Stolpersteine und Spezialitäten in C/C++ mit Hinweisen auf Java
betreut von Aegidius Plüss, Gymnasium Bern-Neufeld

Setzt man Java als Programmiersprache für Anfänger ein, wird dieser gleich zu Beginn mit der objektorientierten Programmierung (OPP) konfrontiert. Bei Verwendung einer hybriden Programmiersprache wie C++ kann diese Schwierigkeit umgangen werden, indem zuerst elementare, prozedurale Programmtechniken eingeübt werden, bevor der Schritt zur OOP vollzogen wird. Mit Java ist der Einstieg schwieriger, denn die Lehrperson muss mit viel didaktischem Geschick gleich zu Beginn Klassendefinitionen erläutern, beispielsweise am Beispiel des bekannten "Hello World"- Programms:

```
public class HelloWorld {
   public static void main( String[] args ) {
      System.out.println("Hello World" );
   }
}
```

Neben der Notwendigkeit, das Konzept von Klassen und deren Instanzen einzuführen, müssen auch Begriffe wie *public*, *static* und der Methodenaufruf mit dem Punktoperator bereits beim einfachsten Programmbeispiel besprochen werden. Dies ist zwar möglich, aber didaktisch keineswegs einfach.

Im Folgenden werden als Hilfestellung die Begriffe der *statischen* Methoden und *statischen* Attribute (Klassenvariablen) behandelt. Die grundsätzlichen Überlegungen gelten sowohl für C++ wie für Java.

P71:
Was sind statische Methoden und statische Attribute?

L71:
Eigentlich ist die Erklärung ganz einfach: Normalerweise gehören Methoden und Attribute zu jeder Klasseninstanz, also zu jedem Objekt. (Zwar wird zur Optimierung der Programmgrösse der Code der Methoden nur einmal in das ausführbare Programm aufgenommen und die Zugehörigkeit zur einzelnen Instanz über einen "versteckten" this-Zeiger geregelt. Davon braucht aber der Programmierer meist nichts wissen.) Es gibt aber durchaus Fälle, wo es notwendig ist, dass eine Methode oder ein Attribut zur ganzen Klasse und nicht zum einzelnen Objekt gehört. In diesem Fall deklariert man die Methode oder das Attribut als *static*. Im folgenden Beispiel wird dies, im Sinne der exemplarischen Methodik, erläutert.

Dazu eignet sich wieder einmal eine Klasse von Turtles hervorragend. Falls man diese zu einer "Familie" zusammenfassen will, so ist die Anzahl der Familienmitglieder eine wichtige Grösse. Darum leiten wir aus der Klasse *Turtle* die Klasse *TurtleChild* ab. Diese soll einen Zähler *counter* für die Anzahl Kinder enthalten, über den man die Kinderzahl abfragen kann. (Wie es sich im Sinn der Datenkapselung gehört, ist der counter von Aussen nicht sichtbar, also *private*.) Offensichtlich darf nicht jedes TurtleChild einen eigenen counter besitzen, daher wird dieser *static* deklariert.

Man erkennt am Beispiel die Eleganz der OOP: Im Konstruktor wird der counter erhöht, im Destruktor erniedrigt. Der Destruktor löscht auch gleichzeitig die Figur des TurtleChilds auf dem Bildschirm. Falls man an Stelle von TurtleChild-Objekten Zeiger auf solche einführt, lässt sich die Konstruktion (mit new) und Destruktion (mit delete) elegant steuern.

```
// STATICEX1.CPP

#define OBJECT_TURTLE
#include <champ.h>

// ==================== Class TurtleChild interface =================
class TurtleChild : public Turtle
{
public:
   TurtleChild ();
   ~TurtleChild ();
   static int howMany ();

private:
   static int counter;
};

// ==================== Class TurtleChild implementaion =============
TurtleChild::TurtleChild ()
{
   counter++;
   setPos( 30*counter, 0 );
}

TurtleChild::~TurtleChild ()
{
   counter--;
}

// --------------- Static methods ---------------------------------
int TurtleChild::howMany ()
{
   return counter;
}

// --------------- Static attributes definition/initialization -----
int TurtleChild::counter = 0;

// ==================== Main ========================================
void gmain ()
```

```
{
   ginit( "StaticEx1" );

   TurtleChild laura;
   TurtleChild mary;
   TurtleChild peter;

   CP::msgBox() << "Number of children: " << laura.howMany() << endl
                << "Press OK to let John appear.";

   TurtleChild * pJohn = new TurtleChild;

   CP::msgBox() << "Number of children now: " << laura.howMany() << endl
                << "Press OK to let John vanish.";

   delete pJohn;

   CP::msgBox() << "Number of children now: " << laura.howMany() << endl
                << "Press OK to terminate.";

   gend();
}
```

