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

Es gibt viele Gründe, warum man die Objektorientierte Programmierung (OOP) frühzeitig im Informatikunterricht einführen sollte. Ich stelle fest:

- Die OOP hat sich in den letzten 20 Jahren bewährt und ist ein Muss für praktisch alle neuen Informatikprojekte
- Die OOP ist ein langfristig gesichertes Konzept, das zum Grundwissen der Informatik zählt
- Die OOP ist zentraler Bestandteil aller schulrelevanten Programmiersprachen (C++, Pascal/Delphi, Java, in Java gibt es nur OOP).
- Die OOP ist auch für einfache, kurze Programme sinnvoll, ähnlich wie die Strukturierte Programmierung
- Die OOP macht der Lehrperson Spass, da sie eine neue didaktische Herausforderung darstellt (weg vom Basic-Stil der 60er Jahre!)

Aus diesen Gründen (und weil ich in den letzten Jahren viele meiner didaktischen Anstrengungen auf die Verbreitung der OOP fokussierte) soll in Zukunft in dieser Kolumne davon ausgegangen werden, dass unsere Leserinnen und Leser wenigsten mit den Grundlagen der OOP vertraut sind. Da viele Lehrpersonen neben C++ auch Java unterrichten oder wenigstens an Java interessiert sind, wird in Zukunft, wo es sinnvoll erscheint, ein Quervergleich gemacht. Dieser soll dem besseren Verständnis der einen **und** anderen Sprache dienen und das gemeinsame friedliche Nebeneinander der sehr ähnlichen Sprachen fördern. Es ist sattsam bekannt, dass sich die Vor- und Nachteile der beiden Sprachen und ihre Bedeutung in der Welt etwa die Waage halten, wobei der Einzelne seine Vorliebe durchaus vertreten darf. Jede Polemik ist aber wenig konstruktiv.

P62:
Manchmal wird behauptet, die OOP eigne sich nur für grosse Informatikprojekte und sei beispielsweise für mathematisch/naturwissenschaftliche Problemlösungen nicht nötig oder gar ungeeignet. Man zeige das Gegenteil an Hand einer typischen Aufgabe aus einem wichtigen Gebiet des Unterrichts in Angewandter Mathematik (Populationsdynamik). Gleichzeitig sollen wichtige Probleme im Zusammenhang mit dem Variablenkonzept aufgezeigt werden.

L62:
Es ist vernünftig, eine Population als eine Klasse zu definieren, damit alle Attribute und Methoden, welche für die Population wichtig sind, zusammengefasst werden. Dazu gehört insbesondere ein Array, welcher die einzelnen Individuum enthält. (Ein Individuum könnte im allgemeinen

wieder ein Objekt aus einer Individuumsklasse sein, auch der Array ist nicht zwingend, eine andere Datenstruktur (vector, list, usw.) wäre durchaus denkbar.) Im Folgenden wird ein einzelnes Individuum nur dadurch charakterisiert, dass es (neben der Anordnung im Array) ein boolesches Attribut besitzt, ob es am Leben oder tot ist.

Wie jede Lehrkraft weiss, ist bereits für den Programmieranfänger ein solides Verständnis des Variablenbegriffs von grosser Wichtigkeit. (Man erinnert sich an Analogien mit "Schuhschachteln", "Schubladen", usw.) In C/C++ ist auch für Objekte eine Auffassung in klassischer Form möglich, in Java hingegen kommt ein völlig neues, für den Anfänger recht schwieriges Konzept zur Anwendung: Für die primitiven Datentypen gibt es (insbesondere aus Effizienzgründen) Variable in klassischem Sinn, für Objekte verhalten sich Variable aber ganz anders. Man nennt sie darum "Referenzen" bzw. "Handles", denn es handelt sich eigentlich um Zeiger (die automatisch derefernziert sind. Die Zuweisung a = b kopiert also für Objekte die Referenzen. Damit kann es leicht zu ungewollten Seiteneffekten (Aliasing) kommen: Verändert man ein Attribut von a, so wird dasjenige von b auch verändert (da beide Referenzen auf das gleiche Objekt "zeigen").

