Adressen und Zeiger

Das Arbeiten mit Adressen und Zeigern hat in der Programmiersprache C++ für den Software – Entwickler eine sehr große Bedeutung. Mit Zeigern lässt sich der Computerspeicher direkt manipulieren. Daher gehören Zeiger einerseits zu den leistungsfähigsten Werkzeugen in der Hand des C++ Programmierers, andererseits aber auch zu den schwierigsten Aspekten von C++ (im Hinblick auf mögliche Programmierfehler!).
Im ersten Teil dieses Skripts werden zunächst die Grundlagen zur Verwendung von Zeigern erläutert. Im zweiten Teil ist dann mit Hilfe von Zeigern und der C – Headerdatei <cstring> ein Software – Adressbuch zu erstellen.

1. Zeiger (Pointer)

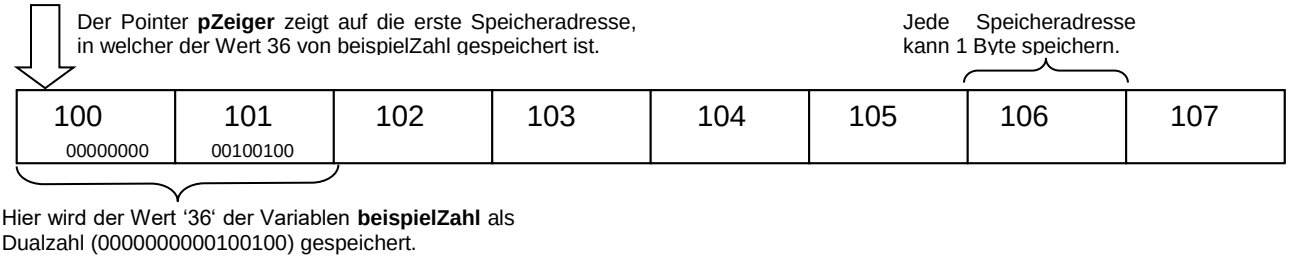
Programme beherbergen die Werte von Variablen im Speicher des Computers. Bei einem *Pointer* handelt es sich um nichts anderes als um eine *Speicher – Adresse*: Ein Pointer zeigt sozusagen auf einen bestimmten Speicherplatz (daher der Name „Pointer“ = Zeiger).

Ein Zeiger ist also eine Variable, die eine Speicheradresse enthält.

Will man den unter einer Adresse gespeicherten Wert im Programm verwenden, so kann man mit Hilfe des Zeigers direkt darauf zugreifen. Der Zeiger dient also als „Merkhilfe“ für den Programmierer, da er sich nur den Zeigernamen merken muss und nicht die komplizierte Speicheradresse.

Schematische Darstellung des Speichers:

- Eine Ganzzahl (short) benötigt 2 Byte Speicherplatz im Speicher des Computers.
- Der Wert 36 der Ganzzahl - Variable `short beispielZahl = 36;` wird im Computerspeicher z.B. unter den Adressen 100 und 101 als Dualzahl gespeichert.
- Der Zeiger pZeiger zeigt auf die erste Speicheradresse: `short* pZeiger = &beispielZahl;`



Im Programm muss auch der Typ (hier: **short*** pZeiger) angegeben werden, auf den der Pointer pZeiger zeigt. Dadurch ist dem Compiler bekannt, dass er zwei Speicheradressen (100 und 101 für 2 Byte beim Typ **short**) auslesen muss, wenn der Zeiger im Programm aufgerufen wird. Deklariert man den Zeiger für den Typ **float** (float* pZeiger), so werden beim Aufruf des Zeigers automatisch die Speicheradressen 100 bis 103 ausgelesen, da der Typ **float** 4 Bytes Speicher benötigt.
Der Programmierer muss bei der Verwendung von Zeigern also nur den Zeigernamen und nicht die Speichergröße kennen. Der Compiler nimmt ihm diese Arbeit ab.

