1

Einführung in C++

und

das objektorientierte Programmieren

(OOP)

2

- Typische Aufgaben für den Rechner

  Addition zweier Zahlen a und b.

  Steuerung einer Ampelanlage.

  Überweisung von einem Konto auf ein anderes.

  Simulation einer biochemischen Reaktion.

Simulation der realen Welt durch Abbildung in Rechner.

Breymann_Folien

3

Simulation der realen Welt durch Abbildung in Rechner.

Virtuelle Welt (OOP)

– Objekte im Sinne des OOP sind Abstraktionen von Dingen der realen Welt.

– Objekte bestehen aus

- Daten = Attribute = Eigenschaften
- Operationen = Methoden = Funktionen

4

Simulation der realen Welt durch Abbildung in Rechner.

Virtuelle Welt (OOP)

– Objekte mit den gleichen Eigenschaften (Daten) und Operationen werden zu Klassen zusammengefasst.

– Beispiel: Beschreibung eines Objekts „Konto“:

- Daten oder Attribute
  - Inhaber: Folge von Buchstaben
  - Kontonummer: Zahl
  - Betrag: Zahl
  - Dispo-Zinssatz: Zahl
- Operationen
  - überweisen(Ziel-Kontonummer, Betrag)
  - abheben(Betrag)
  - einzahlen(Betrag)

5

Simulation der realen Welt durch Abbildung in Rechner.

Virtuelle Welt (OOP)

– Tatsächliche Konten K1 und K2 enthalten konkrete Daten.

Attribut	Werte für Konto K1	Werte für Konto K2
Inhaber	Roberts, Julia	Constantine, Eddy
Kontonummer	12345678	98765432
Betrag	-200,30	1222,88
Dispo-Zinssatz	13,75	13,75

– Die tatsächlichen Objekte heißen auch Instanzen einer Klasse.
– Julia will Eddy 1000 € überweisen. Dem Objekt K1 wird also der Auftrag mit den folgenden Daten mitgeteilt:

```
K1.überweisen(98765432, 1000);
```

6

- Typische Aufgaben für den Rechner

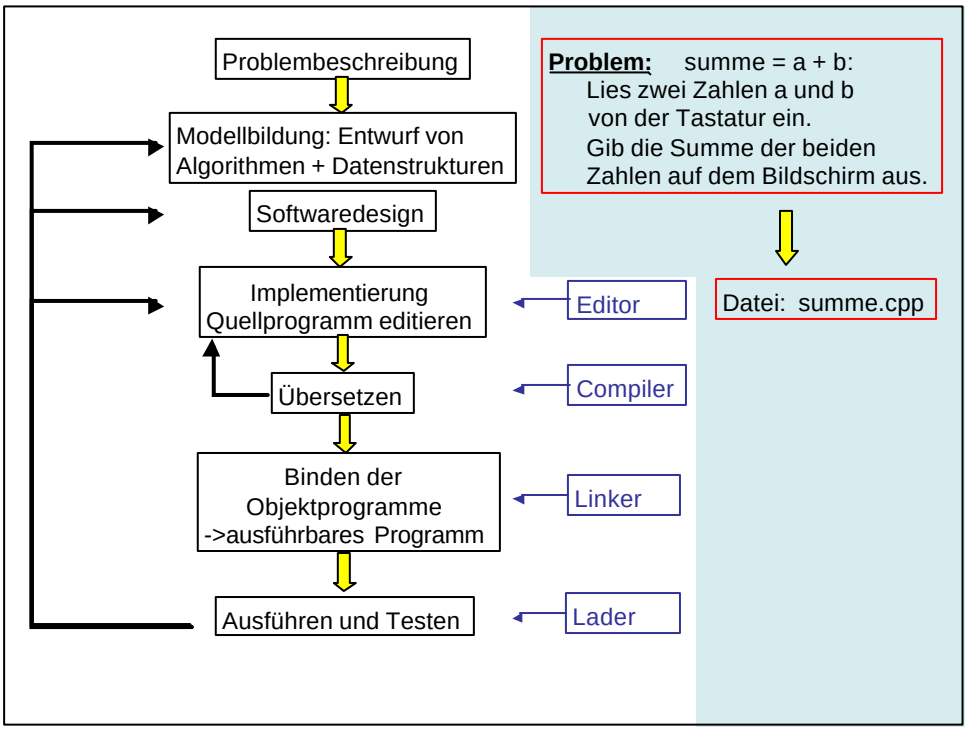
⇨ Addition zweier Zahlen a und b.

Steuerung einer Ampelanlage.

Überweisung von einem Konto auf ein anderes.

Simulation einer biochemischen Reaktion.

Simulation der realen Welt durch Abbildung in Rechner.



8

Unser erstes Programm!

Wir öffnen mit einem Editor eine neue Datei mit dem Namen „summe.cpp“ und schreiben unser erstes Programm (siehe Breymann Folien Seite 16).

Das Grundgerüst eines (Haupt)Programmes

9

```
int main( )
```

main

- ist das Hauptprogramm

main( )

- In der runden Klammer können Parameter an das Hauptprogramm übergeben werden.

int main( )

- Das Hauptprogramm kann nach Beendigung eine ganze Zahl (Datentyp int) an das Betriebssystem zurückgeben.
- Bei ordnungsgemäßem Programmablauf wird die Zahl 0 zurückgegeben.

Das Grundgerüst eines (Haupt)Programmes

10

```
int main( )
{




}
```

int main ( ) { .....}

- Der zu dem Programm gehörende Programmcode wird durch die geschweiften Klammern { und } eingeschlossen.

- Ein mit { und } begrenzter Bereich heißt Block.

Das Grundgerüst eines (Haupt)Programmes


```
/* Dieses Programm berechnet
   die Summe von zwei Zahlen a,b.  */


int main( )
{
    /* Reserviere Speicherplatz
       für die Variablen bzw. die Objekte. */


    // Lies die Zahlen a und b ein



    // Berechne die Summe beider Zahlen


    // Zeige das Ergebnis auf dem Bildschirm


}
```

- Kommentare: Passende und geeignete Kommentare sind für die Lesbarkeit von Programmen von zentraler Bedeutung.

  /* Diese Art von Kommentar kann beliebig viele Zeilen lang sein und endet mit der nächsten Zeichenfolge der Form */

  // K. bis zum Zeilenende

- Kommentare werden vom Präprozessor eliminiert.

Das Grundgerüst eines (Haupt)Programmes


```
/* Dieses Programm berechnet
   die Summe von zwei Zahlen a,b.  */
#include<iostream>

int main( )
{
    /* Reserviere Speicherplatz
       für die Variablen bzw. die Objekte. */


    // Lies die Zahlen a und b ein



    // Berechne die Summe beider Zahlen


    // Zeige das Ergebnis auf dem Bildschirm


}
```

#include<????>

- Einbindung von vorhandenen Programmen (Header).
- Textueller Import von externen „Header-Dateien“.
- Inhalt der Header-Datei ersetzt diese Zeile.
- Zeilen, die mit # beginnen, enthalten Präprozessor-Anweisungen.

#include<iostream>

- Einbindung der Ein-/Ausgabefunktionen.

Das Grundgerüst eines (Haupt)Programmes


```
/* Dieses Programm berechnet
   die Summe von zwei Zahlen a,b.  */
#include<iostream>
using namespace std;
int main( )
{
    /* Reserviere Speicherplatz
       für die Variablen bzw. die Objekte. */

    // Lies die Zahlen a und b ein


    // Berechne die Summe beider Zahlen


    // Zeige das Ergebnis auf dem Bildschirm


}
```

using namespace std;

- Der Namensraum „std“ wird benutzt.
- Alle Symbole der Standard C++ Bibliothek sind im Namespace „std“ definiert.
- Schreiben Sie diese Zeile in jedes Programm an diese Stelle.
- Genauere Erklärung folgt später.

Das Grundgerüst eines (Haupt)Programmes


```
/* Dieses Programm berechnet
   die Summe von zwei Zahlen a,b.  */
#include<iostream>
using namespace std;
int main( )
{
    /* Reserviere Speicherplatz
       für die Variablen bzw. die Objekte. */
    int summe, a, b;

    // Lies die Zahlen a und b ein


    // Berechne die Summe beider Zahlen


    // Zeige das Ergebnis auf dem Bildschirm


}
```

int summe, a, b;

- Deklaration und Definition von 3 Variablen vom Datentyp „int“ (ganze Zahl der Größe 32 Bit).
- Der Compiler reserviert für jede dieser Variablen einen Speicherplatz der Größe 32 Bit.
- Mittels der Namen können wir auf diese Speicherplätze zugreifen und die dort gespeicherten Daten ändern.
- Weitere Möglichkeit:

```
int summe;
int a;
int b;
```

Das Grundgerüst eines (Haupt)Programmes


```
/* Dieses Programm berechnet
   die Summe von zwei Zahlen a,b.  */
#include<iostream>
using namespace std;
int main( )
{
    /* Reserviere Speicherplatz
       für die Variablen bzw. die Objekte. */
    int summe, a, b;

    // Lies die Zahlen a und b ein
    cout << " a und b eingeben : ";


    // Berechne die Summe beider Zahlen


    // Zeige das Ergebnis auf dem Bildschirm
    cout << " Summe = " << summe;

}
```

cout << " a und b eingeben :";

- cout : iostream-Objekt, Standardausgabekanal
- Abkürzung für *character out*
- Operator <<
  << = „Schiebe die Daten zur Ausgabe“
- Sollen mehrere Dinge ausgegeben werden, so sind sie durch << zu trennen.

Das Grundgerüst eines (Haupt)Programmes


```
/* Dieses Programm berechnet
   die Summe von zwei Zahlen a,b.  */
#include<iostream>
using namespace std;
int main( )
{
    /* Reserviere Speicherplatz
       für die Variablen bzw. die Objekte. */
    int summe, a, b;

    // Lies die Zahlen a und b ein
    cout << " a und b eingeben : ";
    cin >>  a  >> b;

    // Berechne die Summe beider Zahlen


    // Zeige das Ergebnis auf dem Bildschirm
    cout << " Summe = " << summe;

}
```

cin >> a >> b;

- cin : iostream-Objekt, Standardeingabekanal
- Abkürzung für *character in*
- Operator >>
  >> = „Schiebe die Daten von der Eingabe (Tastatur) in die entsprechenden Variablen“.
- Die Werte von a und b werden von der Tastatur gelesen.
- Sollen mehrere Dinge eingelesen werden, so sind sie durch >> zu trennen.

Das Grundgerüst eines (Haupt)Programmes


```
/* Dieses Programm berechnet
   die Summe von zwei Zahlen a,b.  */
#include<iostream>
using namespace std;
int main( )
{
    /* Reserviere Speicherplatz
       für die Variablen bzw. die Objekte. */
    int summe, a, b;

    // Lies die Zahlen a und b ein
    cout << " a und b eingeben : ";
    cin >>  a  >> b;

    // Berechne die Summe beider Zahlen
    summe = a + b;

    // Zeige das Ergebnis auf dem Bildschirm
    cout << " Summe = " << summe;

}
```

summe = a + b;

- Die Summe von a und b wird berechnet und „summe“ zugewiesen, d.h., unter dem für „summe“ reservierten Speicherplatz gespeichert.

