Low-Level Ansteuerung der Text-Konsole in C++

Portables Textausgabe-API

Wenn man unter UNIX keinen X-Windows-Server zur Verfügung hat, also pures UNIX auf einer Server-Maschine einsetzt, dann ist eine, auf Text basierende, Benutzeroberfläche ein willkommenes Hilfmittel zur interaktiven Benutzerführung. **Peter Thömmes, Januar 2006**

Konsole

Die Konsole ist der primäre Zugang zu UNIX-Systemen, da reines UNIX keine graphische Benutzeroberfläche anbietet. In der Regel sieht man 80x25 Zeichen (80 Spalten und 25 Zeilen) in nichtproportionaler Schrift auf dem Bildschirm. Erst die Installation eines X-Windows-Servers (oder kurz: X-Server) erweitert das System derart, dass Anwendungen auf einen Graphikbildschirm zugreifen können. Der X-Server entkoppelt die Treiber der Hardware, also der Graphikkarte und des Monitors, von den Anwendungen, wodurch Programme hardwareunabhängig und damit langlebig werden.
Im Gegensatz zu UNIX bietet Windows die Möglichkeit der graphischen Ausgabe jederzeit an, da das GDI (Graphical Device Interface) ein fester Bestandteil des Betriebssystems ist und daher auch immer für Anwendungen zur Verfügung steht. Das war allerdings bei den ersten Windows-Versionen (16-Bit) nicht so, denn damals war der Betriebssystemkern (DOS) nicht mit der Fähigkeit der graphischen Ausgabe ausgestattet. Anders als bei bei UNIX jedoch, musste jede Anwendung eigene Graphik-Treiber mitliefern (zum Beispiel VGA-Treiber), was zur hardwareabhängigkeit und damit kurzlebigkeit der Programmen führte.

Windows-Konsole

Unter Windows gibst es neben der DOS-Box, die in einer VDM (Virtual DOS Machine) läuft auch eine echte Win32-Konsole, die oft mit der DOS-Box verwechselt wird, da die Shell-Kommandos kompatibel zu DOS sind. **<<Bild 1>>**

Zum steuern der Textausgabe benötigt der Programmierer zunächst ein Handle (`hStdOut`):

```
void* hStdOut =
  GetStdHandle(
    STD_OUTPUT_HANDLE);
if(hStdOut ==
   INVALID_HANDLE_VALUE){
   ...//error
}
```

Damit wird dann zuerst die Scroll-Leiste abgeschaltet:

```
COORD coSize;
coSize.X = 80;
coSize.Y = 25;
BOOL bOk =
SetConsoleScreenBufferSize(
              hStdOut,
              coordSize);
```

Dann wird die automatische Darstellung von Tastatureingaben (ECHO) abgeschaltet:

```
unsigned long dwMode = 0;
BOOL bOk =
     GetConsoleMode(
               hStdOut,
               &dwMode);
if(!bOk){
   ...//error
}

dwMode &=
~ENABLE_ECHO_INPUT;
dwMode &=
~ENABLE_PROCESSED_OUTPUT;
dwMode &=
~ENABLE_WRAP_AT_EOL_OUTPUT;
bOk = SetConsoleMode(
                hStdOut,
                dwMode);
if(!bOk){
   ...//error
}
```

Jetzt kann die Kontrolle des Bildschirminhaltes implementiert werden, wozu Methoden zur Farbumstellung des Textes und des Hintergrundes programmiert werden. Folgende Textfarben sind möglich (jeweils mit einem 4-bit-Wert `c` kodiert):

```
c:   R,  G,  B: Farbe
------------------------
0:   0,  0,  0: schwarz
1:   0,  0,127: dunkelbl.
2:   0,127,  0: grün
3:   0,127,127: grün-blau
4: 127,  0,  0: dunkelrot
5: 127,  0,127: purpur
6: 127,127,  0: olive
7: 192,192,192: silber
8: 127,127,127: grau
9:   0,  0,255: blau
A:   0,255,  0: hellgrün
B:   0,255,255: cyan
C: 255,  0,  0: rot
D: 255,  0,255: magenta
E: 255,255,  0: gelb
F: 255,255,255: weiss
```

Tabelle 1: Textfarben der Win32-Konsole

Zum aktivieren einer bestimmten **Textfarbe** `c` ist folgender Kode geeignet (hier gezeigt für **`grün`**):

```
unsigned char c = 2;

CONSOLE_SCREEN_BUFFER_INFO
                     csbi;
BOOL bOk =
 GetConsoleScreenBufferInfo(
                  hStdOut,
                  &csbi);
if(!bOk){
   ...//error
}

unsigned short wAttr =
        csbi.wAttributes;
wAttr &= 0xFFF0;
wAttr |= c;
bOk =
 SetConsoleTextAttribute(
                  hStdOut,
                  wAttr);
if(!bOk){
   ...//error
}
```

