Templates und Containerklassen

Ziel, Inhalt

- Wir lernen wie man Funktionen oder Klassen einmal schreibt, so dass sie für verschiedene Datentypen verwendbar sind

Templates

Übung zur Einführung

Um warm zu werden schreiben wir die Funktion Swap für verschiedene Datentypen.

Die Klasse Person

Erzeuge als erstes wieder einmal eine einfache Klasse Person. Erzeuge eine Headerdatei und eine .cpp-Datei dafür. Die Klasse hat zwei Datenelemente für den Vor- und den Nachnamen. Ob du diese als Konstruktorargumente oder mittels set-Methoden setzen willst ist dir überlassen.

Die Swap Funktionen

In einem main.cpp includierst du die Header-Datei zur Person und schreibst dann eine swap Funktion dafür. Teste diese in der main-Funktion. Schreibe noch zwei weitere Swap-Funktionen mit denen du int Variablen und double Variablen tauschen kannst.

Templates als Schablonen

Betrachte die Funktionen genau. Sie unterscheiden sich eigentlich nur durch den Datentyp der Argumente und der temporären Variable in der Funktion drin.

Definition einer Template-Funktion

C++ bietet als einzige Programmiersprache die Möglichkeit für Funktionen, die für verschiedene Datentypen immer gleich aussehen Schablonen- oder Vorlagenfunktionen zu schreiben und den Compiler anzuweisen diese zu verwenden.
Um eine Vorlage zu definieren verwendet man das Schlüsselwort *template*.

```
template<class T> // erzeuge Schablone, bei der eine Klasse T
                  // bei Bedarf gewählt werden darf.
```

Darunter schreibst du die Funktion und schreibst für den Datentypen T hin.

Beispiel swap-Funktion

Hier als Beispiel die Swap-Funktion

```
template<class T>
void swap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

Ergänze nun dein main.cpp mit dieser *template*-Funktion. Danach kannst du die Swap-Funktionen, die du vorhin geschrieben hast auskommentieren.

Aufruf einer template-Funktion

Der Aufruf der swap-Funktion ist sehr einfach:

```
double d = 2.0;
double e = 3.0;

swap(d, e);
```

Der Compiler erkennt dass eine Funktion swap gebraucht wird, die als Argumente zwei *double* Argumente hat. Er findet nur die Vorlage, die template Funktion und instanziert diese für den Datentypen double.
Probier das auch für die Klasse Person, die du vorhin definiert hast.
Es ist möglich und manchmal nötig dem Compiler unter die Arme zu greifen und anzugeben für welchen Datentypen die Funktion zu erzeugen ist.

```
int k = 33;
int l = 44;

swap<int>(k, l); // swap Funktion für int erzeugen
                 // und verwenden
```

Template-Klassen

Ein ähnlicher Mechanismus existiert auch für Klassen. Es kann Sinn machen gewisse Klassen mit den immer gleichen Datenelementen zu verwenden, wobei es manchmal schön wäre einfach den Datentypen zu ersetzen.

Klasse IntArray

Falls ihr die Klasse MyString dabeihabt, holt ihr am besten diese hervor, denn wir werden eine ähnliche Klasse noch einmal schreiben, aber mit einem dynamisch erzeugten Array von int Werten. Die Klasse sollte etwas folgender Klasse entsprechen:

```
#ifndef INTARRAY_H
#define INTARRAY_H

class IntArray
{
public:
    IntArray(int size);
    ~IntArray();

    // Element mit index holen
    int& operator[](int index);
    // Element anfügen
    void add(int element);
    // Grösse abfragen
    int size() const;

private:
    int* m_array;
    int  m_size;
};

#endif
```

Mit dieser Klasse ist es möglich viele Messwerte oder derartiges zu verwalten.

Falls wir aber double's verwalten wollen, müssen wir den gleichen Code noch einmal schreiben. Die Klasse für double würde etwa so aussehen:

```
#ifndef DOUBLEARRAY_H
#define DOUBLEARRAY_H

class DoubleArray
{
public:
    DoubleArray(int size);
    ~DoubleArray();

    // Element mit index holen
    double& operator[](int index);
    // Element anfügen
    void add(double element);
    // Grösse abfragen
    int size() const;

private:
    double* m_array;
    int     m_size;
};

#endif
```

Überall wo vorhin der Datentyp int für die verwalteten Elemente stand, steht jetzt der Datentyp double. Beachte, dass die Grösse des Arrays immer noch mit einem int verwaltet wird.

Template Klasse für dynamische Arrays

Auch hier hilft uns die Fähigkeit von C++ Schablonen also Templates zu definieren. Bei einer template Klasse ist es am besten alles in die Header-Datei zu schreiben.