Das Grundgerüst eines (Haupt)Programmes


```
/* Dieses Programm berechnet
   die Summe von zwei Zahlen a,b.  */
#include<iostream>
using namespace std;
int main( )
{
    /* Reserviere Speicherplatz
       für die Variablen bzw. die Objekte. */
    int summe, a, b;

    // Lies die Zahlen a und b ein
    cout << " a und b eingeben : ";
    cin >>  a  >> b;

    // Berechne die Summe beider Zahlen
    summe = a + b;

    // Zeige das Ergebnis auf dem Bildschirm
    cout << " Summe = " << summe;

}
```

summe = a + b

- ist ein so genannter Ausdruck.
- Ein Ausdruck besteht aus Operanden (summe, a, b), die miteinander durch Operatoren (+, =) verknüpft sind.
- Die Definition von Ausdrücken wird später ausführlich behandelt.

Das Grundgerüst eines (Haupt)Programmes


```
/* Dieses Programm berechnet
   die Summe von zwei Zahlen a,b.  */
#include<iostream>
using namespace std;
int main( )
{
    /* Reserviere Speicherplatz
       für die Variablen bzw. die Objekte. */
    int summe, a, b;

    // Lies die Zahlen a und b ein
    cout << " a und b eingeben : ";
    cin >>  a  >> b;

    // Berechne die Summe beider Zahlen
    summe = a + b;

    // Zeige das Ergebnis auf dem Bildschirm
    cout << " Summe = " << summe;
    return 0;
}
```

return 0;

- Das Hauptprogramm liefert nach Beendigung an das Betriebssystem eine ganze Zahl zurück, welche Auskunft über den regulären Ablauf (0) gibt.
- Falls Probleme oder Fehler auftreten, kann man das Haupt-programm auch vorher abbrechen, z.B. mit

return -1;

Übersetzen und Binden des Programmes

- Wir übersetzen das Programm wie folgt:

  g++ -c summe.cpp

- Falls der Compiler keine Fehler meldet, wird die Datei „summe.o“ (Objektcode) erzeugt. Dann können wir das Programm binden:

  g++ -o summe summe.o

- Der Linker erzeugt die ausführbare Datei mit dem Namen „summe“ (unter Windows würde man den Namen „summe.exe“ wählen).

- Übersetzen und Binden kann auch wie folgt starten:

  g++ -o summe summe.cpp

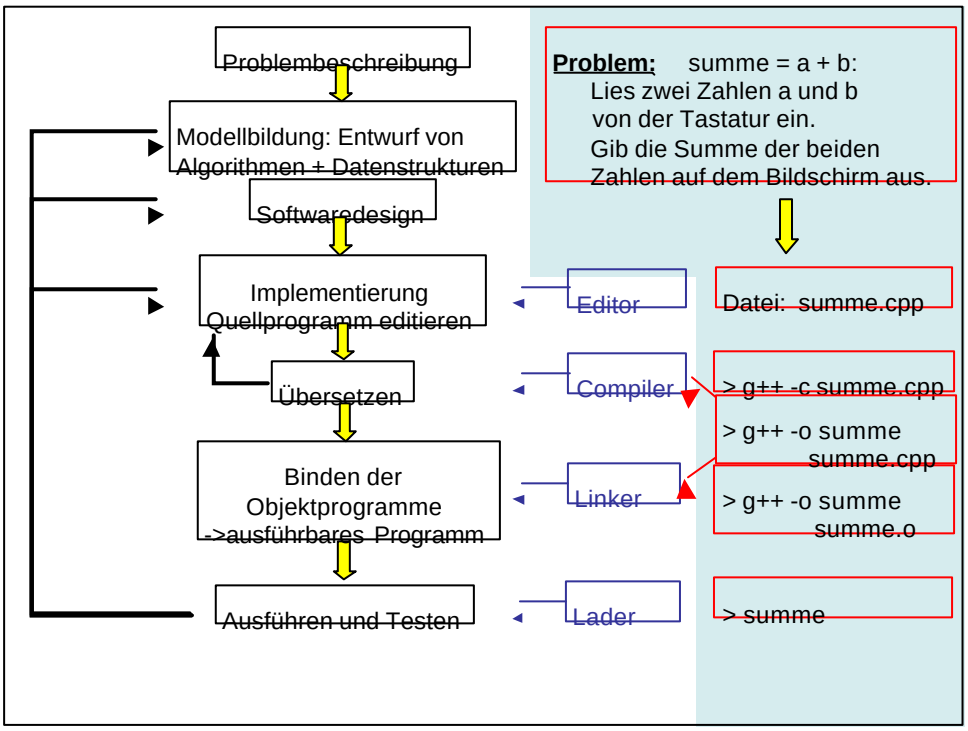
Übersetzen und Binden des Programmes

- Falls Header nicht gefunden werden, dann kann man die Namen der entsprechenden Verzeichnisse wie folgt angeben:

  g++ -I/Pfad_zum_Verzeichnis/include -o summe summe.cpp

- Das Programm kann durch Eintippen von „summe“ gestartet werden.

  > summe
  > a und b eingeben: 10 5
  > Summe = 15
  >



Deklaration und Definition

23

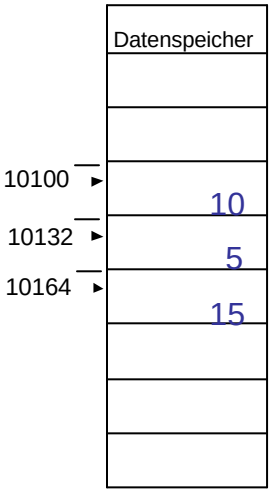
- In Programmen arbeiten wir mit Objekten.
- Durch eine Deklaration führt der Programmierer einen Namen für ein neues Objekt ein und gibt dem Namen eine Bedeutung.
- Der Compiler erstellt eine Tabelle mit allen Objekten, welche die Namen und die Typen (Klassen) der Objekte enthält.
- Eine Deklaration ist auch eine Definition, wenn mehr als nur der Name eingeführt wird, zum Beispiel wenn Speicherplatz reserviert wird (später ausführlicher).

```
int summe, a, b;   /* Diese Deklaration ist auch eine
                      Definition!              */
```

Deklaration und Definition

24

Name	Typ	Adresse
a	int	10100
b	int	10132
summe	int	10164




```
cin >>  a  >>  b;
summe = a + b;
```

Deklaration und Definition


```
int  summe, a, b;

datentyp   variablenname;
datentyp   variablenname = wert;
datentyp   vname1, vname2, ....., vnameX;
```

- Syntax:

```
datentyp   Liste_von_Namen_und_Werten;
```

- Die Liste kann aus nur einem Namen bestehen:

```
Name[=Wert]
```

- Der Initialisierung mit einem Wert ist optional. Der Wert kann auch durch einen Ausdruck des entsprechenden Datentyps angegeben werden.
- Enthält die Liste mehrere Namen (Elemente der Form: Name[=Wert]), so werden diese durch Kommas getrennt.

Deklaration und Definition

- Namenskonvention (siehe Breymann Seite 37):
  - Ein Name ist eine Folge von Zeichen, bestehend aus Buchstaben, Ziffern und Unterstrichen „_“.
  - Ein Name beginnt immer mit einem Buchstaben oder einem Unterstrich (letzteres sollte jedoch vermieden werden).
  - Selbsterfundene Namen sollten nicht mit den vordefinierten Schlüsselwörtern (z.B „main“, „int“ usw.) übereinstimmen.
  - Groß- und Kleinschreibung werden unterschieden:

```
summe    ist nicht dasselbe wie    SuMMe
```

Deklaration und Definition

Regeln für die Verwendung von Variablen:

- Alle Variablen (Namen der Objekte) müssen vor Benutzung deklariert und definiert werden.
- Jede benutzte Variable muss genau einmal im Programm (im Block) definiert werden.
- Definitionen und Deklarationen müssen nicht am Anfang eines Code-Blocks stehen (wie in C).

Empfehlung:

Passende Namen erleichtern das Verständnis des Programms und machen es „lesbarer“.

Einfache Datentypen

- Datentypen für ganze Zahlen:

short	16 Bit	unsigned short
int	32 Bit (16 Bit?)	unsigned int
long	64 Bit (32 Bit?)	unsigned long

- Datentypen für Gleitkommazahlen:

float	32 Bits	$+/- 3.4 * 10^{+/-38}$
double	64 Bits	$+/- 1.7 * 10^{+/-308}$
long double	80 Bits	$+/- 3.4 * 10^{+/-4932}$

- Überschreitet ein Zwischenergebnis einer Folge von arithmetischen Operationen den Zahlenbereich des vorgegebenen Datentyps, so ist das Ergebnis falsch.

Einfache Datentypen

- Für Zeichen, Text und Strings:

char	8 Bit	unsigned char
string	„dynamisch“	#include<string> // Beachte!!!

- Logischer Datentyp:

bool	nur zwei Werte „true“ oder „false“

Einfache Datentypen

- Definition und Initialisierung von Konstanten

```
const  float  pi  =  3.14159254;     // Die Zahl  pi
```

  - Der angegebene Wert kann in dem entsprechenden Programm nicht verändert werden.
  - Auch Variablen können direkt in (nach) der Definition initialisiert werden:

```
int      zaehler = 0;
float    flaeche = 10.0 * 10.0;
string   programm_name(“mein_erstes_Programm“);
```

Ausdrücke

- Ein Ausdruck kann folgende Komponenten enthalten:
  - Operanden (auch Literale)
  - Operatoren
- Einfache Beispiele: a, b, c, summe seien Variablen (int)

```
22                          // Konstante oder Zahl-Literal
12+20
a
a+b
(a+b)*c
summe = a + b + c
“Textliteral“
```

Ausdrücke

- Die Auswertung eines Ausdrucks liefert einen Wert, der an die Stelle des Ausdrucks tritt.
- Im Allgemeinen gelten die Vorrangregeln der Algebra beim Auswerten eines Ausdrucks, inklusive Klammerregeln:
  - Punktrechnung (*, /) vor Strichrechnung (+, -) usw.
- Ist man sich nicht sicher, wie ein Ausdruck ausgewertet wird, so sollte man Klammern verwenden!
- Mehr zu Fragen der Vorrangregeln später!

Kontrollstrukturen: Anweisungen

- Eine Ausdrucksanweisung ist ein Ausdruck gefolgt von einem Semikolon:

```
a = b;
summe = a + b;
```

- Ein Ausdruck repräsentiert nach der Auswertung einen Wert (von einem bestimmten Datentyp).
- Weitere Ausdrucksanweisungen:

```
cin >>  a  >>  b;
cout <<  summe;
```

Kontrollstrukturen: Anweisungen

- Beispiele für Deklarationsanweisungen

```
int summe;
const float pi = 3.1415;
string  text;
```

- Nach Deklarationen sind die Namen im Programm bekannt und können verwendet werden.
- Eine einfache Deklaration wird stets mit einem Semikolon abgeschlossen.

