Klasse Zahl mit Ausnahmebehandlung

Ziel, Inhalt

- Am Beispiel der Klasse *Zahl* wenden wir die Ausnahmebehandlung an
- Wir kennen ab heute die Klasse std::bitset mit der wir bit-Arrays verwalten können
- Wir sind ab heute in der Lage die C++ - Ausnahmebehandlung zu verstehen und anzuwenden

Die Klasse *Zahl* und Ausnahmebehandlung

Die Klasse Zahl

Diese Klasse dient zur Kapselung einer Zahl, die in verschiedenen Formaten dargestellt werden kann. Grundsätzlich geht es darum eine Zahl in Hexadezimaler, Dezimaler, Oktaler und Binärer Form in eine beliebige andere Form zu bringen. Sie kann also verwendet werden um in einem Taschenrechner-Programm die Funktionalität nachzubilden, die der Taschenrechner von Windows auch anbietet.

Öffentliche Schnittstelle von Zahl

Überlegen wir uns also, wie eine solche Klasse aussehen soll. Im allgemeinen wählt der Benutzer durch einen Radio-Button die Basis, die er verwenden will. Danach gibt er in einem Textfeld die Zahl ein. Es sollte also möglich sein aufgrund dieser Angabe ein Objekt unserer Klasse *Zahl* zu erstellen. Falls wir bereits einen Wert haben, sei es als *double* oder als *long* ist es auch nützlich ein solches Objekt erstellen zu können. Zusätzlich definieren wir gleich eine Aufzählung mit der wir die Basis definieren können.

```
#ifndef ZAHL_H
#define ZAHL_H

#include <string>

enum Basis
{
    Binaer = 2,
    Octal = 8,
    Decimal = 10,
    Hexadecimal = 16
};

class Zahl
{
    public:
        // Construction
        Zahl(const std::string& text, Basis basis);
        Zahl(double d);
        Zahl(long l);
};
```

Je nach Berechnung, die wir mit der Zahl dann anstellen wollen, wollen wir die Zahl dann als *long* oder *double* wieder „extrahieren“. Ich wähle hierfür Methoden mit dem Namen *ToLong* oder *ToDouble*. Um die Zahl als String in mit einer wählbaren Basis darzustellen gibt es die Methode *ToString*, die als Argument einen Wert aus der Basis-Aufzählung verwendet.
Was auch ganz nützlich wäre um folgenden Code zu schreiben, wären auch die Zuweisungsoperatoren für *double* und *long*.

```
int main()
{
    Zahl eineZahl(0L); // Konstruktor mit long
    eineZahl = 23.5;   // Zuweisung eines doubles
    eineZahl = 15L;    // Zuweisung eines longs, beachte das L

    return 0;
}
```

Alle diese Wünsche führen zu folgender Schnittstelle:

```
class Zahl
{
    public:
        // Construction
        Zahl(const std::string& text, Basis basis);
        Zahl(double d);
        Zahl(long l);
        // Copy Construction
        Zahl(const Zahl& z);

        // Extraction
        double ToDouble() const;
        long   ToLong() const;
        std::string ToString(Basis basis) const;

        // Assignment
        Zahl& operator=(double d);
        Zahl& operator=(long l);
        Zahl& operator=(const Zahl& z);
};
```

Was hier fehlt sind die Datenelemente und die eine oder andere private Methode, die möglicherweise auch noch nötig sind.

