# Ein- und Ausgabe von Dateien



```
//  iostream.h                      ist in fstream.h eingebunden.
#include<fstream>                   // für Ein- und Ausgabe von Dateien
#include<string>

using namespace std;

int main( )  {
        //  definieren und öffnen der Eingabedatei
        ifstream quelle;            //  Datentyp für den Eingabestrom
        string  quelldateiname;
        cout << "Bitte Namen der Quelldatei angeben:";
        cin >> quelldateiname;
        quelle.open(quelldateiname.c_str(), ios::binary|ios::in);

        if (!quelle) {  // falls quelle nicht geöffnet, Prog. beenden
                return( -1);
        }
```



# Ein- und Ausgabe von Dateien



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

        if (!ziel) {  // falls ziel nicht geöffnet, Prog. beenden
                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 Dateien

3

- Beim Öffnen von Dateien werden neue Ein- und Ausgabeströme (Objekte) eingerichtet.

```
ifstream quelle;      // Variable vom Typ „ifstream“  mit Namen quelle
ofstream ziel;        // Variable vom Typ „ofstream“  mit Namen ziel
```

- 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);
```

- Die Funktion close() schließt die entsprechende Datei.

Breymann_Folien

## Ganze Zahlen

4

- Datentypen für ganze Zahlen:
  - short 16 Bit unsigned short
  - int 32 Bit (16 Bit?) unsigned int
  - long 64 Bit (32 Bit?) unsigned long
- Zahlenbereich bei 32 Bit mit Vorzeichen:
  $-2^{31}$ .... $2^{31}-1$
- 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;
}
```

- Die Operatoren für Ausdrücke mit ganzen Zahlen finden Sie im Breymann Skript auf den Seiten 22-25.

Breymann_Folien

## Reelle Zahlen

5

- 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
- Operatoren für Gleitkommazahlen (siehe Breymann-Skript Seite 28-29).
- Weitere mathematische Funktionen können mittels der Mathe-Bibliothek eingebunden werden:
  #include<cmath>

Breymann_Folien

## Vorrangregeln

6

- 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!

Breymann_Folien

# 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´;
```

- Besondere Zeichenkonstanten der ASCII-Tabelle, die nicht direkt im Druck sichtbar sind, werden als Folge von zwei Zeichen geschrieben und als Escape-Sequenzen “\“ bezeichnet:

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



# Zeichen



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

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

- 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“
using namespace std;

int main() {               // Beispielprogramm mit typischen
                           //  String-Operationen
// String-Objekt definieren und initialisieren
string einString(“hallo“);

// String ausgeben
cout << einstring << endl;                  //  hallo
```



## Strings



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

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

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

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

 eineStringKopie = “Buchstaben“;  //  Zeichenkette zuweisen
 cout << eineStringKopie << endl;   // Buchstaben
```

## Strings



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

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

 // Strings mit dem + Operator verknüpfen
 einString = eineStringKopie + " ABC";
 cout << einString << endl;                     //  Buchstaben ABC

 // Strings mit dem + Operator verknüpfen
 einString = "123" + eineStringKopie ;
 cout << einString << endl;                     //  123Buchstaben

 //  nicht erlaubt ist:  einstring = " ABC" + "123";   Erklärung folgt später

 }  // Ende von main()
```

Breymann_Folien

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

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

Breymann_Folien

## Logischer Datentyp: bool

13

- Operatoren für Boolesche Ausdrücke: a und b seien Variablen 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:

Breymann_Folien

## Zeiger = Pointer

14

- 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:

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

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

- NULL ist als logischer Wert abfragbar:

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

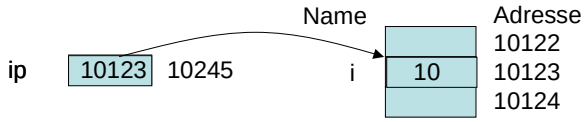
Breymann_Folien

# Zeiger = Pointer

15

- Ein Beispiel für den Verwendung von Zeigern:

```
int  i = 99;        // eine int-Variable i mit Wert 99
                    // wir nehmen an, dass der Compiler für i die
                    // Speicherplatzadresse 10123 festlegt.
```




```
int  *ip;
ip = &i;        //  ip wird die Adresse von i mit dem Adress-
                //  Operator & zugewiesen
*ip = 10;       //  mit dem Stern-Operator können wir nun den
                //  Wert von i (den Wert an der Speicherplatz-
                //   adresse 10123) ändern.
```