Das Aktivieren einer bestimmten **Hintergrundfarbe** geschieht über fast denselben Kode, jedoch setzt man hier das zweite Nibble

in `wAttr`. Windows erlaubt für den Hintergrund die gleiche Palette von Farben wie für den Text (hier gezeigt für **dunkelrot**):

```
unsigned char c = 4;

CONSOLE_SCREEN_BUFFER_INFO
                    csbi;
BOOL bOk =
 GetConsoleScreenBufferInfo(
                  hStdOut,
                  &csbi);
if(!bOk){
   ...//error
}

unsigned short wAttr =
        csbi.wAttributes;
wAttr &= 0xFF0F;
wAttr |= c;
bOk =
 SetConsoleTextAttribute(
                  hStdOut,
                  wAttr);
if(!bOk){
   ...//error
}
```

Wenn man alle Textfarben in Kombination mit allen Hintergrundfarben durchspielt, ergibt sich beispielsweise eine Ausgabe entsprechend Bild 2.

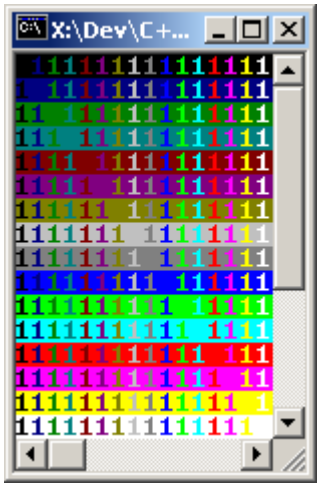

Bild 2: Alle Farben der Win32-Konsole

Nun gibt es ein kleines Problem hinsichtlich der **Portabilität**: UNIX unterstützt zware alle hier gezeigten 16 Textfarben, aber nur 8 Hintergrundfarben und zwar die ersten acht (`c` = 0…7). Deshalb sollte man als **Hintergrundfarbe** nur folgende benutzen:

```
c:   R,  G,  B: Farbe
------------------------
0:   0,  0,  0: schwarz
1:   0,  0,127: dunkelbl.
2:   0,127,  0: grün
3:   0,127,127: grün-blau
4: 127,  0,  0: dunkelrot
5: 127,  0,127: purpur
6: 127,127,  0: olive
7: 192,192,192: silber
```

Tabelle 2: Portable Hintergrundfarben der Win32-Konsole

Dadurch beschränkt man sich auf die Hälfte der sonst möglichen Farbkombinationen, also auf nur 128 statt 256:

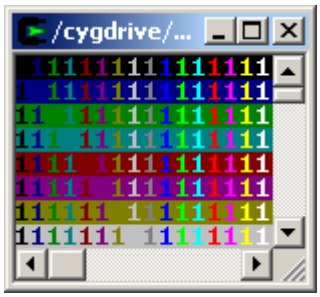

Bild 3: Portable Farben der Win32-Konsole

Um nun letztendlich die volle Kontrolle über den gesamten Bildschirm der Konsole zu bekommen, wird auch noch der Cursor ausgeschaltet, so dass man eine eigene Eingabe-Aufforderung programmieren kann:

```
CONSOLE_CURSOR_INFO cci;
BOOL bOk =
   GetConsoleCursorInfo(
                  hStdOut,
                  &cci);
if(!bOk){
   ...//error
}

cci.bVisible = FALSE;
bOk =
   SetConsoleCursorInfo(
                  hStdOut,
                  &cci);
if(!bOk){
   ...//error
}
```

Als nächstes benötigt man eine Methode zur **Ausgabe eines Textes mit den entsprechenden Attributen**:

```
short x = 5;
short y = 5;

CONSOLE_SCREEN_BUFFER_INFO
                    csbi;
BOOL bOk =
GetConsoleScreenBufferInfo(
                  hStdOut,
                  &csbi);
if(!bOk){
   ...//error
}

COORD coordCursPos;
coordCursPos.X = x;
coordCursPos.Y = y;
unsigned long
       dwNumWritten = 0;
bOk =
FillConsoleOutputAttribute(
        hStdOut,
        csbi.wAttributes,
        1,
        coordCursPos,
        &dwNumWritten);
     return false;
if(!bOk){
   ...//error
}
if(!dwNumWritten){
   ...//error
}

bOk =
WriteConsoleOutputCharacter(
          hStdOut,
          &c,
          1,
          coordCursPos,
          &dwNumWritten);
if(!bOk){
   ...//error
}
if(!dwNumWritten){
   ...//error
}
```

Damit hat man nun im wesentlichen das Rüstzeug zur Programmierung einer Textoberfläche unter Windows. Durch Kapselung der Funktionalität in einer Wrapper-Klasse kann man sich nun ein, auf die eigenen Bedürfnisse zugeschnittenes, Textausgabe-API schreiben.