Anwendung der Klasse Zahl

Diese Klasse kann schon gut gebraucht werden, um ein wenig mit Zahlenformaten zu jonglieren. Hier ein kleines Test-main:

```
#include "Zahl.h"
#include <iostream>
#include <string>
#include <crtdbg.h> // für das _ASSERT

using namespace std;

int main()
{
    string test("2.3456789");
    Zahl testZahl1(test, Decimal);
    Zahl testZahl2(2.3456789);

    double d1 = testZahl1.ToDouble();
    double d2 = testZahl2.ToDouble();

    _ASSERT(d1 == d2);
```

```
    cout << testZahl1.ToString() << endl;

    testZahl1 = 1234L;   // Zuweisung eines long
    string testString = testZahl1.ToString(Hexadecimal);
    cout << testString << endl;
    testString = testZahl1.ToString(Binaer);
    cout << testString << endl;

    return 0;
}
```

Ausnahmebehandlung

Beim Design der Klasse sind wir immer davon ausgegangen, dass die Konvertierung immer funktioniert. Wir wenden die Klasse auch an, als würde sie unter allen Umständen funktionieren. Tatsächlich ist es aber möglich, dass der string, der verwendet wurde um das Objekt zu erzeugen nicht in alle Formate umgewandelt werden kann. Wenn wir das mit Rückgabewerten signalisieren wollten müssten die Extraktoren etwa so aussehen:

```
// Extraction
bool ToDouble(double& d) const;
bool ToLong(long& l) const;
bool ToString(Basis basis, std::string& s) const;
```

Unser Code würde leiden, nur wegen gewissen *Ausnahmen*! Vor allem könnte es sein, dass wir Konvertierungsoperatoren anbieten wollen (leider ist das manchmal gefährlich). Hier als Beispiel der operator um ein Zahl-Objekt automatisch in einen double umzuwandeln:

```
operator double() const
{
    return ToDouble();
}
```

Dadurch ist es möglich ein Zahl-Objekt zu verwenden, als wäre es ein double, denn der Zuweisungsoperator für double existiert auch!

```
int main()
{
    Zahl eineZahl(0L);
    eineZahl = 23.456; // Zuweisung eines doubles
    double test = eineZahl; // operator double()

    return 0;
}
```

Wenn hier aber etwas schief geht, merken wir das nicht! Schlimmstenfalls wird der Kontostand von einem Kunden „verändert“ ;-)
Darum werden wir in der Klasse Zahl mit Ausnahmen arbeiten, die wir mit *catch* fangen können.

Implementation der Klasse Zahl

Die Implementation arbeitet sehr stark mit der *std::stringstream*-Klasse. Diese Klasse bietet ähnlich wie die meisten iostream-Klassen die Fähigkeit Text in Zahlen umzuwandeln und umgekehrt (genau wie das cout- oder das cin-Objekt). Ich habe mich dafür entschieden die Zahl intern als string zu speichern und je nach Bedarf daraus eine Zahl zu machen. Zusätzlich merkt sich die Klasse die Basis, die verwendet wurde um das *Zahl*-Objekt zu erzeugen.
Wie gesagt wird zur Umwandlung die Klasse *std::stringstream* verwendet. Man kann bei ihr wie beim cin- oder cout-Objekt den << oder den >> -operator verwenden. Hier ein kleines Beispiel:

```
#include <string>
#include <sstream> // für stringstream

using namespace std;

int main()
{
    stringstream stream;
    stream << 50; // eine Zahl in den stream
    string test;
    stream >> test; // als Text wieder rausholen

    return 0;
}
```

Nun ist es ganz einfach möglich durch <iomanip> die Basis für diese Extraktion zu bestimmen!

```
#include <sstream> // für stringstream
#include <iomanip> // für setbase

using namespace std;

int main()
{
    stringstream stream;
    stream << setbase(16); // Basis festlegen
    stream << 50;
    string test;
    stream >> test;

    return 0;
}
```

Leider funktioniert das nur für Oktal, Dezimal und Hexadezimal. Beim Binären Format brauchen wir ein wenig mehr STL-Zauberei. Dazu aber später mehr.

