Stolpersteine und Spezialitäten in C/C++

betreut von Aegidius Plüss, Gymnasium Bern-Neufeld

Einer der Vorteile der Programmierung unter Windows besteht darin, dass bei der Ausgabe von Text die vielfältigen Schriftarten von Windows zur Verfügung stehen. Damit können Ausgaben in einem Bildschirm-Fenster, aber auch auf dem Drucker anforderungsgerecht gestaltet werden, was etwa dann wichtig ist, falls Simulations- oder Demonstrationsprogramme erstellt werden. Schriften können positioniert, in der Höhe und Breite verändert, fett, unterstrichen, gedreht, usw. dargestellt werden, und dies mit sehr einfachen Funktionsaufrufen. Das folgende Programm vermittelt einen Eindruck von den Möglichkeiten:

```
// TEXTEX1.CPP

#include <champ.h>

void gmain ()
{
   ginit( "Textex1", CPSize( 400, 200 ) );
//  pinit();
   gwindowEven( 0, 1, 0, 0.5 );

   gtextFont( "Times Roman", 0.1 );
   gtextUnderline( true );
   gtextWeight( FW_BOLD ); gtext( 0.1, 0.40, "Champ's text features" );

   gtextFont( "Arial", 0.05 );       gtext( 0.1, 0.30, "Arial" );
   gtextWeight( FW_BOLD );           gtext( " bold" );
   gtextItalic( true );              gtext( " italic" );

   gtextFont( "Desdemona", 0.1 );    gtext( 0.1, 0.20, "Des" );
   gtextHeight( 0.05 );              gtext( "demona" );
   gtextWeight( FW_BOLD );           gtext( " bold" );
   gtextItalic( true );              gtext( " italic" );
   gtextUnderline( true );           gtext( " under" );
   gtextStrikeout( true );           gtext( "line" );

   gtextFont( "Courier", 0.05 );     gtext( 0.1, 0.15, "Courier" );
   gtextWeight( FW_BOLD );           gtext( " bold" );
   gtextItalic( true );              gtext( " italic" );
   gtextWeight( FW_NORMAL );         gtext( " normal" );

   gtextFont( "Symbol", 0.05 );      gtext( 0.1, 0.10, "aAbBcCdDeE" );
   gtextWeight( FW_BOLD );           gtext( " fFgGhHiIjJkK" );

   gtextFont( "WingDings", 0.05 );   gtext( 0.1, 0.05, "aAbBcCdDeE" );
   gtextWeight( FW_BOLD );           gtext( " fFgGhHiIjJ" );

   gtextFont( DEVICE_DEFAULT_FONT ); gtext( 0.1, 0, "Press any key..." );
   getch();
   gend();
//   pend();
}
```

Die auskommentierten Zeilen führen zum Ausdruck auf dem Systemdrucker. Unabhängig vom verwendeten Drucker ist das Schriftbild immer dasselbe. Dies ist einem weiteren wichtigen Vorteil von Windows zuzuschreiben: Die Programmierung ist weitgehend deviceunabhängig geworden. Dies war in der DOS-Welt ganz anders: Die Aelteren unter uns erinnern sich schmerzhaft, wie manche Stunde damit zugebracht werden musste, die richtigen "Escape-Sequenzen" zur Steuerung eines Druckers oder Plotters zu schreiben!

Mit der einfachen Druckerausgabe kann Champ beispielsweise dafür eingesetzt werden, Standardbriefe mit variablem Feldern, welche mit C++ berechnet werden (oder von Daten aus einer Datei bezogen werden) zu erzeugen. Dies lässt sich zwar mit Macroprogrammierung (beispielsweise Visual Basic) innerhalb einer gängigen Textverarbeitung (bespielsweise MS-Word) auch machen, ja stellt geradezu das Musterbeispiel für den Einsatz der Macroprogrammierung in der Schule dar. Wieviel allgemeiner und flexibler kann dasselbe Problem aber in einer weitverbreiteten höheren Programmiersprache, wie etwa C++ oder Java gelöst werden , ohne dass ein komplizierter Code dazu nötig ist!