UNIX-Konsole

Unter UNIX sieht das Programmier-Interface zur Konsole völlig anders aus, zudem hat man mehrere Angriffspunkte. Am einfachsten ist es mit den vt100-kompatiblen ESCape-Sequenzen zu arbeiten. Das ist zudem auch ein konsistentes Konzept im Sinne von UNIX, da eine Konsole immer auch über eine asynchrone serielle Verbindung , wie RS232, RS422 oder TTY, verbunden sein kann. Asynchrone serielle Verbindung heisst, dass die Steuerung mit ASCII-Zeichenströmen geschieht, wobei die Zeichen unter 0x20 (32) als Steuerzecihen dienen:

Zeichen	ASCII-Code
STX	2 (0x02)
ETX	3 (0x03)
Ctrl+CR (LF)	10 (0x0A)
ENTER (CR)	13 (0x0D)
ESC	27 (0x1B)

Tabelle 3: ASCII Steuerzeichen

Für die UNIX-Konsole wird ESC, also 27 (0x1B), zur Einleitung einer Steuer-Sequenz benutzt. "\033" ist dabei die Octaldarstellung von 27, also ESC. Beipiel:

```
printf("\033[2J");
```

Diese Zeichenkette löscht den gesamten Bildschirm .

Obwohl sich in der Vergangenheit viele Entwickler um eine Vereinheitlichung der Text-Ausgabe-Schnittstelle unter UNIX bemühten, ist die ESCape-Sequenz das einzig wirklich portable Konzept. Eine bekannte Schnittstelle, die auf ESCape-Sequenzen aufsetzt, ist die Datei

```
/etc/termcap.
```

<<Bild 4>>
Diese Datei enthält hinter bestimmten Kürzeln die zugehörigen ESCape-Sequenzen, wobei je nach Anzahl der Parameter (wie z.B. einer Positionsangabe mittels x und y) gewisse Platzhalter enthalten sind. ESC wird dort mit \E codiert. Zum Auswerten der Datei gibt es Funktionen wie

```
tgetstr()
tparm()
putp().
```

Leider wird dieses Verfahren nicht von allen UNIX-Plattformen in der gleichen Weise unterstützt und deshalb auch hier nicht benutzt. Die Nutzung ist aber an vielen Stellen im World Wide Web erläutert.

Eine etwas höher angesiedelte Schnittstelle zur Konsole ist die "curses-library", die praktisch einen Window-Manager für textbasierende Fenster darstellt (wie zum Beispiel die SAA Oberfläche unter DOS). Dabei gibt es Fenster, Unterfenster und sogenannte Pads. Auch hier gibt es je nach Plattform Unterschiede in der Implementierung. (Unter linux heisst diese Bibliothek übrigens "ncurses".) Diese Schnittstelle wird hier nicht benutzt, da sie zu hoch angesiedelt ist.

Es geht also hier um eine Low-Level-Ansteuerung der Text-Konsole, die plattform-übergreifend funktioniert. Alle Ausgaben geschehen also einfach mit der C-Laufzeitbibliotheks-Funktion

```
printf()
```

und nutzen lediglich ESCape-Sequenzen zur Steuerung (siehe Tabelle 4).

Bei der Nutzung der ESCape-Sequenzen gibt es allerdings zwei Probleme:

a) Das **Verstecken des Cursors** funktioniert in einer echten Konsole nicht, sondern nur in einem **xterm**-Fenster, also in einem Terminal unter X-Windows.

b) Das **Schreiben in das letzte Feld rechts unten** (Spalte 80, Zeile 25) führt automatisch zum **Scrollen** um eine Zeile nach oben. Selbst wenn man System-Aufrufe wie pwrite() benutzt wird diese Sache leider nicht immer richtig gehandhabt (linux scrollt trotzdem an dieser Stelle, obwohl es nicht erlaubt ist).

Für den **Punkt a)** habe ich nur eine suboptimale Lösung, aber sie funktioniert zumindest: der Cursor wird nach jeder Ausgabe von Text sofort wieder auf das Feld rechts unten gestellt, so das er immer dort steht und blinkt.
Für den **Punkt b)** gibt es eine besser funktionierende Lösung und die sieht so aus:

```
bool
UpdateBottomRight(char c)
{
  //Cursor setzen
  printf("\033[79;24H");
  fflush(stdout);

  //Cursor sichern:
  printf("\0337");
  fflush(stdout);

  //Zeichen ausgeben:
  printf("%c",c);
  fflush(stdout);

  //25 Zeilen hochscr.:
  printf(""\033[25A");
  fflush(stdout);

  //25 Zeilen runterscr.:
  printf(""\033[25B");
  fflush(stdout);

  //Cursor restaurieren:
  printf("\0338");
  fflush(stdout);

  //Cursor setzen
  printf("\033[79;24H");
  fflush(stdout);
}
```

Es wird also im wesentlichen einmal hoch und einmal wieder runtergescrollt (jeweils um 25 Zeilen), wodurch der Bildschirm wieder richtig zurechtgerückt wird.

