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

Einmal mehr widmen wir uns heute dem graphischen Benützerinterface (Graphics User Interface, GUI). Zwar habe ich immer den Standpunkt vertreten, dass in der Programmierausbildung der Schwerpunkt auf der Entwicklung der Grundkonzepte (Algorithmen, Programm- und Datenstrukturen, OOP, usw.) liegen müsse und das Programmieren des GUI sekundär sei und in Informatik-Grundkursen nur zu einer Verzettelung und Zeitverschwendung führe. Allerdings wünschen gerade auch motivierte Schüler, dass ein Programm nicht nur algorithmisch korrekt sei, sondern dass es auch eine moderne, ansprechende Benützeroberfläche aufweise. Es ist auch bekannt, dass man mit graphisch ansprechenden Programmen, die ein professionelles L&F (look and feel) aufweisen, die Motivation für den Informatikunterricht steigern kann. Aus diesem Grund sind Programmiersprachen bzw. Entwicklungssysteme, mit denen man die Benützeroberfläche ohne grossen Aufwand gestalten kann, im Anfängerunterricht sehr beliebt (VisualBasic, Delphi, JBuilder, C++Builder, C++/Champ).

Die Erstellung einer Benützeroberfläche, die sich in jedem Fall für den Anwender "vernünftig" verhält, ist allerdings keineswegs trivial. Man stelle sich (oder den Schülern) einmal (in irgendeiner bekannten Programmierumgebung) folgendes Problem, das gleichzeitig als Grundgerüst für viele Applikationen dienen kann:

P64:
Eine Applikation soll die Quadratwurzeln der Integer-Werte von 0 bis 100 ausschreiben (stellvertretend für irgendeinen Algorithmus). Das GUI bestehe aus einem Applikationsfenster mit folgenden Elementen:

1) Ausgabefeld mit einem vertikalen Scrollbalken.
2) 3 Buttons *Start*, *Stop*, *Resume*, um die Berechnung zu starten, in irgendeinem Moment zu stoppen und mit *Resume* nachfolgend an der angehaltenen Stelle weiterzugefahren oder mit *Start* neu zu beginnen.
3) Einem Menü mit den Items *Exit*, um das Programm an irgendeiner Stelle zu beenden und *Info*, um einen Hilfetext (oder Copywrite-Text) anzuzeigen.

Die Buttons sollen nur dann aktiv sein, wenn es sinnvoll ist, sie drücken zu können. Sie sollen auch mit der Tastatur bedienbar sein.

L64:

Im Folgenden wird die Lösung mit C++/Champ gezeigt. Wir wollen OOP einsetzen, soweit es als sinnvoll erscheint. Wir gehen aber Top-Down vor und schreiben zuerst das Hauptprogramm. Wieder einmal konstruieren wir einen von mir so geliebten **Automaten** (State machine) mit den Zustandswerten als enum. Es ist kein Verbrechen, den Zustand als globale Variable zu definieren, handelt es sich doch um eine die ganze Applikation dominierende Grösse, die man sicher nicht leichtsinnig ändert.

```
// DLGAPP.CPP

#include <champ.h>
#include "panel.h"

enum State { idle, stopped, running, quitting } state = idle;

void gmain ()
{
   const int maxX = 100;
   int x = 0;
   int y;
   Panel pan( "Dialog Application", x );
   state = idle;

   // State event loop
   while ( state != quitting )
   {
      switch ( state )
      {
         case idle:
         case stopped:
            CP::yield();
            break;

         case running:
            CP::yield();
            y = sqrt( x );
            pan.showResult( x, y );
            if ( x == maxX )
            {
               x = 0;
               state = idle;
               pan.enableStart();
            }
            else
               x++;
            break;
      }
   }
   CP::msgBox() << "Garbage collection (if needed) here.";
}
```

Bemerkungen:

1) In der Eventloop wird zur Sicherheit regelmässig ein `CP::yield()` aufgerufen, damit Windows sicher nicht "hängt".

