

## Das Singleton-Pattern erweitert

### Ziel, Inhalt

- Während des Unterrichts haben wir das Singleton-Pattern erweitert um mehrere Objekte der Klasse *SerialPort* zu verwalten. Hier sehen wir, wie wir darauf gekommen sind.

Das Singleton-Pattern erweitert	1
Ziel, Inhalt	1
Das Singleton-Pattern erweitert	2
Einführung	2
Das Singleton-Pattern, ein Creational-Pattern	2
Die seriellen Ports, mehrere Singletons	2
std::map	2
Die Anwendung der std::map	3
Beispiel Implementation	5

## Das Singleton-Pattern erweitert

### Einführung

#### Das Singleton-Pattern, ein Creational-Pattern

Das Singleton-Pattern gehört zu den Patterns, die verwendet werden um Objekte zu erzeugen, oder um auf Objekte zuzugreifen. Der englische Ausdruck dafür ist Creational-Pattern. Normalerweise erzeugen wir unserer Objekte normal als lokale, globale oder dynamisch erzeugte Objekte. Manchmal ist aber das richtige Design oder die richtige Anwendung auf die übliche Art nicht möglich und unsere Objekte müssen je nach Anwendung speziell erzeugt werden. Hier kann das eine oder andere Creational Pattern hilfreich sein.

#### Die seriellen Ports, mehrere Singletons

Das Singleton-Pattern (siehe auch „[Die Klasse SerialPort, Teil 2](#)“) dient dazu ein einzelnes Objekt zu erzeugen. Ein PC kann aber mehrere serielle Ports besitzen. Jeder Port ist über seine Portnummer identifizierbar. Wenn wir diese Anforderung auf die *Singleton*-Klasse anwenden sieht das Interface zur Design-Zeit etwa so aus:

```
class Singleton
{
public:
    void doIt() const;
    ~Singleton();

    static Singleton& getSingleton(unsigned char id);
};
```

Wir geben der statischen Methode *getSingleton* zusätzlich an welches Objekt wir wollen, hier über eine *id* identifiziert. Wir werden also mehrere Objekte verwalten müssen, also muss irgend ein Container verwendet werden. Die Objekte, die wir im Container halten wollen, sollten über einen *id* zugreifbar sein, was eigentlich mit einem *std::vector* möglich wäre, wobei die *id* der Index wäre. Es kann jedoch sein, dass ein Anwender nur das Objekt mit der *id* 23 und der *id* 55 braucht. Das heisst unser *vector* müsste mindestens 55 Elemente enthalten. Es gibt aber einen besseren Container dafür, die Klasse

#### *std::map*

Die Klasse *std::map* ist ein sogenannter assoziativer Container, ein Container bei dem die enthaltenen Elemente über eine „Assoziation“ gefunden werden. Wir können bei der *map* auch sagen, dass zu jedem

Element auch ein Schlüssel gehört, mit dem das Objekt wieder im Container gefunden werden kann.

### Die Anwendung der `std::map`

Um einen Container zu erzeugen müssen wir nicht nur wie sonst angeben, was wir in den Container packen, sondern auch was der Schlüssel sein soll.

```
#include <map>

std::map<KeyType, ElementType> collection;
```

Hier ein kleines Beispiel mit einer Klasse *Person*, bei denen wir den Vornamen als Schlüssel für die Elemente der Klasse *Person* verwenden.

```
#include <string>
#include <map>

class Person
{
public:
    Person(const std::string& vorname,
           const std::string& nachname);

    Person();

private:
    std::string m_vorname;
    std::string m_nachname;
};

// wir verwenden einen string um auf ein
// Person-Objekt zuzugreifen, also erzeugen
// wir einen Datentypen mit einer map mit
// einem string als Schlüssel und einem
// Person Objekt als Element

typedef std::map<std::string, Person> Persons;
```

