OOP - VOM ASSEMBLER ÜBER C ZU C++

- *Überblick über OOP*
- *Programmierung im Wandel der Zeit*
- *Moderne C++ Compiler für leistungsfähige Mikrocontroller*

Wilhelm Brezovits DSK340//oop

Allgemeines

War vor einigen Jahren bei der Programmierung von Mikrocontrollern noch die übliche Diskussion Assembler oder C, so ist heute das Diskussionsthema C oder C++ .

Im Gegensatz zum zentralen Aufbau eines Programms (ausgehend von `main()`) bietet die OOP die Möglichkeit, einzelne eigenständige Intelligenzinseln (Klassen) zu definieren.

Der wohl größte Vorteil der OO-Programmierung ist die Wiederverwendbarkeit, d.h. Klassen müssen so entwickelt werden, daß sie alle Verantwortlichkeiten über ihre Daten und ihr Verhalten übernehmen und dadurch auch unter völlig verschiedenen Umgebungen in unterschiedlichen Situationen eingesetzt werden können.

Man unterscheidet zwischen 2 Arten der Wiederverwendung: Benutzungsbeziehung und Vererbungsbeziehung.

OOP im Detail

Die drei großen Säulen der objektorientierten Programmierung (OOP):

- Klassenkonzept (Kapselung durch Klassen)
- Vererbung (hierarchische Wiederverwendung in der Vererbung)
- Polymorphie (Vielgestaltigkeit)

Hinweis

Die folgenden Programmbeispiele wurden mit dem Microsoft Visual C++ Compiler (Version 4.1) und dem Tasking C++ Compiler (Version 5.1r0) übersetzt und erfolgreich zum Ablauf gebracht.

Compiler	Zielsystem
Microsoft Visual C++ 4.1	Intel Mikroprozessor im PC
Tasking C++	Siemens Mikrocontroller

Klassenkonzept

Im Prinzip ist eine Klasse (`class`) ein Datentyp, also eine komplexe Datenstruktur so wie in C ein Aufzähltyp (`enum`), eine Struktur (`struct`), ein Bitfeld oder eine Variante (`union`).

Klassen (Datenkapsel, Struktur) können als eine Erweiterung von Strukturen betrachtet werden.

Klassen beinhalten Daten (Attribute, Eigenschaften, data member) und

Funktionen (Elementfunktionen, Member-Functions, Methoden, Botschaften).

Daten und Funktionen bilden in einer Klasse eine abgeschlossene Einheit.

Die Klasse muß dabei die volle Verantwortung für ihre Daten und ihr Verhalten übernehmen!

Interne Daten und Funktionen werden innerhalb der Klasse versteckt (Zugriffsattribut, Gültigkeitsbereich: `private` oder `protected`).

Dem Anwender der Klasse wird eine Schnittstelle (`public`) zur Benutzung zur Verfügung gestellt.

Von einer Klasse können beliebig viele Variablen (Instanzvariablen, Instanzen, Objekte) erzeugt werden.

Vererbung

Eine Klasse (abgeleitete Klasse) kann eine andere Klasse (Basisklasse) erben und damit existierenden Code wiederverwenden.

Natürlich kann eine Klasse auch viele andere Klassen erben (=Mehrfachvererbung).

In der abgeleiteten Klasse sind dabei alle Daten/Funktionen der Basisklasse(n) verfügbar.

Beispiel zu 2.1.) Klassenkonzept:

Datei: bsp_2_1.cpp

```
#include <stdio.h>
#ifndef _MSC_VER  // Compiler==Tasking C++ Compiler
#include "serio.h"
#endif
```

```
class Basisklasse       // Basisklasse = Bezeichnung (Name, Typname) der Klasse
{
};
```

```
class Klassenname : Basisklasse   // Klassenname = Bezeichnung (Name, Typname)
{                                 // : Basisklasse = Vererbung
public:
        // Deklarationen, Definitionen; Schnittstelle nach außen
protected:
        // Deklarationen, Definitionen;
        // in eigener und abgeleiteter Klasse sichtbar
private:
        // Deklarationen, Definitionen; nur innerhalb der Klasse sichtbar
};
```