2) Das Panel-Objekt ist ein nicht modaler Dialog mit den geforderten Controls. Damit das Panel-Objekt den Initialisierungswert von x setzen kann, wird dem Objekt mit dem Konstruktor x als Variablenparameter übergeben und in ein Referenz-Attribut _x kopiert.
3) In C++ muss (im Gegensatz zu Java) der Abfall eigenhändig entsorgt werden (kein automatischer Garbage Collector). Falls nötig, kann dazu Code im Anschluss an die Eventloop eingefügt werden. Dies ist etwa dann nötig, falls Variablen auf dem Heap verwendet werden (mit `new`). Es ergibt sich aber damit eine bekannte Schwierigkeit: Klickt der Anwender den Close-Button in der Titelleiste, so wird gmain sofort verlassen, wo immer sich das Programm gerade befindet und der Code für das Aufräumen daher nicht ausgeführt. Um dies zu vermeiden, fängt man den Event des Close-Button-Klicks ab, indem `evWindowClose()` registriert wird. Im Callback-Code dieses Events wird dann lediglich *state* auf *quitting* gesetzt, wodurch das Programm samft und aufgeräumt beendet wird.

Die Deklaration der Klasse Panel erfolgt in PANEL.H. Der Grundgedanke besteht darin, dass man einen CPModelessDialog exakt in ein CPWindow eingepasst. Die Klasse Panel ist daher von CPModelessDialog abgeleitet. In ihr wird eine FrameWindow-Klasse definiert, welche von CPWindow abgeleitet ist und das Menü und eine Referenz auf "ihr" Panel enthält.

```
// PANEL.H

#ifndef _PANEL_H
#define _PANEL_H


class Panel : public CPModelessDialog
{

// ------------------ Nested class ----------------------------------
class FrameWindow : public CPWindow
{
public:
   FrameWindow ( const char * title, Panel & dlg );

private:
   virtual bool evMenuSelect ( unsigned int id );
   virtual void evWindowClose ();

   CPMenu _menu;
   Panel & _dlg;
};
// ------------------ End of nested class --------------------------


public:
   Panel ( const char * title, int & x );
```

```
   ~Panel ();
   void showResult ( double x, double y );
   void enableStart ();
   void enableStop ();
   void enableResume ();

private:
   virtual void evPushButtonClick ( CPPushButton & btn );
   FrameWindow * _pFrame;

   CPEdit _resultEdit;
   CPButton _startBtn;
   CPButton _stopBtn;
   CPButton _resumeBtn;
   int & _x;
};

#endif // _PANEL_H
```

Die hier gewählten Bezeichner haben sich gut bewährt (Gross-Klein-Schreibung, Klassenvariablen (data member) mit einem Underline eingeleitet, um sie von Parametern und anderen Variablen zu unterscheiden). Die Implementierung der Klasse Panel erfolgt in PANEL.CPP. Der Code wird aus Platzgründen hier nicht gezeigt. Er ist, wie üblich, auf dem Internet erhältlich *(**www.clab.unibe.ch/champ**)*.

Sehr oft müssen im Zusammenhang mit GUI-Oberflächen die Benützereingaben auf Korrektheit geprüft werden, damit sich das Programm auch dann sinnvoll verhält, falls der Benützer gewollt oder bösartig falsche Eingabe macht. Man spricht in diesem Zusammenhang von "**data validation**". Es ist bekannt, dass diese Aufgabe oft nur mit grossem Aufwand zu bewältigen ist, da es sich meist um ein Parsing-Problem handelt (die Eingabe muss vom Programm "verstanden" werden und oft sind mehrere Eingabevarianten korrekt). Im mathematisch-naturwissenschaftlichen Bereich sind Eingabewerte oft Dezimalzahlen. Dies führt uns zu folgendem Problem:

P65:
Man schreibe ein Programm, welches einen Double einliesst und diesen auf Korrektheit überprüft, bevor es ihn wieder ausschreibt.