Der Konstruktor

Dieser setzt einfach die Datenelemente in der Initialisierungsliste.

```
#include <bitset>
#include "Zahl.h"
#include <iomanip>

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

Zahl::Zahl(const std::string& text, Basis basis)
     :m_basis(basis), m_string(text)
{
}
```

Die anderen Konstruktoren mit dem *long* oder dem *double* als Argument sind ein wenig spezieller, denn sie wenden einen *std::stringstream* an, um den Wert in einen string umzuwandeln.

```
Zahl::Zahl(double d)
     :m_basis(Decimal)
{
    std::stringstream stream;
    stream.exceptions(std::ios_base::failbit);
    // exceptions einschalten falls etwas schiefgeht
    stream << std::setprecision(16);
    stream << d;
    stream >> m_string;
}

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

Zahl::Zahl(long l)
     :m_basis(Decimal)
{
    std::stringstream stream;
    stream.exceptions(std::ios_base::failbit);
    // exceptions einschalten falls etwas schiefgeht
    stream << l;
    stream >> m_string;
}
```

Die fetten Zeilen schalten bei der *std::stringstream* Klasse einen Mechanismus ein, der dazu führt, dass bei einem Problem automatisch eine Ausnahme ausgelöst wird. Wir könnten auch nach jeder Zeile folgenden Aufruf machen:

```
if(stream.fail())
{
    throw exception(„Schlecht!“);
}
```

Aber es ist einfacher den bereits eingebauten Mechanismus anzuwenden! Wir schalten ein, dass falls bei einer stream - Operation etwas schief geht

und das fail-Flag gesetzt wird auch ein Objekt der Klasse *exception* geworfen wird. Diese Klasse *exception* ist in der STL definiert.

Zuweisungsoperatoren

Es folgen die Zuweisungsoperatoren, die nach dem gleichen Prinzip arbeiten. Sie erzeugen einen stringstream und fügen mit dem << operator den *long* oder den *double* in den stream ein. Danach wird mit dem >> operator ein string aus dem stream ausgelesen. Es wird wieder der exception-Mechanismus eingeschaltet!

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

Zahl& Zahl::operator =(long l)
{
    m_basis = Decimal;
    std::stringstream stream;
    stream.exceptions(std::ios_base::failbit);
    // exceptions einschalten falls etwas schiefgeht
    stream << l;
    stream >> m_string;

    return *this;
}

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

Zahl& Zahl::operator =(double d)
{
    m_basis = Decimal;
    std::stringstream stream;
    stream.exceptions(std::ios_base::failbit);
    // exceptions einschalten falls etwas schiefgeht
    stream << d;
    stream >> m_string;

    return *this;
}
```

Konvertierungsfunktionen

Die Konvertierungsfunktionen sind eigentlich einfach, wenn man einmal weiss wie (wir lernen viel einfach durch „abschauen“ in C++). Das Datenelement *m_string* enthält ja eine Zahl als string. Diese Zahl ist im Format angegeben, das im Datenelement *m_basis* gespeichert wird. Bevor wir also den string in das stream-Objekt schicken, können wir mit *setbase()* den stream auf das richtige Format einstellen.

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

long Zahl::ToLong() const
{
    std::stringstream stream;
    // exceptions einschalten falls etwas schiefgeht
    stream.exceptions(std::ios_base::failbit);
    // die Basis einstellen, bevor wir
    // den string einfügen
    stream << std::setbase(m_basis);
    stream << m_string;
    long result;
    stream >> result;

    return result;
}

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

double Zahl::ToDouble() const
{
    std::stringstream stream;
    // exceptions einschalten falls etwas schiefgeht
    stream.exceptions(std::ios_base::failbit);
    // die Basis einstellen, bevor wir
    // den string einfügen
    stream << std::setbase(m_basis);
    // möglichst beim double keine Stellen
    // abschneiden darum setprecision
    stream << std::setprecision(16);
    stream << m_string;

    double d;

    if(Decimal == m_basis)
    {
        // nur bei Decimal können
        // wir Nachkommastellen
        // behandeln
        stream >> d;
    }
    else
    {
        // Das Basisformat war nicht
        // Dezimal, darum nur Ganzzahlen
        int t;
        stream >> t;
        d = t;
    }

    return d;
}
```