Für die Begriffsbildung ist es wichtig zu verstehen, was beim Kopieren von Variablen vor sich geht. Dabei unterscheidet man gewöhnlich zwei Arten, eine Variable (bzw. ein Objekt) zu kopieren: Bei der Instanzierung (bzw. bei der Parameterübergabe an eine Funktion oder Funktionsrückgabe) und bei der Zuweisung. In C++ wird der erste Fall durch den Kopierkonstruktor (copy constructor) behandelt, der zweite durch den Zuweisungsoperator (assignment operator). Falls diese nicht explizit definiert sind, kommen ihre Defaults zur Anwendung. In POP1 wird gezeigt, dass diese default Operationen das ganze Objekt (bitweise) kopieren, d.h. auch die Werte des Arrays.

```
// POP1.CPP
// Use of OOP will pack all attributes and methods of a population together

#include <champ.h>
#include <stdlib.h>

#define POPSIZE 10
#define dead false
#define alive true

// --- Class Population : Interface ---
class Population
{
public:
   Population ();
   void fillDead ();
   void fillAlive ();
```

```
   void fillRandom ();
   void show ();

private:
   bool _pop[POPSIZE];
};

// --- Class Population : Implementation ---
Population::Population ()
{
   fillAlive();
}

void Population::fillRandom ()
{
   randomize();
   for ( int i = 0; i < POPSIZE; i++ )
      _pop[i] = (bool)random( 2 );
}

void Population::fillDead ()
{
   for ( int i = 0; i < POPSIZE; i++ )
      _pop[i] = dead;
}

void Population::fillAlive ()
{
   for ( int i = 0; i < POPSIZE; i++ )
      _pop[i] = alive;
}

void Population::show ()
{
   for ( int i = 0; i < POPSIZE; i++ )
      cout << i << " : " << ( _pop[i] == alive ? "alive" : "dead" )
           << endl;
   cout << endl;
}
// --- Main ---
void gmain ()
{
   cinit( "POP1" );
   Population p1;
   p1.fillRandom();
   p1.show();

   Population p2 = p1;    // Default (build-in) copy constructor used
   p2.show();

   p2.fillDead();         // Set only p2 dead
   p1.show();             // p1 remains the same

   Population p3;
   p3 = p1;               // Default (build-in) assignment operator used
   p3.fillAlive();        // Set p3 alive
   p1.show();             // p1 remains the same
}
```

Es stellt sich die interessante Frage, was in der gleichen Situation mit einem Klassenattribut, das selbst ein vollwertiges Objekt ist, geschieht.

Um dies zu demonstrieren, definieren wir eine (altbekannte) Klasse Point und fügen einen Punkt zu unserer Population (sozusagen ihr Aufenthaltsort). C++ verlangt nun, dass wir im Konstruktor von Population auch Point instanzieren. Damit ist gewährleistet, dass Point immer einen definierten Zustand besitzt, d.h. die Koordinaten einen definierten Wert haben. POP1A zeigt, dass beim Kopieren eines Objekts von Population auch die darin enthaltenen Objekte **gleichartig** kopiert werden. Das Objekt wird beim Kopieren vollständig neu erzeugt, es ist ein **perfekter Klone**. Man nennt dies in der Terminologie von Java auch "deep copy". Java kopiert aber im Gegensatz zu C++ bei der Zuweisung von Objekten a = b fast nichts (nur die Referenz, die auf dasselbe Objekt weist). Will man wenigsten die Attribute (primitive Datentypen und Referenzen) mitkopieren, so muss in Java die Methode clone verwendet werden, die ein neues Objekt erzeugt. Dessen Referenzen zeigen aber in der Regel immer noch auf die Objekte des geklonten Objekts (sog. "shallow copy").

```
// POP1A.CPP
// If an object A has a data member B, which is an object,
// B must be initialized by the constructor of A

// Whenever the copy constructor or assignment operator of A is
// called, the same happens to B.
// Therefore whenever A is copied, we get a perfect clone,
// e.g.  C++ performs a "deep copy"
...
class Point
{
public:
   // Constructor
   Point ( double x, double y );
   // Accessors
   double getX () { return _x; }
   double getY () { return _y; }
   void setX ( double x ) { _x = x; }
   void setY ( double y ) { _y = y; }

private:
   double _x;
   double _y;

};

Point::Point ( double x, double y )
{
   cout << "Constructor of Point executing..." <<  endl;
   _x = x;
   _y = y;
}


class Population
{
public:
   Population ();
```

```
   void fillDead ();
   void fillAlive ();
   void fillRandom ();
   void show ();

   Point _point;          // Object data member

private:
   bool _pop[POPSIZE];
};

Population::Population ()
   : _point( 0, 0 )       // All object data members must be initialized or
{                         // compile time error results
   fillAlive();
}

void Population::fillRandom ()
...
void Population::fillDead ()
...
void Population::fillAlive ()
...
void Population::show ()
{
   for ( int i = 0; i < POPSIZE; i++ )
      cout << i << " : " << ( _pop[i] == alive ? "alive" : "dead" )
           << endl;
   cout << "Point( " << _point.getX() << "," << _point.getY() << " )"
        << endl << endl;
}

void gmain ()
{
   cinit( "POP1A" );
   Population p1;
   p1.fillRandom();
   p1.show();

   p1._point.setX( 1 );
   Population p2 = p1;   // Copy constructor of Population AND Point called
   p2.show();            // Values of all data members copied form p1 to p2

   Population p3;
   p3 = p1;              // operator= of Population AND Point called
   p3.show();            // Values of all data members copied form p1 to p3

}
```