Die Implementierung in Java ist ähnlich, falls man die von uns entwickelte Java-Turtleklasse verwendet. Diese ermöglicht wie in C++ mehrere Turtleinstanzen im gleichen Window. (Dieses wird beim Instanzieren der ersten Turtle automatisch erstellt.) Allerdings ist die Vernichtung des Turtleobjekts über den Destruktor nicht möglich, da Java keine Destruktoren kennt.

```
// StaticEx1.java

import turtle.*;

class TurtleChild extends Turtle {
  public TurtleChild ()
  {
     counter++;
     setPos( 30*counter, 0 );
   }

   public static int howMany ()
   {
      return counter;
   }

   private static int counter = 0;
}


public class StaticEx1 {
   public static void main(String[] args) {
      TurtleChild laura = new TurtleChild();
      TurtleChild mary = new TurtleChild();
      TurtleChild peter = new TurtleChild();
      System.out.println( laura.howMany() );
   }
}
```

Ein anderer Klassenentwurf würde darin bestehen, statt einer aus Turtle abgeleiteten Klasse eine Klasse *TurtleFamily* zu definieren, welche die Familienmitglieder, also auch die Kinder enthält. Als "Kinderstube" würde sich am besten eine verkettete Liste eignen, da diese beliebig erweiterbar ist. Meist ist aber die maximale Familiengrösse sowieso beschränkt und man kommt mit einem Array aus. Eine einfache Implementierung zeigt das folgende Beispiel. Es wird ein Array mit Zeigern auf die Familienmitglieder eingeführt, damit man mit new und delete die Instanzen erzeugen und vernichten kann. Dabei zeigt sich eine kleine Schwierigkeit: Die Arraygrösse sollte eine Konstante sein, die sicherlich zur Klasse gehört. Konstante Klassenattribute können aber in C++ im Gegensatz zu Java nicht bei der Deklaration initialisiert werden. (Da sie wie nichtkonstante Attribute Speicherplatz beanspruchen und damit ihr Wert über eine Zuweisung gesetzt wird, der zu Kompilationszeit nicht bekannt ist.) Eine globale Konstante ist wenig schön. Es bietet sich der folgender Workaround an: Man führt die Arraygrösse als *enum* ein, da enums bereits zur Kompilationszeit bekannt sind. Allerdings funktioniert dieser Trick nur mit Integer-Konstanten. (Andere konstante Klassenattribute müssen beim Aufruf des Konstruktors in der Initialisierungsliste initialisiert werden.)

```
// STATICEX2.CPP

#define OBJECT_TURTLE
#include <champ.h>

// const int maxIndex = 3;   // Global class constant (ugly)

// ================== Class TurtleFamily interface ================
class TurtleFamily
{
public:
   TurtleFamily ();
   int howMany ();
   bool addChild ();
   bool removeLastChild ();

private:
   // Maximum number of children: maxIndex + 1
   enum { maxIndex = 3 };    // Enum hack for integer class constants
   int index;
   Turtle * pChildren[maxIndex+1];
};

// ================== Class TurtleFamily implementaion ============
TurtleFamily::TurtleFamily ()
{
   index = 0;
}

int TurtleFamily::howMany ()
{
```

```
   return index;
}

bool TurtleFamily::addChild ()
{
   if ( index > maxIndex )
   {
      CP::msgBox( "Error in addChild" ) << "No place for more children";
      return false;
   }

   pChildren[index] = new Turtle;
   pChildren[index]->setPos( 30*index, 0 );
   index++;
   return true;
}

bool TurtleFamily::removeLastChild ()
{
   if ( index == 0 )
   {
      CP::msgBox( "Error in removeLastChild" ) << "No children left";
      return false;
   }

   index--;
   delete pChildren[index];
   return true;
}


// ================== Main =======================================
void gmain ()
{
   ginit( "StaticEx2" );

   TurtleFamily meyer;
   meyer.addChild();
   meyer.addChild();
   meyer.addChild();
   CP::msgBox() << "Number of children: " << meyer.howMany() << endl
                << "Press OK to let last child vanish.";

   meyer.removeLastChild();

   CP::msgBox() << "Number of children now: " << meyer.howMany() << endl
                << "Press OK to terminate.";

   // Cleanup
   for ( int i = 0; i < meyer.howMany(); i++ )
      meyer.removeLastChild();


   gend();
}
```

In C++ sind wird im Gegensatz zu Java gezwungen, am Ende des Programms die noch vorhandenen dynamischen Objekte (solche, die mit *new* erzeugt wurden) von Hand zu vernichten, damit der Speicherplatz sicher freigegeben wird. Zur einfacheren Fehlerbehandlung geben die

Methoden einen Bool zurück, mit dem geprüft werden kann, ob die Operation erfolgreich war (was hier nicht ausgenützt wird).

