** UND
  + **!** NICHT

**Syntax:**

Bedingung1 Operator Bedingung2

**Operatoren**Damit die Bedingung auch weiß was und wie sie etwas vergleichen soll, gibt es die Operatoren.

* Einfache Vergleichsoperatoren
  + **<** größer als
  + **>** kleiner als
  + **<=** größer oder gleich als
  + **=>** kleiner oder gleich als
  + **==** genau gleich
  + **!=** nicht gleich

**Konditionaloperator**Es ist eine vereinfachte **if-Anweisung** die aber direkt einer Variablen zugewiesen oder im Output verwendet werden kann.





Die Ausgabe hier wäre: *Wert ist: gleich 0*  
Wäre die Variable nicht 0 würde in der Ausgabe stehen: *Wert ist: ungleich 0*

# 

# **Zeichenketten / Listen**

Listen enthalten Variablen/Elemente des gleichen Datentyps.

**Syntax**:  
Modifizierer Datentyp Bezeichner[maxAnzahl] = { Werteliste }

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **ELEMENTE** | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|  |  |  |  |  |  |  |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **POSITION** | | | | | | | |

**Position in Liste**  
0 – maxAnzahl -1

* maxAnzahl = maxAnzahl an Elemente in der Liste  
  bsp.: maxAnzahl = 8 | Maximal 8 Elemente in der Liste

**Element in der Liste**Anzahl der Werte darf nicht größer sein als maxAnzahl  
““ = Zeichenkette  
‘‘ = Zeichen

**Werteliste**Werteliste { Kommagetrennt }

* = { } | alle Elemente werde mit 0 erstellt
* = { 1, 2 } | Element 1 und 2 bekommen den Wert 1 und 2, der Rest bekommt 0

**Zugriff**Anzeigen eines Listenelements

Syntax**:**

* Bezeichner[Pos]

Bsp.:

* cout << Bezeichner[Pos] << endl;

Anzeigen von allen Listenelemente

Bsp.:

* for (int pos = 0; pos < maxAnzahl, pos = pos +1) {  
  cout << Bezeichner[pos];  
  }

**Ändern**Ändern eines Listenelements

Syntax:

* Bezeichner[Pos] = 7;

# **Felder / Listen**

Multidimensionale Felder & Listen.

Datentyp Bezeichner[maxAnzahl][maxAnzahl] ….. [maxAnzahl] = { Werteliste };

**Syntax:**

char Monatsnamen[12][12] = {“Januar“,“Februar“,“Maerz“,“April“,“Mai“,“Juni“};

* Erstellt eine char List mit dem Bezeichner Monatsnamen
* Die erste Liste hat maxAnzahl = 12
* Die zweite Liste hat maxAnzahl = 12
* Die erste Liste wird mit der Zeichenkette der Monatsnamen befüllt
* “ | Wird benutzt für Zeichenketten
* ‘ | Wird benutzt für einzelne Zeichen

**Speicherabbild:**

Datentyp Bezeichner [2][2] = { {1,2} {3,4} };

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
| 1 | 3 | 4 |
|  | 0 | 1 |

**Einlesen von Nummerischen Werte:**  
// Konstante MaxElemente für die Liste  
const int MAXELEMENT = 4;

// Nummerische Liste

// { 1, 2 } Rest wird mit 0 belegt

// { 1, 2, 3, 4 } Jedes Element bekommt seinen eigenen Wert

// { 0 } Jedes Element bekommt den Wert 0

int listNumber[MAXELEMENT] = {0};

// Einlesen der Liste per UserInput

for (int i = 0; i < MAXELEMENT; i = i + 1) {

cin >> listNumber[i];

}

**Einlesen von Zeichenketten:**char Buchstabenkette[30] = "AB";

// Einlesen von maximal 29 Zeichen ohne Leerzeichen

// cin.width(MAXANZAHL)

cin >> Buchstabenkette;

cin.width(30);

// Ausgabe der Zeichenketten

cout << "Liste: " << Buchstabenkette << endl;

// Einlesen von maximal 29 Zeichen mit Leerzeichen

// cin.getline(LISTE,MAXANZAHL)

cin.getline(Buchstabenkette, 30);  
// löscht alle zuvor gesetzten Errorbits.

cin.clear();

cin.sync();

// Auslesen der Anzahl an eingebenden Zeichen

cout << "Zeichen Eingegeben: " << cin.gcount() << endl;

// Ausgabe der Zeichenketten

cout << "Liste: " << Buchstabenkette << endl;

**Ändern von Werten:**Datentyp Bezeichner [2][2] = { {1,2} {3,4} };

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
| 1 | 3 | 5 |
|  | 0 | 1 |
| Änderung im Speicherabbild | | |

Bezeichner[1][1] = 5;

// Ändern von Werten

listNumber[2] = 42 \* listNumber[1];

listNumber[2] = 4 + listNumber[1];  
Buchstabenkette[0] = ‘A‘;