Was gefällt Ihnen an der Implementierung der Klasse "Population" überhaupt nicht? Richtig: die Popuplationsgrösse muss zu Kompilationszeit angegeben werden. Viel besser der Realität angepasst wäre aber eine dynamische Populationsgrösse, d.h. eine, die erst zu Laufzeit angegeben (und damit auch verändert) werden kann. Dynamischen Speicherplatz (auf dem Heap) reserviert man in C++ mit **new** und gibt ihn mit **delete** wieder frei. (In Java gibt es nur dynamische Objekte und die explizite Freigabe ist wegen des automatischen Garbage-Kollektors überflüssig.) POP2 zeigt, wie dies funktioniert, aber

auch, dass jetzt beim Kopieren von Objekten nur der Zeiger auf den dynamischen Array (und nicht dessen Inhalte) kopiert werden (entspricht dem oben erwähnten "shallow copy"). Der neu eingeführte Destructor sorgt für das "Entsorgen", d.h. für die Freigabe des mit new reservierten Speicherplatzes.

```
// POP2
// Copy object will not copy dynamic array

#include <champ.h>
#include <stdlib.h>

#define dead false
#define alive true

class Population
{
public:
   Population ( int size );
   ~Population () { delete [] _pPop; }
   void fillDead ();
   void fillAlive ();
   void fillRandom ();
   void show ();

private:
   int _size;
   bool * _pPop;
};

Population::Population ( int size )
   : _size( size )
{
   _pPop = new bool[_size];
   fillAlive();
}

void Population::fillRandom ()
{
   randomize();
   for ( int i = 0; i < _size; i++ )
      _pPop[i] = (bool)random( 2 );
}

void Population::fillDead ()
{
   for ( int i = 0; i < _size; i++ )
      _pPop[i] = dead;
}

void Population::fillAlive ()
{
   for ( int i = 0; i < _size; i++ )
      _pPop[i] = alive;
}

void Population::show ()
{
   for ( int i = 0; i < _size; i++ )
      cout << i << " : " << ( _pPop[i] == alive ? "alive" : "dead" )
```

```
            << endl;
   cout << endl;
}

void gmain ()
{
   cinit( "POP2" );
   Population p1( 10 );
   p1.fillRandom();
   p1.show();

   Population p2 = p1;         // Default copy constructor used
   p2.show();

   p2.fillDead();              // Will also set p1 dead
   p1.show();                  // Here you see it

   Population p3( 10 );
   p3 = p1;                   // Default assignment operator used

   p3.fillAlive();            // Will also set s1 alive
   p1.show();                 // Here you see it
}
```

