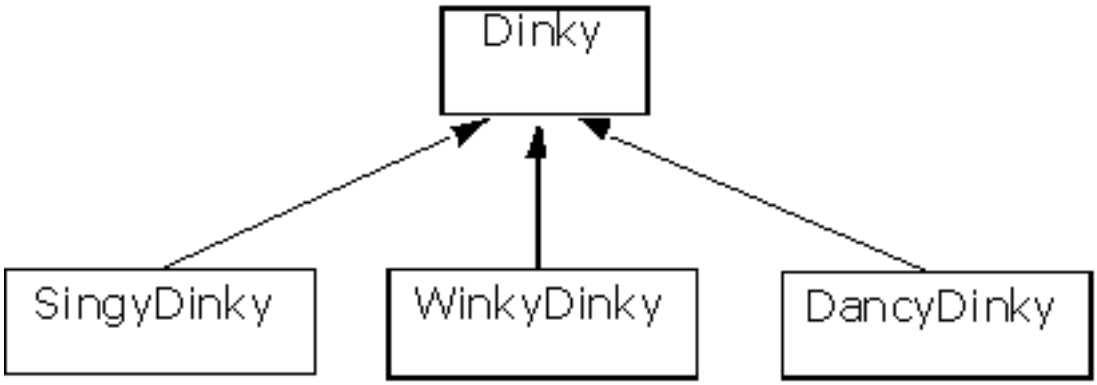
6. Abend

Vererbung und Polymorphismus

Beim Ableiten oder Erben von einer Klasse erweitern wir diese Klasse um Datenelemente und Methoden, können aber gewisse Datenelemente und Methoden der Basisklasse weiterhin verwenden. Wir kennen aus der Prüfung bereits die unsägliche WinkyDinky Klasse, die unglücklicherweise nur eine aus einer Familie von Dinky-Klassen ist. Hier ist die gesamte "Familie" :



Anhand dieser Klassen werden wir die Prinzipien der Vererbung und des Polymorphismus besser kennenlernen. Gewisse Eigenschaften sind allen Dinkys gemeinsam. Jedes Objekt der Klasse Dinky hat eine Farbe ! Genauso hat jedes Dinky eine Lieblingbeschäftigung. Solche Eingeschaften und Verhaltensweisen, die wirklich für jedes Objekt eines Dinky's vorhanden sind, werden in der Basisklasse definiert. Wir beginnen mit :

```
#ifndef DINKY_H
#define DINKY_H

class Dinky
{
    public:
        Dinky();
        ~Dinky();

        // Wir definieren in diesem
        // enum einige Farben, die
        // für ein Dinky grundsätzlich
        // möglich sind
        enum Farbe
        {
            undefiniert = -1,
            rot = 0,
```

```
            blau,
            gruen,
            gelb,
            violett,
            grau,
            schwarz
        };

        // Jedes Dinky hat ein
        // Hobby, das mit diesem
        // Funktionsaufruf ausgeführt
        // wird
        void machHobby();

        // protected Datenelemente
        // sind nur für abgeleitete
        // Klassen sichtbar.
    protected:
        // Jeder Dinky hat eine Farbe
        Farbe   m_farbe;

};


#endif
```

Ich definiere einen Konstruktor und einen Destruktor. Es kann aber sein, dass diese gar nicht gebraucht werden. Als weiteres definiere ich einen enum, der zur Klasse Dinky gehört. Es ist also auch möglich in einer Klasse weitere Datentypen zu definieren. Man kann sogar in einer Klasse eine andere Klasse definieren. Dadurch wird gezeigt, dass dieser enum einzig mit der Klasse Dinky Sinn macht. Man sagt auch, dass dieser Datentyp "Farbe" im Sichtbarkeitsbereich der Klasse Dinky ist. Da er public ist, kann man ihn aber auch ausserhalb der Klasse verwenden, nämlich auf diese Art :

```
#include "Dinky.h"


int main()
{
    // Neue Variable vom
    // Datentyp Farbe aus der
    // Klasse Dinky erstellen.
    Dinky::Farbe eineDinkyFarbe = Dinky::rot;

    return 0;
}
```

Eine Besonderheit stellt das protected - Schlüsselwort dar. Datenelemente, die im protected-Teil einer Klasse deklariert sind, sind von ausserhalb der Klasse nicht sichtbar, sind also privat für andere Klassen und Methoden, die nicht zur Klasse Dinky gehören, mit einer Ausnahme : protected Datenelemente und Methoden können ihn abgeleiteten Klassen verwendet werden. Wir werden das bei den Dinky's noch sehen...