Kontrollstrukturen: Anweisungen

- Eine Verbundanweisung, auch Block genannt, ist ein Paar von geschweiften Klammer { und }, das eine Folge von Anweisungen enthalten:

```
{   }                       // leerer Block

{                           // Block mit einer Anweisung
  Anweisung1
}


{                           // Block mit mehreren Anweisungen
  Anweisung1
  Anweisung2
  .....
}
```

Kontrollstrukturen: Anweisungen

- Eine Kontroll- oder Verzweigungsanweisung erlaubt es, die Ausführung von Anweisungen von Bedingungen abhängig zu machen:

```
Falls          ich heute Abend noch Zeit habe
dann           setze ich noch die Übung auf
andernfalls    werde ich die Übung morgen anfertigen
```

- Die if-Anweisung ermöglicht diese Art von Fallunterscheidungen:

```
if  (Bedingung)  Anweisung1

if (Bedingung)  Anweisung1
else            Anweisung2
```

Kontrollstrukturen: Anweisungen

- Bedingung = Ausdruck vom Typ „bool“
- Auch Blöcke von Anweisungen können in Abhängigkeit von der Bedingung ausgeführt werden:

```
if (Bedingung)
   {
     Anweisung1
     Anweisung2
      ....
    }
else
   {
     Anweisung_else_1
     Anweisung_else_2
     .....
   }
```

Kontrollstrukturen: Anweisungen

- Berechne den Betrag der Differenz von a und b: ||a-b||

```
#include<iostream>
using namespace std;

int main()  {
   int  betrag, a, b;
   cout << “a und b eingeben:“;      // Lies die Zahlen a und b ein
   cin >>  a  >>  b;

   if  (a  > b)   betrag  =  a – b;      //  Berechne den Betrag
   else           betrag  =  b – a;

   cout <<  “ ||a-b|| = “  <<  betrag  << endl;
   return 0;
}
```

Zeilenumbruch

Kontrollstrukturen: Anweisungen

- Umwandlung römischer Ziffern

```
#include<iostream>
using namespace std;

int main()  {
  int  a = 0;                          // Variable für das Resultat
  char c;                              // Variable zum Einlesen der Ziffer
  cout <<  "Ziffer :" ;                //  Eingabeaufforderung
  cin  >> c;                           //  Einlesen der Ziffer

  if  (c == 'I')         a = 1;        // Berechnung der arabischen Zahlen
  else if (c == 'V')  a = 5;
  else if (c == 'X')  a = 10;
  else if (c == 'L')   a = 50;
  else if (c == 'C')  a = 100;
  else if (c == 'D')  a = 500;
  else if (c == 'M')  a = 1000;

  if  (a == 0)  cout << "keine römische Ziffer!" << endl;
  else          cout <<  a  << endl;          // Ausgabe des Resultats
}
```

Kontrollstrukturen: Anweisungen

Die wichtigsten Vergleichsoperatoren

- Seien a und b Variablen oder Ausdrücke des gleichen Datentyps:

==	a == b	true	falls	a gleich b
!=	a != b	true	falls	a ungleich b
>	a > b	true	falls	a größer als b
<	a < b	true	falls	a kleiner als b
>=	a >= b	true	falls	a größer gleich b
<=	a <= b	true	falls	a kleiner gleich b

✍ Jeder Datentyp besitzt eine spezielle Menge von solchen Operatoren.

✍ Siehe Breymann-Folien Seite 22--38.

Kontrollstrukturen: Anweisungen

- Mehrere Bedingungen können durch logische UND- oder ODER-Operatoren kombiniert werden:

&&	Logisches UND
||	Logisches ODER
!	Logische Negation

```
if  ( a > 0 ||  a < 0)   cout <<  " a ist ungleich 0" << endl;
else                     cout <<  " a ist gleich 0"  << endl;

char c = 'B';
bool  grossbuchstabe;

if ( (c >= 'A') && (c <= 'Z'))   grossbuchstabe = true;
else                             grossbuchstabe = false;
```

Kontrollstrukturen: Anweisungen

- If-Anweisungen können natürlich geschachtelt werden:

```
int a, b, c, maximum;
cin >> a >> b >> c;

// Bestimme das Maximum von a, b und c
if ( a >= b)
   {
     if ( c >= a)  maximum = c;
     else          maximum = a;
   }
else
   {
     if (c >= b)  maximum = c;
     else         maximum = b;
   }
```

Kontrollstrukturen: Anweisungen

- switch-Anweisungen erlauben die übersichtliche Auswahl (Fallunterscheidung) unter vielen Anweisungen:

```
switch(Ausdruck) {
   case  const1 :    Anweisungen_1
                     break;
   case  const2 :    Anweisungen_2
                     break;
   // usw.
   default          :    ErsatzAnweisungen
}
```

- Der Ausdruck muss vom Typ „int“ oder leicht in „int“ konvertierbar sein, wie zum Beispiel „char“.
- Das Ergebnis des Ausdrucks wird mit den case-Konstanten verglichen, die zum Einsprung an die richtige Stelle dienen.

Kontrollstrukturen: Anweisungen

- Umwandlung römischer Ziffern

```
#include<iostream>
using namespace std;

int main()  {
   int  a = 0;                          // Variable für das Resultat
   char c;                              // Variable zum Einlesen der Ziffer
   cout <<  “Ziffer :“ ;                //  Eingabeaufforderung
   cin  >> c;                           //  Einlesen der Ziffer

   if  (c == ‘I’)          a = 1;       // Berechnung der arabischen Zahlen
   else if (c == ‘V‘)  a = 5;
   else if (c == ‘X‘)  a = 10;
   else if (c == ‘L‘)   a = 50;
   else if (c == ‘C‘)  a = 100;
   else if (c == ‘D‘)  a = 500;
   else if (c == ‘M‘)  a = 1000;

   if  (a == 0)  cout << “keine römische Ziffer!“ << endl;
   else          cout <<  a  << endl;          // Ausgabe des Resultats
}
```

Kontrollstrukturen: Anweisungen


```
/*  Berechnung der arabischen Zahlen
    mittels einer switch-Anweisung                    */

switch (c)   {
                case  'I'      :  a = 1;      break;
                case  'X'      :  a = 10;     break;
                case  'L'      :  a = 50;     break;
                case  'C'      :  a = 100;    break;
                case  'D'      :  a = 500;    break;
                case  'M'      :  a = 1000;   break;
                default        :  a = 0;
             }
```

Fragen und Tests

Welcher Variablenname ist nicht zulässig?

A) 3D_Point

B) Point_3D

C) _Point_3D

D) point_3D

Fragen und Tests

Welche Deklaration/Initialisierung ist korrekt?

A) int a, b, summe,

B) const float theta;

C) char c = “text“;

D) double square_x = 10.5*10.5;

Fragen und Tests

Seien a, b, c Variablen vom Typ „float“. Welcher Ausdruck enthält einen Fehler?

A) (a+b)==(a*c)

B) (a/b)*(a-c)

C) !((a+b)!=c)

D) (a+b)%c

Fragen und Tests

49

Welchen Wert hat der Ausdruck
cin >> a >> b >> c ?

A) a

B) c

C) cin

D) 0

Fragen und Tests

50

Welche der folgenden if-Anweisungen
ist korrekt ?

A) if ((a+b)=bc) c = a+b;

B) if((a*b)!=c) c == a-b;

C) if((a>b) && (b>c)) c = a/b;

D) if ((a>= b+c)!) c = a*b

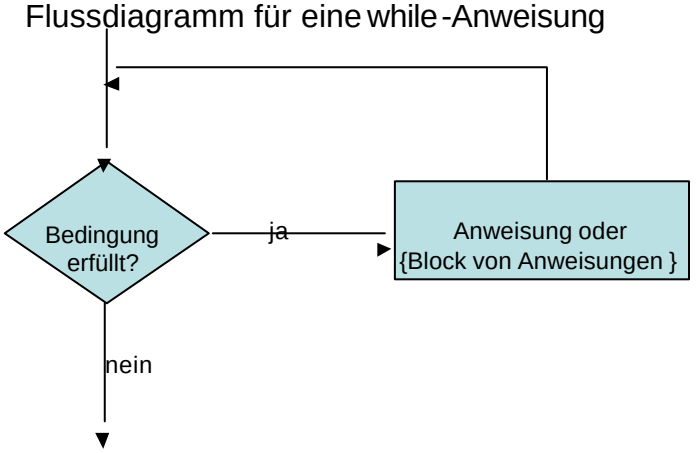
Schleifenanweisungen: while-Schleife

- In vielen Anwendungen muss die gleiche Aufgabe oft wiederholt werden.
- Hierfür gibt es in C++ so genannte Schleifen.
- Nach Abarbeitung des Schleifenblocks beginnt man wieder am Blockanfang.
- Dies geschieht solange, wie die Schleifenbedingung erfüllt ist.
- Syntax von while-Schleifen:

```
while (Bedingung)   Anweisung
while (Bedingung)  {  Block_von_Anweisungen }
```

Schleifenanweisungen: while-Schleife

Flussdiagramm für eine while-Anweisung



Auch while-Anweisungen können geschachtelt werden.

Schleifenanweisungen: while-Schleife

Beispiel: Berechnung von n! (= fak)

```
int  i = 2, n;                  // i ist unser Zähler
long fak = 1;                   // speichert Resultat n!
cin >> n;                       // Eingabe von n

while ( i <= n)
    {
        fak = fak * i ;         // kurz:   fak *= i;
        i++;                    // entspricht  i = i + 1;
    }
```

Kurzform-Operatoren

Binäre Kurzform-Operatoren

+=	Beispiel:	a += b;	entspricht a = a + b;
-=	Beispiel:	a -= b;	entspricht a = a – b;
*=	Beispiel:	a *= b;	entspricht a = a * b;
/=	Beispiel:	a /= b;	entspricht a = a / b;
%=	Beispiel:	a %= b;	entspricht a = a%b;

Der Operator % (= modulo) wird nur für „ganzzahlige“ Typen verwendet.