Die vorhergehende Klassendefinition ist in Bezug auf das Kopieren denkbar schlecht und führt zu gefährlichen Seiteneffekten. Man sollte sie entweder verbessern oder das Kopieren verbieten. Beides ist möglich, zuerst verbessern wir sie, indem wir den Kopierkonstruktor und den Zuweisungsoperator selbst neu definieren (In Java können weder der Kopierkonstruktor noch Operatoren frei definiert werden, immerhin könnte man die clone-Methode entsprechend programmieren.)

```
// POP3.CPP
// Copy constructor and assignment operator defined
...

class Population
{
public:
   Population ( int size );          // Ctor
   Population ( const Population & p );   // Copy ctor
   Population & operator= ( const Population & p );    // Assignment op
   void fillDead ();
   void fillAlive ();
   void fillRandom ();
   void show ();

private:
   int _size;
   bool * _pPop;
};

Population::Population ( int size )
   : _size( size )
{
   if ( size < 1 )
   {
      CP::msgBox( "Fatal Error" )
```

```
        << "Illegal parameter in constructor of class Population";
      exit( 1 );
   }
   _pPop = new bool[_size];
   fillAlive();
}

Population::Population ( const Population & p )
{
   cout << "Copy constructor executing..." << endl;
   _size = p._size;
   _pPop = new bool[_size];
   for ( int i = 0; i < _size; i++ )
      _pPop[i] = p._pPop[i];
}

Population & Population::operator= ( const Population & p )
{
   cout << "Assignment operator executing..." << endl;
   _size = p._size;
   delete [] _pPop;
   _pPop = new bool[_size];
   for ( int i = 0; i < _size; i++ )
      _pPop[i] = p._pPop[i];
   return *this;
}

void Population::fillRandom ()
...
void Population::fillDead ()
...
void Population::fillAlive ()
...
void Population::show ()
...

void gmain ()
{
   cinit( "POP3" );
   Population p1( 10 );
   p1.fillRandom();
   p1.show();

   Population p2 = p1;                // Copy constructor used
   p2.show();

   p2.fillDead();
   p1.show();

   Population p3( 2 );
   // Size not important because space will be released by following =
   p3 = p1;                          // Assignment operator used
   // Same as p3.operator=( p1 );

   p3.fillAlive();
   p1.show();
}
```

Es gehört zum guten Programmierstil, das Kopieren zu verhindern, falls dieses wegen dem "shallow copy" zu schwer auffindbaren Programmfehlern führen kann. Dazu deklariert man den

Kopierkonstruktor und den Zuweisungsoperator **private**. Es ist nicht nötig, eine Implementierung anzugeben.

```
// POP4.CPP
// Private copy ctor and assignment op will inhibit dangerous operations
...
class Population
{
public:
   Population ( int size );
   ~Population () { delete [] _pPop; }
   void fillDead ();
   void fillAlive ();
   void fillRandom ();
   void show ();

private:
   Population ( const Population & p );
   Population & operator= ( const Population & p );
   int _size;
   bool * _pPop;
};
```

P63:

Alle kennen das wichtige Grundprinzip der Programmierung: "**Keine Codeduplizierung**". Dies betrifft natürlich auch Programmteile mit "**fast**" gleichem Code. Die Gründe brauchen hier nicht erneut dargelegt zu werden. Manchmal kommt man aber doch in Versuchung zu sündigen: man hat üblicherweise gleiche Algorithmen für verschiedene Datentypen zu formulieren (klassisches Beispiel: sortieren). Auch der Mathematiker, den ich hier anspreche, kennt die Problematik: Viele Algorithmen betreffen Funktionen (gewöhnlich mit einer Variablen), z.B. Nullstellen-bestimmung, Integration, Extremwertbestimmung, Graphische Darstellung, usw.). Man hat die Absicht, den Algorithmus nur einmal zu schreiben und die Funktion als Parameter zu übergeben. Besitzen die betrachteten Funktionen denselben Typ, beispielsweise double f(double x), so löst man bekanntlich in C/C++ das Problem elegant durch Uebergabe eines **Funktionszeigers** pF, indem man folgenden Typ einführt:

```
typedef double (*pF)( double x )
```

