Vector und List

Ziel, Inhalt

- Wir lernen die Klassen vector und list aus der Standard-C++ Library kennen und anwenden.
- In einer Übung wenden wir diese Klassen an um einen Medienshop (CD's und Bücher) zu implementieren

Vector und List

Überblick

Diese zwei Klassen sind in der STL (Standard Template Library) definiert. Sie sind als Template-Klassen implementiert. Das Buch „C++ Lernen und professionell anwenden“ bietet in Kapitel 32 „Templates“ eine Übersicht über Templates und geht im folgenden Kapitel „Container“ genauer auf die Themen vom heutigen Abend ein. Im Unterricht haben wir diese Klassen in der letzten Lektion vom 3. Semester (http://www.devmentor.ch/teaching/additional/01I/Semester3/Abend9/Abend9.pdf) kurz gestreift.
Diese Klassen bieten die Möglichkeit Objekte in beliebiger Anzahl und von beliebigen Datentypen zu speichern, es sind also Behälter für beliebige Elemente, daher der Name „Container“-Klassen.

Vector und List

Anwendung

Die Anwendung ist denkbar einfach. Man erzeugt ein Objekt einer Container-Klasse (vector oder list) auf folgende Weise:

```
#include <vector>
#include <list>

int main()
{
    // Einen vector für den Datentypen
    // double erzeugen
    std::vector<double> doubleVector;
    // Eine liste für den Datentypen
    // long erzeugen
    std::list<long> longList;
    return 0;
}
```

In den eckigen Klammern steht der Datentyp, für den man eine Klasse erzeugen will. Häufig ist es auch nützlich einen Datentypen zu definieren, man spart sich ein wenig Schreibarbeit und weiter unten werden wir einen anderen nützlichen Effekt kennenlernen.

```
// Datentypen definieren
typedef std::vector<double> DoubleVector;
typedef std::list<long> LongList;
int main()
{
    DoubleVector doubleVector;
    LongList longList;

    return 0;
}
```

Einfügen und Entfernen von Elementen

Die beiden Container vector und list sind sich in der Anwendung sehr ähnlich. Das Einfügen eines Elementes funktioniert bei beiden Containern gleich.

```
// Datentypen definieren
typedef std::vector<double> DoubleVector;
typedef std::list<long> LongList;

int main()
{
    DoubleVector doubleVector;
    LongList longList;

    // Elemente hinten anfügen
    doubleVector.push_back(4.5);
    longList.push_back(5675);

    return 0;
}
```

Das letzte Element lässt sich mit pop_back() löschen.

```
    // letzte Elemente löschen
    doubleVector.pop_back();
    longList.pop_back();
```

Die Anzahl Elemente in einem Container

Die Anzahl Elemente in einem Container lässt sich mit der Methode size() abfragen. Bei diesem Beispiel wende ich auch gleich das assert noch einmal an (in der speziellen Version von Microsoft. Wer keinen Microsoft Compiler verwendet kann auch #include <cassert> und die Funktion assert(…) verwenden).

```
#include <vector>
#include <list>
#include <crtdbg.h>

// Datentypen definieren
typedef std::vector<double> DoubleVector;
typedef std::list<long> LongList;

int main()
{
    DoubleVector doubleVector;
    LongList longList;

    // Elemente hinten anfügen
    doubleVector.push_back(4.5);
    longList.push_back(5675);

    // size_type entspricht einem long
    // dieses size_type wurde per typedef
    // in der jeweiligen Klasse definiert
    // darum der Zugriff mit ::
    DoubleVector::size_type vectorLength = doubleVector.size();
    LongList::size_type listLength = longList.size();

    // da je ein Element eingefügt wurde
    // müssen beide container die
    // Länge 1 haben.
    _ASSERT(1 == vectorLength);
    _ASSERT(1 == listLength);

    // letzte Elemente löschen
    doubleVector.pop_back();
    longList.pop_back();

    return 0;
}
```

Das Iterieren

Unter Iterieren versteht man den Zugriff auf mehrere Elemente in einem Container, wie es in einer Schleife häufig verwendet wird.
Beim vector, der sehr ähnlich zu einem dynamischen Array ist, gibt es den Index Operator (operator – []).

```
// Funktion aufrufen, die einige
// Elemente einfügt
fillDoubleVector(doubleVector);
// Anzahl Element finden
vectorLength = doubleVector.size();
// Schleife
for(unsigned long i = 0; i < vectorLength; ++i)
{
    std::cout << doubleVector[i] << std::endl;
}
```