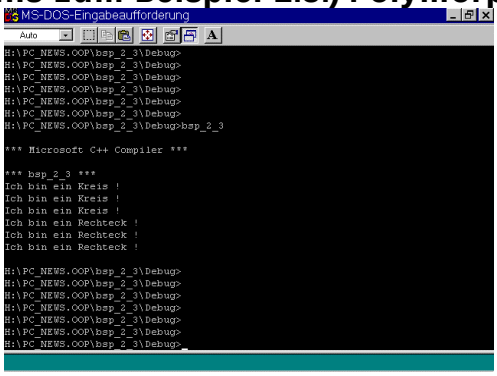
```
void main (void)
{
#ifdef _MSC_VER     // Compiler==Microsoft C++ Compiler
    printf("\n*** Microsoft C++ Compiler ***\n");
#else               // Compiler==Tasking C++ Compiler
    init_serio();
    printf("\n*** Tasking C++ Compiler ***\n");
#endif

    printf("\n*** bsp_2_1 ***\n");

    Basisklasse EinObjekt_der_Basisklasse;          // Instanz, Variable, Objekt
    Klassenname EinObjekt_der_abgeleiteten_Klasse;  // Instanz, Variable, Objekt

    while(1);
}
```

Beispiel zu 2.2.) Vererbung

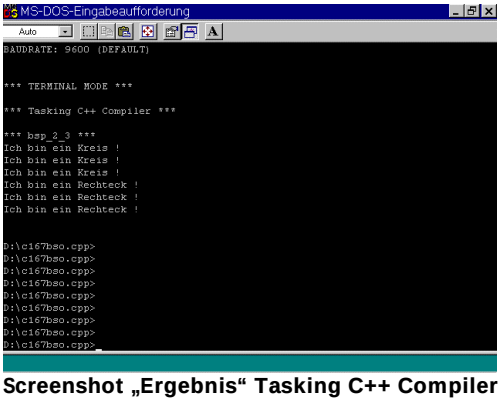
Datei: bsp_2_2.cpp

```
#include <stdio.h>
#ifndef _MSC_VER  // Compiler==Tasking C++ Compiler
#include "serio.h"
#endif

struct datum
{
        int tag,monat,jahr;
};

class mensch
{
        datum geburtstag;
        int groesse;
};

class mann : public mensch                  // Vererbung
{
        char bart;
};

class frau : public mensch                  // Vererbung
{
        int kinder;
};

class Paar : public mann, public frau       // Mehrfachvererbung
{
        datum ehe;
};

void main (void)
{
# ifdef _MSC_VER      // Compiler==Microsoft C++ Compiler
       printf("\n*** Microsoft C++ Compiler ***\n");
# else                // Compiler==Tasking C++ Compiler
       init_serio();
       printf("\n*** Tasking C++ Compiler ***\n");
# endif
      printf("\n*** bsp_2_2 ***\n");
      Paar paar1, paar2, paar3;
      while(1);
}
```

Ergebnis zum Beispiel 2.3.) Polymorphie



Screenshot „Ergebnis“ Microsoft C++ Compiler <



Screenshot „Ergebnis“ Tasking C++ Compiler

Polymorphie

Polymorphie beschreibt die Fähigkeit, an unterschiedliche Objekte (entstanden durch Vererbung, also Ableitung) gleichnamige Botschaften zu senden, die abhängig vom Objekt zu unterschiedlichen Reaktionen führen.

Das bedeutet, bei einem Methodenaufruf wird anhand des Typs des gebundenen Objektes vom Compiler realisiert durch eine vom Compiler selbst generierte VMT-Tabelle (virtuelle Methoden Tabelle) herausgefunden, welche Methode er rufen soll.