Oft ist es nicht sinnvoll, Ausgaben in ein Graphikfenster zu schreiben, besonders dann, wenn die Ausgabe mit anderen GUI-Elementen, wie Buttons oder Listboxes, usw. kombiniert werden soll. In diesem Fall eignet sich ein modaler oder nichtmodaler Dialog mit eingebautem Textfenster. Sollen nur Ausgaben gemacht werden, so ist dazu ein Control der Klasse CPStatic zu verwenden. Falls ein Control der Klasse CPEdit eingesetzt wird, steht sogar ein voll funktionsfähiges Editierfenster (mit "cut and paste", Scrollbalken, usw.) zur Verfügung. Instanzen von CPStatic und CPEdit besitzen die Methode text, mit der formatierter Text auf einfachste Art in das Control geschrieben werden kann: text ist nämlich ein Stream analog zu cout, so dass alle dort

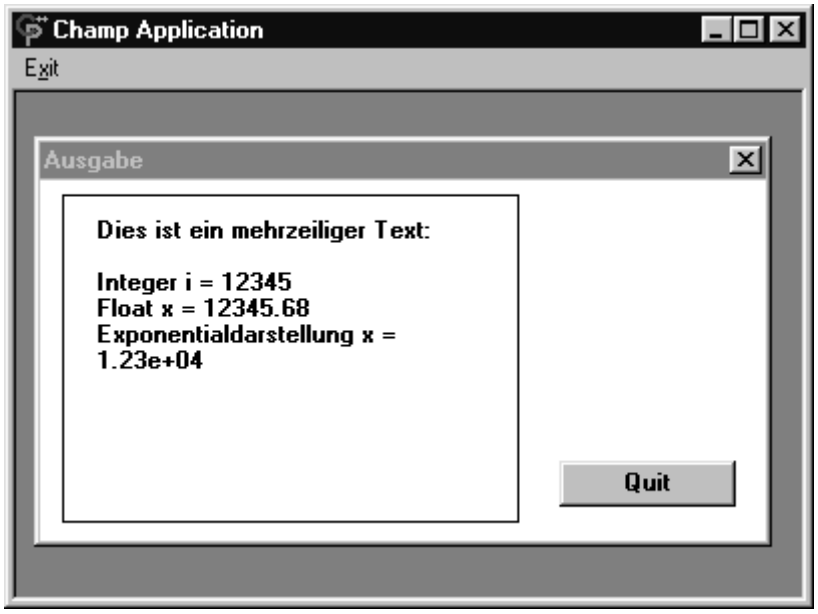
bekannten Formatierungsmöglichkeiten eingesetzt werden können. Im folgenden Beispiel werden Integer- und Float-Zahlen ausgeschrieben:

```
// TEXTEDI.CPP

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

void quitProc ();

void gmain ()
{
   int i = 12345;
   float x = 12345.6789;

   CPFrame f;
   CPModelessDialog dlg( "Ausgabe" );
   CPStatic display( dlg, IDSTATIC );
   CPButton quit( dlg, IDQUIT, quitProc );
   dlg.showModeless();

   display.text()
      << "Dies ist ein mehrzeiliger Text:" << endl << endl
      << "Integer i = " << i << endl
      << setiosflags( ios::fixed )
      << setprecision(2)
      << "Float x = " << x << endl
      << setiosflags( ios::scientific )
      << "Exponentialdarstellung x = " << x << ends;

   CP::run();
}

void quitProc ()
{
   CP::quit();
}
```

(Mit dem Resource-Workshop wird mithilfe graphischer Methoden ein Dialog mit dem Namen "Ausgabe" erstellt, welcher das Static-Control mit der Id **IDSTATIC** und ein Button mit der Id **IDQUIT** enthält. Die Resourcen sind in der Datei "textedi.rc" abgespeichert, die dazugehörenden Symbole in der Datei "textedi.rh".)

Das Programm erzeugt ein "Frame"-Fenster, in das der Dialog eingebettet ist. Man beachte auch die einfache Verwendung der "Callback-Function" quitProc: Das Hauptprogramm läuft auf die Zeile mit CP::run(). Klickt der Benützer auf den Button "Quit", so wird diese Funktion automatisch aufgerufen, da wir sie bei der Erzeugung des Buttonobjekts "quit" registriert, d.h. angegeben haben.

quitProc ruft dann die Methode CP::quit auf, was dazu führt, dass CP::run() verlassen und das Programm beendet wird.