1.1 Die Deklaration von Zeigern

Ein Zeiger muss, genau wie jede andere Variable in C++ deklariert werden. Um einen Zeiger im Programm bereits am Namen zu erkennen, wird am Anfang eines Zeigernamens ein „p“ vorangestellt; z.B. pmeinZeiger. Dem Zeigernamen wird bei der Deklaration der Verweisoperator ***** vorangestellt:

Zeiger deklariert man, indem man den Typ angibt, auf den der Zeiger zeigt, und danach den Verweisoperator (*****) und den Zeigernamen schreibt. Das “p“ vor dem Zeigernamen kennzeichnet einen Zeiger (Pointer).

```
Typ* Zeigername;
```

z.B.: `int* pZeiger;`

Der Pointer pZeiger ist nun deklariert, aber auf welche Adresse zeigt er? 💣 Vorsicht, hierin liegt eine große Gefahr beim Arbeiten mit Zeigern. Weist man dem Zeiger keine Adresse zu, so deutet er auf *irgendeine* Adresse im Speicher des Computers. Ruft man den Zeiger dann im Programm

auf, kann nicht vorhergesagt werden, was geschieht. Bestenfalls stürzt das Programm ab, eventuell formatieren Sie aber auch gerade Ihre Festplatte ☺.

Um diese Gefahr zu vermeiden initialisiert man Zeiger bei der Deklaration mit der Adresse NULL. Jetzt kann beim versehentlichen Aufruf des Zeigers nichts mehr schiefgehen, da von dieser Adresse nicht gelesen werden kann.

```
int* pZeiger = NULL;
```

Bevor man den Zeiger später im Programm nutzt, muss ihm eine Adresse zugewiesen werden.

1.2 Die Initialisierung von Zeigern

Ein Zeiger ist eine Variable, die eine Speicheradresse speichert. Darum muss einem Zeiger auch eine Adresse zugewiesen werden. Dies erfolgt in C++ mit Hilfe des *Adressoperators* &. Der Adressoperator & liefert die Adresse, unter welcher der Wert einer Variablen gespeichert ist.
Betrachten wir uns folgenden Programmausschnitt:

Zunächst wird die Variable zahl deklariert und mit dem Wert 16 initialisiert. Danach deklariert man den Zeiger pzahl und weist ihm mit dem Adressoperator & die Speicheradresse zu, unter welcher der Wert 16 der Variablen zahl gespeichert ist.

```
...
int zahl = 16;
int* pzahl = &zahl;
...
```

Der Zeiger pzahl deutet jetzt auf die Adresse im Computerspeicher, in welcher der Wert 16 der Variable zahl abgespeichert ist. Man muss dem Zeiger also eine Adresse zuweisen!

☞ **Was würde die Zuweisung `int* pzahl = zahl` ohne den Adressoperator bewirken?**

1.3 Adresse oder Wert?

Beim Arbeiten mit Zeigern muss man sich immer bewusst machen, ob man gerade mit einer Adresse oder mit einem Wert hantiert.

Um mit der *Adresse* eines Zeigers zu arbeiten, verwendet man den Zeigernamen:

Die Anweisung cout << pzahl; gibt die Adresse auf dem Bildschirm aus, auf die der Pointer zeigt.

```
cout << pzahl;
```

Um mit dem *Wert* zu arbeiten, der unter der Adresse gespeichert ist, auf die der Pointer zeigt, benutzt man den *Indirektionsoperator* ***** und den Zeigernamen. Man greift „indirekt“ (über die Speicheradresse) auf den gespeicherten Wert zu:

Die Anweisung cout << *pzahl; gibt den Wert auf dem Bildschirm aus, auf dessen Speicheradresse der Pointer zeigt.

```
cout << *pzahl;
```

Als kurze Merkregel gilt:

Wird der Zeiger **ohne** * benutzt, geht es um **Adressen**, wird der Zeiger **mit** * benutzt, geht es um **Werte**!