Beispiel zu 2.3.) Polymorphie

Datei: bsp_2_3.cpp

```
#include <stdio.h>
#ifndef _MSC_VER  // Compiler==Tasking C++ Compiler
#include "serio.h"
#endif

class Zeichnung                    // abstrakte Basisklasse
{
public:
        virtual void drucke() = 0;
};

class Kreis : public Zeichnung     // abgeleitete Klasse
{
public:
        void drucke()
        {
                printf("Ich bin ein Kreis !\n");
        }
};

class Rechteck : public Zeichnung      // abgeleitete Klasse
{
public:
      void drucke()
      {
        printf("Ich bin ein Rechteck !\n");
      }
};

void main (void)
{
#ifdef _MSC_VER       // Compiler==Microsoft C++ Compiler
      printf("\n*** Microsoft C++ Compiler ***\n");
#else                 // Compiler==Tasking C++ Compiler
      init_serio();
      printf("\n*** Tasking C++ Compiler ***\n");
#endif

      printf("\n*** bsp_2_3 ***\n");

      Kreis kreis1;         // 1 Objekt der Klasse Kreis
      Rechteck rechteck1;   // 1 Objekt der Klasse Rechteck

      Zeichnung *p;         // Basisklassenzeiger anlegen

      kreis1.drucke();      // statische Variante der
                            // Polymorphie
      p=&kreis1;
      p->drucke();          // dynamische Variante
                            // der Polymorphie
      (*p).drucke();        // ... andere Schreibweise

      rechteck1.drucke();   // statische Variante
                            // der Polymorphie
      p=&rechteck1;
      p->drucke();          // dynamische Variante
                            // der Polymorphie
      (*p).drucke();        // ... andere Schreibweise

      while(1);
}
```

Templates (Codeschablonen)

Natürlich beherrschen Mikrocontroller C++ Compiler auch Templates:

Bei den Grundalgorithmen der Informatik (Listen, Bäume. u.s.w.) ist der Algorithmus immer der gleiche, nur der Datentyp der verwalteten Elemente ist immer ein anderer.

Templates bieten die Möglichkeit, einen oder mehrere Platzhalter für Datentypen bei der Definition von Funktionen oder Klassen anzugeben und so einen allgemeingültigen, typunabhängigen aber typsicheren Algorithmus zu formulieren, wobei der Compiler die entsprechende Funktion/Klasse erst bei der Verwendung für die Platzhalter generiert.

Beispiel zu 3.) Templates

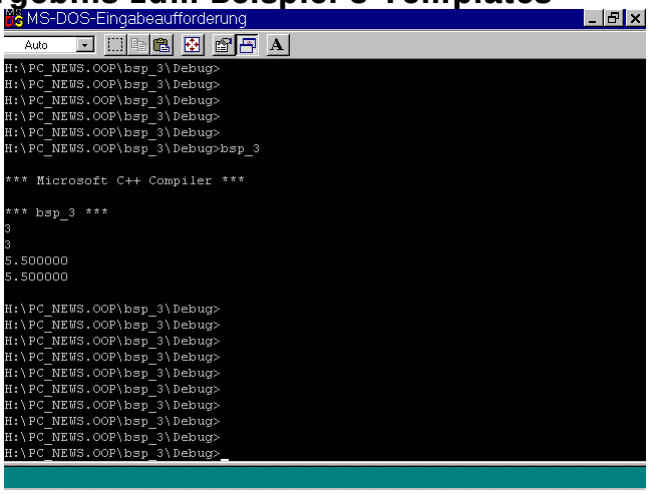
Datei: bsp_3.cpp

```
#include <stdio.h>
#ifndef _MSC_VER  // Compiler==Tasking C++ Compiler
#include "serio.h"
#endif

// Achtung:
// Typen wie int oder double können als bereits definierte
// Klassen betrachtet werden.
// Bei selbstdefinierten Typen (Klassen) müssen alle
// vorkommenden Operationen definiert werden (hier: >)!
```

```
template <class T1>    // T1... Platzhalter für Template
T1 max(T1 a, T1 b)
{
    return a>b?a:b;
}

#ifndef _MSC_VER  // Compiler==Tasking C++ Compiler
#pragma instantiate int max(int,int)
#pragma instantiate double max(double,double)
#endif
```

```
void main(void)
{
#ifdef _MSC_VER     // Compiler==Microsoft C++ Compiler
    printf("\n*** Microsoft C++ Compiler ***\n");
#else               // Compiler==Tasking C++ Compiler
    init_serio();
    printf("\n*** Tasking C++ Compiler ***\n");
#endif

    printf("\n*** bsp_3 ***\n");

    int a=2,b=3;
    double x=5.5,y=4.4;

    printf("%d\n",max(a,b));  // entspricht: max(int,int)
    printf("%d\n",max(b,a));  // entspricht: max(int,int)

    printf("%f\n",max(x,y));  // entspricht: max(double,double)
    printf("%f\n",max(y,x));  // entspricht: max(double,double)

    while(1);
}
```

