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

P56:
Die objektorientierte Programmierung hat sich in letzter Zeit auch an Ausbildungsstätten etabliert. Ein Grund dafür ist wohl, weil Java und C++ als hauptsächliche Programmiersprachen in der Ausbildung gelten. OOP basiert ganz wesentlich auf Klassen, Klassenableitungen und Polymorphie. Letzteres stellt allerdings für die Informatikdidaktik eine Herausforderung dar, und man ist immer wieder froh, sinnvolle, d.h. praxisrelevante Demonstrationsbeispiele kennen zu lernen. Hier ein Problem, sich vor allem bei der Erstellung und Anwendung von Klassenbibliotheken in Java und C++ stellt und von der Polymorphie wesentlich Gebrauch macht.

Es gibt Situationen, in welchen man gerne eine "generische" Klasse verwenden möchte, deren Methoden grundsätzlich automatisch aufgerufen werden, wobei man es aber einem "Anwender" der Klasse überlassen möchte, einige davon gelegentlich verändern möchte.

Ich denke dabei etwa an eine Servlet-Klasse, wo die meisten "Default"-Methoden bei jedem Zugriff auf das Servlet, d.h. bei jedem HTTP-Request von einem Webbrowser, automatisch ausgeführt werden. Man möchte nun jeweils nur diejenigen Methoden überschreiben, die man auch tatsächlich verändern will. (Die Implementierung entspricht der Java-Servlet-Klasse *HttpServlet*.)

L56:
Die Implementierung macht davon Gebrauch, Methoden **virtual** zu definieren und verwendet die Polymorphie, sowie eine Liste (d.h. einen Container) mit Zeigern. Sie ist darum ein Musterbeispiel für die Eleganz der objektorientierten Programmiertechnik.

Das folgende Beispiel soll das Vorgehen illustrieren. Die Basisobjekte stammen aus der Klasse *Bird*. Man möchte nun durch Klassenableitung gewisse Methoden, etwa *whistle* in der Klasse *Sparrow* ersetzen, d.h. überschreiben. Dazu erstellt man in *Handler* eine Liste mit Zeiger *Bird** auf die Klasse *Bird*. In dieser lassen sich aber auch Zeiger *Sparrow** auf *Sparrow* speichern. (Man nennt dies "upcasting".) Beim Aufruf von *whistle* über einen solchen Zeiger, "erinnern" sich diese sozusagen, auf welchen Objekttyp sie zeigen (genau dies ist der springende Punkt der Polymorphie). Der Konstruktor von *Bird* fügt die Zeiger in diese Liste ein.

Die Methode *go* des Handlerobjekts führt für alle eingefügten Objektzeiger die Methode *whistle* aus. Man vergewissere sich, dass beim Weglassen des Schlüsselwortes *virtual*, die Methode *whistle* von *Bird* statt derjenigen von *Sparrow* aufgerufen wird. Man beachte auch die Eleganz der Template-Klasse *CPList.* (Man könnte an ihrer Stelle auch den Listencontainer aus der Standard Template Library (STL) benützen. Siehe den Artikel "Vom frust- zum lustvollen Programmieren mit wiederverwendbaren Komponenten" in diesem Bulletin.)

Das Vorgehen ist in Java analog, allerdings sind hier sowieso alle Methoden virtuell und alle Objektvariablen Referenzen (d.h. derefernzierte Zeiger). Allerdings gibt es hier leider keine Template-Klassen, so dass die Liste etwas schwieriger zu handhaben ist.

```
// BIRDS.CPP
#include <champ.h>

// -------------  Interface normally in BIRDS.H
class Bird
{
public:
   Bird ();
   virtual void whistle ();   // Must be virtual
};

class Handler
{
public:
   static void go ();
   static CPList<Bird *> list;
};
// ------------- End of interface

CPList<Bird *> Handler::list;

void Handler::go ()
{

  for ( list.restartAtHead();  // Start with the first element
        list;                  // Loop while list is ok
        list.next() )          // Advance to next element
   {
      list.current()->whistle();
   }

}

Bird::Bird ()
{
   Handler::list.insert( this );
}

void Bird::whistle ()
{
   cout << "sheweeee......";
}

//  Sparrow may or may not override method whistle (outcommented here)
class Sparrow : public Bird
{
//   virtual void whistle();
};
```

```
/*
void Sparrow::whistle ()
{
   cout << "zwaawaa.......";
}
*/

void gmain ()
{
   cinit();
   Handler handler;
   Sparrow sparrow;
   handler.go();
}
```