1.4 Ein kurzes Beispielprogramm

Zuerst wird die Variable zahl deklariert und mit dem Wert 16 initialisiert. Danach deklariert man den Zeiger pzahl und weist ihm NULL zu. Im nächsten Schritt wird dem Zeiger pzahl mit dem Adressoperator die Adresse zugewiesen, unter welcher der Wert der Variablen zahl gespeichert ist.
Danach werden jeweils der Wert und die Adresse der Variablen zahl und des Zeigers pzahl ausgegeben.
Zuletzt weist man der Variablen zahl den neuen Wert 17 zu und gibt wieder Wert und Adresse von Variable und Zeiger aus.
Wie zu erwarten ist, sind sowohl die Werte von zahl und *pzahl als auch die Adressen &zahl und pzahl bei der Ausgabe gleich. Ändert man den Wert der Variable zahl, so wird der neue Wert an der alten Adresse gespeichert.

```
// C++ Datei zur Funktion von Zeigern:
#include <iostream>
using namespace std;
int main( )
{
    int zahl = 16;
    int*  pzahl = NULL;

    pzahl = &zahl;

    cout << "\nDer Wert der int - Variable 'zahl':\t" << zahl;
    cout << "\nDie Adresse der int - Variable '&zahl':\t" << &zahl;
    cout << "\nDer Wert, auf den '*pzahl' deutet:\t" << *pzahl;
    cout << "\nDie Adresse, auf die 'pzahl' deutet:\t" << pzahl;

    cout << "\n\nJetzt aendern wir den Wert nach 'zahl = 17'!\n";
    zahl = 17;

    cout << "\nDer Wert der int - Variable 'zahl':\t" << zahl;
    cout << "\nDie Adresse der int - Variable '&zahl':\t" << &zahl;
    cout << "\nDer Wert, auf den '*pzahl' deutet:\t" << *pzahl;
    cout << "\nDie Adresse, auf die 'pzahl' deutet:\t" << pzahl;

    cout << "\n\n";
    return 0;
}
```

1.5 Die Bedeutung von Zeigern

Warum verwendet man eigentlich Zeiger, wenn im Programm auch mit „ganz normalen“ Variablen gearbeitet werden kann?

Eine Erklärung für diese Frage bietet das **Beispielprogramm** zum Vertauschen von Zahlen:

Im Hauptprogramm werden die Variablen a=8 und b=5 definiert und auf dem Bildschirm ausgegeben. Danach ruft man die Funktion Vertausche(a, b) auf und übergibt die Werte der Variablen a und b.
In der Funktion Vertausche werden die Werte in a und b mit Hilfe der Variable temp vertauscht und ausgegeben. a hat jetzt den Wert 5 und b den Wert 8.
Danach springt man wieder zurück ins Hauptprogramm und gibt a und b erneut auf dem Bildschirm aus.

- ☞ **Testen Sie das Beispielprogramm!**
- ☞ **Achten Sie auf die Ausgabe von a und b im Hauptprogramm, nachdem die Funktion Vertausche ausgeführt wurde!**

```
#include <iostream>
using namespace std;
void Vertausche(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
    cout << "\n\nFunktion Vertausche aufgerufen."
         << "\nDie Werte nach der Vertauschung:"
         << "\na: " << a << "\tb: " << b
         << "\nzurueck zu main.\n\n";
}
int main( )
{
    int a = 8;
    int b = 5;
    cout << "a und b in main vor Vertauschung:"
         << "\na: " << a  << "\tb: " << b;
    Vertausche(a, b);
    cout << "a und b in main nach Vertauschung:"
         << "\na: " << a  << "\tb: " << b;
    cout << "\n\n";
    return 0;
}
```

Nach dem Ausführen der Funktion Vertausche werden die Werte von a und b nochmals im Hauptprogramm main ausgegeben. Trotz der Funktion Vertausche haben a und b aber wieder die alten Werte. Die Vertauschung hat auf das Hauptprogramm also keinen Einfluss!