Die Implementation, das heisst die .cpp-Datei offenbart keine Neuigkeiten :

```
#include "Dinky.h"
#include <iostream>
using namespace std;

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

Dinky::Dinky()
{
    // Bei diesem
    // Grund-Dinky ist
    // die Farbe undefiniert
    m_farbe = undefiniert;
    cout << "Ein Dinky kommt" << endl;
}

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

Dinky::~Dinky()
{
    cout << "Ein Dinky geht" << endl;
}

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

void Dinky::machHobby()
{
    cout << "Ich betreibe ";
    cout << "mein Hobby" << endl;
}

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

Als einziges fällt möglicherweise vielleicht der Konstruktor auf. Im Gegensatz zum main müssen wir um eine Farbe anzuwenden nicht den Sichtbarkeitsoperator schreiben (Dinky::undefiniert), da wir uns bereits im Sichtbarkeitsbereich der Klasse Dinky befinden.
Nun gilt es die anderen Dinky's zu definieren. Der Einfachheit halber befinden sich alle im gleichen Header- und Implementations-File.

```
 // Da wir ableiten, müssen
// wir die Basisklasse mittels
// include bekannt machen
#include "Dinky.h"

class SingyDinky : public Dinky
{
    public:
        SinkyDinky();
        ~SingyDinky();

        // dieses Dinky hat eine
        // eigene machHobby
        // Methode
        void machHobby();

        // Diese Funktion unterscheidet
        // das SingyDinky vom Dinky
        void Singe();
};


class WinkyDinky : public Dinky
{
    public:
        WinkyDinky();
        ~WinkyDinky();

        void machHobby();

        void Winke();
};


class DancyDinky : public Dinky
{
    public:
        DancyDinky();
        ~DancyDinky();

        void machHobby();

        void Dance();
};
```

Jede Klasse wird nun so erweitert, dass jede Dinky-Spezialisierung eine eigene Methode wie Dance oder Winke hat. Zusätzlich "**überschreiben**" wir in jeder Klasse die Methode "machHobby".

```
#include "DinkyKinder.h"
#include <iostream>

using namespace std;

SingyDinky::SingyDinky()
{
    // Zugriff auf das
    // protected Daten-
    // element aus der
    // Basisklasse
    m_farbe = rot;

    cout << "Ein SingyDinky ";
    cout << "kommt" << endl;
}

SingyDinky::~SingyDinky()
{
    cout << "Ein SingyDinky ";
    cout << "geht" << endl;
}

void SingyDinky::machHobby()
{
    // Rufe die eigene Hobby-
    // funktion auf
    Singe();
}

void SingyDinky::Singe()
{
    cout << "Ich singe !" << endl;
}

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

WinkyDinky::WinkyDinky()
{
    m_farbe = gruen;

    cout << "Ein WinkyDinky ";
    cout << "kommt" << endl;
}

WinkyDinky::~WinkyDinky()
{
```

```
    cout << "Ein WinkyDinky ";
    cout << "geht" << endl;
}

void WinkyDinky::machHobby()
{
    Winke();
}

void WinkyDinky::Winke()
{
    cout << "Ich winke !" << endl;
}

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

DancyDinky::DancyDinky()
{
    m_farbe = blau;

    cout << "Ein DancyDinky ";
    cout << "kommt" << endl;
}

DancyDinky::~DancyDinky()
{
    cout << "Ein DancyDinky ";
    cout << "geht" << endl;
}

void DancyDinky::machHobby()
{
    Dance();
}

void DancyDinky::Dance()
{
    cout << "Ich tanze !" << endl;
}
```

Anhand verschiedener Testprogramme werden wir einige Regeln besser kennenlernen, die bei der Anwendung von abgeleiteten Klassen wichtig sind.
Eine davon ist, dass wenn man eine ein Objekt einer abgeleiteten Klasse erzeugt, wird zuerst der Basisklassenkonstruktor aufgerufen ! Hier ein Beispiel dazu :