Kurzform-Operatoren

55

Unäre Operatoren:

++	Beispiel:	i++;	entspricht	i = i+1;
		++i;	entspricht	i = i+1;
--	Beispiel	i--;	entspricht	i = i-1;
		--i;	entspricht	i = i-1;

Was ist der Unterschied zwischen vor- und nachgestelltem Operator (++, --)?

```
while ( i <= n)  fak *= i++;    // gleich:  fak *= i; i = i + 1;
while ( i <= n)  fak *= ++i;    // gleich:  i = i + 1; fak *= i;
```

Schleifenanweisungen: while-Schleife

56

Berechnung des größten gemeinsamen Teilers GGT(x,y) zweier Zahlen x und y:

- GGT(x,x) = x
- Falls x < y , dann gilt: GGT(x,y) = GGT(x,y-x)
- Falls y < x , dann gilt: GGT(x,y) = GGT(x-y,y)

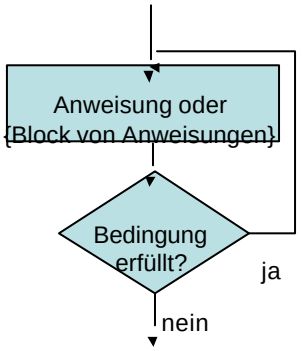
Schleifenanweisungen: while-Schleife


```
// Dieses Programm berechnet den GGT von zwei Zahlen x und y
#include<iostream>
using namespace std;

int main( )  {
      unsigned int x, y;

      cout  << "2 Zahlen größer 0 eingeben:";
      cin >> x >> y;
      cout  << " Der GGT von "  <<  x  <<  " und "  << y << " ist: ";

      while ( x != y )
            if ( x > y)  x -= y;      //  if … else zählt als eine Anweisung
            else         y -= x;      //  deshalb ohne {  }

      cout  <<  x   <<   endl;
}
```

Schleifenanweisungen: while-Schleife

- C++ stellt auch do-while-Schleifen zur Verfügung:
- Syntax:

```
do
      Anweisung   oder
      { Block_von_Anweisungen }
while (Bedingung);
```



Schleifenanweisungen: for-Schleife

- for-Schleifen: sehr kompakt und übersichtlicht
- Syntax der for-Schleife:

```
for (Initialisierung ; Bedingung ; Veränderung)  Anweisung
                                                 oder {Block}
```

- Beispiel:

```
int  n, i;
long fak = 1;
cin >> n;
for ( i = 2 ;  i <= n ; i++)   fak *= i;
```

Schleifenanweisungen: for-Schleife

- Beispiel:

```
int  n, i;
long fak =1;
cin >> n;
for ( i = 2 ;  i <= n ; i++)   fak *= i;
```

- Bedeutung:
  - Durchführung der Initialisierung: Setze Startwerte. Diese Anweisung wird nur einmal am Start durchgeführt.
  - Überprüfe die Bedingung. Falls Sie erfüllt ist, dann führe die Anweisung (oder Anweisungen im Block) durch. Anschließend die Veränderung(en).
  - Dieser Vorgang wird wiederholt bis die Bedingung nicht mehr erfüllt ist.

Schleifenanweisungen: for-Schleife

61

- Die Zählvariable i kann auch in der Initialisierung der for-Schleife definiert werden:

```
for (int  i = 2 ; i <= n ; ++i)  fak *= i;
```

- Die Variable i ist dann nur innerhalb der for-Schleife „bekannt“ (definiert).
- Mittels der „break“-Anweisung kann man aus einer for-Schleife (while-Schleife, switch-Anweisung) aussteigen (die Ausführung beenden).

```
for ( ... ;  ... ;  ...)
   {
     ...
     if ( ...)  break;
     ...
   }
```

- Man beachte, dass die „break“-Anweisung bei geschachtelten Schleifen nur die Ausführung der innersten Schleife beendet.

Schleifenanweisungen: for-Schleife

62

- Mittels der „continue“-Anweisung kann man die Ausführung von bestimmten Teilen der Schleife überspringen:

```
for ( ... ;  ... ;  ...)
   {
     ...
     if ( ...)  continue;        // Falls die Bedingung erfüllt ist,
     ...                         //  werden alle Anweisungen, die der
     ...                         // „continue“-Anweisung folgen,
     ...                         //  übersprungen.
   }
```

- In einer for-Schleife geht man sofort zur Ausführung der Veränderung(en) und dann wieder zur Überprüfung der Bedingung über.

Schleifenanweisungen: for-Schleife

- Der Komma-Operator ermöglicht sehr kompakte for-Schleifen.

```
for (int  i = 2 , fak = 1 ; i <= n ; fak *= i, ++i);
```

- Hier haben wir die Anweisung im Schleifenkörper in die Veränderung integriert.
- Dieser Programmierstil führt nicht zu leicht verständlichen Programmen und ist deshalb nicht empfehlenswert.

Unterprogramme / Module

- Große Programme müssen in übersichtliche Teile (Funktionen/Module) zerlegt werden:
  - Eine Funktion erledigt eine Teilaufgabe.
  - Die notwendigen Daten werden der Funktion mitgegeben.
  - Sie gibt das Ergebnis an den Aufrufer zurück.
  - Eine Funktion muss nur einmal definiert werden.
  - Anschließend kann sie beliebig oft durch Nennung ihres Namens aufgerufen werden.

Unterprogramme / Module


```
// Implementation (Definition): Fakultaet einer Zahl
unsigned long fakultaet( int  zahl )  {
    unsigned long fak = 1;
    for (int i = 2; i<= zahl; ++i)  fak * = i;
    return fak;
}
```

- Eine Definition veranlasst den Compiler,
  - entsprechenden Code zu erzeugen und
  - den notwendigen Speicherplatz zu reservieren.

- Synthax der Funktionsdefinition:

Rückgabetyp	Funktionsname	(Formalparameterliste)	Block
unsigned long	fakultaet	(int zahl)	{ ….}

Unterprogramme / Module


```
#include<iostream>
using namespace std;

// Funktionsprototyp (Deklaration):
unsigned long fakultaet( int);

int main ( ) {
     int n;
     unsigned long erg;

     cout << “Fakultaet berechnen. n >= 0? :“
     cin >> n;
     if (n > 0)  {
              erg = fakultaet(n);
              cout << n << “! = “ << erg << endl;
     }
}

// Funktionsimplementation (Definition):
unsigned long fakultaet(int zahl)  {
              unsigned long fak = 1;
              for (int i = 2; i<= zahl; ++i)  fak * = i;
              return fak;
}
```

Syntax eines Funktionsprototypen (Funktions-Deklaration)

```
unsigned long     fakultaet          ( int );
      :                  :                 :
Rückgabetyp   Funktionsname (Parameterliste)
```

Die Deklaration sagt dem Compiler, dass eine Funktion (oder eine Variable) mit diesem Aussehen irgendwo definiert ist.

Syntax des Funktionsaufrufs

```
          fakultaet( n )
               :      :
Funktionsname(Aktualparameterliste)
```

Die Aktualparameterliste enthält Ausdrücke und /oder Namen der Objekte oder Variablen, die an die Funktion übergeben werden.

Die Aktualparameterliste kann gegebenenfalls auch leer sein.

Unterprogramme / Module

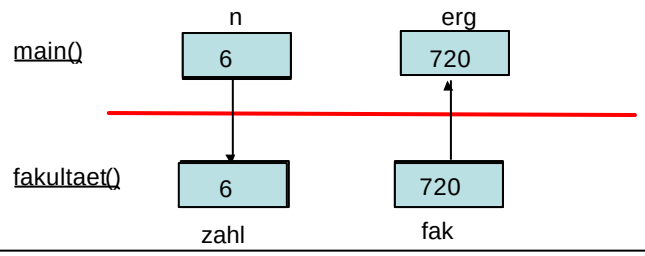
- Datentransfer in und aus Funktionen wird durch die Beschreibung der Schnittstelle festgelegt.
- Schnittstelle = formale Vereinbarung
  - über die Art und Weise des Datentransports
  - und darüber, was die Funktion leistet.
- Funktionsprototyp beschreibt die Schnittstelle:
  - den Rückgabetyp der Funktion
  - den Funktionsnamen
  - Parameter, die der Funktion bekannt gemacht werden
  - die Art der Parameterübergabe

Unterprogramme / Module

- Zwei Arten des Datentransports
  - Übergabe per Wert
  - Übergabe per Referenz
- Übergabe per Wert
  - Die Funktion erhält eine Kopie des Objekts.
  - Sie arbeitet mit der Kopie.
  - Das Original beim Aufrufer bleibt unverändert.
- Übergabe per Referenz
  - Die Funktion arbeitet auf dem Originalobjekt und verändert gegebenenfalls dessen Daten.

Unterprogramme / Module

- Übergabe per Wert
  - Die Funktion erhält eine Kopie des Objekts.
  - Sie arbeitet mit der Kopie.
  - Das Original beim Aufrufer bleibt unverändert.
- Beispiel: fakultaet
  - Funktionsprototyp: unsigned long fakultaet(int);
  - Funktionsaufruf: erg = fakultaet(n);
  - Der Wert von n wird an die Variable „zahl“ übergeben.



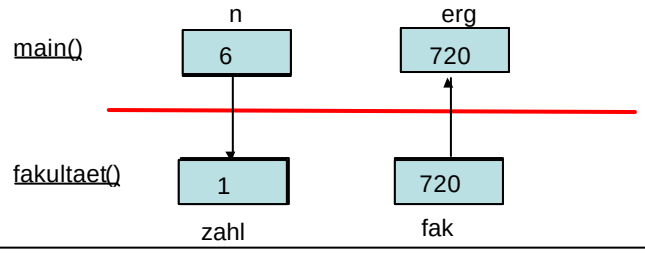
Unterprogramme / Module


```
// Implementation (Definition): Fakultaet einer Zahl
unsigned long fakultaet(int zahl) {
    unsigned long fak = 1;
    for (int i = 2; i<= zahl; ++i)  fak * = i;
    return fak;
}
```

```
// Modifizierte Version von fakultaet
unsigned long fakultaet(int zahl) {
    unsigned long fak = 1;
    for ( ; zahl > 1; --zahl)  fak * = zahl;
    return fak;
}
```

Unterprogramme / Module

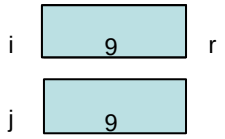
- Übergabe per Wert
  - Die Funktion erhält eine Kopie des Objekts.
  - Sie arbeitet mit der Kopie.
  - Das Original beim Aufrufer bleibt unverändert.
- Beispiel: Modifizierte Version von fakultaet
  - Funktionsprototyp: `unsigned long fakultaet( int );`
  - Funktionsaufruf: `erg = fakultaet(n);`
  - Der Wert von n wird an die Variable „zahl“ übergeben.



Unterprogramme / Module

- Eine Referenz ist ein Alias-Name für ein Objekt:

```
int i = 2, j = 9;
int &r = i;        // r ist Referenz auf i
r = 10;            // ändert i ✍ i = 10
r = j;             // Wirkung :  i = 9
```



- Das & Zeichen vor einer Variable deklariert diese als Referenz.
- Wo das Zeichen & zwischen Datentyp (int) und Variablennamen (r) steht ist unerheblich.
- Jede Referenz muss bei der Deklaration initialisiert werden, damit der Compiler weiß, für welche Variable sie ein Alias-Name ist.