Die **Erklärung** für diesen Sachverhalt liegt in der **Übergabe** der Parameter **als Wert**:
In C++ werden Variablen aus dem Hauptprogramm nicht wirklich an Funktionen übergeben. Der Compiler legt eine *Kopie* von a und b für die Funktion Vertausche an. Innerhalb der Funktion Vertausche werden dann die Kopien von a und b vertauscht und demnach auch getauscht auf dem Bildschirm ausgegeben. Kehrt das Programm nach main zurück, so werden die Kopien zerstört. Im Hauptprogramm werden dann wieder die *Originalwerte* verwendet. Die „Originale“ sind aber nie vertauscht worden (sondern nur die Kopien). Für a und b erscheinen die alten Werte auf dem Bildschirm.

Damit die Funktion Vertausche auch wirklich ihren Zweck erfüllt, müssen a und b **als Zeiger** übergeben werden.

- ☞ **Ändern Sie Ihr Programm in folgenden Punkten ab, so dass a und b als Zeiger übergeben werden!**

Der Funktionskopf von Vertausche muss geändert werden, damit Zeiger und nicht mehr Variablen übergeben werden können.
temp übernimmt jetzt zuerst den Wert, auf dessen Adresse pa zeigt (*pa!). Danach wird der Wert in *pa mit dem Wert von *pb überschrieben. *pb wird schließlich mit dem Wert von temp initialisiert. Die Werte sind vertauscht. Da mit Zeigern gearbeitet wird, muss auch bei der Ausgabe mit dem Indirektionsoperator (*) gearbeitet werden!
Im Hauptprogramm müssen beim Funktionsaufruf jetzt nicht mehr die Variablen a und b, sondern die Adressen der Variablen übergeben werden. Dies geschieht mit Hilfe des Adressoperators (&)

```
// Die Übergabe von Zeigern an Funktionen
...
void Vertausche(int* pa, int* pb)
{
    int temp =*pa;
    *pa = *pb;
    *pb = temp;
    cout << "\n\nFunktion Vertausche aufgerufen,"
         << "\nDie Werte nach der Vertauschung:"
         << "\na: " << *pa << "\tb: " << *pb
         << "\nzurueck zu main.\n\n";
}

int main( )
    ...
    Vertausche(&a, &b);
    ...
```

Im Programm werden jetzt die *wirklichen* Adressen von a und b übergeben und nicht nur die Kopien der Variablen. Dadurch wirkt sich die Änderung der Werte in der Funktion Vertausche auch im Hauptprogramm main( ) aus.

Das Beispielprogramm zeigt folgendes:

- Variablen werden in C++ nur als *Kopien* an Funktionen übergeben. Ändert man die Kopien in einer Funktion, so bleiben die „*Originalvariablen*“ im Hauptprogramm erhalten. Diese lassen sich in der Funktion nicht verändern. Um Abhilfe zu schaffen muss also die *Adresse* der Variablen mit einem *Zeiger* an die Funktion übergeben werden.
- Das Anlegen von Kopien der Variablen bedeutet *Speicher- und Rechenaufwand*. Ein Programm wird deshalb *langsamer* ausgeführt, wenn man mit Variablen arbeitet. Verwendet man hingegen Zeiger, so entfallen diese Kopien. Das macht sich vor allem bei umfangreicheren Programmen bemerkbar. Das Programm wird schneller.

Insgesamt kann die Effektivität eines Programms also durch den gezielten Einsatz von Zeigern wesentlich gesteigert werden.

2. Arbeiten im Freispeicher (Heap) des Computers

Der Programmierer hat es im allgemeinen mit fünf Speicherbereichen zu tun:

- Bereich der globalen Namen,
- Heap,
- Register,
- Codebereich,
- Stack (Stapelspeicher).

