

1. Samstag

Zeigerarithmetik

Betrachte folgenden Code mit, der ein dynamisch erstelltes Array mit einer Schleife bearbeitet wird:

```
int main()
{
    // platz für dreissig characters
    char* buchstaben = new char[30];

    for(int i = 0; i < 30; ++i)
    {
        // jeweiligen character auf
        // den Wert 'A' setzen
        buchstaben[i] = 'A';
    }

    return 0;
}
```

Es ist möglich diesen Code auch anders zu schreiben, wobei ich auch gleich den Datentypen wechsele:

```
int main()
{
    // platz für zehn doubles      double* wertStart = new
double[10];

    double* wertEnde = wertStart + 10;
    // der Zeiger wertEnde erhält
    // den numerischen Wert
    // wert + (30 * 8) !

    for(double* wertAkt = wertStart; // Initialisierung
        wertAkt != wertEnde;         // Laufbedingung
        ++wertAkt)                   // Reinitialisierung
    {
        *wertAkt = 3.5;
    }

    return 0;
}
```

Kleine Beschreibung:

Zuerst wird Platz für zehn double-Werte dynamisch auf dem Heap alloziert. Die zweite Codezeile zeigt ein erstes Beispiel von Zeigerarithmetik. Wir erzeugen den neuen Zeiger `werteEnde` und initialisieren ihn mit dem Wert von `werteStart` plus 10. **Vorsicht !** Da der Datentyp vom `werteEnde` ein Zeiger auf double ist, rechnet der Compiler nicht einfach 10 zum Wert des Zeigers `werteStart`. Angenommen der Zeiger `werteStart` hätte den Wert 1000, erhält `werteEnde` **nicht** den Wert 1010. Der Compiler sorgt dafür, dass falls ein double-Zeiger wie `werteStart` auf eine double zeigt, dass ein Zeiger `werteStart+1` auf den nächsten double im Speicher zeigt. Das heisst falls `werteStart = 1000` ist `werteStart+1 = 1008`, denn ein double braucht 8 Bytes. In der Schleife wird in der Reinitialisierung der double Zeiger mit ++ heraufgezählt. Er wird also jedesmal so verändert, dass er auf den nächsten double im Speicher zeigt. Der Zeiger `werteEnde` zeigt in unserem Beispiel auf einen double, der gerade hinter unserem allozierten Speicher liegt. Man darf also an die Speicherstelle, auf die `werteEnde` zeigt nicht schreiben. Als Laufbedingung ist der Zeiger aber geeignet, die Schleife wird nämlich rechtzeitig beendet. Überzeuge dich selbst mit dem Debugger davon !

Hier noch weitere Beispiele mit Zeigerarithmetik:

```
int main()
{
    // platz für dreissig characters
    double* werteStart = new double[10];

    double* werteEnde = werteStart + 10;
    // der Zeiger wertEnde erhält
    // den numerischen Wert
    // werte + (30 * 8) !

    // Zeiger auf zweites Element
    double* zweitWert = werteStart + 1;
    *zweitWert = 0.0;
    // andere Schreibweise :
    werteStart[1] = 5.5;

    // also
    bool esGeht = ( &werteStart[1] == zweitWert);
    // die Adresse des Elements mit index 1
    // ist gleich der Adresse des ersten
    // Elements + 1

    // Inhalt des letzten Elements auf 666
    // setzen
    *(werteStart+9) = 666;

    return 0;
}
```

Wir werden eine ähnliche Art der Schleifenbildung wie oben verwenden, wenn wir mit den **Containern** und den **Iteratoren** der Standard-C++ Library arbeiten.

typedef

Typendefinitionen kennen wir bereits, denn wer wir eine eigene Klasse, eine eigene Struktur oder eine eigene union definieren, definieren wir einen Datentypen.