Für xterm-Fenster gäbe es zwar eine elegantere Möglichkeit: man

könnte sich im Speicher die letzten beiden Zeilen merken und zum Auffrischen der letzten Zeile folgendes tun:
1.) Cursor in die zweitletzte Zeile platzieren.
2.) Die zweitletzte Zeile löschen.
3.) Die zweitletzte Zeile nun mit Inhalt der letzten Zeile beschreiben.
4.) Cursor erneut in die zweitletzte Zeile platzieren.
5.) Eine leere Zeile einfügen.
6.) Cursor erneut in die zweitletzte Zeile platzieren.
7.) Die zweitletzte Zeile mit ihrem eigentlichen Inhalt beschreiben.

Dies funktioniert aber nicht in einer echten Konsole und scheidet somit als portable Lösung aus.

Um nun, wie schon oben bei Windows gezeigt, die volle Kontrolle über den gesamten Bildschirm der Konsole zu bekommen, wird folgendermaßen verfahren:

Zunächst sichert man alle Attribute um sie später wieder zurückzusetzen:

```
struct termios old_tios;
tcgetattr(
      fileno(stdout),
      &old_tios);
```

Dann aktiviert man den vt100-Modus:

```
printf("\033[61;1\"p");
```

Danach wird die Bildschirmgröße und der Scrollbereich eingestellt:

```
int L = 25; //Zeilen
int C = 80; //Spalten

//Bildschirmgröße:
printf(
     "\033[8;%d;%dt",L,C);
fflush(stdout);

//Löschen:
printf("\033[2J");
fflush(stdout);

//Scrollbereich:
printf(
     "\033[%d;%dr",0,L-1);
fflush(stdout);
```

Last but not least wird die automatische Ausgabe von Zeichen, die über die Tastatur eingegeben wurden (ECHO), abgeschaltet:

```
termios tio;
int iError = tcgetattr(
          fileno(stdout),
          &tio);
if(iError) {
  ... //error
}

tio.c_lflag &= ~ECHO;
iError =tcsetattr(
         fileno(stdout),
         TCSANOW,
         &tio);
if(iError) {
  ... //error
}
```

Jetzt kann die eigentliche Ausgabe geschehen, wozu zunächst die Textfarbe gewählt wird. Jede Textfarbe ist mit einer Teilsequenz (String) ST kodiert:

ST:	R,	G,	B:	Farbe
30:	0,	0,	0:	schw.
34:	0,	0,127:		dk.bl
32:	0,127,	0:		grün
36:	0,127,127:			gn-bl
31:	127,	0,	0:	dk.rot
35:	127,	0,127:		purpur
33:	127,127,	0:		olive
37:	192,192,192:			silber
30;1:	127,127,127:			grau
34;1:	0,	0,255:		blau
32;1:	0,255,	0:		hellgn
36;1:	0,255,255:			cyan
31;1:	255,	0,	0:	rot
35;1:	255,	0,255:		magen.
33;1:	255,255,	0:		gelb
37;1:	255,255,255:			weiss

Tabelle 5: Textfarben der UNIX-Konsole

Die Textfarbe läßt sich nur in Kombination mit einer Hintergrundfarbe umstellen, welche durch die Teilsequenz SB (String) kodiert wird:

SB:	R,	G,	B:	Farbe
40:	0,	0,	0:	schw.
44:	0,	0,127:		dk.bl
42:	0,127,	0:		grün
46:	0,127,127:			gn-bl
41:	127,	0,	0:	dk.rot
45:	127,	0,127:		purpur
43:	127,127,	0:		olive
47:	192,192,192:			silber

Tabelle 6: Hintergrundfarben der UNIX-Konsole

Der Code zur Farbumstellung sieht dann so aus:

```
printf(
  "\033[0;%s;%sm",ST,SB);
fflush(stdout);
```

Dies ist also im Endeffekt eine ganz einfache Sache, man muss sich lediglich an die Vorgehensweise gewöhnen.

Portables Textausgabe-API

Um ein portables Textausgabe-API zu programmieren, beschränkt man sich auf die Schnittmenge der Fähigkeiten der Windows- und der UNIX-Konsole. Dann deklariert man für jede dieser Basisfunktionen eine Interface-Methode und schreibt jeweils 2 Implementierungen, die durch den Preprozessor-Switch `_Win32` separiert werden.

```
#ifdef _WIN32
  ... //Windows-Code
#else
  ... //UNIX-Code
#endif
```

Solche Methoden könnten folgendermaßen deklariert sein:

```
short Initialize();
void Uninitialize();

unsigned char
 SetTextColor(
  unsigned char byTCIdx);
unsigned char
 SetBackgroundColor(
  unsigned char byBCIdx);

bool Clear();
bool Fill(char c);
```

```
bool SetCursorXY(
    short x,short y);
bool GetCursorXY(
    short& x,short& y);
bool PutChar(char c);
bool PutStr(
    const char* szStr);
bool PutFmtString(
    const char* szFmtStr,
    ...);
bool PutCharXY(
    short x,
    short y,
    char c);
bool PutStrXY(
    short x,
    short y,
    const char* szStr);
```

Eine mögliche Implementierung von `SetTextColor()` zeigt Listing 1.