Lokale Variablen befinden sich zusammen mit den Funktionsparametern auf dem Stack. Der Code steht natürlich im Codebereich und globale Variablen im Bereich der globalen Namen. Die Register dienen internen Verwaltungsaufgaben (z.B. zur Programmsteuerung). Der restliche Speicher geht an den sogenannten *Heap* (Freispeicher).
Bis jetzt wurde in den C++ Programmen für Variablen und Zeiger nur der Stack verwendet. C++ bietet mit den Schlüsselwörtern *new* und *delete* eine einfache Möglichkeit den Heap in einem Programm zu nutzen.

2.1 Benutzen des Freispeichers (Heap) mit new

In C++ weist man Speicher im Heap mit dem Schlüsselwort ***new*** zu. Nach new steht der Typ des betreffenden Objekts, damit der Compiler die erforderliche Speichergröße kennt.

So reserviert z.B. `new int` 4 Byte und `new double` 8 Byte Speicher auf dem Heap.

Der *Rückgabewert* von new ist eine *Speicheradresse*. Damit ist klar, dass man zum „Merken“ für diese Speicheradresse einen Zeiger benötigt.

In der ersten Zeile des Programmausschnitts wird ein Zeiger auf ein int Objekt mit dem Namen `pZeiger` deklariert. Dann weist man dem Zeiger eine Speicheradresse auf dem Heap zu. `pZeiger` deutet jetzt auf 4 Byte Speicher im Heap. In der letzten Zeile wird der Wert `4` unter der Adresse im Heap gespeichert, auf die `pZeiger` zeigt.

```
...
int* pZeiger;
pZeiger = new int;
*pZeiger = 4;
...
```

Mit Hilfe des Indirektionsoperators (*) kann man im Programm auf den gespeicherten Wert zurückgreifen (z.B.: `cout << *pZeiger;`).

Die Deklaration des Zeigers und das Zuweisen einer Speicheradresse im Heap können auch in einem Schritt erfolgen:

```
int* pZeiger = new int;
```

Der Zeiger pZeiger deutet jetzt wieder auf eine Adresse im Heap, in der eine Ganzzahl (Typ int) mit 4 Byte gespeichert werden kann.

Problem:

Wie verhält sich das Programm, wenn der gesamte Freispeicher bereits mit Daten belegt ist?
In diesem Fall liefert *new* einen NULL – Zeiger zurück. `pZeiger` deutet dann auf NULL und nicht auf eine Speicheradresse im Heap. Es kann kein Speicher mehr im Heap freigegeben werden.
Um zu überprüfen, ob noch genug Speicherplatz vorhanden ist, muss beim Verwenden von new immer kontrolliert werden, dass kein NULL – Zeiger zurückgeliefert wird:

Nach der Deklaration des Zeigers und der Zuweisung der Adresse im Heap wird überprüft, ob genügend Speicher vorhanden ist: `if(pZeiger = = NULL)`. Wird ein NULL – Zeiger zurückgeliefert, so muss das Programm abgebrochen werden!

```
int* pZeiger = new int;
if( pZeiger = = NULL)
{
   cout << “Kein Speicher frei. Programmabbruch!“;
   ...
}
```

2.2 Freigeben des Speichers mit delete

Arbeitet man in einer Funktion mit lokalen Variablen auf dem Stack, so wird der Speicher automatisch wieder freigegeben, sobald die Funktion beendet ist. Man braucht sich um die Freigabe des Speichers nicht zu kümmern.
Anders sieht es beim Arbeiten im Heap aus. Hat man einem Zeiger mit ***new*** eine Speicheradresse im Heap zugewiesen, so wird der Speicher erst bei Programmende wieder freigegeben. Benötigt man den Speicher im Programm nicht mehr, so muss man ihn mit Hilfe des Schlüsselwortes ***delete*** wieder zerstören. Hier ist also der Programmierer verantwortlich. Vergisst man die Freigabe, so erzeugt man Speicherlücken, die im Programm nicht mehr verwendet werden können. Der Computer „verliert quasi an freiem Speicher“, das Programm wird langsamer oder kann eventuell gar nicht mehr ausgeführt werden.