```
#include "Dinky.h"
#include "DinkyKinder.h"
```

```
#include <iostream>

using namespace std;

int main()
{
    // Konstruktor-Aufruf
    Dinky mamma;

    mamma.machHobby();

    cout << endl;
    cout << "***";
    cout << endl << endl;

    // Abgeleitete Klasse
    // anwenden
    WinkyDinky kind1;
    kind1.machHobby();

    cout << endl;
    cout << "***";
    cout << endl << endl;
    return 0;
}
```

Dieses Programm erzeugt folgende Ausgabe :

```
Ein Dinky kommt
Ich betreibe mein Hobby

***

Ein Dinky kommt
Ein WinkyDinky kommt
Ich winke !

***

Ein WinkyDinky geht
Ein Dinky geht
Ein Dinky geht
```

Man sieht also, dass in der Zeile nach den Sternen, bevor das WinkyDinky erzeugt wird, der Dinky-Teil dieses Objekts erzeugt wird, also der Konstruktor von Dinky zuerst ausgeführt wird.

Der Grund dafür ist, dass man in der abgeleiteten Klasse auf die Datenelemente der Basisklasse **immer** zugreifen kann. Insbesondere kann man im Konstruktor der abgeleiteten Klasse auf Datenelemente der Basisklasse zugreifen, wie in diesem Beispiel auf "m_Farbe".
Ähnliches gilt für den Destruktor. Beim Destruktoraufruf der abgeleiteten Klasse muss sichergestellt sein, dass die Datenelemente der Basisklasse noch vorhanden sind, also wird der Destruktor der Basisklasse erst nach dem Destruktor der abgeleiteten Klasse aufgerufen.
Betrachten wir noch ein Beispiel :

```
int main()
{
    // Abgeleitete Klasse
    // erzeugen
    WinkyDinky kind1;
    kind1.machHobby();

    cout << endl;
    cout << "***";
    cout << endl << endl;

    // es ist zulässig
    // einen Basisklassen-
    // zeiger auf eine ab-
    // geleitete Klasse zeigen
    // zu lassen
    Dinky* dinky = &kind1;

    dinky->machHobby();

    cout << endl;
    cout << "***";
    cout << endl << endl;

    return 0;
}
```

Ausgabe :

```
Ein Dinky kommt
Ein WinkyDinky kommt
Ich winke !

***

Ich betreibe mein Hobby
```

```
***

Ein WinkyDinky geht
Ein Dinky geht
```

Da wir beim zweiten Aufruf von "machHobby" nur einen Zeiger auf die Basisklasse haben, wird auch die Basisklassenfunktion aufgerufen, obwohl wir ein WinkyDinky-Objekt haben und nicht nur ein Dinky !
Eine Regel, die auch im Code-Kommentar steht ist die folgende, man darf einen Zeiger vom Basisklassentyp auf ein Objekt einer abgeleiteten Klasse zeigen lassen. Das gilt ebenso für Referenzen. Hier der passende Code-Abschnitt:

```
    // es ist zulässig
    // eine Referenz auf Basisklasee
    // auf eine abgeleitete Klasse
    // zu haben.
    Dinky& dinky = kind1;

    dinky.machHobby();
```

Wieder wird nur die Basisklassenmethode "machHobby" aufgerufen, obwohl das Objekt von der abgeleiteten Klasse WinkyDinky ist.
Dieses Verhalten tritt bei dynamisch erstellten Objekten noch viel stärker hervor, denn dort kann es passieren, dass nur der Destruktor der Basisklasse aufgerufen wird, falls der Zeiger vom Basisklassentyp ist, was wir am nächsten Beispiel sehr gut sehen.