Viel eleganter ist es, die Kinder in eine verkettete Liste aufzunehmen, die dynamisch verwaltet werden kann. Man gewinnt zwei Vorteile: die Zahl der Kinder kann beliebig wachsen und man kann leicht auch Objekte entfernen, die sich an irgendeiner Stelle der Liste befinden. Wie bereits in den vorhergehenden Beispielen ist es besser, an Stelle einer Liste mit den Kindern selbst eine Liste mit Zeigern auf die Kinder zu führen. Damit lässt sich das Entstehen und Vernichten der Objekte leichter steuern. In Java gibt es für Objekte gar keine Wahl, da die Variablen immer (deferenzierte) Zeiger (genannt *Referenzen*) sind.

Die Implementierung von Listen ist bekanntlich ein interessantes Lernobjekt; es ist aber nicht sinnvoll, dieser Arbeit immer und immer wieder zu leisten. In C++ gibt es bekanntlich für die wichtigsten Datenstrukturen und Algorithmen vorgefertigte Modelle, genannt *Templates*. Diese befinden sich in der Standard Template Library (STL). Für unseren Fall ist es einfacher, das Listentemplate *cplist* aus der Champ-Library zu verwenden. Das Programmlisting zeigt, wie elegant diese Lösung ist.

```
// STATICEX3.CPP

#define OBJECT_TURTLE
#include <champ.h>


// ================== Class TurtleFamily interface =================
class TurtleFamily
{
public:
   TurtleFamily ();
   int howMany ();
   void addChild ();
   bool removeChild ( int nb );
   void cleanup ();


private:
   CPList<Turtle*> children;
   int nbChildren;
};

// ================== Class TurtleFamily implementaion =============
TurtleFamily::TurtleFamily ()
{
   nbChildren = 0;
}

int TurtleFamily::howMany ()
{
   return nbChildren;
```

```
}

void TurtleFamily::addChild ()
{
   Turtle * pChild = new Turtle;
   pChild->setPos( 30*nbChildren, 0 );
   children.insertTail( pChild );
   nbChildren++;
}

bool TurtleFamily::removeChild ( int nb )
{
   if ( nb < 0 || nb > nbChildren-1 )
   {
      CP::msgBox( "Error in removeChild" ) << "Child's index not valuable";
      return false;
   }

   children.restartAtHead();
   for ( int i = 0; i < nb; i++ )
      children.next();
   delete children.current();   // Delete object
   children.removeCurrent();    // Remove it from list
   nbChildren--;
   return true;
}

void TurtleFamily::cleanup ()
{
    for ( children.restartAtHead(); children; children.next() )
      delete children.current();
}

// ================== Main ========================================
void gmain ()
{
   ginit( "StaticEx3" );

   TurtleFamily meyer;
   for ( int i = 0; i < 4; i++ )
      meyer.addChild();

   CP::msgBox() << "Number of children: " << meyer.howMany() << endl
                << "Press OK to let child # 1 child vanish.";

   meyer.removeChild( 1 );

   CP::msgBox() << "Number of children now: " << meyer.howMany() << endl
                << "Press OK to terminate.";

   meyer.cleanup();
   gend();
}
```

P72:

Statische Methoden und Attribute lassen sich an Stelle von globalen Funktionen und Variablen einsetzen. Wann ist dies angebracht?

L72:

Die Technik stammt von Java, und ist dort weit verbreitet, da die Sprache keine globalen Funktionen und Variablen kennt. Der Trick (in einem gewissen Sinn ist es ein solcher) besteht darin, eine Klasse zu definieren, welche ausschliesslich public und static Methoden und Variablen besitzt, beispielsweise in Java die Klasse Math mit dem Methoden sin(), usw. und den Attributen PI, usw. Statische Methoden und Attribute kann man mit dem Punktoperator entweder unter Angabe eines Instanznamens oder aber mit dem Klassennamen allein verwenden. Letzteres macht gerade hier Sinn, indem man keine Klasseninstanz erzeugt, sondern lediglich Math.sin(), bzw. Math.PI schreibt.

In C++ kann dieser Trick dann Sinn machen, falls man globale Grössen verwenden will, ohne dass es zu einem Namenskonflikt kommt. Eleganter ist allerdings der Einsatz von *namespace*.

P73:
In gewissen Klassenentwürfen kann es sinnvoll sein, zu verhindern, dass irrtümlicherweise von einer Klasse eine Instanz erzeugt wird. Wie kann man dies erreichen?

L73:
Im Problem P62 (Bulletin Nr. 35) habe ich gezeigt, dass man mit einem privaten (Copy-)Konstruktor verhindern kann, dass eine Klasse instanziert wird. Der angegebene Konstruktor besass einen leeren Körper, d.h. bestand nur aus zwei Klammern {}. Unser wohlbekannte C++-Guru und Entwickler von Champ, Hans Salvisberg, hat mich freundlicherweise darauf aufmerksam gemacht, dass es besser ist, den Konstruktor zu *deklarieren*, ohne ihn zu definieren, d.h. die Klammern wegzulassen. Damit verhindert man zusätzlich, dass in einer Methode der Klasse oder von einem allfälligen Friend eine Instanz erzeugt wird.

Wie üblich befinden sich alle Sourcen auf *www.clab.unibe.ch/champ*