Konvertierung in string mit gewünschter Basis

Diese Methode ist ziemlich happig. Wir erzeugen auch wieder eine stream, bei dem wir die exceptions einschalten. Danach wandeln wir unsere Zahl einfach in einen *double* mit der Methode *ToDouble*. Diesen *double* können wir problemlos in einen Decimal-Wert umwandeln, wir können sogar die Nachkommastellen behalten. Bei der Umwandlung in eine Hex- oder Oktalzahl gehen die Nachkommastellen verloren.

```
std::string Zahl::ToString(Basis basis) const
{
    std::stringstream stream;
    stream.exceptions(std::ios_base::failbit);
    // exceptions einschalten falls etwas schiefgeht
    std::string copy(m_string); // Kopie erstellen, damit
                                // unser Original-String
                                // nicht verändert wird

    // am besten holen wir uns einfach
    // einen double, mit lässt es sich
    // am besten weiterarbeiten
    double d = ToDouble();

    if(Decimal == basis)
    {
        stream << std::setbase(Decimal);
        stream << d;

        copy = stream.str();
    }
    else if(Binaer == basis)
    {
        // hier die nette kleine
        // Klasse bitset, die aus einer
        // Ganzzahl ein binaeres Muster
        // als String erzeugen kann
        int t = (int)d;
        std::bitset<32> bits(t);

        // leider ist diese Zeile kaum zu verstehen...
        copy = bits.to_string<char, std::char_traits<char>,
std::allocator<char> >();
    }
    else
    {
        // oktal oder hexadezimal wandeln
        // wir in eine Ganzzahl
        int t = (int)d;
        stream << std::setbase(basis);
        stream << t;

        copy = stream.str();
    }

    return copy;
}
```

Die Klasse std::bitset

Um in eine Binärdarstellung zu wandeln verwenden wir die Klasse *std::bitset*. Diese Klasse ist geeignet um einzelne bits als Array zu speichern. Es ist ähnlich wie ein Array von bool, braucht aber viel weniger Platz im Speicher und kann auch aus einem string (z.B. „00011100100“) erzeugt werden oder aus einem unsigned long. Um ein einzelnes bit abzufragen kann man einfach den []-operator verwenden. Hier ein kleines Beispiel:

```
#include <bitset>
#include <iostream>

using namespace std;

int main()
{
    // ein bitset mit 8 bits
    std::bitset<8> bits(155);

    for(int i = 0; i < 8; ++i)
    {
        if(bits[i] == true)
        {
            cout << "Bit " << i << " ist gesetzt" << endl;
        }
        else
        {
            cout << "Bit " << i;
            cout << " ist nicht gesetzt" << endl;
        }
    }

    return 0;
}
```

Die *std::bitset* Klasse ist eine template-Klasse, die als Template Argument die Anzahl bits verwendet. Wir sehen hier, dass als template-Argumente auch einfache Datentypen verwendet werden können (es müssen nicht immer Klassen sein). Hier ein Beispiel für eine solche template-Funktion:

```
template<int anzahl>
void HalloSchleife()
{
    for(int i = 0; i < anzahl; ++i)
    {
        cout << "Hallo" << endl;
    }
}
int main()
{

    HalloSchleife<40>();
    return 0;
}
```

Anwendung mit Ausnahmen

Jetzt da unsere Klasse fertig ist, können wir sie mit einem netten try-catch Block verwenden. Die Klasse Zahl wirft dann eine Exception, wenn etwas mit dem stream nicht stimmt. Dieser stream wirft im Fehlerfall eine Exception der Klasse *exception*. Wir können diese also so fangen:

```
try
{
    Zahl test("Hallo", Decimal);
    long blah = test.ToLong();
}
catch(exception& e)
{
    cout << "Etwas ist schief gegangen ";
    cout << e.what() << endl;
}
```

Die Klasse *exception* hat sogar eine Methode *what* die in einem string angibt, was schiefgelaufen ist. Manchmal ist diese Information aber doch recht dürftig...