Mit dem Schlüsselwort **typedef** können wir aus einem bestehenden Typen einen anderen definieren. Der zugrundeliegende Datentyp kann ein beliebiger eingebauter oder selbstdefinierter Typ sein. Der typedef wird häufig gebraucht um einem Datentypen einen anderen Namen zu geben, der entweder besser passt, oder auch hilft Tipparbeit zu sparen. Die Syntax ist wie folgt:

```
typedef DATENTYP NEUE_BEZEICHNUNG;
```

Hier zwei Beispiele:

```
class KlasseMitUnpraktischemNamen
{
public:
    KlasseMitUnpraktischemNamen()
    {
        m_zahl = 0;
    }
private:
    int m_zahl;
};

//      Name des bestehenden Typs      neue Bezeichnung
typedef KlasseMitUnpraktischemNamen ZahlKlasse;

typedef int Anzahl;

int main()
{
    // Variable vom Typ
    // Zahlklasse
    ZahlKlasse element1;

    // Variable vom Typ
    // Anzahl;
    Anzahl eineAnzahl = 0;

    return 0;
}
```

Verwendung der Container-Klassen aus der STL

Erinnern wir uns an die Klasse MyString, mit der wir die Verwendung von dynamischen Datenelementen kennegelernt haben.

Wir haben dort die Daten in einem Array gespeichert, das wir dynamisch, das heisst bei Bedarf vergrössert haben. Das Ändern jenes Arrays war aufwendig und mit vielen Kopieroperationen verbunden.

Das Problem, das wir eine sich ständig ändernde Anzahl von Elementen irgendwo speichern müssen kommt in C++ so häufig vor und wurde ebenso häufig falsch gelöst, dass man diese Probleme mit dem C++-Standard ein für allemal gelöst hat. Die C++ Standard Library (STL) enthält einige Klassen, um viele Elemente damit zu speichern. Man nennt diese Klassen Container. Unsere selbstprogrammierte CD-Liste ist ein Container für CD-Elemente.

Mittels der **template**-Technik von C++ sind die Container-Klassen der STL für beliebige Datentypen anwendbar. Die Schreibweise wehen wir weiter unten.

Man kennt in der Informatik einige bestimmte Container, die je nach Anwendung ausgewählt werden, da sie verschiedene Eigenschaften haben. Alle diese Container sind sehr ähnlich aufgebaut. Zum Beispiel erlauben die meisten davon mit der Funktion **push_back** neue Elemente hinzuzufügen.

Ein **vector** ist ähnlich wie ein Array. Das heisst wenn wir Elemente (von irgend einem Datentypen) in einem vector speichern, können wir mittels eines index-operators beliebig auf die Elemente zugreifen. Beim **vector** besteht aber beim anfügen von neuen Elementen die Möglichkeit, dass der ganze **vector** wie in unserer MyString-Klasse ständig umkopiert wird, was manchmal unerwünscht ist.

```
#include <iostream>
#include <vector>

using namespace std;

// typedef zur besseren Lesbarkeit
typedef vector<double> doubles;

int main()
{
    // ein doubles-Objekt erstellen
    doubles meineDoubles;

    // ein paar Werte hinzufügen
    meineDoubles.push_back(1.0);
    meineDoubles.push_back(1.1);
    meineDoubles.push_back(1.2);
    meineDoubles.push_back(1.3);
    meineDoubles.push_back(1.4);
    meineDoubles.push_back(1.5);

    int AnzahlWerte = meineDoubles.size();
```

```
// Werte ausgeben
for(int i = 0; i < AnzahlWerte; ++i)
{
    cout << meineDoubles[i] << endl;
}

return 0;
}
```

Den Nachteil von der "herumkopiererei" hat die **list** nicht. Das hinzufügen eines neuen Elementes ist weniger aufwendig, wie wir in unserer CD-Liste nachsehen können. Dafür erlaubt sie keinen direkten Zugriff auf ein Element mitten in der Liste. Wir müssen uns zum gewünschten Element durcharbeiten.