```
#include "Dinky.h"
#include "DinkyKinder.h"
#include <iostream>

using namespace std;

int main()
{
    // Abgeleitete Klasse
    // dynamisch erzeugen
    Dinky* kind1 = new WinkyDinky;
    kind1->machHobby();

    cout << endl;
    cout << "***";
    cout << endl << endl;

    // dynamisch angelegtes
```

```
    // Objekt zerstören
    delete kind1;

    return 0;
}
```

Wir verwenden einen Zeiger auf die Basisklasse von WinkyDinky mit dem namen kind1, was wie bereits gesagt absolut korrekt ist, denn der WinkyDinky ist immer auch ein Dinky, wenn auch ein spezieller. Die Ausgabe des Programms zeigt auch genau was passiert.
Zuerst wird der Basisklassenkonstruktor aufgerufen, danach der Kosntruktor von WinkyDinky. Es macht also keinen Unterschied ob wir das Objekt auf dem Stack wie oben erzeugen oder dynamisch auf dem Heap. Danach rufen wir mit dem Zeiger (Zeiger auf Basisklassentyp, wohlgemerkt) "machHobby" auf. Es wird auch die Basisklassenfunktion aufgerufen. Das Schlimme ist aber der Aufruf von **delete,** denn es wird nur der Destruktor der Basisklasse aufgerufen. Hätten wir in der Klasse WinkyDinky irgendwo Speicher alloziert und diesen korrekterweise im Destruktor erst freigegeben, wäre dieses Verhalten fatal.
Hier die Ausgabe des Programms:

```
Ein Dinky kommt
Ein WinkyDinky kommt
Ich betreibe mein Hobby

***

Ein Dinky geht
```

Hier Code, der im obigen Beispiel zu Speicherlecks geführt hätte:

```
WinkyDinky::WinkyDinky()
{
    m_farbe = gruen;

    cout << "Ein WinkyDinky ";
    cout << "kommt" << endl;

    m_p = new char[50];
}

WinkyDinky::~WinkyDinky()
{
    cout << "Ein WinkyDinky ";
    cout << "geht" << endl;

    delete [] m_p;
}
```

Dieses Verhalten, das dazu führt, dass immer nur die Methode aufgerufen wird, für deren Typ man den Zeiger oder die Referenz hat nennt man also **statische Bindung**, wo der Compiler zur Kompilierzeit festlegt was aufgerufen wird.

Dynamische Bindung

Es gibt einige Gründe dafür ein anderes, flexibleres Verhalten zu wünschen. Angenommen wir wollen dem Benutzer die Möglichkeit geben einige Dinky's zu erstellen und ihm dabei die Wahl zu geben was genau für ein Dinky erzeugt wird. Diese Dinky's würden wir in einer Liste halten. Folgender Code könnte diesem Wunsch ungefähr entsprechen.

```
#include "Dinky.h"
#include "DinkyKinder.h"
#include <iostream>
#include <list>

using namespace std;

// Im neuen Typ "Dinkys"
// können wir unsere Dinkys
// als Zeiger verwalten
typedef list<Dinky*> Dinkys;

int main()
{
    // leere Liste erzeugen
    Dinkys meineDinkys;

    // ewige Schleife
    while(1)
    {
        cout << "[1] für DancyDinky" << endl;
        cout << "[2] für WinkyDinky" << endl;
        cout << "[3] für SingyDinky" << endl;
        cout << "[0] für Ende" << endl;

        int Eingabe = 0;
        cin >> Eingabe;

        // Abbrechen ?
        if(0 == Eingabe)
        {
            // Aus Schleife springen !
            break;
        }
```

```
        Dinky* neuerDinky = 0;

        switch(Eingabe)
        {
            case 1:
                neuerDinky = new DancyDinky;
                break;
            case 2:
                neuerDinky = new WinkyDinky;
                break;
            case 3:
                neuerDinky = new SingyDinky;
                break;
            default:
                break;
        }
        if(Eingabe != 0)
        {
            meineDinkys.push_back(neuerDinky);
        }
    }