Benutzeroberfläche

Um eine eigene, auf Text basierende, Benutzeroberfläche zu programmieren, bündelt man Aufrufe des selbst implementierten Textausgabe-APIs in einer Art, das man ein höher angesiedeltes Interface erhält, also so etwas wie die 'curses-library'. Man muss also einen Window-Manager programmieren. Eine vereinfachte Funktion zum Zeichnen eines Fensters zeigt **Listing D**. Die Ausgabe könnte dann dann wie in **Bild 5** aussehen.

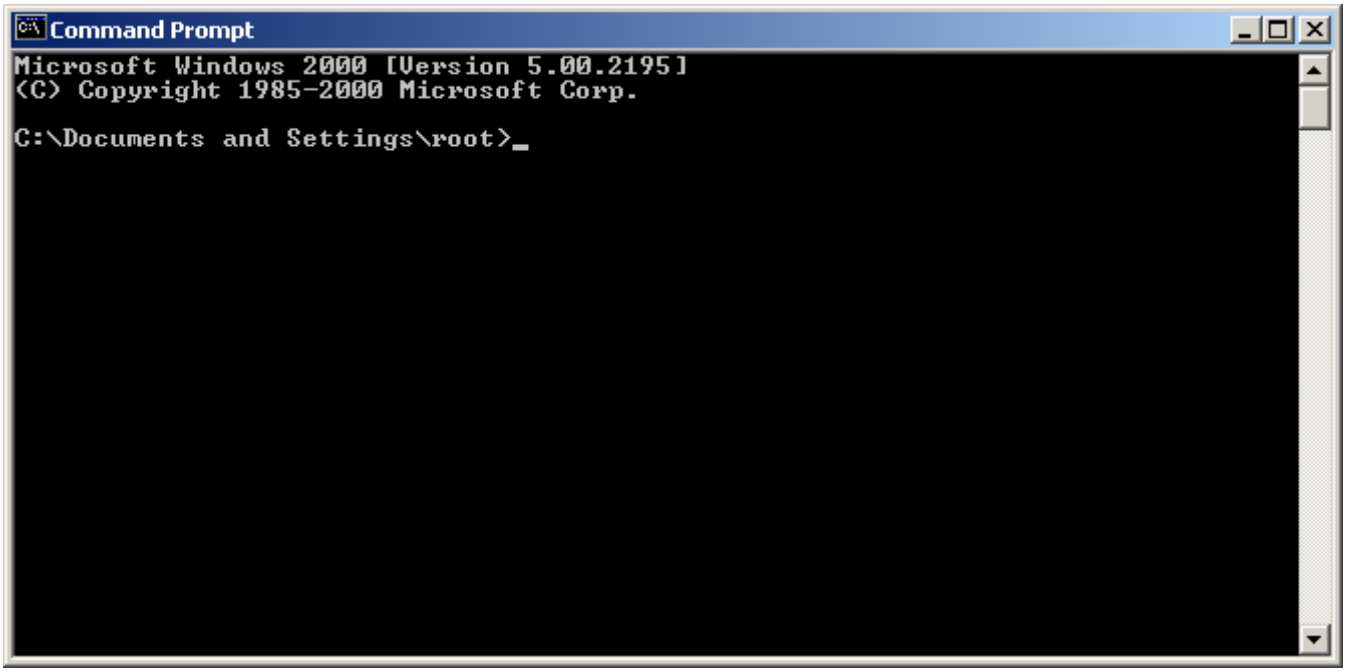

Bild 1: Win32-Konsole

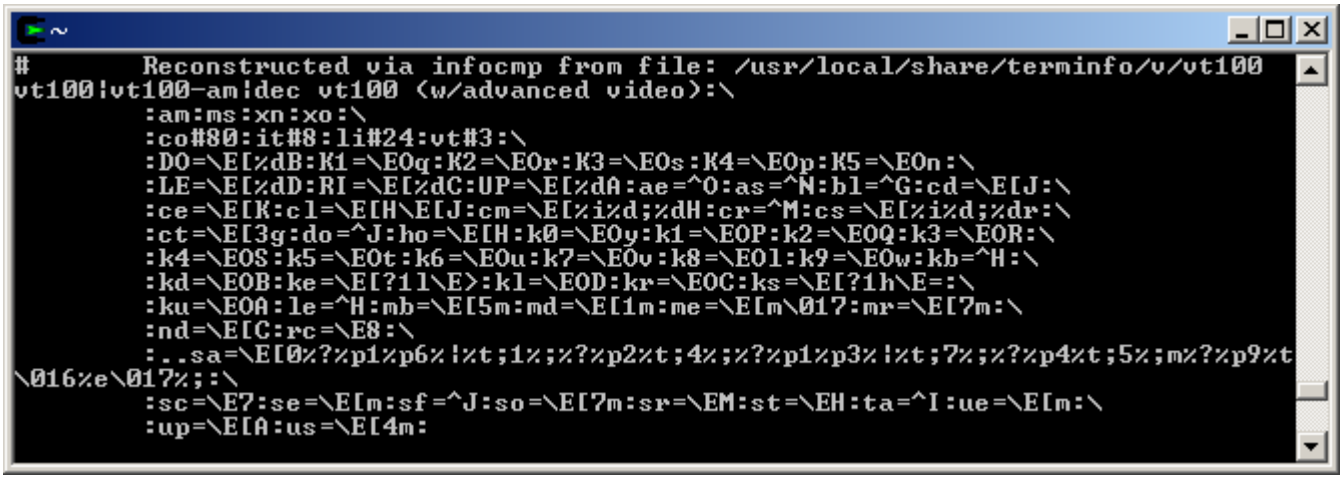

Bild 4: /etc/termcap

Aktion	ESCape-Sequenz	Eintrag (cap) in `/etc/termcap`
Allgemein (PORTABEL):		
vt100 terminal setzen	`printf("\033[61;1\"p");`	
Terminal in original Zustand zurücksetzen	`printf("\033c");`	
Bildschirmgröße: L Zeilen und C Spalten	`printf("\033[8;%d;%dt",L,C);`	
Textfarbe T + Hintergrundfarbe H setzen	`printf("\033[0;%s;%sm",T,H);`	
vt100 (PORTABEL):		
Bildschirm löschen	`printf("\033[2J");`	`"cd"`
Cursor nach (x,y) bewegen	`printf("\033[%d;%dH",x,y);`	`"cm"`
N Zeilen hochscrollen	`printf("\033[%dA",N);`	`"UP"`
N Zeilen runterscrollen	`printf("\033[%dB",N);`	`"DO"`
Scrollbereich setzen: Zeile L0 bis L1	`printf("\033[%d;%dr",L0,L1);`	`"cs"`
Cursor sichern	`printf("\0337");`	`"sc"`
Cursor restaurieren	`printf("\0338");`	`"rc"`
xterm:		
N Zeilen hinter Cursor einfügen	`printf("\033[%dL",N);`	`"AL"`
N Zeilen hinter Cursor löschen	`printf("\033[%dM",N);`	`"DL"`
Cursor verstecken	`printf("\033[?25l");`	`"vi"`
Cursor zeigen	`printf("\033[?25h");`	`"ve"`

Tabelle 4: ESCape-Sequenzen

```
enum
{
   T_COLOR_BLACK   =  0,//RGB = (  0,  0,  0), black
   T_COLOR_NAVY    =  1,//RGB = (  0,  0,127), navy (dark-blue)
   T_COLOR_GREEN   =  2,//RGB = (  0,127,  0), green
   T_COLOR_TEAL    =  3,//RGB = (  0,127,127), teal (green-blue)
   T_COLOR_MAROON  =  4,//RGB = (127,  0,  0), maroon (dark-red)
   T_COLOR_PURPLE  =  5,//RGB = (127,  0,127), purple
   T_COLOR_OLIVE   =  6,//RGB = (127,127,  0), olive