Der Index-Operator ist bei der Klasse std::list nicht überschrieben.
Es gibt einen anderen Weg, der bei den meisten Container-Klassen gleich funktioniert. Hier treten die Iteratoren ins Rampenlicht.
Ein Iterator entspricht einem Zeiger auf ein Element in der Kollektion.
Dieser Zeiger hat den ++ und den -- Operator überschrieben. Zur Erinnerung:

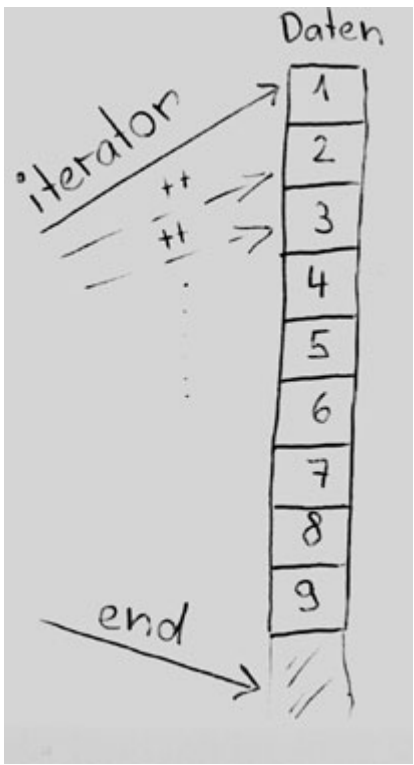
```
#include <iostream>

int main()
{
    long daten[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // dieser Zeiger zeigt nun
    // auf das erste Element
    long* iterator = daten;

    // länge berechnen
    long size = sizeof(daten) / sizeof(daten[0]);
    // Zeiger end hinter das letzte Element
    // zeigen lassen
    long* end = iterator + size;
    while(iterator != end)
    {
        // Wert holen, indem
        // man den Zeiger dereferenziert
        long wert = *iterator;
        std::cout << wert << std::endl;
        ++iterator;
    }

    return 0;
}
```



Genau das gleiche geht, wenn wir eine Container-Klasse definieren. Dabei gibt es spezielle Iterator-Klassen, die man hierfür verwenden kann. Bitte beachtet dabei auch, dass ein iterator eine Referenz auf das Element in der

Kollektion zurückgibt. Mit den beiden Methoden begin() und end() können wir direkt einen Iterator (Zeiger) auf das erste Element erhalten und einen, der hinter das letzte Element zeigt. Beachte auch, dass hier der typedef von vorhin ein wenig Tipparbeit ersparen hilft. Hier das Beispiel:

```
// hier eine while-Schleife zum Iterieren
DoubleVector::iterator it = doubleVector.begin();
DoubleVector::iterator end = doubleVector.end();
while(it != end)
{
    double d = *it;
    std::cout << d << std::endl;
    ++it;
}


// Hier das gleich als for-Schleife
for(LongList::iterator it2 = longList.begin();
    it2 != longList.end(); ++it2)
{
    // hier holen wir uns nur
    // eine Referenz
    long& l = *it2;
    std::cout << l << std::endl;
}
```

Bei mir hat sich folgende Schreibweise eingebürgert:

```
// so wie ich das immer mache
DoubleVector::iterator it = doubleVector.begin();
DoubleVector::iterator end = doubleVector.end();

for( ; it != end; ++it)
{
    double& d = *it;
    std::cout << d << std::endl;
}
```

Löschen von Elementen

Mit einem Iterator auf ein bestimmtes Element ist es auch möglich dieses zu löschen. Die Methode ist erase(), mit einem Iterator als Argument.

```
LongList::iterator elementIt = longList.begin();
// auf zweites Element vorrücken
++elementIt;
// Dieses Element löschen
longList.erase(elementIt);
```

Besondere Eigenschaft von vector

Die Klasse vector bietet als Besonderheit den Index Operator, den wir bereits gesehen haben. Dadurch ist es möglich direkt auf ein bestimmtes Element zuzugreifen.

```
unsigned long index = 4;
// stelle sicher, dass mindestens
// fünf Elemente im vector sind
_ASSERT(index < doubleVector.size());
double& el4 = doubleVector[index];
```