    // ein iterator
    // ist wie ein Zeiger
    // mit dem man durch ein Array
    // hindurchgeht (iteriert)
    Dinkys::iterator it;
    for(it = meineDinkys.begin(); it != meineDinkys.end(); ++it)
    {
        // Da it wie ein Zeiger auf
        // Dinky* ist, müssen wir ihn
        // dereferenzieren.
        // it zeigt auf ein Element in der
        // Kollektion, wo Dinky* drin sind
        Dinky* aktDinky = *it;

        aktDinky->machHobby();

        // gleich auch löschen
        delete aktDinky;
    }

    return 0;
}
```

Mit den vorhandenen Klassen ist das Verhalten dieses Programms nicht wünschenswert. Durch **dynamische Bindung** bekommen wir aber ein vernünftigeres und hier erwünschtes Verhalten.
Durch das Schlüsselwort **virtual** welches in der Klassendeklaration (also in der Header-Datei) vor eine Methode gesetzt werden kann, entbinden wir den Compiler schon beim kompilieren die Methodenaufrufe aufzulösen. Der Compiler erzeugte bei der statischen Bindung Code, der die Methode des jeweiligen Zeigertyps aufrufen liess. Durch das **virtual**-Schlüsselwort wird erst zur Laufzeit entschieden welche "machHobby" Methode aufgerufen wird und zwar aufgrund des Objektes, das sich hinter einem Zeiger oder einer Referenz verbirgt. Wir ändern einfach die Dinky-Deklaration :

```
#ifndef DINKY_H
#define DINKY_H

class Dinky
{
    public:
        Dinky();
        // Der Destruktor
        // sollte immer virtuell
        // sein, wenn von der Klasse
        // Ableitungen möglich sein
        // sollen !!!
        virtual ~Dinky();

        // Wir definieren in diesem
        // enum einige Farben, die
        // für ein Dinky grundsätzlich
        // möglich sind
        enum Farbe
        {
            undefiniert = -1,
            rot = 0,
            blau,
            gruen,
            gelb,
            violett,
            grau,
            schwarz
        };

        // Jedes Dinky hat ein
        // Hobby, das mit diesem
        // Funktionsaufruf ausgeführt
        // wird
        virtual void machHobby();
```

```
        // protected Datenelemente
        // sind nur für abgeleitete
        // Klassen sichtbar.
    protected:
        // Jeder Dinky hat eine Farbe
        Farbe   m_farbe;
};

#endif
```

Nach dieser Änderung, die nur aus den beiden **virtual-**Deklarationen in der Dinky-Headerdatei besteht, reagiert unser Programm völlig anders. Obwohl wir nur Zeiger auf die Basisklasse verwenden, verhält sich jedes Objekt so wie es eigentlich gedacht ist.
Das **virtual**-Schlüsselwort ist dabei nur in der Basisklasse nötig. Bei den abgeleiteten Klassen muss es bei den virtuellen Methoden nicht noch einmal angegeben werde. Es ist aber häufig klarer wenn man das Schlüsselwort wiederholt, denn es erhöht die Übersichtlichkeit bei vielen Vererbungsstufen beträchtlich. Here you go :

```
// Da wir ableiten, müssen
// wir die Basisklasse mittels
// include bekannt machen
#include "Dinky.h"

class SingyDinky : public Dinky
{
    public:
        SingyDinky();
        virtual ~SingyDinky();

        // dieses Dinky hat eine
        // eigene machHobby
        // Methode
        virtual void machHobby();

        // Diese Funktion unterscheidet
        // das SingyDinky vom Dinky
        void Singe();
};


class WinkyDinky : public Dinky
{
    public:
        WinkyDinky();
        virtual ~WinkyDinky();
```