   T_COLOR_SILVER  =  7,//RGB = (192,192,192), silver
   T_COLOR_GREY    =  8,//RGB = (127,127,127), grey
   T_COLOR_BLUE    =  9,//RGB = (  0,  0,255), blue
   T_COLOR_LIME    = 10,//RGB = (  0,255,  0), lime (light-green)
   T_COLOR_CYAN    = 11,//RGB = (  0,255,255), cyan (light-blue)
   T_COLOR_RED     = 12,//RGB = (255,  0,  0), red
   T_COLOR_MAGENTA = 13,//RGB = (255,  0,255), magenta
   T_COLOR_YELLOW  = 14,//RGB = (255,255,  0), yellow
   T_COLOR_WHITE   = 15 //RGB = (255,255,255), white
};

enum
{
   BG_COLOR_BLACK  =  0,//RGB = (  0,  0,  0), black
   BG_COLOR_NAVY   =  1,//RGB = (  0,  0,127), navy (dark-blue)
   BG_COLOR_GREEN  =  2,//RGB = (  0,127,  0), green
   BG_COLOR_TEAL   =  3,//RGB = (  0,127,127), teal (green-blue)
   BG_COLOR_MAROON =  4,//RGB = (127,  0,  0), maroon (dark-red)
   BG_COLOR_PURPLE =  5,//RGB = (127,  0,127), purple
   BG_COLOR_OLIVE  =  6,//RGB = (127,127,  0), olive
   BG_COLOR_SILVER =  7,//RGB = (192,192,192), silver
};

unsigned char TextScreen::SetTextColor(unsigned char byTextColorIdx)
{
   #ifdef _WIN32
      return WSetTextColorImpl(byTextColorIdx);
   #else
      return USetTextColorImpl(byTextColorIdx);
   #endif
}

#ifdef _WIN32 //WINDOWS
unsigned char TextScreen::WSetTextColorImpl(unsigned char byTextColorIdx)
{
   //Get handle to stdout:
   void* hStdOut = ::GetStdHandle(STD_OUTPUT_HANDLE);
   if(hStdOut == INVALID_HANDLE_VALUE)
      return 0xFF;

   //Get the screen settings and keep the old values:
   CONSOLE_SCREEN_BUFFER_INFO csbi;
   if(!::GetConsoleScreenBufferInfo(hStdOut,&csbi))
      return 0xFF;
   unsigned short wAttr = csbi.wAttributes;
   unsigned char byOldTextColorIdx =
                              WGetTextColorIndexFromAttrImpl(csbi.wAttributes);

   //Set new attributes:
   wAttr &= 0xFFF0; //delete textcolor
   wAttr |= WGetTextColorAttrFromIndexImpl(byTextColorIdx); //put new textcolor
   if(!::SetConsoleTextAttribute(hStdOut,wAttr))
      return 0xFF;
   return byOldTextColorIdx;
}
#endif
```