Unterprogramme / Module

- Übergabe per Referenz
  - Die Funktion arbeitet auf dem Originalobjekt und verändert gegebenenfalls dessen Daten.
  - Funktionsprototyp:

```
unsigned long fakultaet( int&);    // int& = Referenz auf int
```

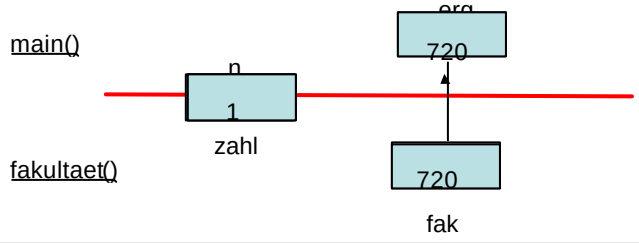
  - Funktionsimplementation (Definition)

```
// Modifizierte Version von fakultaet
unsigned long fakultaet(int&  zahl)  {
        unsigned long fak = 1;
        for ( ; zahl > 1; --zahl)  fak * = zahl;
        return fak;
}
```

  - Funktionsaufruf ändert sich nicht: `erg = fakultaet(n);`

Unterprogramme / Module

- Übergabe per Referenz
  - Die Funktion arbeitet auf dem Originalobjekt und verändert gegebenenfalls dessen Daten.
- Beispiel: Modifizierte Version von fakultaet
  - Funktionsprototyp: `unsigned long fakultaet( int& );`
  - Funktionsaufruf: `erg = fakultaet(n);`
  - Der Variablen „n“ und „zahl“ sind Namen für das gleiche Objekt.



Unterprogramme / Module

- Funktionen können viele Parameter besitzen:
  - leere Liste: `int func(); // = int func(void);`
  - ein Param. `float square_root( float );`
  - zwei Param. `int addition(int, int);`
  - drei Param. `void xyz(float, int, char);`
  - usw.
- Jeder Parameter kann per Referenz oder per Wert übergeben werden (später: Zeiger/Pointer).

Unterprogramme / Module

- Funktionen können sich selbst aufrufen.

```
// Rekursive Version von fakultaet
unsigned long fakultaet(int zahl) {
    if (zahl == 1) return 1;
    else return( zahl * fakultaet( zahl – 1 ) );
}
```

- Der Aufruf einer Funktion durch sich selbst heißt Rekursion.
- Natürlich können Funktionen auch andere Funktionen aufrufen.

Polymorphismus

- Funktionen können überladen werden.
- Verschiedene Funktionen für gleichartige Operationen können den gleichen Funktionsnamen tragen.

```
float   max(float , float );
int     max(int, int);
```

- Die Funktionen müssen sich jedoch in ihrer Signatur unterscheiden.
- Signatur = Funktionsname + Reihenfolge + Typen der Parameter


```
// Beispielprogramm
#include<iostream>
using namespace std;

double  max(double x, double y) { return x > y ? x : y; }
int     max(int x, int y)       { return x > y ? x : y; }

int main() {
   int a = 100, b = 200;
   double c = 25.4, d = -10.0;
   cout << max(a,b) << endl;   // Aufruf max(int, int)
   cout << max(c,d) << endl;   // Aufruf max(double, double)
}
```

Unterprogramme / Module

- Überladen von Funktionen:
  - Der Compiler trifft die Zuordnung anhand der Signatur, die er mit dem Aufruf vergleicht.
  - Der Compiler versucht nach bestimmten Regeln immer die beste Übereinstimmung mit den Parametertypen zu finden.

```
const float e = 2.7182, pi = 3.14159;
cout << max(e,pi);
```

führt zum Aufruf von „max(double,double)“, weil float-Werte in double-Werte konvertiert werden.

```
cout <<  max(31, ´A´);
```

führt zum Aufruf von „max(int,int)“, weil der Datentyp „char“ auf „int“ abgebildet wird.

Modulare Gestaltung

- Es empfiehlt sich große Programme in einzelne, getrennt übersetzbare Dateien aufzuteilen (Wiederverwendbarkeit).
- Standard-Header haben die Form `#include<....>`
- Eigene (und fremde, nicht-standard) Header-Dateien können hinzugefügt werden (siehe folgendes Beispiel).

```
#include “meine_mathe_funktionen.h“
```

- Die entsprechende Header-Datei mit dem Namen „meine_mathe_funktionen.h“ könnte wie folgt aussehen:

```
// meine_mathe_funktionen.h
unsigned int  ggt( unsigned int, unsigned int);
```

- Im Header findet man hier nur die Funktionsdeklaration.

Modulare Gestaltung

- Die Einbindung in das Hauptprogramm erfolgt wie bei einem Standard-Header:

```
// Hauptprogramm „berechne_ggt.cpp“
#include<iostream>
#include “meine_mathe_funktionen.h“
using namespace std;

int main( ) {
    unsigned int x, y, resultat;

    cout << “2 Zahlen größer 0 eingeben:“;
    cin >> x >> y;
    cout << “ Der GGT von “ << x << “ und “ << y << “ ist: “;

    resultat = ggt(x,y);
    cout << “GGT(“ << x << “,“ << y << “)=“ << resultat << endl;
}
```

Modulare Gestaltung

- Die Implementation der entsprechenden Funktion(en) findet man in der Datei „meine_mathe_funktionen.cpp“:

```
// meine_mathe_funktionen.cpp
#include “meine_mathe_funktionen.h“

unsigned int ggt( unsigned int zahl1, unsigned int zahl2) {
        while ( zahl1 != zahl2 )
                if ( zahl1 > zahl2) zahl1 -= zahl2;
                else                zahl2 -= zahl1;
        return zahl1;
}
```

- Bevor das Hauptprogramm (Datei „berechne_ggt.cpp“) gelinkt werden kann, muss „meine_mathe_funktionen.cpp“ übersetzt werden:

```
g++ -c meine_mathe_funktionen.cpp
g++ -c berechne_ggt.cpp
g++ -o berechne_ggt berechne_ggt.o meine_mathe_funktionen.o
```

- Das ausführbare Programm hat den Namen „berechne_ggt“.

Gültigkeitsbereich und Sichtbarkeit

Für alle Namen gelten Gültigkeits- und Sichtbarkeitsregeln.

```
#include<iostream>
using namespace std;

int main() {

    // Äußerer Block
    {
        int a = 10;
        // Innerer Block
        {
            int a = 200;
            cout << a ; // Gibt 200 aus.
        }

        cout << a ;      // Gibt 10 aus.
    }

}
```

- Namen sind gültig
  - nach der Deklaration
  - und nur innerhalb des Blockes, in dem sie deklariert wurden.
- Sie sind lokal bzgl. des Blocks.
- Sie sind auch gültig für innerhalb des Blocks angelegte innere Blöcke.
- Die Sichtbarkeit kann durch die Deklaration von Variablen mit gleichem Namen eingeschränkt werden.

Gültigkeitsbereich und Sichtbarkeit

Dateiübergreifende Gültigkeit und Sichtbarkeit

```
// Datei1.cpp
// Deklaration + Defintion
int global;
int main() {

}
```

```
// Datei2.cpp
// Deklaration, aber keine Definition
extern int global;
int func1 {
    global = 123;
}
```

- Globale Variablen werden außerhalb von main und jeglicher anderen Funktion definiert.
- Sie sind in allen Teilen eines Programms gültig, auch in anderen Dateien.
- Um eine globale Variable in einer anderen Datei zu benutzen, muss sie dort als „extern“ deklariert werden.
- „extern“ bedeutet
  = ist irgendwo anders definiert.

Gültigkeitsbereich und Sichtbarkeit

Dateiübergreifende Gültigkeit und Sichtbarkeit

```
// Datei1.cpp
// Deklaration + Defintion
// mit Initialisierung
extern const float pi = 3.1415;
```

```
// Datei2.cpp
// Deklaration, ohne Definition
// und Initialisierung
extern const float pi;
```

- Konstanten sind (ohne Schlüsselwort extern) nur in der Definitionsdatei sichtbar.
- Sollen sie anderen Dateien zugänglich gemacht werden, so müssen sie als extern deklariert werden.

Gültigkeitsbereich und Sichtbarkeit

Dateiübergreifende Gültigkeit und Sichtbarkeit

```
// Datei1.cpp
// Deklaration + Defintion
static int global;
int main() {

}
```

```
// Datei2.cpp
// Deklaration + Definition
static int global;
int func1 {
    global = 123;
}
```

- Mittels des Schlüsselworts „static“ kann man den Gültigkeitsbereich auf eine Datei beschränken.
- In unserem Beispiel haben wir es mit zwei verschiedenen Variablen mit gleichem Namen zu tun.
- Ziel = Vermeidung von Namenskonflikten zwischen verschiedenen Dateien.

Gültigkeitsbereich und Sichtbarkeit

Dateiübergreifende Gültigkeit und Sichtbarkeit

```
// Funktion mit Gedächnis
#include<iostream>
using namespace std;

void func( ) {
   static int anz = 0;
   cout << "anz = " << anz << endl;
}

int main() {
  for (int i = 0; i < 3; ++i)  func( );
  /*  main gibt folgendes aus:
        anz = 0
        anz = 1
        anz = 2   */
}
```

- „static“ kann auch andere Bedeutungen besitzen:
- „static“ steht auch vor Variablen,
  - die innerhalb von Funktionen deklariert sind und
  - die zwischen verschiedenen Funktionsaufrufen nicht ihren Wert verlieren sollen.
- Diese static-Variablen erhalten einen festen Speicherplatz.
- Sie werden nur einmal initialisiert.

Gültigkeitsbereich und Sichtbarkeit

- Alle nicht globalen und nicht-static-Variablen sind automatisch auto-Variablen.
- „register“ ist ein Hinweis an den Compiler, dass diese Variablen aus Geschwindigkeitsgründen am besten in ein Register der CPU gepackt würden.

Ein- und Ausgabe von (in) Dateien

- Um Daten aus einer Datei einzulesen oder in eine Datei auszugeben, müssen neue „Datenströme“ (fstream = file stream) deklariert, definiert und geöffnet werden.
- Zunächst binden wir den Header „fstream“ ein:

```
#include<fstream>
// iostream.h ist in fstream.h eingebunden.
```

- Zum Einlesen aus einer Datei benötigen wir ein Objekt (eine Instanz) der Klasse „ifstream“ (input file stream):

```
ifstream quelle;        // Das Objekt nennen wir „quelle“
```

Ein- und Ausgabe von (in) Dateien

- Dem Objekt „quelle“ ordnen wir eine Eingabedatei zu:

```
string  quelldateiname;
cout << “Bitte Namen der Quelldatei angeben:“;
cin >> quelldateiname;
quelle.open(quelldateiname.c_str(), ios::binary|ios::in);
```

- „open“ öffnet die Datei „quelldateiname“

ios::in	✍	zum Lesen
ios::binary	✍	auch binäre Dateien können gelesen werden
c_str( )	✍	macht einen char-String aus dem Objekt „quelldateiname“ vom Typ „string“ (später).

