

Abend 20

Zeiger und Vektoren, Dateiverarbeitung

Ziel

- Wir lernen heute, wie man mit Zeigern auf Elemente in Arrays zugreift und wie man mit Zeigern rechnen kann.
- Wir lernen mit Dateien ähnlich umzugehen wie mit den cin- und cout-Objekten mit dem Unterschied, dass wir anstelle der Konsole Dateien verwenden

Zeiger und Vektoren

Gleichheit von Zeigern und Vektoren

Definieren wir ein Array

```
int einArray[5] = { 9, 8, 7, 6, 5 };
```

so steht der Name *einArray* nicht nur für das ganze Array sondern ist auch ein Zeiger auf das erste Element des Arrays! Nehmen wir an, dass das Array im Speicher an der Adresse 1000 beginnt, das sieht unser Speicher ungefähr so aus:

	Adresse	einArray
einArray	1000	9
	1004	8
	1008	7
	1012	6
	1016	5

Den Beweis liefert ein kleines Testprogramm:

```
#include <iostream>

using namespace std;

int main()
{
    int einArray[5] = { 9, 8, 7, 6, 5 };
```

```
int* zeigerAufErstesElement = &einArray[0];
cout << "Wert von einArray : ";
cout << einArray << endl;
cout << "Adresse des ersten Elementes : ";
cout << zeigerAufErstesElement << endl;

return 0;
}
```

Das Programm gibt bei mir folgende Werte aus :

```
Wert von einArray : 0012FEC4
Adresse des ersten Elementes : 0012FEC4
```

Beachte, dass wir nicht die Elemente im Array ausgeben, sondern das was der Compiler für *einArray* intern benutzt, nämlich eine Adresse.

Der Name, dem wir einem Array geben ist zugleich ein konstanter Zeiger auf das erste Element im Array! Konstant deshalb, weil wir ihn nachträglich nicht mehr neu zuweisen können.

Ein weiteres Beispiel, diesmal mit Zeiger auf char zeigt, dass allgemein gilt:

char[] \approx char*
(gilt auch für andere Datentypen!)

```
#include <iostream>

using namespace std;

int main()
{
    char einText[] = "Hallo";
    char* auchEinText = "Auch Hallo";

    cout << einText << endl;
    cout << auchEinText << endl;

    return 0;
}
```

Beide Variablen *einText* und *auchEinText* werden gleich behandelt, beide Texte erscheinen auf der Console.

Was auch möglich ist:

```
#include <iostream>

using namespace std;

int main()
{
    char einText[] = "Hallo";
    char* auchEinText = einText;
```

```
        cout << einText << endl;
        cout << auchEinText << endl;

        return 0;
    }
```

Da *einText* ja nichts anderes als ein Zeiger auf das erste Element des Arrays ist, können wir den Zeiger *auchEinText* auf die gleiche Stelle zeigen lassen.

Adressierung mit Zeigern und Vektoren

Um Elemente aus Vektoren/Arrays einzeln anzusprechen verwenden wir die Index-Schreibweise:

```
#include <iostream>

using namespace std;

int main()
{
    int einArray[5] = { 78, 4, 9, 33, 78 };

    int zweitesElement = einArray[1];

    cout << zweitesElement << endl;

    return 0;
}
```

Aufgrund der Aussage weiter oben, sollte folgendes auch möglich sein:

```
#include <iostream>

using namespace std;

int main()
{
    int einArray[5] = { 78, 4, 9, 33, 78 };
    int* zeiger = einArray;

    int zweitesElement = zeiger[1];

    cout << zweitesElement << endl;

    return 0;
}
```