Besondere Eigenschaft der Liste

Die Liste ist intern als doppelt verkettete Liste implementiert. Es gibt hier keinen direkten Zugriff auf ein bestimmtes Element. Der Zugriff geschieht immer über einen Iterator. Die Liste bietet als Besonderheit eine Methode sort(), mit der sich die Liste der Grösse nach sortieren lässt. Dabei ist es wichtig, dass es für den Datentypen, der sich in der Liste befindet einen operator < (kleiner-Operator) gibt. Für die eingebauten Datentypen ist das kein Problem, bei eigenen Datentypen muss man das aber beachten.

```
longList.sort();
```

Diese Beispiele findest du alle in folgender main Datei:
http://www.devmentor.ch/teaching/additional/01I/Semester5/Abend2/main.cpp

Eigene Datentypen in Containern

Es ist mit den Containern möglich beliebige Datentypen zu verwalten, also auch Selbstdefinierte.

```
#include <vector>
#include <string>

class Person
{
public:
    Person(const std::string& vorname,
           const std::string& nachname)
           :_vorname(vorname),
            _nachname(nachname)
    {
    }

    void aendereNachname(const std::string& nachname)
    {
        _nachname = nachname;
    }

private:
    std::string _vorname;
    std::string _nachname;
};

// neuen Datentypen definieren
typedef std::vector<Person> Personen;

int main()
{
    Person einePerson("Eine", "Person");
    Person zweitePerson("Zweite", "Person");

    Personen allePersonen;

    allePersonen.push_back(einePerson);
    allePersonen.push_back(zweitePerson);

    return 0;
}
```

Was man diesem Beispiel kaum ansieht ist die Anzahl an Kopien, die von den Person-Objekten erzeugt werden. Es ist aber so, dass wenn man einen vector von einem Datentypen erzeugt, dieser Datentyp kopierbar sein muss. Zur Demonstration kann man hier bei der Klasse Person einen Kopierkonstruktor privat definieren:

```
class Person
{
public:
    Person(const std::string& vorname,
           const std::string& nachname)
           :_vorname(vorname),
            _nachname(nachname)
    {
    }

    void aendereNachname(const std::string& nachname)
    {
        _nachname = nachname;
    }

private:
    // zur Demo machen wir den
    // Kopierkonstruktor privat!
    Person(const Person& c)
        :_vorname(c._vorname),
         _nachname(c._nachname)
    {
    }

private:
    std::string _vorname;
    std::string _nachname;
};
```

Die main-Funktion von vorhin wird jetzt nicht mehr kompilieren, da der Kopierkonstruktor nicht aufgerufen werden kann.

Wenn wir keinen eigenen Kopierkonstruktor definieren, erzeugt der Compiler wenn möglich einen. Dieser automatisch erzeugte Kopierkonstruktor ruft den Kopierkonstruktor der Datenelemente der Klasse auf. Einen eigenen Kopierkonstruktor braucht es, wenn eine Klasse Zeiger auf dynamisch allozierten Speicher hat (oder ähnliches).

Manchmal gibt es also Gründe, dass man keine Kopien erzeugen will. Es ist dann nötig Zeiger im Container zu verwalten. Dadurch wird es auch möglich den Polymorphismus von C++ auszunutzen (siehe auch Übung). Es wird dabei nötig den Speicher selber zu verwalten (new und delete).

```
// neuen Datentypen definieren
// hier nur Zeiger verwalten
typedef std::vector<Person*> Personen;

int main()
{
    Person* einePerson = new Person("Eine", "Person");
    Person* zweitePerson = new Person("Zweite", "Person");

    Personen allePersonen;

    allePersonen.push_back(einePerson);
    allePersonen.push_back(zweitePerson);

    // Schleife zum Löschen aller
    // Elemente.
    Personen::iterator it = allePersonen.begin();
    Personen::iterator end = allePersonen.end();

    for( ; it != end; ++it)
    {
        Person* person = *it;
        delete person;
    }

    return 0;
}
```