- Es muss gestestet werden, ob „open“ erfolgreich war:

```
if (!quelle)        // auf Fehlermeldung wird hier verzichtet!
   return(-1);
```

Ein- und Ausgabe von (in) Dateien

- Zum Auslesen in eine Datei benötigen wir ein Objekt (eine Instanz) der Klasse „ofstream“ (output file stream):

```
ofstream ziel;        // Das Objekt nennen wir „ziel“
```

- Dem Objekt „ziel“ ordnen wir eine Ausgabedatei zu:

```
string  zieldateiname;
cout << “Bitte Namen der Zieldatei angeben:“;
cin >> zieldateiname;
quelle.open(zieldateiname.c_str(), ios::binary|ios::out);
```

- „open“ öffnet die Datei „zieldateiname“

ios::out	✍ zum Schreiben
ios::binary	✍ auch binäre Dateien können erstellt werden
c_str( )	✍ macht einen char-String aus dem Objekt „quelldateiname“ vom Typ „string“ (später)

Ein- und Ausgabe von (in) Dateien

- Einlesen aus „quelle“:

```
char c;
quelle.get(c);     // Liest ein Zeichen aus der Datei.
int  i;
quelle >> i;       // Liest einen Wert für i  aus der Datei.
```

- Ausgabe nach „ziel“ :

```
char c = `a`;
ziel.put(c);       // Schreibt ein Zeichen in die Datei.
int i = 100;
ziel << i;         // Schreibt den Wert von i in die Datei.
```

- Dateien schließen:

```
quelle.close( );

ziel.close( );
```

Ein- und Ausgabe von (in) Dateien


```
// Beispielprogramm: Kopiere Quelldatei nach Zieldatei
#include<fstream>
#include<string>

using namespace std;

int main( ) {

        // Quelldatei öffnen
        ifstream quelle;
        string quelldateiname;
        cout << “Bitte Namen der Quelldatei angeben:“;
        cin >> quelldateiname;
        quelle.open(quelldateiname.c_str(), ios::binary|ios::in);

        if (!quelle) return( -1);
```

Ein- und Ausgabe von (in) Dateien


```
        // Zieldatei öffnen
        ofstream ziel;
        string zieldateiname;
        cout << “Bitte Name der Zieldatei angeben:“;
        cin >> zieldateiname ;
        ziel.open(zieldateiname.c_str(),ios::binary|ios::out);

        if (!ziel) return( -1);

        // Inhalt von Quelle nach Ziel kopieren
        char c;
        while (quelle.get(c)) ziel.put(c);

        // Dateien schließen
        quelle.close();
        ziel.close();
}
```

Ein- und Ausgabe von (in) Dateien

- Die Richtung des Datenflusses wird beim Öffnen der Datei mit der Funktion open() als Parameter angegeben:

```
quelle.open(quelldateiname.c_str(), ios::binary|ios::in);
ziel.open(zieldateiname.c_str(), ios::binary|ios::out);
```

- Ohne „ios::binary“ können nur Textdateien bearbeitet werden.
- Die Operatoren „<<“ und „>>“ können wie bei der Standardein- und ausgabe verwendet werden.

```
int a, b;
quelle >> a >> b;
ziel << a << b;
```

- Die Funktionen get() und put() können zum Einlesen oder zur Ausgabe einzelner Zeichen verwendet werden:

```
while (quelle.get(c))  ziel.put(c);
```

Ganze Zahlen

- Datentypen für ganze Zahlen:

short	16 Bit	unsigned short
int	32 Bit	unsigned int
long	64 Bit	unsigned long

- Zahlenbereich bei 32 Bit mit Vorzeichen: -2^{31} .... $2^{31}-1$
- Die Operatoren für Ausdrücke mit ganzen Zahlen finden Sie im Breymann Skript auf den Seiten 22-25.

Unäre Operatoren:	+	-	++	--		
Binäre Operatoren:	+	-	*	/	%	=
Kurzform-Operatoren:	+=	-=	*=	/=	%=	
Vergleichsoperatoren:	==	>	<	!=	<=	>=

Ganze Zahlen


```
/*  Die in Ihrem C++-System zutreffenden Zahlenbereiche
    finden Sie im Header <limits>:                          */

    #include<iostream>
    #include<limits>
    using namespace std;

    int main ( ) {
       cout << "der Zahlenbereich für ints geht von "
            << numeric_limits<int>::min( )  <<  " bis "
            << numeric_limits<int>::max( ) << endl;
    }
```

Reelle Zahlen

- Datentypen für Gleitkommazahlen:

float	32 Bits	$+/- 3.4 * 10^{+/-38}$
double	64 Bits	$+/- 1.7 * 10^{+/-308}$
long double	80 Bits	$+/- 3.4 * 10^{+/-4932}$

- Gleitkommazahlen können wie folgt dargestellt werden:
  - Vorzeichen (optional)
  - Vorkommastellen
  - Dezimalpunkt
  - Nachkommastellen
  - e oder E und ganzzahliger Exponent (optional)
  - Suffix f, F oder l, L (optional, Zahlen ohne Suffix sind double)

- Beispiele:

120.234 -123.234 -1.0e6 -2.5E-03 1.8L

Vorrangregeln

- Es gelten im Allgemeinen die Vorrangregeln der Algebra beim Auswerten eines Ausdrucks, inklusive Klammerregeln:

  *, /, % vor +, - usw. (siehe BS. Seite 30).

- Auf gleicher Prioritätsstufe wird ein Ausdruck von links nach rechts abgearbeitet (linksassoziativ), mit Ausnahme gewisser Opertoren (=, +=, -=, *=, !, unäres + und -, präfix ++ und --, ~ , ...).

  a = b + d + c; ist gleich a = ((b+d) + c);
  a = b = d = c; ist gleich a = (b= (d = c));

- Ist man sich nicht sicher, wie ein Ausdruck ausgewertet wird, so sollte man immer Klammern verwenden!

Zeichen

- Zeichen sind
  - Buchstaben: a b .... A B ....
  - Ziffern: 0 1 9
  - Sonderzeichen: ! “ § % ....
- Die ASCII-Tabelle ordnet jedem Zeichen eine Zahl (bis 256) zu.
- Zeichenkonstanten (Zeichenliterale) werden in Hochkommata eingeschlossen:

```
char stern = ´*´;
char eins  = ´1´;
```

Zeichen

- Besondere Zeichenkonstanten der ASCII-Tabelle werden als Folge von zwei Zeichen geschrieben und als Escape-Sequenzen "\" bezeichnet:

```
char zeilenumbruch = ´\n´;
char tabulator;
tabulator = ´\t´;
```

- Die Position eines Zeichens in der ASCII-Tabelle kann durch eine Typumwandlung von „char" nach „int" bestimmt werden:

```
char stern = ´*´;
int i  = static_cast<int>(stern);      // int i = (int) stern;
char c = static_cast<char>(i);
```

Zeichen

- Für Zeichen gibt es die „üblichen" Vergleichsoperatoren:

```
==    !=    >    <    >=    <=
```

```
char ende_while = ´n´;

do {
      // Anweisung oder Block von Anweisungen
      cout << "Soll die Schleife beendet werden?" << endl;
      cin >> ende_while;
      // oder cin.get(ende_while);
   }
while (ende_while == ´n´);
```

Strings

- Eine Zeichenkette, auch String genannt, ist aus Zeichen des Typs „char“ zusammengesetzt.
- Die wichtigsten Funktionen diskutieren wir anhand eines Beispielprogramms:

```
#include<iostream>
#include<string>
#include<cstddef>                    // für Datentyp „size_t“

/*  Der Datentyp „size_t“ wird für positive, ganzzahlige
    Größenangaben (wie zum Beispiel einen Zähler in
    Schleifen) verwendet. Der Typ entspricht entweder
    „unsigned int“ oder „unsigned long“ (je nach Compiler).
*/

using namespace std;
```

Strings


```
/* Es folgt das Hauptprogramm, das sich über mehrere
   Folien ausdehnt.   */

int main() {

// String-Objekt definieren und initialisieren
string einString(“hallo“);

// String ausgeben
cout << einstring << endl;        // Ausgabe:   hallo

// String zeichenweise ausgeben (ungeprüfter Zugriff)
for (size_t  i = 0; i < einString.size(); ++i) cout << einString[i];
cout << endl;
```

Strings


```
// Zeichenweise Ausgabe mit Indexprüfung
for (size_t i = 0; i < einString.length(); ++i)
      cout << einString.at(i);
cout << endl;
/* at(i) prüft, ob dieser Index i „existiert“, und beendet
   das Programm, falls der Index „außerhalb“ ist. */

// Kopie des Strings erzeugen
string eineStringKopie(einString);
cout << eineStringKopie << endl;   // Ausgabe: hallo

string diesIstNeu(“neu!“);              // neuen String anlegen
eineStringKopie = diesIstNeu;          // Zuweisung
cout << eineStringKopie << endl;   // Ausgabe:  neu!
```

Strings


```
eineStringKopie = “Buchstaben“;
cout << eineStringKopie << endl;  // Ausgabe: Buchstaben

// Zuweisung eines Zeichens vom Typ char
einString = ´X´;
cout  << einString << endl;          // Ausgabe:  X

// Strings mit dem += Operator verknüpfen
einString += eineStringKopie;
cout << einString << endl;           //  Ausgabe: XBuchstaben

// Strings mit dem + Operator verknüpfen
einString = eineStringKopie + “ ABC“;
cout << einString << endl;
//  Ausgabe: Buchstaben ABC
```

Strings


```
// Strings mit dem + Operator verknüpfen
einString = “123“ + eineStringKopie ;
cout << einString << endl;
//  Ausgabe: 123Buchstaben

/*  nicht erlaubt ist:  einstring = “ ABC“ + “123“;
    Erklärung folgt später!    */

}  // Ende von main()
```

Logischer Datentyp: bool

- Benannt nach dem englischen Mathematiker George Boole (1815-1864): Boolesche Algebra
- Instanzen dieses Typs können nur zwei Werte

  „true“ und „false“ annehmen.
- Falls notwendig, wird „bool“ zu „int“ gewandelt, wobei

  „true“ der Wert 1 und
  „false“ der Wert 0 ist.
- Bedingungen sind Ausdrücke vom Typ „bool“.

Logischer Datentyp: bool

- Operatoren für Boolesche Ausdrücke:
  a und b seien Variablen oder Ausdrücke vom Typ „bool“:

!	logische Negation	Beispiel: !a
&&	logisches UND	Beispiel: a && b
||	logisches ODER	Beispiel: a || b
==	Vergleich	Beispiel: a == b
!=	Vergleich	Beispiel: a != b
=	Zuweisung	Beispiel: a = a && b

- Mit Hilfe dieser Operatoren und den runden Klammern „(“ und „)“ können komplizierte Boolesche Ausdrücke dargestellt werden.

