Windows-Programmierung, BORLAND

Franz Fiala, N, TGM DSK-404:\BD

BORLAND verwendet zwei verschiedene Benutzeroberflächen (IDE=Integrated Development Environment) für DOS und Windows. Während man mit der DOS-IDE alle Formen von Programmdateien entwickeln kann (also auch EXE-Files für Windows), kann man mit der Windows-IDE ausschließlich Windows-Programme generieren. Die Windows-IDE präsentiert sich so:

EasyWin

Das einfachste unter Windows ablaufende Programm wird mit **EasyWin** erzeugt. Man kann die gewohnten Funktionen aus `conio.h` und `stdio.h` verwenden, allerdings mit Einschränkungen:

- Verzicht auf Farbe und Grafik
- Beschränkung auf:
  `gotoxy(), wherex(), wherey(), clrscr(), clreol()`

Und so schaut ein Programmgerüst mit **EasyWin** aus:

```
#include <windows.h>
#pragma argsused

int PASCAL WinMain
(
  HANDLE hInstance,
  HANDLE hPrevInstance,
  LPSTR lpszCmdLine,
  int cmdShow
)
{
  _InitEasyWin();
  ...
  alles weitere ist beliebig
}
```

Wie man sieht, ersetzt `WinMain()` das bekannte `main()` aus C und C++. Der Startup-Kode ist auch unterschiedlich. Etwa benötigt man mehr Parameter als noch unter DOS:

`hInstance` Jede Windows-Applikation unterscheidet sich von einer anderen durch ein `HANDLE`, eine ganze Zahl.
`hPrevInstance` Gibt es mehr als nur eine Instanz, dann enthält dieser Wert das `HANDLE` auf die vorige Instanz. Gibt es keinen Vorläufer, ist dieser Wert 0.
`lpszCmdLine` Pointer auf die Kommandozeile.
`cmdShow` Zeigt auf das erste darzustellende Fenster
`_InitEasyWin()` Initialisiert das Fenster und die verwendbaren `stdio`- und `conio`-Funktionen.

Erstes Windows-Programm, `EASY1.CPP`

Dieses Programm zeigt die Verwendung der `stdio`-Funktionen. Die Zeile

```
#pragma argsused
```

verhindert die Fehlermeldung, daß die übergebenen Parameter ungenutzt bleiben.

// EASY1.CPP

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#pragma argsused

int PASCAL WinMain
(
  HANDLE hInstance,
  HANDLE hPrevInstance,
  LPSTR lpszCmdLine,
  int cmdShow
)
{
  _InitEasyWin();
  printf("Hello world\n\x80");
  gotoxy(10,10);
  printf("Hallo Seminar\n");
  return 0;
}
```

Übergabeparameter, `EASY2.CPP`

Dieses Programm zeigt die Werte der vier Übergabeparameter an. Man sieht, daß die Kommandozeile mit Ausschluß des Programmnamens gezählt wird.

Am besten verwendet man `EasyWin` zum Debuggen, wenn der eingebaute Debugger nicht komfortabel genug arbeitet.

// EASY2.CPP

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#pragma argsused

int PASCAL WinMain
(
  HANDLE hInstance,
  HANDLE hPrevInstance,
  LPSTR lpszCmdLine,
  int cmdShow
)
{
  _InitEasyWin();
  printf("HANDLE hInstance    : %d\n", hInstance);
  printf("HANDLE hPrevInstance: %d\n", hPrevInstance);
  printf("LPSTR  lpszCmdLine  : %s\n", lpszCmdLine);
  printf("int    cmdShow      : %d\n", cmdShow);
  return 0;
}
```

Object Windows

Unser einfaches Programm soll lediglich einen Begrüßungstext kreieren, eine Abfrage, ob man denn wirklich aufhören will (Verabschiedung) und ein Window-gemäßes "Hello-World", in dem der Text nicht als ASCII-Text sondern mit der Maus als Grafik eingegeben wird.

Es wäre kein richtiges Programm, hätte es nicht auch einen Bug. Hier werden zwar die Linien richtig gezeichnet, beim nochmaligen Zeichnen aber gibt es unerwünschte Verbindungslinien, die noch zu beseitigen wären. Der Fehler entsteht durch die vereinfachte Aufzeichnung der Bildpunkte in einer Liste. Eigentlich müßte man eine Liste von Listen konstruieren und jede dieser Einzellisten wäre ein durchgehender Linienzug, daher sind Unterbrechungen möglich. So aber werden beim erstmaligen Neuzeichnen des Bildschirms auch jene Verbindungen gezogen, bei denen die Maus nicht gedrückt war. Als Übung könnten Sie daher dieses Beispiel entsprechend korrigieren und in Anlehnung an das Handbuch, in dem dieser Fehler natürlich nicht vorkommt, erweitern.

Diese Bildschirm steht sinngemäß stellvertretend für alle enstehenden Programmversionen OW0..OW06.CPP:

Windows-Programme werden mit der Windows-IDE hergestellt. Folgende Voraussetzungen sind für das Arbeiten mit Object Windows erforderlich:

```
Options
  Compiler - LARGE Model
  Linker Libraries
    Container(Static)
    Object Windows(Static)
    StandardRunTime (Static)
  Directories
    Pfade auf INCLUDE- und LIB-Subdirectories
    von RTL,CLASSLIB und OWL
```

Ein Fenster, OW0.CPP

Eine Object-Windows-Anwendung wird von der Klasse `TApplication` abgeleitet. Über die Größe des Fensters muß man sich in Windows zunächst nicht kümmern. Einer `TApplication` übergibt man den Titel (hier "Hello") die entsprechenden `HANDLE` und auch die Kommandozeile.