Oft besitzen die Funktionen aber unterschiedlich viele Parameter. Man müsste nun viele verschiedene Typen einführen

```
typedef double (*pF1)( double x, double a )
typedef double (*pF2)( double x, double a, double b )
```

könnte diese aber, da sie unterschiedlichen Typ besitzen, nicht derselben Algorithmusfunktion übergeben (auch das Ueberladen hilft nicht weiter). Wie kann man das Problem elegant lösen? (Der

nachfolgende Lösungsvorschlag stammt von Qiang Liu, Implementing Reusable Mathematical Procedures Using C++, C/C++ Users Journal, June 2001.)

L63:

Die beschriebene Problematik führt zu einem anderen Paradigma der modernen Informatik, der Entwicklung von "**Generischen Algorithmen"**. Darunter versteht man Rechenverfahren, die für eine weite Klasse von Datentypen gültig sind. Das Verfahren ist aus der Mathematik und den exakten Naturwissenschaften wohlbekannt: Man ist bestrebt, Theorien so zu formulieren, dass sie für eine breite Klasse von Problemen anwendbar sind. Für die Formulierung von Generischen Algorithmen stellt C++ ein elegantes, mächtiges Sprachkonstrukt zur Verfügung: die **Templates**.

Im Folgenden soll ein klassisches Problem des Anfängerunterrichts gelöst werden: Die Darstellung einer Funktion in einem Graphikfenster. Nach demselben Prinzip könnten (sollten!) Nullstellenbestimmungen, usw. behandelt werden.

Zur Demonstration packen wird das Problem vorerst altertümlich an. Der hauptsächliche Nachteil dieser Lösung besteht darin, dass das Programm mit globalen Grössen, nämlich den Funktionsparametern, belastet wird, die eigentlich nur zur Funktion selbst gehören. Da andere Funktionen andere Parameter verlangen, führt dieses Vorgehen sehr schnell ins Chaos, insbesondere auch, weil das Hauptprogramm zu viele Einzelheiten (Anpassen der Fensterkoordinaten, usw.) enthält.

```
// FUNC1.CPP

#include <champ.h>

// Normalverteilung
double s = 1;
double u = 2;

double f( double x )
{
   return 1 / ( sqrt( 2*M_PI )*s ) * exp( - (x - u)*(x - u) / ( 2*s*s ) ) ;
}

void gmain ()
{
   ginit( "FUNC1" );

   double step = 0.01;
   double xmin = -10;
   double xmax = 10;
   double x;
   double ymin = f( xmin );
   double ymax = ymin;

   // Anpassen der Fensterkoordinaten
   for ( x = xmin; x <= xmax; x += step )
```

```
   {
     if ( ymin > f( x ) )
        ymin = f( x );
     if ( ymax < f( x ) )
        ymax = f( x );
   }
   gwindow( xmin, xmax, ymin, ymax );

   gpos( xmin, f( xmin ) );
   for ( x = xmin; x <= xmax; x += step )
      gdraw( x, f( x ) );
}
```

Als nächstes wählen wir einen objektorientierten Ansatz, der für den unvoreingenommenen Schüler oft natürlicher erscheint als für den in alten Mustern denkenden Lehrer. Dazu führen wir Funktionen als Objekte ein und geben ihnen damit die Eigenständigkeit, die sie verdienen. Die Parameter gehören nun als Attribute zum Funktionsobjekt. Durch das Ueberladen des Klammeroperators erreichen wir eine Verwendung in völliger Uebereinstimmung mit der bekannten Notation y = f(x)!

```
// FUNC2.CPP

#include <champ.h>

class Normalverteilung
{
public:
   Normalverteilung ( double s, double u )
      : _s( s ), _u( u )
   {}

   double operator () ( double x )
   {
      return
       1 / (sqrt( 2*M_PI)*_s ) * exp( - (x - _u)*(x - _u) / ( 2*_s*_s ) ) ;
   }

private:
   double _s;
   double _u;
};

void gmain ()
{
   ginit( "FUNC2" );

   double s = 1;
   double u = 2;
   Normalverteilung f( s, u );

   double step = 0.01;
   double xmin = -10;
   double xmax = 10;
   double x;
   double ymin = f( xmin );
   double ymax = ymin;

   // Anpassen der Fensterkoordinaten
   for ( x = xmin; x <= xmax; x += step )
   {
     if ( ymin > f( x ) )
        ymin = f( x );
     if ( ymax < f( x ) )
```

```
         ymax = f( x );
   }
   gwindow( xmin, xmax, ymin, ymax );

   gpos( xmin, f( xmin ) );
   for ( x = xmin; x <= xmax; x += step )
      gdraw( x, f( x ) );
}
```