**Ausgabe von Werten:**// Ausgabe ALLER Nummerischen Listeneinträge

for (int Element : listNumber) {

cout << "ELEMENT: " << Element << " ";

}

// Zeichen/Ketten Liste

char Buchstaben[2] = { 'A','B' };

char Buchstabenkette[30] = "AB";

// Ausgabe von Buchstabenlisten

cout << "Liste Buchstabenkette: " << Buchstabenkette << endl;

cout << "Liste Buchstaben: " << Buchstaben << endl;

**Zählen von Elementen:**// Zählen der Elemente in Liste mit Schleife

int Laenge = 0;

while (Buchstabenkette[Laenge] != '\0') {

Laenge = Laenge + 1;

}

cout << "Laenge von Buchstabenkette: " << Laenge << endl;

**Verändern der Ausgabe von Elementen:**// Rückwerts Ausgeben von Elementen  
cout << "Liste Rueckwarts ausgeben\n";

for (int i = Laenge; i >= 0; i = i - 1) {

cout << Buchstabenkette[i];

}

cout << endl;

// Alternative

for (int i = 0; i < Laenge; i = i + 1) {

cout << Buchstabenkette[Laenge - 1 - i];

}

cout << endl;

**Vertauschen von Elementen**  
// Vertauschen von Elementen

for (int i = 0; i < Laenge / 2; i = i + 1) {

char temp = Buchstabenkette[i];

Buchstabenkette[i] = Buchstabenkette[Laenge - 1 - i];

Buchstabenkette[Laenge - 1 - i] = temp;

}

cout << Buchstabenkette << endl;

**Verknüpfen von Zeichenketten**

// Verknüpfen von Zeichenketten

char Text1[40] = "Hallo, ";

char Text2[5] = "Welt";

for (int i = 0; i <= 5; i = i + 1) {

Text1[7+i] = Text2[i];

}

cout << Text1 << endl;

**Kopieren von Zeichenketten**

// Kopieren von Zeichenketten

for (int i = 0; i <= 5; i = i + 1) {

Text1[i] = Text2[i];

}

cout << Text1 << endl;  
  
  
**Multidimensional**  
char Monatsnamen[12][12] = { "Januar", "Februar", "Maerz", "April" };

**Ausgabe**

// Ausgabe = MAERZ

cout << Monatsnamen[2] << endl;

// Gibt nur den 4ten Buchstaben aus

cout << Monatsnamen[2][3] << endl;

// In der For-Schleife werden erst alle 12 Inhalte von der ersten Liste geloopt

// In der zweiten For-Schleife werden jetzt nur die ersten 2 Buchstaben ermittelt  
// und Ausgegeben

for (int i = 0; i < 12; i = i + 1) {

for (int j = 0; j < 2; j = j + 1) {

cout << Monatsnamen[i][j];

}

cout << endl;

}

**Ändern**

// Ändert bei Maerz das große M durch ein kleines M

Monatsnamen[2][0] = 'm';

cout << Monatsnamen[2] << endl;

# **Benutzerdefinierte Datentypen**

**Aufzählungen mit enum**

enum Bezeichner { WERTELISTE } ;

Bsp:

enum geschlecht { m, w, d };

**Struktur**

struct Bezeichner { ELEMENTLISTE };

Bsp.:

struct person {

char nachname [40],

geschlecht Geschlecht,

int id

};

**Erstellen mit Inhalt**

Structname Bezeichner { ELEMENTLISTE };

person Maier = { “Maier“, m, 1 };

nachname, geschlecht, id

**Ändern**Bezeichner Elementname = element.WERT;

Maier.Geschlecht = geschlecht.d;

# **Funktionen**

Deklaration = Bekanntgeben  
Definition = Funktionalität

Modifizierer Rückgabetyp Bezeichner (‘Parameterliste‘);

**Rückgabetyp**

* Elementare Datentype
  + int, double, ect.
* Datentyp \*
  + Adresse in der Speicherstelle
* Datentyp &
* void

**Parameterliste**

* Parameter mit kommagetrennt
* Parameter = Variablen Definition
* call by value  
  Kopie der Parameterwerte
* call by value with reference  
  Adresse einer Speicherzelle übergeben
* call by reference  
  Verweis auf den originalen Speicherplatz / Aliasname

**Parametertypen**

* elementare Datentypen
  + int, double, float, ect.
* Datentyp \*
* Datentyp &
* Datentyp Bezeichner []

**Aufruf**

Bezeichner(WERTELISTE);  
Werteliste = Variablen

**BEISPIEL**

// Variablen Definieren  
int zahl = 10;  
int zahlen[2] = { 1,2 };

// Aufruf der Funktion mit den Variablen  
berechne(zahl, zahlen);

// Funktion Deklaration mit Definition  
int berechne(int pzahl, int pzahlen[]) {

int sum;

sum = pzahl + pzahlen[];

return sum;

}