Listing 1A: Beispiel-Implementierung, Teil A

```
#ifdef _WIN32 //WINDOWS
unsigned char TextScreen::WGetTextColorIndexFromAttrImpl(unsigned short wAttr)
{
   return (unsigned char) (wAttr & 0x000F);
}
#endif

#ifdef _WIN32 //WINDOWS
unsigned short TextScreen::WGetTextColorAttrFromIndexImpl(
                                       unsigned char byTextColorIdx)
{
   static unsigned short TEXT_COLOR_PALETTE[16] =
   {
      0x0000, //RGB = (  0,  0,  0), black
      0x0001, //RGB = (  0,  0,127), navy (dark-blue)
      0x0002, //RGB = (  0,127,  0), green
      0x0003, //RGB = (  0,127,127), teal (green-blue)
      0x0004, //RGB = (127,  0,  0), maroon (dark-red)
      0x0005, //RGB = (127,  0,127), purple
      0x0006, //RGB = (127,127,  0), olive
      0x0007, //RGB = (192,192,192), silver
      0x0008, //RGB = (127,127,127), grey
      0x0009, //RGB = (  0,  0,255), blue
      0x000A, //RGB = (  0,255,  0), lime (light-green)
      0x000B, //RGB = (  0,255,255), cyan (light-blue)
      0x000C, //RGB = (255,  0,  0), red
      0x000D, //RGB = (255,  0,255), magenta
      0x000E, //RGB = (255,255,  0), yellow
      0x000F  //RGB = (255,255,255), white
   };

   if(byTextColorIdx > 15)
      byTextColorIdx = 15;
   return TEXT_COLOR_PALETTE[byTextColorIdx];
}
#endif
```

Listing 1B: Beispiel-Implementierung, Teil B

```
#ifndef _WIN32 //UNIX
unsigned char TextScreen::USetTextColorImpl(unsigned char byTextColorIdx)
{
   unsigned char byOldIdx = theTextColorIdx();
   theTextColorIdx() = byTextColorIdx;
   printf(
      ESC_CHANGE_ATTR_TC_BGC,
      UGetTextColorSequenceFromIndex(theTextColorIdx()),
      UGetBGColorSequenceFromIndex(theBGColorIdx()));
   fflush(stdout);
   return byOldIdx;
}
#endif

#ifndef _WIN32 //UNIX
char* TextScreen::UGetTextColorSequenceFromIndex(unsigned char byTextColorIdx)
{
   static char TEXT_COLOR_PALETTE[16][10] =
   {
      "30",//black
      "34",//navy (dark-blue)
      "32",//green
      "36",//teal (green-blue)
      "31",//maroon (dark-red)
      "35",//purple
      "33",//olive
      "37",//silver
      "30;1", //grey
      "34;1", //blue
      "32;1", //lime (light-green)
      "36;1", //cyan (light-blue)
      "31;1", //red
      "35;1", //magenta
      "33;1", //yellow
      "37;1"  //white
   };
   if(byTextColorIdx > 15)
      byTextColorIdx = 15;
   return TEXT_COLOR_PALETTE[byTextColorIdx];
}
#endif

#ifndef _WIN32 //UNIX
char* TextScreen::UGetBGColorSequenceFromIndex(unsigned char byBGColorIdx)
{
   static char BG_COLOR_PALETTE[8][10] =
   {
      "40",//black
      "44",//navy (dark-blue)
      "42",//green
      "46",//teal (green-blue)
      "41",//maroon (dark-red)
      "45",//purple
      "43",//olive
      "47",//silver
   };
   if(byBGColorIdx > 7)
      byBGColorIdx = 7;
   return BG_COLOR_PALETTE[byBGColorIdx];
}
#endif
```