L65:
In C++ löst man dieses Parsing-Problem am einfachsten unter Verwendung von Streams, und zwar hier eines `istrstream`-Objekts (Input-String-Stream). Dieses Objekt kann den eingelesenen String (character array) in ein anderes Format umwandeln. Dabei gibt die Methode `fail()` an, ob bei der Umwandlung etwas schief gelaufen ist. Wir verwenden dann auch noch ein `ostrstream`-Objekt, um eine Eingabe, die aus mehreren Wörtern besteht, abzufangen.

```
// FLOATIN.CPP
// Validate floating point number in edit field

#include <champ.h>
#include "floatin.rh"

void doOk ( CPDialog & dlg );
bool validate ( char * valueStr, double & value );

void gmain ()
{
   CPModelessDialog dlg( "UserDialog" );

   dlg.showModeless();
   CPPushButton okButton( dlg, IDOK, doOk );

   while ( !dlg.isClosed() )
      CP::yield();
}

void doOk ( CPDialog & dlg )
{
   char valueStr[20];
   double value;

   CPEdit input( dlg, IDC_INPUT );
   CPEdit output( dlg, IDC_OUTPUT );
   input.text().getline( valueStr, sizeof( valueStr ), '\n' );
   if ( validate( valueStr, value ) )
   {
      output.text() << value << ends;
   }
   else
   {
      CP::msgBox( "Error" ) << "Illegal entry";
   }
}

bool validate( char * valueStr, double & value )
{
   istrstream is( valueStr );
   if ( is.eof() )
       return false; // Empty entry
   else
   {
      is >> value;    // Try to convert
      if ( is.fail() )         // Can't convert
         return false;
      else
      {
         is >> ws;  // Throw away whitespaces
         // We test if there is an illegal trailing part
         ostrstream os;
         os << is.rdbuf() << ends;  // Get the trailing part
         int len = strlen( os.str() );
         delete os.str();
         if ( len > 0 )
            return false;
         return true;
      }
   }
}
```

Selbst wenn man möglicherweise die Streams nicht ganz durchblickt, lässt sich der Code im Sinne einer Black-Box-Library einsetzen. Im nächsten Problem geht es noch einmal um die graphische Benützeroberfläche.

P66:
Man möchte zur Auflockerung ein Bild in einen Dialog einbauen. Wie muss man vorgehen?

L66:
Vorbemerkung: Es muss ein Bild im BMP-Format vorliegen, das in der Pixelgrösse dem Dialog angepasst ist. Das Vorgehen unter BC++ Version 4.52 ist wie folgt:

1. Den Dialog mit dem Resource Workshop wie üblich erstellen und ihn wie bekannt im Programm anzeigen lassen. (Siehe Beispiel DLGIMG1.CPP mit einem nicht modalen Dialog, der einen Button zum Beenden des Programms enthält.)
2. Mit dem Resource Workshop einen "Borland-Button" einfügen (Werkzeug-Ikone wie ein Button.) Unter Eigenschaften | Schaltertyp Bitmap wählen. Den ID-Wert entsprechend abändern. Sich den numerischen Wert merken (im Beispiel 103).
3. Den Resource Workshop schliessen und in der Borland-IDE das File DLGIMG1.RC öffnen. Am Ende der Datei die Zeile
   `1103 BITMAP "cp-logo.bmp"`
   einfügen, wo der numerische Wert 1000 + ID-Wert des Buttons ist (im Beispiel 1103) und die Bilddatei in Anführungsstrichen steht.
4. Resourse Workshop durch Doppelklick auf das RC-File im Projekt wieder öffnen und den Dialog anzeigen lassen. Das Bild muss im Dialog sichtbar sein. Man kann es im Dialog positionieren.
5. Beim Ausführen des Programms erscheint das Bild im Dialog.

Durch Doppelklicken auf die Bild-Resource im Resource Workshop wird der interne Pixeleditor gestartet, mit dem man das Bild leicht bearbeiten kann. Es folgt der Code in den drei Dateien .CPP, .RC, .RH:

```
// DLGIMG1.CPP

#include <champ.h>
#include "dlgimg1.rh"

void doQuit();

bool quitting = false;
```

```
void gmain ()
{
   CPModelessDialog dlg( "DLGIMG" );
   dlg.showModeless();
   CPPushButton quitBtn( dlg, ID_QUIT, doQuit );

   while ( !quitting )
      CP::yield();
}


void doQuit ()
{
   quitting = true;
}


// DIALOG.RC
// Relevant section only

DLGIMG DIALOG 6, 15, 207, 111
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog with Image"
FONT 8, "MS Sans Serif"
{
 PUSHBUTTON "Quit", ID_QUIT, 82, 82, 50, 14
 CONTROL "Schalter", IDI_LOGO, "BorBtn", BBS_BITMAP | WS_CHILD | WS_VISIBLE
| WS_TABSTOP, 54, 30, 86, 25
}

1103 BITMAP "cp-logo.bmp"



// DLGIMG1.RH

#define DIALOG_1  1
#define ID_QUIT   102
#define IDI_LOGO  103
#include <champ.rh>
```

Im Folgenden möchte man in einem Dialog eine Ikone verwenden, die programmgesteuert entweder sichtbar oder unsichtbar ist. (Beispielsweise ein Knopf, der programmgesteuert wie eine LED grün oder rot erscheint.) Dazu werden zwei Bilder mit dem Pixeleditor erzeugt, die bis auf den roten bzw. grünen Kreis identisch sind. Ihre ID seien 1005 bzw. 1006.