Die `map` verwaltet intern die Eintragungen als Pärchen (`std::pair<KeyType, ElementType>`). Wir könnten also so ein Pärchen bilden und dann die Methode *insert* und das Pärchen in die `map` einfügen. Noch einfacher ist die Anwendung des operator[], also des Index-Operators. Dabei ist können wir den Schlüsseltypen als Index verwenden! Betrachte folgendes Beispiel:

```
#include "Person.h"

int main()
{
    Persons thePersons;

    // hier erzeugen wir ein std::pair-Objekt
    // mit dem Namen eintrag1. Dieses Objekt
    // nimmt als Konstruktor-Argument einen KeyType also
    // ein string und ein ElementType also ein Person-Objekt
    // Beachte, dass zum Erstellen des Person-Objektes
    // die Konstruktor-Argumente anzugeben sind.
    std::pair<std::string, Person>
        eintrag1("Markus", Person("Markus", "Burri"));

    thePersons.insert(eintrag1);

    // Das pair-Objekt wird automatisch als typedef definiert
    // wir können also das gleiche wie oben auf
    // diese Art erreichen
    Persons::value_type
        eintrag2("Fred", Person("Fred", "Feuerstein"));

    thePersons.insert(eintrag2);

    // Am einfachsten ist das einfügen mit dem index-Operator
    thePersons["Bart"] = Person("Bart", "Simpson");

    return 0;
}
```

Um ein Objekt „nachzusehen“ können wir die *find*-Methode verwenden, oder aber auch den Index-Operator. Bei der Möglichkeit mit dem Index-Operator wird aber ein Objekt erzeugt, falls noch keines vorhanden war!

```
// Zugriff über find
Persons::iterator it = thePersons.find("Markus");
if(it != thePersons.end())
{
    // gefunden !
    Person& markusPerson = it->second;
}

// oder mit Index-Operator
Person& bartPerson = thePersons["Bart"];
```

Der iterator, den wir zurück erhalten zeigt auf ein Objekt vom Datentyp *Persons::value\_type* was auch dem Datentypen *std::pair<std::string, Person>* entspricht. Das pair enthält ein als *first* den Schlüssel und als *second* das Element. Der Zugriff auf den Schlüssel und das Element ist also wie folgt:

```
std::string key = it->first;
Person& element = it->second;
```

## Beispiel Implementation

Hier findest du die Anwendung um das spezielle „Singleton“-Verhalten zu erreichen, wie es für die seriellen Ports nötig ist.

```
#ifndef SINGLETON_H
#define SINGLETON_H

#include <map>

// Vorwärtsdeklaration damit wir die
// map definieren können
class Singleton;

// typedef für eine map mit einem unsigned char
// als Schlüssel und einem Singleton-Objekt
// als Element
typedef std::map<unsigned char, Singleton> Singletons;

class Singleton
{
public:
    void doIt() const;
    ~Singleton();

    // angepasste get-Methode
    static Singleton& getSingleton(unsigned char port);

private:
    // Die map, die wir erzeugt haben
    // muss den Standardkonstruktor aufrufen
    // können, der aber privat ist.
    // Wir machen darum die map zum Freund!
    friend Singletons;

    Singleton();
    void open(unsigned char port);
};

#endif
```

Hier die .cpp Datei:

```
// Der Compiler erzeugt zum Teil zu lange
// interne Bezeichner, was zu einer Warnung
// führt. Diese schalten wir mit pragma ab.
#pragma warning (disable : 4786)

#include "Singleton.h"
#include <iostream>

Singleton::Singleton()
{
}

Singleton& Singleton::getSingleton(unsigned char port)
{
    // Diese Variable (also map) wird nur einmal
    // erzeugt und bleibt dann immer vorhanden
    static Singletons dieSingletons;

    // Wir sehen ob das Objekt bereits existiert
    Singletons::iterator it = dieSingletons.find(port);

    if(it == dieSingletons.end())
    {
        // Das Objekt existiert noch nicht, wir
        // erzeugen es jetzt also mit dem Index-Operator
        Singleton& neuesObjekt = dieSingletons[port];
        // und rufen eine Methode davon auf
        neuesObjekt.open(port);
    }

    return dieSingletons[port];
}

void Singleton::doIt() const
{
}

void Singleton::open(unsigned char port)
{
}

Singleton::~Singleton()
{
}
```

Du findest die Dateien auch zum Download auf:

<http://www.devmentor.ch/teaching/additional/001/Semester5/Abend7/Singleton2.zip>