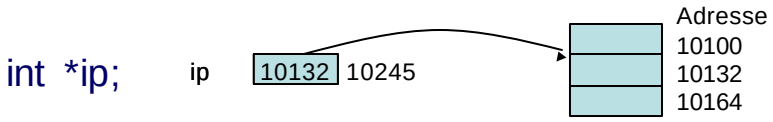
Logischer Datentyp: bool

- Beispiel:
  - Test, ob ein Zeichen ein Grossbuchstabe ist:

```
bool grossBuchstabe;
char c;
cin >> c;
grossBuchstabe = (c >= ´A´) && (c <= ´Z´);
cout  << grossBuchstabe;
```

Zeiger / Pointer

- Zeiger sind ähnlich wie andere Variablen, d.h., sie haben
  - einen Namen und
  - einen Wert und
  - sie können mit Operatoren verändert werden.
- Der Wert eines Zeigers ist eine Adresse im Speicher.
- In Deklarationen bedeutet ein * „Zeiger auf“. Beispiel:

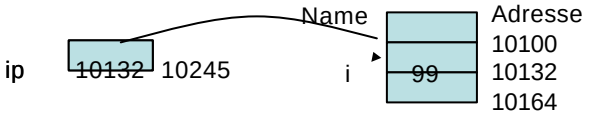


- ip ist ein Zeiger auf einen int-Wert, d.h, in der Speicherzelle, deren Adresse in ip gespeichert ist, befindet sich ein int-Wert.

Zeiger / Pointer

Wie initialisiert man Zeiger?

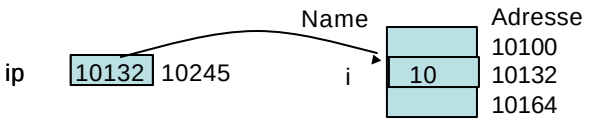
```
int  i = 99;    // eine int-Variable i mit Wert 99

/* Wir nehmen an, dass der Compiler für i die
   Speicherplatzadresse 10132 bestimmt hat.  */

int  *ip;       //  Deklaration und Definition des Zeigers ip
ip = &i;        //  ip wird mit der Adresse von i initialisiert

/*  Der Operator & wirkt hier als Adressoperator und
   übergibt die Adresse der Variablen i an den Zeiger ip.  */
```



Zeiger / Pointer

Wie ändert man den Wert an der Adresse 10132 (Wert von i) mit Hilfe des Zeigers ip?

```
*ip = 10;       // ✍ i = 10

/*  *ip bedeutet, dass der Wert an der Stelle betrachtet
    wird, auf die der Zeiger verweist
    (* = Dereferenzierung).  */
```



Zeiger / Pointer

- Zeiger erhalten bei der Deklaration zunächst eine beliebige Adresse.
- In C gibt es einen speziellen Zeigerwert, nämlich NULL. Ein mit NULL initialisierter Zeiger zeigt auf „nichts“.

```
int *ip = NULL;     // Zeiger ip auf NULL initialisiert
```

- NULL ist als logischer Wert abfragbar:

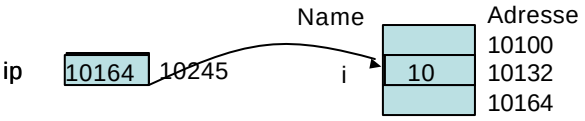
```
if (ip != NULL)     // ..dann tue etwas, andernfalls nicht
```

- Die Initialisierung kann mit der Definition erfolgen:

```
int *ip = &i;
```

Zeiger / Pointer

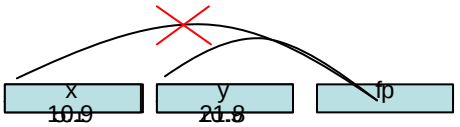
- Operatoren:
  - Adressoperator: & `ip = &i;`
    - liefert die Adressen von Variablen.
  - Derefernzierungs-Operator: * `*ip = 10;`
    `int j = *ip;`
    - ermöglicht die Veränderung des Werts der entsprechenden Speicherzelle, auf die der Zeiger zeigt, oder den Zugriff auf den dort gespeicherten Wert
  - Die Operatoren ++ -- + - += -= `++ip;`
    - ermöglichen die Änderung der gespeicherten Adressen.



Zeiger / Pointer

- Zeiger sind „Variablen“, d.h., dass ihre Werte (Adressen) geändert werden können.
- Mit Zeigern kann man sich also im Speicher bewegen:

```
float  x = 5.1, y = 10.9;
float  *fp = &x;
*fp = y;
fp = &y;
*fp = x + y;
```



Zeiger / Pointer

Wiederholung zum Thema „Parameterübergabe an Funktionen“!

- Übergabe per Wert: Beispiel „ggt“

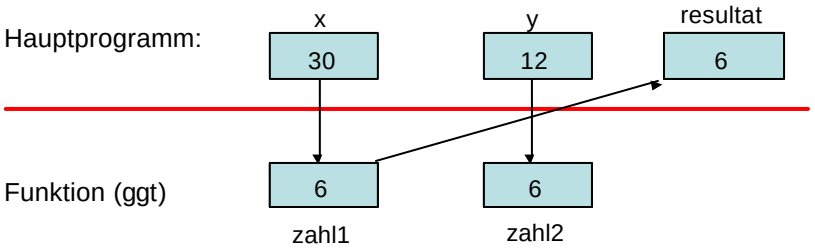
```
unsigned int  ggt( unsigned int zahl1, unsigned int zahl2) {
        while ( zahl1 != zahl2 )
                if ( zahl1 > zahl2)  zahl1 -= zahl2;
                else                 zahl2 -= zahl1;
        return zahl1;
}
```

Zeiger / Pointer


```
#include<iostream>
using  namespace std;

unsigned int ggt( unsigned int, unsigned int);
int main( )  {
        unsigned int x, y, resultat;

        cout  << “2 Zahlen größer 0 eingeben:“;
        cin >> x >> y;

        resultat = ggt(x,y);
        cout  <<  “GGT(“ << x << “,“ <<  y << “)=“
              <<  resultat  <<   endl;
}
```

Zeiger / Pointer

- In der obigen Version erfolgte die Parameterübergabe per Wert:



Zeiger / Pointer

Wiederholung zum Thema „Parameterübergabe an Funktionen“!

- Übergabe per Referenz: Beispiel „ggt“

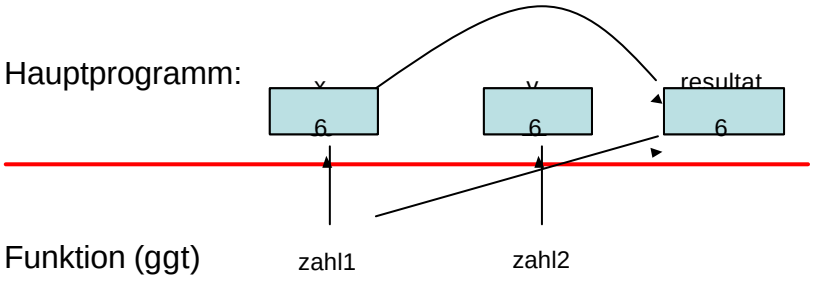
```
unsigned int  ggt( unsigned int & zahl1, unsigned int & zahl2)
    {
        while ( zahl1 != zahl2 )
            if ( zahl1 > zahl2)  zahl1 -= zahl2;
            else                 zahl2 -= zahl1;
        return zahl1;
    }
```

Zeiger / Pointer


```
#include<iostream>
using namespace std;

unsigned int ggt( unsigned int&, unsigned int&);
int main( ) {
        unsigned int x, y, resultat;

        cout << "2 Zahlen größer 0 eingeben:";
        cin >> x >> y;

        resultat = ggt(x,y);
        cout << "GGT(" << x << "," << y << ")="
             << resultat << endl;
}
```

Zeiger / Pointer

- In unserer zweiten Version erfolgte die Parameterübergabe per Referenz:



Zeiger / Pointer

- Parameterübergabe per Zeiger: Beispiel „ggt“

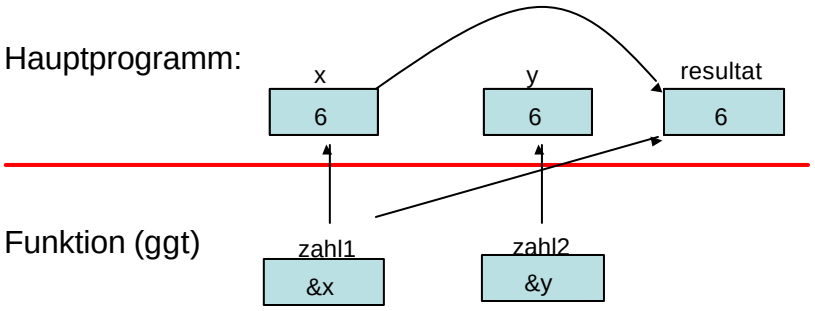
```
unsigned int  ggt( unsigned int* zahl1, unsigned int* zahl2)
    {
        while ( *zahl1 != *zahl2 )
            if ( *zahl1 > *zahl2)  *zahl1 -= *zahl2;
            else                   *zahl2 -= *zahl1;
        return *zahl1;
    }
```

Zeiger / Pointer


```
#include<iostream>
using  namespace std;

unsigned int ggt( unsigned int*, unsigned int*);
int main( )  {
        unsigned int x, y, resultat;

        cout  << “2 Zahlen größer 0 eingeben:“;
        cin >> x >> y;

        resultat = ggt(&x,&y);
        cout  <<  “GGT(“ << x << “,“ <<  y << “)=“
              <<  resultat  <<   endl;
}
```

Zeiger / Pointer

- In unserer dritten Version erfolgte die Parameterübergabe per Zeiger:



Aufzählungs- oder Enumerationstypen

- Wir lernen nun eine erste Möglichkeit kennen, wie man als Programmierer selbst Typen definieren kann.
- Diese Aufzählungstypen machen dann Sinn, wenn man eine kleine endliche Menge von nicht-numerischen Werten hat. Beispiele:

```
enum Geschlecht { maennlich, weiblich };

enum Wochentage { sonntag, montag, dienstag, mittwoch,
                  donnerstag, freitag, samstag };

Wochentag  heute = dienstag, morgen;

if (heute == dienstag)   morgen = mittwoch;
```

Aufzählungs- oder Enumerationstypen

- Syntax der Deklaration/Definition:

```
enum  [Typname] { Liste möglicher Werte }[Variablenliste];
```

- Die Werte in der Liste werden durch Kommas getrennt.
- Die eckigen Klammern bedeuten, dass Typname oder Variablenliste weggelassen werden können (nicht notwendig sind).

```
enum Wochentage { sonntag, montag, dienstag,
                  mittwoch, donnerstag, freitag, samstag } heute;
```

Strukturierte Datentypen

- Um logisch zusammengehörende Daten von verschiedenen Datentypen zusammenfassen zu können, stellt C/C++ die Struktur(en) zur Verfügung:
- Beispiel: Struktur für Punkte auf dem Bildschirm.

```
enum Farbe { rot, gruen, blau };        // unsere einfache Farbpalette

struct Punkt  {
   int x, y;              // die x- und y-Koordinate eines Punktes
   Farbe farbe;           // die Farbe des Punktes
};
```

- Syntax von Strukturen:

```
struct [Typname] {Definition der inneren Daten} [Variablenliste];
```