Ergebnis zum Beispiel 3 Templates



Screenshot „Ergebnis“ Microsoft C++

Screenshot „Ergebnis“ Tasking C++ Compiler

Ein leistungsfähiger µC C++ Compiler

Jetzt möchte ich die WWW-Seite eines C/C++ Compiler-Hersteller für viele Mikrocontroller (µC) und Mikroprozessor (µP) Familien anführen:

Mikrocontroller wollen mehr!

Mikrocontroller C/C++ Compiler verfügen natürlich über zusätzliche Erweiterungen um auf die Architektur voll zugreifen zu können:

- zusätzliche Datentypen (*)
- Steuerwörter (*)
- intrinsic-Funktionen (*)

(*) **Hinweis**: Diese Erweiterungen sind natürlich nicht (wie ANSI C) genormt. Das bedeutet, daß die entsprechende Syntax hier von Compiler zu Compiler abweicht.

zusätzliche Datentypen

Bits

Mikrocontroller haben Bit-Befehle (boolsche Verknüpfung, Bit-Setzen, Bit-Loeschen, Sprünge wenn gesetzt/nicht gesetzt, ...). Zur Unterstützung verfügen Mikrocontroller Compiler deshalb über den Datentyp Bit!

SFR (Special Function Register)

Die On-Chip-Peripherals (Port-Zustände, Timer-Control-Register, u.s.w.) werden im Mikrocontroller durch den Befehlsvorrat (Zugriff) unterstützt.

Im Prinzip handelt es sich hier um den Datentyp volatile unsigned int mit der Eigenschaft eine fixe Adresse im Adressraum zu sein (SFR-Bereich).

Zur Unterstützung der On-Chip-Peripherie verfügen Mikrocontroller deshalb über den Datentyp sfr.

Steuerwörter

Um für eine Variable eine bestimmte Adresse (einen bestimmten Platz) im internen oder externen Speicher festzulegen, bzw. ein bestimmtes Adressierungsschema auszuwählen gibt es Steuerwörter.

z.B.: `idata/iram, bdata/bitword, sdata/system, near/near, far/far, huge/huge, xhuge/huge,` (Implementierung: KEIL/TASKING-Syntax)

Intrinsic-Funktionen

Der Befehlsvorrat des Mikroprozessors im Mikrocontroller sieht Befehle vor wie z.B. `IDLE` (Enter_Idle_Mode), `NOP` (No_Operation), `PWRDN` (Enter_power_Down_Mode), `SRVWDT` (Service_Watchdog_Timer), welche die Hochsprache (ANSI-C) normalerweise nicht unterstützt.

Damit der Mikrocontroller-Hochsprachen-Programmierer auch Zugriff auf solche Befehle hat, gibt es die intrinsic-Funktionen:

einige Beispiele

`_idle_()/_idle(), _nop_()/_nop(), _pwrdn_()/_pwrdn(), _srvwdt_()/_srvwdt()`, (Implementierung: KEIL/TASKING-Syntax)