Sortieren mit Zeigern

Bei der Liste gibt es die Möglichkeit zu Sortieren. Wenn wir aber Zeiger in der Liste haben, werden nur die Zeigerwerte zum Sortieren verwendet, was bei einer Namensliste falsch ist.
Hier das Beispiel ohne erfolgreiche Sortierung:

```
class Person
{
public:
    Person(const std::string& vorname,
           const std::string& nachname)
           :_vorname(vorname),
            _nachname(nachname)
    {
    }

    void ausgeben(std::ostream& out) const
    {
        out << _vorname << " ";
        out << _nachname << std::endl;
    }

private:
    std::string _vorname;
    std::string _nachname;
};

// neuen Datentypen definieren
// hier nur Zeiger verwalten
typedef std::list<Person*> Personen;

int main()
{
    Person* einePerson = new Person("Hans", "Zwahlen");
    Person* zweitePerson = new Person("Markus", "Burri");

    Personen allePersonen;

    allePersonen.push_back(einePerson);
    allePersonen.push_back(zweitePerson);

    allePersonen.sort();
    // Schleife zum Ausgeben aller
    // Elemente.
    Personen::iterator it = allePersonen.begin();
    Personen::iterator end = allePersonen.end();
    for( ; it != end; ++it)
    {
        Person* person = *it;
        person->ausgeben(std::cout);
    }
    // Schleife zum Löschen aller
    // Elemente.
    it = allePersonen.begin();

    for( ; it != end; ++it)
    {
        Person* person = *it;
        delete person;
    }
    return 0;
}
```

Das Ergebnis überrascht kaum, denn wie gesagt werden nur die Elemente im Container sortiert und das sind in unserem Fall Zeiger.
Es ist aber möglich der sort-Methode eine Funktion als Argument mitzugeben (es wird dabei der Funktionszeiger verwendet), die mit je zwei Elementen aus dem Container (hier Zeiger auf Person) aufgerufen wird.

```
class Person
{
public:
    Person(const std::string& vorname,
           const std::string& nachname)
           :_vorname(vorname),
            _nachname(nachname)
    {
    }

    static bool vergleiche(Person* p1, Person* p2)
    {
        if(p1->_nachname == p2->_nachname)
        {
            return (p1->_vorname < p2->_vorname);
        }
        else
        {
            return p1->_nachname < p2->_nachname;
        }
    }

private:
    std::string _vorname;
    std::string _nachname;
};

// neuen Datentypen definieren
// hier nur Zeiger verwalten
typedef std::list<Person*> Personen;

int main()
{
    Person* einePerson = new Person("Hans", "Zwahlen");
    Person* zweitePerson = new Person("Markus", "Burri");

    Personen allePersonen;

    allePersonen.push_back(einePerson);
    allePersonen.push_back(zweitePerson);

    // Sortiere mit Hilfe der
    // statischen vergleiche Methode
    allePersonen.sort(Person::vergleiche);
```

Dieses Beispiel findest du zum Download hier:
http://www.devmentor.ch/teaching/additional/01I/Semester5/Abend2/main2.cpp

Übung

Anwendung von Liste und Vector

- Schreibe eine Klasse Auto mit zwei Datenelementen (Marke und Modell).
- Ergänze diese Klasse mit einer Methode „ausgeben“, damit die Daten in einen ostream ausgegeben werden.
- In einer main-Funktion erzeugst du nun einige Auto-Objekte und verwaltest diese in einem vector. Verwende zuerst keine Zeiger in der Kollektion.
- Rufe für alle Objekte im Container die Methode „ausgeben“ auf.
- Durch eine ganz kleine Änderung sollte es möglich sein eine list-Klasse anstelle der vector-Klasse zu verwenden.
- Rufe sort auf. Dafür musst du für die Klasse Auto einen operator < definieren!

Kleiner Medienshop

Die nächste Anwendung ist ein kleiner Medienshop. In diesem Medienshop verkaufen wir Bücher und CDs, also Medien. Ein solches Medium hat einen Preis und einen Titel. Ein Buch hat zusätzlich eine ISBN-Nummer (10-stellige Nummer) und einen Autor, eine CD hat einen Interpreten.
Um eine Übersicht über das Inventar zu erhalten möchte der Betreiber des Shops eine Liste aller Medien ausgeben können. Sehr schön wäre natürlich eine Ausgabe als HTML. Der Betreiber ist mit einem einfachen Programm zufrieden, in dem man neue CDs oder Bücher eintragen kann und die Liste ausgeben kann.

Finde zuerst die Klassen, die du brauchen wirst.
Gibt es eine Basisklasse?
Welche Klassen leiten wir davon ab?
Welchen Container verwendest du? Was wirst du im Container speichern?
Einige Substantive ergeben nicht unbedingt Klassen, möglicherweise sind es einfache (eingebauter Datentyp oder Datentyp aus der STL) Datenelemente in einer Klasse.