Als nächstes definieren wird showFunc als Template und erkennen sofort den Vorteil des Generischen Programmierens: showFunc kann für Funktionen mit unterschiedlicher Zahl und Typ der Parameter verwendet werden. Das Hauptprogramm wird nun endlich auch im Sinn der Strukturierten Programmierung klar und übersichtlich.

```
// FUNC3.CPP

#include <champ.h>

template<class T>
   void showFunc( T & pF, double xmin, double xmax, double step )
{
   double ymax = pF( xmin );
   double ymin = ymax;
   double x;

   // Anpassen der Fensterkoordinaten
   for ( x = xmin; x <= xmax; x += step )
   {
     if ( ymin > pF( x ) )
        ymin = pF( x );
     if ( ymax < pF( x ) )
        ymax = pF( x );
   }
   gwindow( xmin, xmax, ymin, ymax );

   gpos( xmin, pF( xmin ) );
   for ( x = xmin; x <= xmax; x += step )
      gdraw( x, pF( x ) );
}

class Normalverteilung
{
public:
   Normalverteilung ( double s, double u )
      : _s( s ), _u( u )
   {}

   double operator () ( double x )
   {
      return
       1 / (sqrt( 2*M_PI )*_s) * exp( - (x - _u)*(x - _u) / ( 2*_s*_s ) ) ;
   }

   void show ( double xmin, double xmax )
   {
      showFunc( *this, xmin, xmax, 0.01 );
   }

private:
   double _s;
   double _u;
};


void gmain ()
{
```

```
   ginit( "FUNC3" );

   double s = 1;
   double u = 2;
   Normalverteilung f( s, u );

   double xmin = -10;
   double xmax = 10;
   f.show( xmin, xmax );
}
```

Der gleiche Code funktioniert nun auch für die Funktionsklasse Schwingung.

```
// FUNC4.CPP
...
template<class T>
   void showFunc( T & pF, double xmin, double xmax, double step )
...

class Normalverteilung
...

class Schwingung
{
public:
   Schwingung ( double a, double b, double w )
      : _a( a ), _b( b ), _w( w )
   {}

   double operator () ( double x )
   {
      return _a * exp( -_b*x ) * sin( _w* x );
   }

   void show ( double xmin, double xmax )
   {
      showFunc( *this, xmin, xmax, 0.01 );
   }

private:
   double _a;
   double _b;
   double _w;
};


void gmain ()
{
   ginit( "FUNC4" );

   Normalverteilung f( 1, 2 );
   f.show( -10, 10 );

   Schwingung g( 2, 0.1, 4 );
   g.show( -10, 10 );

}
```

Spezialfälle von Funktionen können sehr elegant als abgeleitete Funktionsobjekte eingeführt werden.

```
// FUNC5.CPP
...

template<class T>
```

```
   void showFunc( T & pF, double xmin, double xmax, double step )
class Normalverteilung
...
class Schwingung
...

class SinusSchwingung : public Schwingung
{
public:
   SinusSchwingung ( double a, double w )
      : Schwingung( a, 0, w )
   {}
};

void gmain ()
{
   ginit( "FUNC5" );

   Normalverteilung f( 1, 2 );
   f.show( -10, 10 );

   SinusSchwingung g( 2, 4 );
   g.show( -10, 10 );
}
```

Ein solch ästhetisches Programm muss doch auch den Mathematikern, die für den Hang zur Vollkommenheit bekannt sind, gefallen.

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