Breymann_Folien

# Zeiger = Pointer

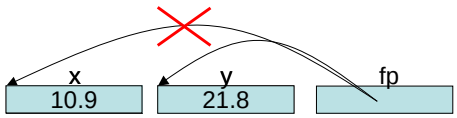
16

- Operatoren:

| | | |
|---|---|---|
| & | liefert die Adressen von Variablen: | `ip = &i;` |
| * | ermöglicht die Veränderung des Werts der entsprechenden Speicherzelle, auf die der Zeiger zeigt, oder den Zugriff auf den dort gespeicherten Wert | `*ip = 10;`<br>`int j = *ip;` |
| ++, --, +, - ,+=,-= | ermöglichen die Veränderung der gespeicherten Adresse | `++ip;` |

- Da Zeiger auch „Variablen“ sind, kann man die dort gespeicherten Adressen auch ändern, so dass man sich mit Zeigern im Speicher bewegen kann:

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




Breymann_Folien

## Unterprogramme/Funktionen



```
#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;
        cout << " Der GGT von " << x << " und " << y << " ist: ";

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

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



## Unterprogramme/Funktionen



- Eine Funktionsdefinition (Implementation) veranlasst den Compiler entsprechenden Code zu erzeugen.
- Syntax der Funktionsdefinition (Implementation):

Rückgabetyp Funktionsname (Formalparameterliste) Block

Unser Beispiel :

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

- Hier müssen zwingend Namen für die Eingabeparameter angegeben werden, mit denen dann auch im Block gearbeitet werden kann.
- Die Namen der Eingabeparameter sind frei wählbar und völlig unabhängig vom Aufruf.
- Die Rückgabe des Resultats erfolgt mit der „return“-Anweisung.

## Unterprogramme/Funktionen



- Syntax des Funktionsaufrufs:

  Funktionsname (Aktualparameterliste);

  Unser Beispiel : resultat = ggt( x, y);

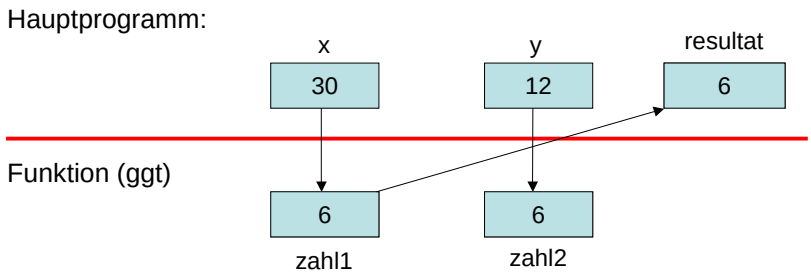
- Die Aktualparameterliste enthält die Namen der Variablen (Objekte) oder Ausdrücke, die an die Funktion übergeben werden sollen und für die die Funktion ausgeführt werden soll.
- Die Datentypen, die Zahl und Reihenfolge der Parameter müssen natürlich mit der Funktionsdeklaration/Funktions-definition übereinstimmen.



## Unterprogramme/Funktionen



- In unserem Beispiel erfolgte die Parameterübergabe per Wert:




- Später werden wir andere (effizientere) Möglichkeiten der Parameterübergabe kennen lernen.

# Parameterübergabe per Zeiger



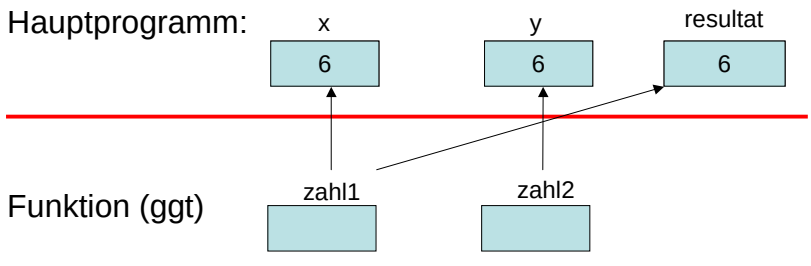
```
#include<iostream>
using namespace std;

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

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

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

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



# Übergabe per Referenz



- In unserem zweiten Beispiel erfolgte die Parameterübergabe per Referenz:



## Referenzen



- Eine Referenz ist ein Datentyp, der einen Verweis auf ein Objekt liefert.
- Eine Referenz ist ein Alias-Name für ein Objekt, über den es ansprechbar ist. Beispiel:

```
int i = 2, j = 9;
int &r = i;          // Referenz auf i  (r ist Alias für i)
r = 10;              // ändert i
r = j;               //  Wirkung :  i = j
```

- 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.