Tatsächlich können wir einen Zeiger verwenden als wäre er ein Vektor/Array. Mit einem Zeiger können wir aber auch rechnen. Die Variable *zeiger* zeigt zuerst auf das erste Element im Array. Wir können den Zeiger einfach um eins erhöhen, danach sollte er auf das zweite Element zeigen!

```
#include <iostream>

using namespace std;

int main()
{
    int einArray[5] = { 78, 4, 9, 33, 78 };
    int* zeiger = einArray;

    // zeiger um eins erhöhen
    zeiger = zeiger + 1;
    // hole Element worauf
    // zeiger zeigt
    int zweitesElement = *zeiger;

    cout << zweitesElement << endl;

    return 0;
}
```

Hier eine kleine Tabelle, die diesen Zusammenhang verdeutlichen soll.

zeiger	78	einArray[0]
zeiger+1	4	einArray[1]
zeiger+2	9	einArray[2]
zeiger+3	33	einArray[3]
zeiger+4	78	einArray[4]

Das heisst dass folgende Ausdrücke gleichwertig sind :

Adresse des dritten Elementes:

&einArray[2] zeiger + 2

Wert des dritten Elementes:

einArray[2] *(zeiger + 2)

Rechnen mit Zeigern

Zeiger können wie wir bereits gesehen haben durch Addition verändert werden. Genauso können wir auch subtrahieren oder mit den ++ oder -- Operatoren Zeiger verändern. Mit einem kleinen Beispiel wollen wir das auch probieren. Wenn wir einen Zeiger auf int (int*) mit cout verwenden erscheint in der Konsole der Wert des Zeigers. Verwenden wir aber ein Zeiger auf char (char*) ist das Verhalten anders. Hier werden alle Zeichen ausgegeben bis eine abschliessende 0 erscheint.

Wir schreiben eine kleine Funktion, die genau das gleiche macht, wie wenn man einen char* mit << an das cout schickt:

```
#include <iostream>

using namespace std;

// Funktionsprototyp
void TextAusgeben(char* text);

int main()
{
    char* einText = "Hallo";
    // ich hätte genauso schreiben können
    // char einText[] = "Hallo";

    cout << einText;
    cout << endl;
    // Diese Funktion soll das gleiche machen
    // wie cout << einText
    TextAusgeben(einText);
    cout << endl;

    return 0;
}

// TextAusgeben Funktionsdefinition
void TextAusgeben(char* text)
{
    char zeichen = *text;
    while(0 != zeichen)
    {
        // einzelnes Zeichen ausgeben
        cout << zeichen;
        // Zeiger um eins erhöhen
        text++;
        zeichen = *text;
    }
}
```

Die Funktion *TextAusgeben* könnte jedoch auch so aussehen, wenn wir anstatt der Zeiger- die Index-Schreibweise verwenden:

```
// TextAusgeben Funktionsdefinition
void TextAusgeben(char text[])
{
    int index = 0;
    while(0 != text[index])
    {
        // einzelnes Zeichen ausgeben
        cout << text[index];
        // Index um eins erhöhen
        index++;
    }
}
```

Wir können auch einen Zeiger von einem anderen subtrahieren:

```
#include <iostream>

using namespace std;

int main()
{
    char* einText = "Hallo";
    char* zeigerAufDas0 = &einText[4];

    int indexVon0 = zeigerAufDas0 - einText;

    return 0;
}
```

Vektoren/Arrays von Zeigern

Wir kennen bereits Arrays von char (C-Strings) von int und von anderen selbstdefinierten Klassen (CDListe, Notenliste). Es ist auch möglich Arrays von Zeigern zu bilden.

Nehmen wir an wir wollen Termine verwalten und merken uns dafür auch den Wochentag. Dafür definieren wir folgenden Aufzählung:

```
enum Wochentag
{
    Montag = 0,
    Dienstag,
    Mittwoch,
    Donnerstag,
    Freitag,
    Samstag,
    Sonntag
};
```

Zu jedem Wochentag gehört ein string, den wir für eine Ausgabe verwenden können. Schön wäre wir könnten ein Array bilden, bei dem beim Index = 0

der string für den Montag steht, bei Index = 1 der für Dienstag.
Einzelne Strings zu definieren nützt hier nichts :

```
char* MontagEng = "Monday";  
char* DienstagEng = "Dienstag";
```

