Beispiel

Ziel, Inhalt

- Ich will an einem kleinen Beispiel zeigen, wie man durch eine einfache Abstraktion ein Problem vereinfachen kann und Fehler vermeidet.

Kleines Design-Beispiel

Ziel

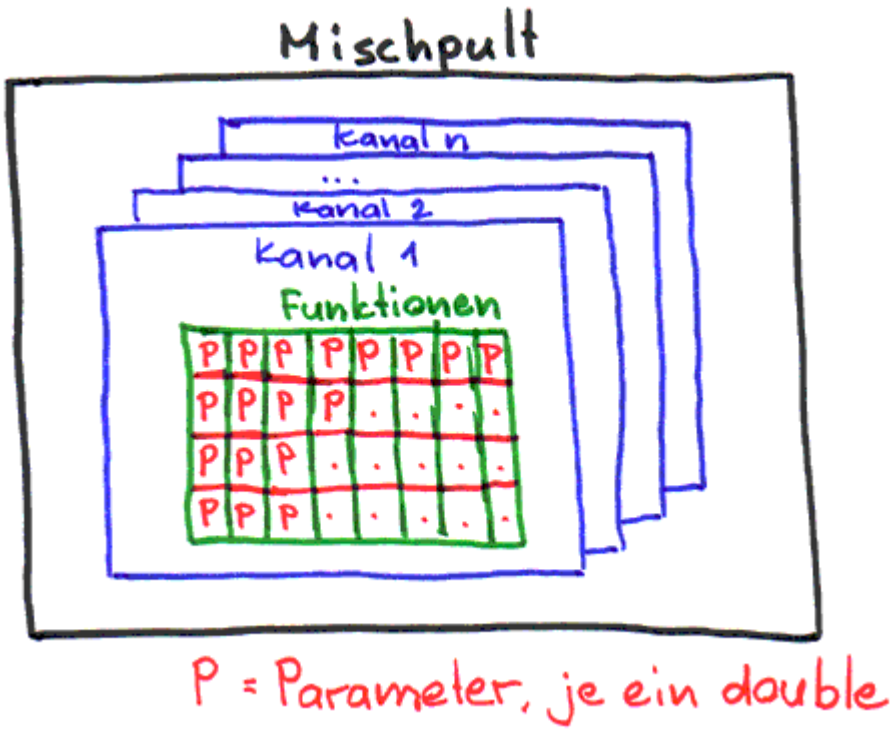
Ähnlich wie bei den Prüfungen, stellen wir uns eine kleine Aufgabe und versuchen die Klassen richtig zu definieren. Es sollte dabei offensichtlich werden, wie etwas durch objektorientiertes Design vereinfacht werden kann.

Problemstellung

Wir schreiben die Software für ein einfaches Mischpult. Das Mischpult hat mehrere Kanäle. In jedem Kanal können sich mehrere Funktionen befinden. In jedem dieser Funktionen befinden sich mehrere Parameter. Jeder Parameter ist im Grunde genommen nichts anderes als ein *double* Wert. Die Anzahl der Kanäle, die Anzahl der Funktionen und die Anzahl der Parameter pro Funktion sollen über eine Datei (Config.txt) konfigurierbar sein.

Erste Analyse

Eine kleine Analyse führt zu folgender kleinen Zeichnung.



Erster Lösungsansatz

Die erste Idee könnte uns zu einer einfachen aber unsicheren Lösung führen. Man erkennt nämlich eine dreidimensionale Matrix, in der sich nur Double-Werte befinden.

Erster Code

Mit dieser Idee ist es bereits möglich Code zu schreiben.

```
#include <fstream>

int main()
{
    // dynamisches dreidimensionales Array
    double*** data = 0;

    // Konfigurationsdaten auslesen
    std::ifstream config("config.txt");

    // Anzahl Kanäle
    int channelCount = 0;
    config >> channelCount;

    // Anzahl Funktionen
    int functionCount = 0;
    config >> functionCount;

    // Anzahl Parameter
    int parameterCount = 0;
    config >> parameterCount;

    // Kanäle erzeugen
    data = new double**[channelCount];

    // Funktionen erzeugen
    for(int channelIndex = 0;
        channelIndex < channelCount;
        ++channelIndex)
    {
        data[channelIndex] = new double*[functionCount];
        // Parameter erzeugen
        for(int functionIndex = 0;
            functionIndex < functionCount;
            ++functionIndex)
        {
            data[channelIndex][functionIndex] = new
double[parameterCount];
        }
    }

    // Zugriff auf Channel 0, Function 2, Parameter 4
    data[0][2][4] = 3.5;

    double test = data[0][2][4];

    // Freigeben des Speichers
    for(channelIndex = 0;
        channelIndex < channelCount;
        ++channelIndex)
    {
        for(int functionIndex = 0;
            functionIndex < functionCount;
            ++functionIndex)
```

```
            {
                delete [] data[channelIndex][functionIndex];
            }
            delete [] data[channelIndex];
        }
        delete [] data;

        return 0;
    }
```

Schaut Euch das genau an! Wir erzeugen dynamisch ein dreidimensionales Array. Das braucht ein wenig Hirnschmalz und Konzentration. In englischen Fachbüchern steht jeweils "This is not for the faint of heart", das ist nichts für solche mit Herzproblemen.

