Templates und STL

C++ Übung am 09. Juni 2016

Templates

- Schablonen für konkrete Funktionen oder Klassen

- Datentyp noch offen -> Platzhalter-Datentyp

- Objekttyp wird erst bei Deklaration festgelegt

- Schreibweise:

  - *template*<class *X*> vor Kopf der Funktion/ Klasse

  - statt Datentyp wird z.B. *X* geschrieben

Funktions-Templates

```
template <class X> void my_swap(X &a, X &b) {
    X temp;
    temp = a;
    a = b;
    b = temp;
}
int main() {
    wstring wort1 = L"Erster";
    wstring wort2 = L"Zweiter";
    my_swap(wort1, wort2);
    wcout << L"Wstring 1 und 2: " << wort1 << wort2 << endl;
    int zahl1 = 1;
    int zahl2 = 2;
    my_swap(zahl1, zahl2);
    wcout << L"Zahl 1 und 2: " << zahl1 << zahl2 << endl;
}
```

Standard Template Library

- Sammlung von
  - Containern,
  - Iteratoren,
  - speziellen Klassen und
  - Algorithmen als Schablonen

Container

- Datenstruktur zur Zusammenfassung gleichartiger Objekte
- interne Struktur nach außen verborgen
- Zugriff auf Elemente über Iteratoren
- müssen eingebunden werden, z.B. **#include <vector>**
- der Typ der Elemente muss angegeben werden, z.B.

  vector<int> my Vector;

```
#include <vector>

int main() {
    vector<wstring> worte;
    worte.push_back(L"hallo");
}
```

Container

wichtigste Funktionen: (am Beispiel Vektor)

- **myVector.begin():**

  -> erzeugt Iterator, der auf das erste Element zeigt

- **myVector.end():**

  -> erzeugt Iterator, der *hinter* das letzte Element zeigt

- **myVector.push_back(object):**

  -> verlängert den Vektor hinten um das Element *object*

- **myVector.pop_back():**

  -> löscht das letzte Element des Vektors

- **myVector.empty():**

  -> leert den Vektor

Container: Überblick

- **Vector:** eindimensional, am Ende effizient
- **Deque:** eindimensional, am Ende und am Anfang effizient
- **List:** eindimensional, überall effizient

- **Set:** geordnete Menge von Objekten
- **Map:** geordnete Menge von Tupeln von Objekten

- **Multiset:** Menge von Objekten
- **Multimap:** Menge von Tupeln von Objekten

- **Unordered set:** ungeordnete Menge von Objekten
- **Unordered map:** ungeordnete Menge von Tupeln von Objekten (Hash)

Iteratoren

- Projektionsfunktion, verweist auf Objekt
- ermöglicht Zugriff auf Elemente in Containern
- iteriert ('wandert') über Elemente eines Containers
- Grundoptionen:
  - **Dereferenzieren:** * liefert Objekt als Ergebnis
  - **Inkrementieren:** ++ Iterator wandert ein Objekt weiter
  - **Vergleichen:** == bzw. != prüft, ob zwei Iteratoren auf das selbe Objekt verweisen

Iteratoren

- Output-Iteratoren: zum Schreiben
- Input-Iteratoren: zum Lesen
- Forward-Iteratoren: Lesen und Schreiben vorwärts
- Bidirektionale-Iteratoren: beide Richtungen
- **Random-Access-Iteratoren:** können alles
  - direkten Zugriff auf Elemente mit dem Indexoperator []
  - direkte Verschiebung durch iter += n, iter -= n;
  - arithmetisches Rechnen wie z.B. iter + n, iter - n;
  - Abstände wie z.B. iter1 - iter2
  - relationale Operationen: <, >, <=, >=
- **Reverse-Iterator:** läuft rückwärts

Iteratoren

Random-Access-Iteratoren:

```
container<type>::iterator name;
```

```
vector<int>::iterator iter;
for (iter = myVector.begin(); iter < myVector.end(); ++iter) {
    cout << *iter << endl;
}
```

Reverse-Iterator:

```
container<type>::reverse_iterator name;
```

```
vector<int>::reverse_iterator rit;
for (rit = myVector.rbegin(); rit < myVector.rend(); ++rit) {
    cout << *rit << endl;
}
```

Funktionen überladen

- gleichnamige Funktion für mehrere Zwecke implementieren
- verschiedene Argumente

```
int addiere (int zahl1, int zahl2) {
    // Funktionsrumpf
}
int addiere (float zahl1, float zahl2) {
    // Funktionsrumpf
}
int addiere (int zahl1, int zahl2, int zahl3) {
    // Funktionsrumpf
}
```

Operatoren überladen

- Operator für eine Klasse neu definieren
- bisherige Definition nicht verändert

```
void ClassName::operator++() { // Präoperator
    // neue Funktion
}
```

Aufruf: ++*object*;

```
void ClassName::operator++(int notused) { // Postoperator
    // neue Funktion
}
```

Aufruf: *object*++;

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

int main() {
    vector<wstring> myVector;
    wstring zeile;
    wstring word;
    wcout << "Bitte geben Sie eine Zeile ein: " << endl;
    getline(wcin, zeile);
    wstringstream myStream;
    myStream << zeile;

    while (myStream >> word) {
        myVector.push_back(word);
    }
    vector<wstring>::iterator myIter;

    for (myIter = myVector.begin();
         myIter != myVector.end(); ++myIter) {
        wcout << "Aktuelles Element: " << *myIter << endl;
    }
}
```

Klausurübung

Schreibe ein C++ Programm, das eine deutsche Wortliste aus der Datei wortliste.txt (utf-8Datei, pro Zeile ein Wort) liest und die einzelnen Wörter testet, ob sie ein Palindrom sind. Schreibe eine Funktion is_pali, die wahr zurückliefert, wenn das Argument ein Palindrom ist. Gib jedes eingelesene Wort aus und falls das Wort ein Palindrom ist, dann gib auch die Meldung „Es ist ein Palindrom“ aus.