Aber folgendes ist möglich, wir definieren ein Array von Zeigern :

```
#include <iostream>  
  
using namespace std;  
  
enum Wochentag  
{  
    Montag,  
    Dienstag,  
    Mittwoch,  
    Donnerstag,  
    Freitag,  
    Samstag,  
    Sonntag  
};  
  
char* WochentageEng[] =  
{  
    "Monday",  
    "Tuesday",  
    "Wednesday",  
    "Thursday",  
    "Friday",  
    "Saturday",  
    "Sunday"  
};  
  
int main()  
{  
    Wochentag einWochentag = Donnerstag;  
  
    char* WochentagString = WochentageEng[einWochentag];  
  
    cout << WochentagString << endl;  
  
    return 0;  
}
```

WochentageEng ist also ein Array von Zeigern. Oder auch anders ausgedrückt : ein Array von Arrays, oder auch ein mehrdimensionales Array. Hier ein einfacheres Beispiel für ein mehrdimensionales Array, eine Matrix:

```
int matrix[2][2] =  
{  
    { 11, 12 },  
    { 21, 22 }  
};
```

Wobei die erste Dimension angibt wieviele Zeilen (sozusagen y) unsere Matrix hat und die zweite wieviele Kolonnen (entsprechend dem x).

Argumente an die main-Funktion

Erinnert ihr euch an das Registrieren des Programms TsuZeichnen.exe? Das Programm musste dafür so gestartet werden:

```
c:\c++\tsuzeichnen.exe -regserver
```

Dieser string „-regserver“ ist ein sogenanntes Programm-Argument. Es ist nämlich möglich einem Programm direkt beim Starten Parameter zu übergeben. Um in einem Programm Argumente zu verwenden muss unsere main-Funktion anders definiert werden. Betrachte folgendes Beispiel:

```
#include <iostream>

using namespace std;

// dieses main erhält die Anzahl Argumente
// und ein Array von C-Strings wo diese Argumente
// drinstecken
int main(int anzahlArgumente, char* argumente[])
{
    if(anzahlArgumente > 0)
    {
        cout << "Dieses Programm hat den Pfad" << endl;
        cout << argumente[0] << endl;
    }

    if(anzahlArgumente <= 1)
    {
        cout << "Keine weiteren Argumente :(" << endl;
    }
    else
    {
        for(int i = 1; i < anzahlArgumente; ++i)
        {
            char* argument = argumente[i];
            cout << i << ". Argument : ";
            cout << argument << endl;
        }
    }

    return 0;
}
```

Das erste Argument im Array ist der Pfadname, mit dem das Programm gestartet wurde. Die nächsten sind Argumente, die wir dem Programm zusätzlich übergeben können.

Dateiverarbeitung

Die File-Stream Klassen

Wie die *iostream*-Klassen, die wir zur Ein- und Ausgabe von der Konsole verwenden, gibt es Klassen um Dateien zu schreiben und zu lesen. Das *cin*-Objekt ist ein Objekt der Klasse *istream*. Das Objekt *cout* ist von der Klasse *ostream*. Um in eine Datei zu schreiben, erzeugen wir ein Objekt der Klasse *ofstream* (beachte das f für file). Um aus einer Datei zu lesen verwenden wir ein Objekt der Klasse *ifstream*.

Grundfunktionen der File-Stream Klassen

Die wichtigsten Funktionen der File-Stream Klassen sind sicherlich die Konstruktoren. Beim Konstruktor der Klasse *ofstream* oder *ifstream* ist es möglich direkt einen Dateinamen anzugeben.

Wie bei *cin* und *cout* ist es möglich mit den Operatoren << und >> in eine Datei zu schreiben beziehungsweise zu lesen.