## Referenzen



```
int i = 2, j = 9;
int &r = i;          // Referenz auf i  (r ist Alias für i)
r = 10;              // ändert i
r = j;               //  Wirkung :  i = j
```

- Eine Variablendefinition reserviert im Speicher einen gewissen Bereich.
- Die Größe des reservierten Speicherplatzes ist abhängig vom Datentyp der Variable.
- Durch Zuweisungen über den Namen der Variablen können wir die dort gespeicherten Daten verändern.
- Eine Referenz auf eine Variable ist einfach ein zusätzlicher Name (Alias) für Zugriffe auf diesen Speicherplatz.

# Parameterübergabe per Referenz



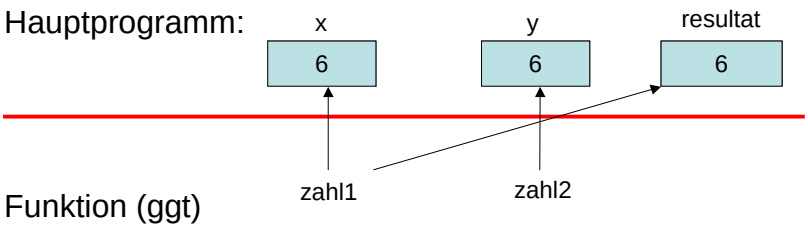
```
#include<iostream>
using namespace std;

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

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

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

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



# Übergabe per Referenz



- In unserem zweiten Beispiel erfolgte die Parameterübergabe per Referenz:



## Aufzählungs- oder Enumerationstypen

27

- 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 = freitag, morgen;
if (heute == freitag)   morgen = samstag;
```

- Syntax der Deklaration/Definition:

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

Die eckigen Klammern bedeuten, dass Typname oder Variablenliste weggelassen werden können.

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

Breymann_Folien

## Strukturierte Datentypen

28

- Um logisch zusammengehörende Daten von verschiedenen Datentypen zusammenfassen zu können, stellt C/C++ die Struktur(en) zur Verfügung:

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

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;
                };
```

- Syntax von Strukturen:

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

Breymann_Folien

## Strukturierte Datentypen



- Beispiel-„Student“:

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

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

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



## Strukturierte Datentypen



- Beispiel-„Student“:

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

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 „->“.

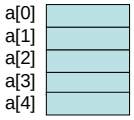
# C-Arrays

31

- Vektoren „vector< ... >“ sind aus der ersten Übung bekannt.
- C-Arrays sind ein primitiver C-Felddatentyp, der ähnlich zu Vektoren, aber viel primitiver ist.

```
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):




- Bei Initialisierungen der obigen Form muss die in eckigen Klammern angegebene Feldgröße (im Beispiel 5) eine Konstante sein:

```
// Nicht erlaubt ist eine Initialisierung mit einer nicht-konstanten Variablen :
int  groesse = 5;
int  a[groesse];            // -> Fehlermeldung des Compilers.
```

Breymann_Folien

# C-Arrays

32

- 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  = 3;
a[i] = 50;
int k  =  a[i];
```

- Hierbei sind folgende Terme gleichwertig:

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

- Der Name des Feldes „a“ zeigt also auf die Startadresse des Feldes = „a“ ist ein Zeiger.

Breymann_Folien

## C-Arrays



- C-Arrays bieten eine einfache und komfortable Möglichkeit mehrdimensionale Felder (Matrizen) konstanter Größe zu definieren:

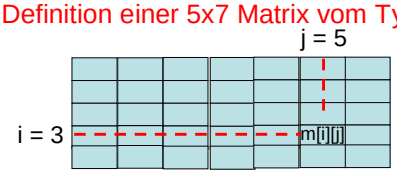
  Datentyp Matrixname[Anzahl der Zeilen][Anzahl der Spalten];

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

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

```
int m[5][7];
int i = 3, j = 5;
int k  =  m[i][j];
m[i+1][j+1] = m[i][j] + 5;
```

```
//  Definition einer 5x7 Matrix vom Typ int
```




Breymann_Folien

## C-Arrays



- 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 $\in$ { 0, ...., groesse }

```
void  transponiereMatrix( int a[ ][ ], int groesse) {
   int  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;
                                        }
}
```

Breymann_Folien

## 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.

Breymann_Folien