Benötigt man den Speicher im Heap nicht mehr, so muss man ihn mit dem `delete pZeiger;` C++ Schlüsselwort **delete** wieder freigeben!

Als Faustregel gilt:

Mit **new** erzeugte Objekte müssen mit **delete** wieder gelöscht werden!

2.3 Ein Programmbeispiel unter Nutzung des Heaps

Im Hauptprogramm wird zuerst der Zeiger pZeiger deklariert und durch new mit einer Speicheradresse im Heap initialisiert.
Danach überprüft man in der if - Anweisung, ob genug Speicher im Heap vorhanden ist. Hat new einen NULL-Zeiger zurückgegeben, so wird das Hauptprogramm nach einer Fehlermeldung (“Speichermangel!“) abgebrochen (return 0;).
Ist genug Freispeicher vorhanden, so werden der Wert 4 unter der Speicheradresse im Heap abgelegt, auf die pZeiger deutet und sowohl die Adresse als auch der Wert ausgegeben.
Zum Schluss gibt man den Speicher mit delete wieder frei und beendet das Programm.

```
// Programm zur Nutzung des Heap
…
int main( )
{
    int* pZeiger = new int;
    if(pZeiger = = NULL)
    {
        cout << "Speichermangel!";
        return 0; // Programmabbruch
    }
    *pZeiger = 4;
    cout << "\nSpeicheradresse im Heap: " << pZeiger;
    cout << "\ngespeicherter Wert im Heap: " << *pZeiger;
    cout << "\n\n";
    delete pZeiger;
    return 0;
}
```

3. Das Arbeiten mit Strings (Zeichenketten)

3.1 Char – Arrays

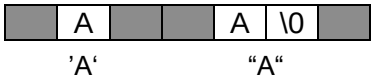
Charakter Strings werden in C++ dazu verwendet, um Text zu speichern oder zu verarbeiten. Um einen Charakter String innerhalb eines Programms zu definieren, wird ganz einfach ein Array des Typs char deklariert, das über genügend Elemente verfügt, um die gewünschte Anzahl von Zeichen zu speichern.

Das Array aString wird für 26 Zeichen deklariert und mit einem String “Jetzt....Text.“ initialisiert. Danach gibt man den Text aus.

```
...
char aString[26] = “Jetzt speichere ich Text.“;
cout << aString;
...
```

☞ **Schreiben Sie ein kurzes C++ Programm um die Speicherung von Strings zu testen!**
☞ **Wie reagiert das Programm, wenn Sie mehr als 26 Zeichen speichern wollen?**

Der obige Beispieltext enthält 25 Zeichen (inklusive Leerzeichen). Weshalb benötigt man aber 26 Felder im Array, um den Text zu speichern?
Die Erklärung liegt im Unterschied zwischen *Zeichen* ‘A‘ und *String* “A“:
Bei einem einzelnen Buchstaben (‘A‘) handelt es sich um eine Charakter – Konstante (char). Vom Compiler wird dafür nur 1 Byte Speicher reserviert. Die zweite Art von Zeichen (“A“) wird als String (Zeichenkette) bezeichnet. Sie kann unterschiedlich lang sein und benötigt daher eine NULL (\0) zur Markierung ihres Endes.



Es besteht ein Unterschied zwischen Zeichen und String.
‘A‘ und “A“ sind also in C++ nicht das Gleiche!

Betrachten wir noch einmal den Beispieltext "Jetzt speichere ich Text" :
Da der Text in Anführungszeichen eingegeben wird, handelt es sich um einen String. Der String benötigt ein NULL - Zeichen zum Markieren seines Endes. Durch die Anführungszeichen wird dieses NULL - Zeichen automatisch angehängt. Insgesamt benötigen wir also 26 Felder im String Array: aString[0], aString[1] ... aString[25].