1. Man geht wie oben beschrieben vor und erzeugt mit dem Pixeleditor eine grüne LED mit der ID 1005 bzw. eine rote ID 1006. Die IDs der entsprechenden "Borland-Buttons" müssen 105 bzw. 106 sein. Man positioniert die beiden LEDs exakt übereinander (Positionen im .RC-File eventuell von Hand anpassen).
2. Die beiden Buttons sind Controls, aber auch gewöhnlich Fenster im Sinn des Windows-API. Man kann diese im Programm über ihren

"Handle" ansprechen. Dazu braucht man vorerst den Handle des Dialogs, den man mit der Methode `hwnd()` nach Anzeige des Dialogs erhält.

```
HWND hwndDlg = dlg.hwnd(); // Call after dialog is shown!
```

3. Mit der API-Function `GetDialogItem()` erhält man den gesuchten Handle des Button-Controls und kann dieses mit der API-Funktion `ShowWindow()` anzeigen oder verstecken lassen.

```
void setGreen ()
 {
    ShowWindow( GetDlgItem( hwndDlg, ID_GREENLED ), SW_SHOW );
    ShowWindow( GetDlgItem( hwndDlg, ID_REDLED ), SW_HIDE );
 }
```

Zum Schluss folgen noch Workarounds zu drei Bugs, die sich in der Version 4.52 von Borland C++ unangenehm bemerkbar machen.

P67:
Klickt man auf das RC-File im Projektknoten, so geht der Resource Workshop automatisch auf. Man editiert damit und kann im Prinzip kompilieren, ohne dass man den Resource Workshop schliesst. Dies führt unter BC++ Version 4.52 je nach Betriebssystem manchmal zu einem fatalen Kompilationsfehler: "**Unerwarteter Abbruch während des Compilierens**". Anschliessend muss man meist BC neu starten.

L67:
Es wird folgender Workaround empfohlen: Nach dem Editieren schliesse man den Resource Workshop immer (speichern & schliessen), bevor das Projekt neu kompiliert wird.

P68:
Unter BC++ Version 4.52 führen gewisse Einleseoperationen im Zusammenhang mit Floats oder Doubles zu einem Runtime-Error mit dem Hinweis auf einen "**Floating Point Overflow**", obschon kein Programmierfehler vorliegt.

L68:
Es wurde an verschiedenen Stellen darauf hingewiesen, dass man den Fehler vermeiden kann, falls man unmittelbar vor dem Einlesen die Library-Funktion `_fpreset()`(aus float.h) aufruft. In der neusten Champ-Version 1.11 wird dieser Fehler behoben, so dass der Workaround mit `_fpreset()` nicht mehr nötig ist. (der Champ-Update ist auf www.salvisberg.com/champ erhältlich).

P69:
Falls man versucht, BC++ Version 4.52 unter Windows 2000 zu installieren, so misslingt dies, da bei der Installation gewisse DLLs nicht ins Windows-System-Verzeichnis kopiert werden.

L69:
Installieren Sie zuerst die Version BC++ Version 5.02 und erst nachher die Version 4.52. Da beide Versionen teilweise dieselben DLLs verwenden, gibt sich nun auch die Version 4.52 zufrieden.

P70:
Mit BC++ Version 4.52 kann es vorkommen, dass das Champ-Help nicht funktioniert, obschon die Installation laut Anleitung (mit ohelpcfg.exe) durchgeführt wurde.

L70:
Man öffnet mit einem Texteditor (z.B. notepad) die Datei openhelp.ini, die sich im Windows-Systemverzeichnis befindet und sucht nach der Zeile

```
Descriptor=11111...1110
```

Die Anzahl der Zahlen 1 entspricht der Anzahl der Help-Dateien. Die letzte Zahl sollte aber keine 0, sondern auch eine 1 sein.

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