Strukturierte Datentypen

- Wie deklariert, definiert und verwendet man Strukturen?

```
Punkt  p1, p2;                  // Wir definieren zwei Punkte.
p1.x = 100;                     // Die Werte von p1 werden gesetzt.
p1.y = 12;
p1.farbe = blau;                .

cin >> p2.x >> p2.y;            // Einlesen der Koordinaten von p2.
p2.farbe = p1.farbe;            // p2 bekommt die Farbe von p1.

Punkt p3;                       // ein neuer Punkt
p3.x = p1.x + p2.x;
p3.y = p1.y + p2.y;

if  (p2.farbe == blau)  p3.farbe = gruen;
else                    p3.farbe = rot;
```

Strukturierte Datentypen

- Ein komplexeres Beispiel:

```
enum Geschlecht { weiblich, maennlich };
enum Studienfach { Informatik, Bioinformatik, CuK, Computerlinguistik,
                   AngewInformatik, Wirtschaftsinformatik};

struct Student {
                   string          name;
                   Geschlecht      geschlecht;
                   unsigned short  semesterzahl;
                   Studienfach     studienfach;
                   unsigned long   matrikelnummer;
                   unsigned short  uebungsnummer;
                   string          name_des_bremsers;
                   unsigned short  zahl_uebungspunkte;
                   bool            ist_zur_klausur_zugelassen;
                   float           note;
                };
```

Strukturierte Datentypen


```
// ... Student namens Paul ....
Student  paul;
paul.name = “Paul Hacker“;
paul.geschlecht = maennlich;
paul.matrikelnummer = 23456789;
paul.studienfach = Informatik;
// ... und so weiter

if (paul.zahl_uebungspunkte >= 60)
      Paul.ist_zur_klausur_zugelassen = true;
else  Paul.ist_zur_klausur_zugelassen = false;

//  Ein Feld vom Typ „vector“ zur Speicherung aller Studierender
vector<Student>   alle_unsere_schuetzlinge(500);
```

Strukturierte Datentypen

- Verwendung von Zeigern (Pointer):

```
Student  *einZeigerAufPaul = &paul;
einZeigerAufPaul->note = 1.7;
cout <<  einZeigerAufPaul->name  << “ hat mit der Note “
      <<  einZeigerAufPaul->note    << “ bestanden.“
      <<  endl;
```

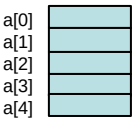
- Bei Zugriff auf Daten einer Struktur mittels eines Zeigers verwendet man den Pfeil-Operator „->“.

C-Array

- C-Array ist ein primitiver Datentyp für Felder:

```
int  a[5];      // Wir definieren ein Feld von ganzen Zahlen der Größe 5.
```

- Der Compiler reserviert hierbei einen Speicherblock von der Größe 5*sizeof(int):



- Initialisierung: Die in eckigen Klammern angegebene Feldgröße (im Beispiel 5) muss eine Konstante sein:

```
/* Initialisierung mit nicht-konstanten Variablen sind nicht erlaubt. */
int  groesse = 5;
int  a[groesse];            // -> Fehlermeldung des Compilers.
```

C-Array

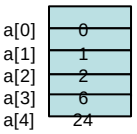
- Die Syntax einer einfachen Felddefinition lautet:

  Datentyp Feldname[Anzahl der Elemente];

- Hierbei muss „Anzahl der Elemente“
  - eine Konstante sein oder
  - ein Ausdruck, der ein konstantes Ergebnis hat.

- Der Zugriff auf das i-te Feldelement erfolgt mit dem [ ]-Operator:

```
int a[5];
int i;
for (i=0; i < 5; ++i)   a[i] = fakultaet(i);
```




```
int  k  =  a[i-1];              // k = a[4] = 4! = 24
int produkt = a[3]*a[4];        // produkt = 144
```

C-Array

- Die beiden folgenden Terme sind gleichwertig:

```
a[i]  = 50;
*(a + i) = 50;
```

- „a“ ist ein Zeiger (Pointer), der auf die Startadresse des Feldes zeigt.
- C-Arrays bieten eine einfache und komfortable Möglichkeit mehrdimensionale Felder (Matrizen) konstanter Größe zu definieren:

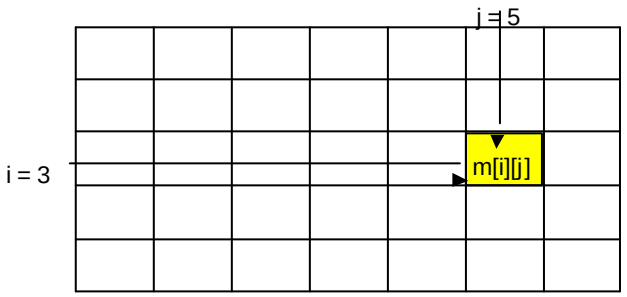
```
Datentyp Matrixname[Anzahl der Zeilen][Anzahl der Spalten];
```

- Hierbei müssen die Anzahl der Zeilen und die Anzahl der Spalten Konstanten sein oder Ausdrücke, die ein konstantes Ergebnis haben.

C-Array

- Der Zugriff auf das Element in der i-ten Zeile und der j-ten Spalte erfolgt mit dem [ ][ ]-Operator:

```
float m[5][7];          // Definition einer 5x7 Matrix vom Typ float
int i = 3, j = 5;
m[i][j] = 11.5;
m[i+1][j+1] = m[i][j] + 5.0;
```



C-Array

- Funktion zur Berechnung der transponierten Matrix A^T zu einer vorgegebenen Matrix A, wobei die Matrix A überschrieben wird:

$a_{ij}^T = a_{ji}$ für alle i, j ? { 0, ..., groesse }

```
void  transponiereMatrix( float a[ ][ ], int groesse) {
   float  a_ji;      // Hilfsvariable zum Zwischenspeichern von a[j][i]

   for (int i = 0; i < groesse ; i++)
     for (int j = i + 1; j < groesse ; j++) {
                                          a_ji = a[ j ][ i ];
                                          a[ j ][ i ] = a[ i ][ j ];
                                          a[ i ][ j ] = a_ji;
                                        }
}
```

C-Strings

- oder „char“-Felder:

```
char  *text  =  “Dies ist ein Beispiel fuer ein \“char\“-Feld“;
```

- Der Compiler reserviert hier Speicherplatz für alle Zeichen des obigen Literals plus ein Byte für das Abschlusszeichen `\0`.
- Das Abschlusszeichen `\0` wird vom Compiler ergänzt.

```
for (int i = 0; text[i] != `\0`; i++)  cout  <<  text[i];
cout << endl;
```

- In dem Header <cstring> (Datei string.h) findet man eine große Zahl von vordefinierten Funktionen für C-Strings.
- Diese haben durch die C++-Standardklasse „string“ an Bedeutung verloren.

Die C++-Klasse „vector“

- Vektoren sind komfortable Felder bzw. Tabellen.

```
vector<int> V(10);    // Definition und Deklaration
```

- Sie enthalten (size) viele Elemente desselben Datentyps <Datentyp>.
- Der Datentyp der Elemente ist beliebig wählbar.
- Der Zugriff erfolgt mit dem Klammeroperator [ ]:

```
for (size_t  i = 0; i < V.size(); i++)   V[i]  =  i*i;
```

- Vektoren sind dynamische Datenstrukturen:

```
V.push_back(10*10);            //  Es gibt nun V[10] mit Wert 100
cout <<  V.size();  << endl;   //  Größe ist 11
```

- Der Header <vector> muss eingebunden werden: #include<vector>

Funktionstemplates

- Gleiche Aufgabe für verschiedene Datentypen: Sortieren
- Schablonen (Templates) erlauben C++ Nutzern Funktionen mit parametrisierten Datentypen zu schreiben.
- Für den noch unbekannten Datentyp wird ein Platzhalter (Parameter) eingefügt.
- Die allgemeine Form für Templates ist:

```
template<class Typbezeichner> Funktionsdefinition
```

Funktionstemplates: Sortieren mit Mergesort


```
//mergesort.cpp
#include<vector>
using namespace std;

template<class T>
void merge(vector<T>& a, int left, int middle, int right) {
    vector<T> help;    // Hilfsvektor
    for (int i = left, int j = middle + 1; i <= middle && j <= right; ) {
        for ( ; a[i] <= a[j] &&  i <= middle; ++i)  help.push_back(a[i]);
        for ( ; a[i]  > a[j] &&  j <= right ; ++j)  help.push_back(a[j]);
    }
    for ( ; i <= middle; ++i)  help.push_back(a[i]);// Rest des linken Felds
    for ( ; j <= right; ++j)  help.push_back(a[j])     // R. des rechten Felds

    for (int i = 0, int j = left; j <= right; ++i, ++j)  a[j] = help[i];
    delete [ ] help;
}
```

Funktionstemplates: Sortieren mit Mergesort


```
// Fortsetzung der Datei: mergesort.cpp


template<class T>
void mergesort(vector<T>& a, int left, int right) {
   int middle = (right – left)/2;
   if (middle >=1) {
     middle += left;
     mergesort(a, left, middle);
     mergesort(a, middle+1, right);
     merge(a, left, middle, right);
   }
}
```

Funktionstemplates

- Einbinden von Templates:
  - Eine *.o Datei, die durch Übersetzen einer Datei mit Templates erzeugt wurde, enthält **keinen Programmcode und keine Daten**.
  - Templates sind keine Funktionsdefinitionen im üblichen Sinn, sondern eben Schablonen, mit der der Compiler **bei Bedarf** eine Funktion zu einem konkreten Datentyp erzeugt:

```
int main ( ) {
   vector<int>  vint;
   // vint wird gefüllt.
   mergesort(vint, ..., ...);

   vector<float> vfloat;
   // vfloat wird gefüllt.
   mergesort(vfloat, ..., ....);
}
```

Compiler erzeugt ein erstes echtes Programm, in dem er „T“ in der Schablone durch „int“ ersetzt und das resultierende Programm übersetzt und einbindet.

Compiler erzeugt ein zweites echtes Programm, in dem er „T“ in der Schablone durch „float“ ersetzt und das resultierende Programm übersetzt und einbindet.

Funktionstemplates

- In den Funktionstemplates kann man Variablen vom Typ T (<class T> oder <typename T>) deklarieren, definieren und man kann mit diesen Variablen arbeiten:

```
T  a, b, c;
a = b;                    // Der Zuweisungsoperator muss in der
                          // Klasse T definiert sein.
if (a < c)  funct1(a,c);  // Der Vergleichsoperator muss in T
                          // definiert sein und die Funktion funct1
                          // mit den entsprechenden Parametern
                          // muss definiert (implementiert) sein.
```

- Man beachte: Alle Operatoren und Funktionen, die in Verbindung mit den entsprechenden Variablen aufgerufen werden, müssen für die Klassen bzw. für den Datentyp T definiert sein.

Funktionstemplates: Einbinden von Templates

– Template-Dateien mit **#include** einlesen:

```
// my_template.h
#ifndef my_template_h
#define my_template_h
// hier folgen die Template-Deklarationen (Prototypen)
//...
//...
// hier werden die Definitionen eingebunden
#include “my_template.cpp“
#endif
```

– Man beachte: Hier macht es Sinn, die Funktions-deklarationen und –definitionen in der gleichen Datei zu speichern.