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
J	e	t	z	t		s	p	e	i	c	h	e	r	e		i	c	h		T	e	x	t	.	\0

Mit cout wird das String - Array Buchstabe für Buchstabe ausgelesen, bis die Markierung für das Ende (\0) erreicht ist. Die Ausgabe wird gestoppt.

Über den Index des Arrays ist es möglich, einzelne Zeichen auszugeben:

Hier wird das Zeichen im Feld 10 des Arrays (c) ausgegeben.

```
cout << aString[10] << endl;
```

Das Array kann auch gezielt bis zu einem bestimmten Buchstaben ausgegeben werden:

Zunächst definiert man die Laufvariable i mit 0.
In einer do...while – Schleife wird jedes Feld von aString einzeln ausgegeben, solange das 'r' noch nicht erreicht ist.
Ist der Buchstabe 'r' erreicht, so wird die Schleife und damit auch die Ausgabe abgebrochen.

```
int i = 0;
do
{
   cout << aString[i];
   i++;
}while (aString[i] != 'r');
```

3.2 Zeiger auf Strings

Strings können in C++ auch ganz einfach gespeichert und bearbeitet werden, indem man einen *Zeiger* auf den String deklariert:

Mit der Zuweisung string = "Ein String Zeiger"; deutet der Zeiger auf die Adresse, in welcher der erste Buchstabe des Strings gespeichert ist.
Bei der Ausgabe über cout wird die Zeigeradresse automatisch immer um 1 Byte erhöht, bis das NULL – Zeichen (Ende des Strings!) gefunden wird. Die Ausgabe ist beendet.

```
const char* string;
string = "Ein String Zeiger";
cout << string;
```

Verwendet man im Programm Zeiger auf einen String, so muss die Länge des Strings nicht extra angegeben werden, wie dies der Fall bei String – Arrays ist.

Man kann mit Hilfe von String - Zeigern auch die Länge eines Textes bestimmen:

Im Hauptprogramm main wird der String – Zeiger string deklariert und mit einem String "Wie viele...es?" initialisiert.
Die Zählvariable i für die Anzahl der Zeichen wird auf 0 gesetzt und der String ausgegeben.
Weil bei der Ausgabe des Strings der Zeiger bei jedem Buchstaben um 1 Byte erhöht wird, kann man diese Inkrementierung auch in einer while – Schleife mitverfolgen. Jedesmal wenn der String - Zeiger auf den nächsten Buchstaben erhöht wird, wird i ebenfalls um 1 erhöht. Das Programm zählt also die Buchstaben.

```
#include <iostream>
using namespace std;
int main( )
{
   const char* string = "Wie viele Zeichen sind
   es?";
   int i = 0;
   cout << string;
   while(*(string++) != '\0')
        i++;
   cout << " = " << i << " Zeichen.";
   return 0;
}
```

In diesem Beispiel enthält der string die vorgegebenen Zeichen, 26 an der Zahl. Innerhalb der while – Schleife wird nun der Zeiger so lange erhöht, bis das Begrenzungszeichen gefunden wird, analog dazu wird i mit jedem Durchlauf erhöht. Als Ergebnis wird die Anzahl der Zeichen im String gezählt.
In der Standard – Headerdatei **<cstring>** von C/C++ sind eine ganze Reihe von nützlichen Funktionen zur Manipulation von Zeichenketten gespeichert. Dabei werden die Strings mit Hilfe von Zeigern übergeben. Als Beispiel soll hier kurz die Funktion **strcpy** angeführt werden:

char* strcpy(char* string_1, const char* string_2);

Funktion:	Kopiert den übergebenen string_2 in den Ziel – String string_1.	
Parameter:	string_1:	Ziel - String
	string_2:	zu kopierender String
Rückgabewert:	Ein Zeiger auf den neuen String.	

Die beiden Strings string_1 und string_2 werden als Zeiger übergeben.