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
    // Datei erzeugen und öffnen
    ofstream outFile("Test.txt");

    // ist die Datei wirklich offen ?
    if(outFile.is_open())
    {
        // genau wie in das cout-Objekt
        outFile << "Hallo Datei" << endl;
    }

    // Datei schliessen, damit wir diese
    // wieder öffnen können
    outFile.close();

    // Datei zum Lesen öffnen
    ifstream inFile("Test.txt");

    if(inFile.is_open())
    {
        string text;
        // lese Linie bis zum
        // endl (ohne endl)
        getline(inFile, text);

        cout << "In der Datei steht : ";
        cout << text << endl;
    }
}
```

```
        // inFile wird automatisch geschlossen
        // wenn das Objekt inFile zerstört wird
        // also ist close nicht unbedingt nötig
        inFile.close();

    return 0;
}
```

Um eine ganze Zeile aus einer Datei zu lesen verwenden wir die Funktion *getline*.

Weitere Funktionen der File-Stream Klassen

Mit *close* wird eine Datei wieder geschlossen. Die Datei wird auch geschlossen, wenn das File-Stream Objekt zerstört wird. Wichtig ist das im Beispiel oben, weil wir eine Datei zuerst zum Schreiben erzeugen und diese danach zum Lesen wieder öffnen. Es kann aber sein, dass eine Datei nicht gleichzeitig zum Schreiben und Lesen geöffnet werden kann.

Mit der Methode *is_open* wird geprüft, ob eine Datei geöffnet werden konnte.

Probleme beim Mischen von Text und Zahlen

Text und Zahlen beim Schreiben in ein Datei zu mischen stellt kein Problem dar:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
    // Datei erzeugen und öffnen
    ofstream outFile("Test.txt");

    // ist die Datei wirklich offen ?
    if(outFile.is_open())
    {
        // genau wie in das cout-Objekt
        outFile << "Datei" << endl;
        outFile << 20 << endl;
        outFile << "Zeile 1" << endl;
        outFile << 30 << endl;
        outFile << "Zeile 2" << endl;
        outFile << 40 << endl;
        outFile << "Zeile 3" << endl;
    }

    return 0;
}
```

Die Aufgabe, die es zu lösen gilt ist, die Textzeilen in ein *string*-Objekt zu lesen und die Zahlen in eine *int*-Variable. Folgender Code funktioniert nicht!

```
// Datei zum Lesen öffnen
ifstream inFile("Test.txt");

if(inFile.is_open())
{
    string text;
    int zahl;

    getline(inFile, text);
    inFile >> zahl;
    getline(inFile, text);
    inFile >> zahl;
    getline(inFile, text);
    inFile >> zahl;
    getline(inFile, text);
}
```

Die erste Zeile und die erste Zahl werden korrekt eingelesen, aber die nachfolgenden nicht mehr. Nach dem ersten Einlesen einer Zahl mit `inFile >> zahl;` bleibt das endl im File-Stream und das nachfolgende `getline` liest nur bis zum nächsten endl, der string bleibt leer. Danach stimmt nichts mehr, denn das Lesen der nächsten Zahl schlägt auch fehl... etc.

Mit der Methode *ignore* können wir das endl aus dem stream auslesen und ignorieren, danach funktioniert das Lesen so wie wir uns das vorstellen:

```
if(inFile.is_open())
{
    string text;
    int zahl;

    getline(inFile, text);
    inFile >> zahl;
    inFile.ignore();
    getline(inFile, text);
    inFile >> zahl;
    inFile.ignore();
    getline(inFile, text);
    inFile >> zahl;
    inFile.ignore();
    getline(inFile, text);
}
```

Ähnlichkeit von stream-Klassen

Wir können ein Objekt der Klasse *ofstream* anstelle eines Objektes der Klasse *ostream* verwenden. Auf diese Weise haben wir in der Übung *Notenliste* zur Ausgabe nur eine Funktion geschrieben, die als Parameter ein Objekt der Klasse *ostream* verwendet. Dieser Funktion können wir als Parameter das *cout*-Objekt übergeben genauso wie ein *ofstream*-Objekt, das wir als Datei erzeugen.