```
        virtual void machHobby();

        void Winke();
};


class DancyDinky : public Dinky
{
    public:
        DancyDinky();
        virtual ~DancyDinky();

        virtual void machHobby();

        void Dance();
};
```

Wichtig ist es vor allem den Destruktor virtuell zu machen, wenn man erwarten kann, das von einer Klasse abgeleitet wird !
Dieses Grundgerüst, wie wir es mit unseren Dinky's gebaut haben (wir werden sie bald los sein diese Dinky's), wo in einer Liste oder in einem Vektor nur Basisklassenzeiger verwaltet werden, die Objekte sich aber trotzdem "typgerecht" verhalten wird sehr sehr häufig verwendet.
Beispiel : Wenn wir mit dem Borland C++Builder ein Formular erstellen sind die Elemente, die wir darauf anordnen zwar unterschiedlich (ein Knopf ist anders als ein Eingabefeld) haben aber auch gemeinsame Datenelemente, wie zum Beispiel die Position und andere Dinge, die wir im Objektinspektor betrachten können. Unter anderem aber auch Methoden, die das Control frisch zeichnen lassen. So sit ein TEditControl genau wie ein TButton von TWinControl abgeleitet. Die Klasse TWinControl enthält unter anderem eine Methode "Repaint". Diese Methode ist virtuell und wird von der jeweiligen Spezialisierung (so nennt man abgeleitete Klassen auch) überschrieben. Auf unserer Form liegen also viele von TWinControl abgeleitete Objekte. Halten wir diese Objekte alle in einer Liste von TWinControl* - Zeigern, z.B. std::list<TWinControl*> m_myControls, können wir in einer Schleife alle diese Objekte neu zeichnen, indem wir "Repaint" aufrufen. Durch die Virtualität dieser Methode wird sichergestellt, dass immer der richtige Code aufgerufen wird, ein TEdit also wie eine EditBox aussieht und ein TButton wie ein Button.
Dieser Mechanismus ist für diesen speziellen Fall vor uns versteckt. Der BCB stellt ja genau dafür ein "Framework" zur Verfügung, damit wir uns nicht mehr um solche "Kleinigkeiten" kümmern müssen. Wir sehen aber auch, dass wenn wir ein Form erzeugen, dieses so deklariert ist : class TForm1 : public TForm und obwohl irgendwo im Code des Borland-Frameworks stehen könnte : TForm* userForm = GetUserForm(); delete userForm; um das Formulat zu löschen, wird ein Destruktor, den wir in unsere Klasse einbauen können auch aufgerufen, denn der Destruktor ist virtuell.

Abstrakte Basisklassen

Ich muss leider noch mal unsere Dinky's bemühen. Im Grunde genommen ist unsere Basisklasse "Dinky" zu nichts nütze, denn es hat keine definierte Farbe (siehe Konstruktor) und auch kein genau definiertes Hobby. Es beschreibt aber das Verhalten eines jeden Dinky's (z.B. WinkyDinky) insofern, als dass jedes Dinky ein Hobby und eine Farbe hat, macht also als Basisklasse Sinn. Es macht aber keinen Sinn solch ein Objekt zu erstellen. In einer solchen Situation ist es sinnvoll diese Klasse als **abstrakte** Klasse zu definieren. Eine abstrakte Klasse ist eine Klasse, von der man keine Objekte erzeugen/instantieren kann. Wir deklarieren eine abstrakte Klasse indem wir die Methoden (Funktionen), die in der Basisklasse nicht implementiert werden soll so :

```
virtual void machHobby() = 0;
```

Durch diese Änderung können wir die Implementation dieser Funktion in der Datei Dinky.cpp entfernen. Die Funktion "machHobby" ist eine rein virtuelle Funktion.
In einer Basisklasse wird also nur noch implementiert, was bei allen Objekten gemeinsam ist, während die Methoden, die in den Spezialisierungen überschrieben werden sollen als **"= 0;"** deklariert werden.
Dieses Konzept der Abstraktion kann so weit geführt werden, dass alle Funktionen =0 also rein virtuell deklariert werden. Eine solche Klasse ist eine **rein** abstrakte Basisklasse und beschreibt nur eine "Schittstelle", also ein gemeinsames Verhalten aller von ihr abgeleiteten Klassen.