Eine Schwierigkeit ergibt sich mit dieser Methode: Hat man einmal in das Static- oder Editfenster geschrieben, so kann man den Text nicht mehr ergänzen, sondern muss in als Ganzes neu schreiben. Um dies möglichst einfach zu bewerkstelligen, kann man die neue String-Klasse von C++ einsetzen. Diese ermöglicht es, den String ohne jede Platzreservierung beliebig zu ergänzen. Folgendes Programm zeigt das Verfahren:

```
// TEXTSTR.CPP

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

void nextProc ();
bool hasClicked = false;

void gmain ()
{
   CPFrame f;
   CPModelessDialog dlg( "Ausgabe" );
   CPStatic display( dlg, IDSTATIC );
   CPButton quit( dlg, IDNEXT, nextProc );
   dlg.showModeless();

   string s = "Dies ist ein mehrzeiliger Text:\n\n";
   display.text() << s.c_str() << ends;

   while ( !hasClicked )
      CP::yield();
   hasClicked = false;

   s = s + "Hier ist eine erste Textzeile\n";
   display.text() << s.c_str() << ends;

   while ( !hasClicked )
```

```
      CP::yield();
   hasClicked = false;


   s += "Hier ist eine zweite Textzeile\n";
   display.text() << s.c_str() << ends;

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

void nextProc ()
{
   hasClicked = true;
}
```

Das Programm verweilt jeweils in einer Endlosschlaufe, in der nur "geyieldet" wird, d.h. wo der Prozesser für andere Aufgaben freigegeben wird. Dies ist typisch für Windows 3.1 und 95, welche keine preemtiven, sondern kooperative Betriebssysteme sind. Wir der Button "next" gedrückt, so läuft das Programm weiter.

Wie ersichtlich, wird der String mit + bzw. mit += konkateniert und ein C-typischer String (character array) mit der Methode c_str herausgeholt.

Leider handelt es sich bei den Strings um keine Streams, so dass es nun schwierig wird, formatierten oder konvertierten Text auszuschreiben. Auch hier gibt es eine elegante Methode: Man verwendet die Klasse ostrstream! Mit einem Objekt dieser Klasse kann man wie mit cout verfahren: Man benützt den inserter << und die üblichen Manipulatoren. Auch ein ostrstream-Objekt besitzt einen dynamischen Speicher, der sich den nötigen Platz selbst (auf dem Heap) holt. Mit der Methode str() wird der Inhalt als C-typischer String (character array) zurückgegeben. Aus Sicherheitsgründen wird beim Aufruf dieser Methode der Speicherplatz aber "eingefroren". Bevor man das Objekt vernichtet, sollte man dieses Einfrieren mit der Methode freeze(0) wieder rückgängig machen, damit der Platz beim Vernichten des Objekt (hier fällt es aus dem Scope, weil zusätzliche Blockklammern vorhanden sind) wieder freigegeben wird.

```
// TEXTOSTR.CPP

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

void nextProc ();
bool hasClicked = false;

void gmain ()
{
   int i = 12345;
   float x = 12345.6789;
```

```
   CPFrame f;
   CPModelessDialog dlg( "Ausgabe" );
   CPStatic display( dlg, IDSTATIC );
   CPButton quit( dlg, IDNEXT, nextProc );
   dlg.showModeless();

   string s = "Dies ist ein mehrzeiliger Text:\n\n";
   display.text() << s.c_str() << ends;

   while ( !hasClicked )
      CP::yield();
   hasClicked = false;

   {
      ostrstream os;
      os << "Integer i = " << i << endl << ends;
      s = s + os.str();
      os.rdbuf()->freeze( 0 );
      display.text() << s.c_str() << ends;
   }

   while ( !hasClicked )
      CP::yield();
   hasClicked = false;

   {
      ostrstream os;
      os << setiosflags( ios::fixed )
         << setprecision(2)
         << "Float x = " << x << endl
         << setiosflags( ios::scientific )
         << "Exponentialdarstellung x = " << x << ends;
      s = s + os.str();
      os.rdbuf()->freeze( 0 );
      display.text() << s.c_str() << ends;
   }

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

void nextProc ()
{
   hasClicked = true;
}
```

Hier wird auch ersichtlich, dass man in C++ mit dem Datenspeicher sehr vorsichtig umgehen muss, da es keine automatische Garbage-Collection gibt, wo der nicht mehr benötigte Speicherplatz freigegeben wird. In diesem Bezug hat man es in Java besser: Eine periodisch ausgeführte Routine entfernt nicht mehr verwendeten Speicherplatz selbständig. Allerdings muss man damit in Kauf nehmen, dass zu unbestimmten Zeiten eine Auflaufverzögerung des laufenden Programms auftritt. Für nicht zeitkritische Anwendungen ist dies aber belanglos.

Dieser Text und die beschriebenen Programme sind auf der SIS-Website erhältlich, und zwar unter **http://www.kl.unibe.ch/kl/lab/champ**.