Besseres Design

Wir haben uns solche Mühe gemacht mit der Zeichnung. Wieso erzeugen wir daraus nicht ein anständiges Design und eine gute Implementation? Wir brauchen eine Klassenhierarchie, ein Objektmodell!

Top Down oder Bottom Up?

Wir können die Klassen von oben nach unten definieren oder von unten nach oben. Dabei scheint mir die Klasse *Mischpult* die oberste in der Hierarchie und die Klasse *Parameter* die unterste. Ich denke hier ist es einfacher von unten nach oben zu arbeiten.

Parameter

Der Parameter ist gemäss Zeichnung nur ein *double*. Es ist aber nicht auszuschliessen, dass dieser später mehr beinhaltet. Ein Parameter könnte auch einen Namen haben und weitere Attribute. Darum ist es nicht schlecht wir definieren einen Typen mit dem Namen *Parameter*. Das kann eine Klasse sein oder einfach nur ein *typedef*.

```
typedef double Parameter;
```

Function als Sammlung von Parametern

Ich würde für die Funktion, die eine Sammlung von Parameter ist eine eigene Klasse definieren. Die einfache Möglichkeit würde so aussehen:

```
#include <vector>

typedef double Parameter;

typedef std::vector<Parameter> Function;
```

Mit einer Klasse *Function* gewinnen wir aber die Möglichkeit, die Array-Grenzen zu prüfen:

```
#include <vector>
#include <crtdbg.h> // für ASSERT

typedef double Parameter;

class Function
{
public:
    // Konstruktor
    Function(int count)
    {
        // Vector vergrössern
        // und Werte initialisieren
        _parameters.resize(count, 0);
    }

    Parameter& operator[] (int index)
    {
        // Grenzen prüfen
        _ASSERT(index >= 0 && index < _parameters.size());
        return _parameters[index];
    }
private:
    std::vector<Parameter> _parameters;
};
```

Diese Klasse ist einfach und sicher. Wir setzen als Container hier einen vector ein.
Beachte den überschriebene Index – Operator (operator []). Wenn wir ein Objekt der Klasse *Function* haben, können wir es verwenden als wäre es ein normales Array von *Parameter*. Wichtig ist dabei, dass wir eine Referenz auf den Parameter zurückgeben, sonst wird eine Kopie des Wertes erzeugt, was meistens unerwünscht ist.

Der Kanal, oder Channel als Sammlung von Function

Parallel dazu können wir den Kanal in der Klasse *Channel* als Sammlung von *Function* definieren. Es wäre wieder eine einfacher typedef möglich, aber mir einem Konstruktor wird die Sache einfacher!

```
class Channel
{
public:
    Channel(int functionCount, int parameterCount)
    {
        _functions.resize(functionCount, parameterCount);
    }

    Function& operator[] (unsigned int index)
    {
        _ASSERT(index < _functions.size());
        return _functions[index];
    }

private:
    std::vector<Function> _functions;
};
```

Der Konstruktor nimmt hier als Argument nicht nur die Anzahl Funktionen sondern auch die Anzahl Parameter, die bei der resize Methode verwendet wird um den Konstruktor der Klasse *Function* aufzurufen.

Die Klasse Mixer

Jetzt fehlt nur noch die Klasse, für das gesamte Mischpult, so kommen wir zu einer akzeptablen Abstraktion unseres Problems. Bei dieser Klasse geben wir drei Konstruktor-Argumente mit. Leider haben wir hier aber nicht mehr die Möglichkeit mit der resize-Methode zu arbeiten, denn wir können nicht zwei Argumente mitgeben. Wir erzeugen daher die *Channel* Objekte dynamisch. Beim index – Operator müssen wir daher dereferenzieren, denn wir halten uns ja Zeiger auf die Objekte. Zusätzlich brauchen wir einen Destruktor um die dynamisch allozierten Objekte zu löschen.

```
class Mixer
{
public:
    Mixer(int channelCount,
          int functionCount,
          int parameterCount)
    {
        for(int i = 0; i < channelCount; ++i)
        {
            Channel* channel = new Channel(functionCount,
                                           parameterCount);
            _channels.push_back(channel);
        }
    }

    ~Mixer()
    {
        const size_t count = _channels.size();
        for(unsigned long i = 0; i < count; ++i)
        {
            Channel* channel = _channels[i];
            delete channel;
        }
    }

    Channel& operator[](unsigned int index)
    {
        _ASSERT(index < _channels.size());
        return *_channels[index]; // dereferenzieren
    }
private:
    std::vector<Channel*> _channels;
};
```

Die main - Funktion

Die main – Funktion ist jetzt von betörender Schönheit und Einfachheit:

```
int main()
{
    // Konfigurationsdaten auslesen
    std::ifstream config("config.txt");

    // Anzahl Kanäle
    int channelCount = 0;
    config >> channelCount;

    // Anzahl Funktionen
    int functionCount = 0;
    config >> functionCount;

    // Anzahl Parameter
    int parameterCount = 0;
    config >> parameterCount;

    Mixer meinMixer(channelCount, functionCount,
parameterCount);

    meinMixer[0][2][4] = 3.5;

    double test = meinMixer[0][2][4];

    // schöner:
    Channel& channel = meinMixer[0];
    Function& function = channel[2];
    Parameter& parameter = function[4];

    test = parameter;

    return 0;
}
```