Dieser Zugriff auf die Elemente erfolgt bei der Liste mit einem sogenannten **Iterator**. Ein Iterator ist ähnlich wie ein Zeiger, den man mittels `operator++` inkrementieren kann, so dass er auf das nächste Element im Container (man spricht auch von Kollektion) zeigt. Das Beispiel jetzt also mit einer Liste und mit einem Iterator.

```
#include <iostream>
#include <list>

using namespace std;

// typedef zur besseren Lesbarkeit
typedef list<double> doubles;

int main()
{
    // ein doubles-Objekt erstellen
    doubles meineDoubles;

    // ein paar Werte hinzufügen
    meineDoubles.push_back(1.0);
    meineDoubles.push_back(1.1);
    meineDoubles.push_back(1.2);
    meineDoubles.push_back(1.3);
    meineDoubles.push_back(1.4);
    meineDoubles.push_back(1.5);           // durch die ganze
Kollektion
    // "durchwaten" mit Iteratoren

    // einen Iterator auf das erste Element
    // wie ein Zeiger auf das erste Element
    doubles::iterator akt = meineDoubles.begin();
    // und einer auf das Ende
    doubles::iterator end = meineDoubles.end();
}
```

```
    for(akt = meineDoubles.begin(); akt != end; ++akt)
    {
        double Wert = *akt;
        cout << Wert << endl;
    }

    return 0;
}
```

Übrigens funktioniert dieser Code auch mit **vector** als Container. Auch bei der Klasse **vector** sind die Iteratoren definiert ! Das heisst nur durch ändern des Wortes **list** auf **vector** funktioniert der Code oben genau gleich ! Der Programmierer muss für den jeweiligen Fall die Klasse auswählen, die besser passt.

Ein **stack** (übersetzt ein Stapel) ist ein wenig anders aufgebaut. Man legt etwas zuoberst auf den Stack und kann immer nur auf das oberste Element zugreifen. Dieses Verhalten wird in der Computertechnik häufig angewendet um kurzzeitig etwas (eine Variable oder Daten) kurzzeitig abzulegen. Ein UPN Taschenrechner funktioniert auch mit einem Stack. Man legt die Zahlen auf den Stack und wählt dann die Funktion, die sich die benötigten Operatoren vom Stack abholt und das Ergebnis wieder dorthin zurücklegt. Hier zuerst ein kleines Beispiel für die Anwendung des Stacks:

```
#include <stack>

using namespace std;

int main()
{
    // stack für doubles anlegen
    stack<double> doubleStack;

    // einen Wert auf den Stack
    // legen
    doubleStack.push(3.5);

    // Stackgrösse zur Kontrolle
    int stackSize = doubleStack.size();

    // oberstes Element holen
    // Das Element bleibt aber
    // auf dem Stack
    double d = doubleStack.top();

    // Stackgrösse zur Kontrolle
    stackSize = doubleStack.size();

    // oberstes Element entfernen
```

```
doubleStack.pop();

// Stackgrösse zur Kontrolle
stackSize = doubleStack.size();

return 0;
}
```

Das folgende Beispiel zeigt die Anwendung des Stacks für einen Rechner ! Versuche das Beispiel nachzuvollziehen !

```
#include <iostream>
#include <string>
#include <strstream>
#include <stack>

using namespace std;

int main()
{
    // stack für doubles anlegen
    stack doubleStack;

    // eingabe initialisieren
    string eingabe = "";

    // mit x Schleife verlassen
    while(eingabe != "x")
    {
        cin >> eingabe;

        // addieren ?
        if(eingabe == "+")
        {
            // addieren nur falls
            // mindestens 2 Elemente
            // auf dem Stack sind
            if(doubleStack.size() >= 2)
            {
                // oberstes Element holen
                double op1 = doubleStack.top();
                // oberstes Element entfernen
                doubleStack.pop();

                double op2 = doubleStack.top();
                doubleStack.pop();
```

```
        // rechnen
        double ergebnis = op1 + op2;
        // ergebnis zurück auf den
        // Stack
        doubleStack.push(ergebnis);
        // Ausgabe zu Kontrolle
        cout << ergebnis << endl;
    }
}
else
{
    // stringstream ist ein
    // stream, der einfach
    // im Speicher ist, sonst
    // aber ähnlich ist wie
    // cin oder cout
    stringstream stream;

    // die Eingabe in den Stream
    stream << eingabe;

    // und als double wieder
    // aus dem Stream lesen
    double wert;
    stream >> wert;

    // Wert auf den Stack
    doubleStack.push(wert);
}
}

return 0;
}
```