```
#ifndef DYNARRAY_H
#define DYNARRAY_H

////////////////////////////////////////

template<class T>
class DynArray
{
public:
    DynArray(int size = 0);
    ~DynArray();

    // Element mit index holen
    T& operator[](int index);
    // Element anfügen
    void add(const T& element);
    // Grösse abfragen
    int size() const;

private:
    T*  m_array;
    int m_size;
};

////////////////////////////////////////

template<class T>
DynArray<T>::DynArray(int size)
         :m_array(0)
{
    if(size > 0)
    {
        m_array = new T[size];
    }
    m_size = size;
}

////////////////////////////////////////

template<class T>
DynArray<T>::~DynArray()
{
    delete [] m_array;
}

////////////////////////////////////////

template<class T>
T& DynArray<T>::operator [](int index)
{
    // Hier müsste man unbedingt
    // eine Prüfung auf index < m_size
    // einbauen
    return m_array[index];
}
```

```
////////////////////////////////////

template<class T>
void DynArray<T>::add(const T& element)
{
    T* temp = new T[m_size+1];
    for(int i = 0; i < m_size; ++i)
    {
        temp[i] = m_array[i];
    }
    temp[m_size] = element;
    ++m_size;
    delete [] m_array;
    m_array = temp;
}

////////////////////////////////////

template<class T>
int DynArray<T>::size() const
{
    return m_size;
}

////////////////////////////////////

#endif
```

Hier eine main.cpp, die diese Array-Klasse für den Datentyp double erzeugen lässt:

```
#include "DynArray.h"
#include <iostream>

using namespace std;

int main()
{
    DynArray<double> doubleArray;

    doubleArray.add(5.6);
    doubleArray.add(3.5);

    int size = doubleArray.size();
    for(int i = 0; i < size; ++i)
    {
        cout << doubleArray[i] << endl;
    }

    return 0;
}
```

Containerklassen

Die Standard C++ Library macht sehr starken Gebrauch von Templates, sie heisst nicht umsonst die STL, Standard Template Library!
Ein Problem, das beim Programmieren häufig lösen muss, ist das Verwalten von vielen Daten. Als Beispiel könnte man Personen- und andere Datenbanken nennen (CD-Sammlung, etc.). Auch ständig wachsende Messwertreihen gehören dazu. Immer wenn viele Daten von einer bestimmten Art zusammenkommen und gesammelt werden müssen, werden Behälter wie unsere dynamischen Array verwendet.
Dabei werden immer wieder gleiche Strukturen verwendet, dazu gehören dynamische Arrays, verkettete Listen oder Queues (Schlangen) und Stacks. Diese Container sind immer gleich programmiert unabhängig von den Datenelementen, die damit verwaltet werden. In der STL wurden darum die häufigsten dieser Container als Template-Klassen definiert.

Der vector

Für uns am einfachsten zu verstehen ist die *vector*-Klasse. Im Grunde genommen ist es eine sauber programmierte *DynArray*-Klasse. Sie verwaltet also Daten in einem dynamisch allozierten Array.
Wir erzeugen am besten schnell so einen vector:

```
#include <vector>

using namespace std;

int main()
{
    vector<double> doubleVector;

    return 0;
}
```

Anwendung der vector-Klasse

Die wichtigten Methoden der vector-Klasse sind:

- push_back, zum anfügen eines Elementes
- der operator[] (index-operator), zum Holen eines Elementes
- die Methode size() zum Abfragen der Anzahl Elemente

Hier das Beispiel mit der Klasse vector anstelle der Klasse DynArray.

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

using namespace std;

int main()
{
    vector<double> doubleVector;

    doubleVector.push_back(5.6);
    doubleVector.push_back(3.5);

    int size = doubleVector.size();
    for(int i = 0; i < size; ++i)
    {
        cout << doubleVector[i] << endl;
    }

    return 0;
}
```