Listing 1C: Beispiel-Implementierung, Teil C

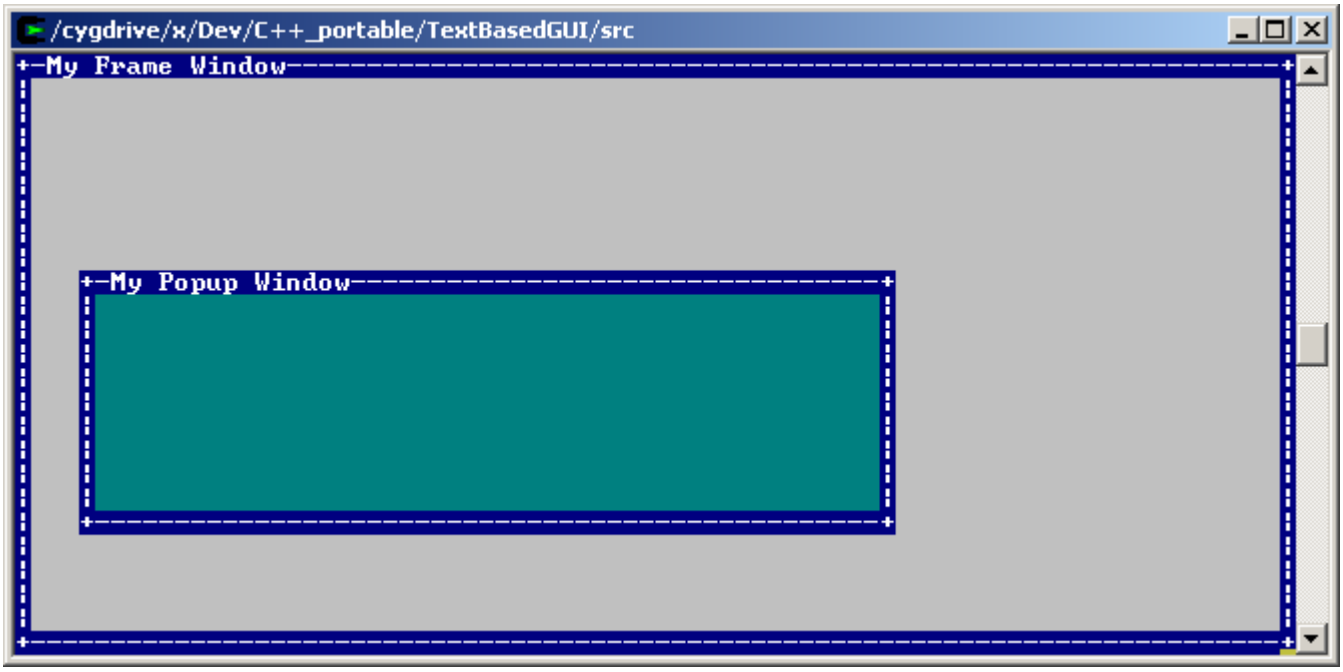

Bild 5: Basisfunktion DrawWindow des Window-Managers

```
void WindowManager::DrawWindow(
       const char* const szTitle,
       unsigned char byFrameTextColorIdx,
       unsigned char byFrameBGColorIdx,
       unsigned char byWindowBGColorIdx,
       unsigned char x,
       unsigned char y,
       unsigned char cx,
       unsigned char cy)
{
  //Draw an empty window (background):
  TextScreen::SetBackgroundColor(byWindowBGColorIdx);
  short i = 0;
  short j = 0;
  for(i = (short) (y + 1);i < (y + cy);++i)
  {
    TextScreen::SetCursorXY((short) (x + 1),i);
    for(j = (short) (x + 1);j < (x + cx);++j)
      TextScreen::PutChar(' ');
  }

  //Set the frame colors:
  TextScreen::SetTextColor(byFrameTextColorIdx);
  TextScreen::SetBackgroundColor(byFrameBGColorIdx);

  //Draw the frame:
  TextScreen::SetCursorXY(x,y);
  TextScreen::PutChar('+');
  for(i = (short) (x + 1);i < (x + cx);++i)
    TextScreen::PutChar('-');
  TextScreen::SetCursorXY((short) (x + cx),y);
  TextScreen::PutChar('+');

  for(i = (short) (y + 1);i < y + cy;++i)
  {
    TextScreen::SetCursorXY(x,i);
    TextScreen::PutChar('|');
    TextScreen::SetCursorXY((short) (x + cx),i);
    TextScreen::PutChar('|');
  }

  TextScreen::SetCursorXY(x,(short) (y + cy));
  TextScreen::PutChar('+');
  for(i = (short) (x + 1);i < x + cx;++i)
    TextScreen::PutChar('-');
  TextScreen::SetCursorXY((short) (x + cx),(short) (y + cy));
  TextScreen::PutChar('+');

  //Draw the title:
  if(cx > 4)
  {
    char szTitleShrinked[80 + 1] = "";
    strncpy(szTitleShrinked,szTitle,cx - 4);
      szTitleShrinked[cx - 4] = 0x00; //to be sure
    TextScreen::SetCursorXY((short) (x + 2),y);
    TextScreen::PutStr(szTitleShrinked);
  }
}
```

Listing D: Beispiel-Implementierung eines Fensters