Die einfachste Anwendung (ein einzelnes Fenster) nimmt uns einmal die Aufgabe ab, sich um die Übergabeparameter kümmern zu müssen, indem die Klasse `TApplication` diese als Parameter übernimmt.

Das Objekt `a` der Klasse `TApplication` wird mit `a.Run()` ausgeführt.

```
//OW0.CPP
#define WIN31
#include <owl.h>

int PASCAL WinMain
(
  HANDLE hInstance,
  HANDLE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow)
{
  TApplication a      // Konstruktor rufen
  (
    "Hello",
    hInstance,
    hPrevInstance,
    lpCmdLine,
    nCmdShow
  );
  a.Run();           // Anwendung ausführen
  return a.Status;   // Beenden
}
```

Box zur Verabschiedung, OW01.CPP

Wie kann man nun etwas an diesem Basisvorgang verändern? Zunächst könnte man versuchen, eine bestehende Funktion von `TApplication` zu überladen; z.B. `CanClose()`.

Wie kann man den Benutzer fragen, ob er aufhören will? Man öffnet eine `MessageBox` und teilt dieser Box mit, welchen Titel ("Verabschiedung"), welche Botschaft ("Wollen Sie wirklich aufhören?") sie haben soll und welche Schalter (`MB_YESNO | MB_ICONQUESTION`) sie enthalten soll.

Die möglichen anderen Werte der Schalter und der Antworten der `MessageBox` erfährt man am besten über die eingebaute on-line-Hilfe.

Die `MessageBox` muß aber, wie viele andere Fenster auch, wissen, von welchem Fenster sie abstammt. `hInstance` ist ein Mitglied von `TModule` und enthält das aktuelle `HANDLE` der Applikation.

In diese neuen Klasse können die Funktionen zum Zeichnen des Fensters `InitMainWindow()` und `CanClose()` zum löschen des Fensters überladen werden.

Das Überladen des Fensters selbst generiert einen anderen als den Anfangstext durch Öffnen eines Fensters `TWindow`. Zum Verlassen des Programms einer `MessageBox` mit einem JA/NEIN-Knopf. `IDYES` bedeutet, daß der `OK`-Knopf selektiert wurde.

Anmerkung: Dieses und die folgenden Programme benutzen inline-Elementfunktionen, das sind solche, die innerhalb der Klassendeklaration definiert werden oder solche, die das zusätzliche Schlüsselwort `inline` vorangestellt bekommen. Die zweitere Art mit explizitem `inline`-erfordert mehr Platz und ist für Anfänger nicht so übersichtlich; daher wurden die Funktionen in die Klassendefinition mitaufgenommen. Es wird aber darauf hingewiesen, daß diese Schreibweise nur bei sehr kurzen Funktionen verwendet wird, hier aber der Einfachheit halber für alle Funktionen verwendet wurde. >>>

```
//OW01.CPP
#define WIN31
#include <owl.h>

class MyApp : public TApplication
{
public:
  MyApp
  (
    LPSTR AName,
    HANDLE hInstance,
    HANDLE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
  ) : TApplication
      (
        AName,
        hInstance,
        hPrevInstance,
        lpCmdLine,
        nCmdShow
      )
  {
  }

  virtual void InitMainWindow()
  {
    MainWindow = new TWindow (NULL,"Hallo");
  }
  BOOL CanClose()
  {
    return MessageBox
    (
      MainWindow->HWindow,
      "Wollen Sie wirklich aufhören?",
      "Verabschiedung",
      MB_YESNO | MB_ICONQUESTION
    ) == IDYES;
  }
};

int PASCAL WinMain
// ... wie in OW0.CPP
}
```

Text im Fenster, OW02.CPP

Jede Anwendung `TApplication` hat so etwas wie einen Konstruktor, nämlich eine Funktion, die aufgerufen wird, wenn eine Instanz dieser Anwendung kreiert wird: `InitInstance();`

```
  void InitInstance()
  {
    TApplication::InitInstance();
    HDC DC;
    char S[100];
    sprintf(S,"Begrüßungstext");
    DC = GetDC(MainWindow->HWindow);
    TextOut(DC,100,100,S,strlen(S));
    ReleaseDC(MainWindow->HWindow,DC);
  }
```

aber auch eine Funktion, die beim allerersten Aufruf dieser Anwendung aufgerufen wird: `InitApplication()`.

Dieses `InitInstance()` druckt einen Begrüßungstext aus. Ganz so einfach geht das aber nicht:

Man schreibt nicht direkt in den Bildschirm, sondern in einen Display-Kontext `HDC`, den man von der Anwendung mit `GetDC()` erhält. Alle Ausgaben werden auf diesen Display-Kontext bezogen, indem dieser als Argument in der Ausgabefunktione `TextOut()` übergeben wird. Das Wichtigste aber ist, den Display-Kontext durch `ReleaseDC()` wieder freizugeben, sonst reagiert Windows ähnlich wie bei einem Stapelüberlauf auf Grund fehlender POP-Befehle.

Das Hauptfenster in `TApplication` ist einfach ein leeres Fenster. Will man etwas in das Fenster schreiben, muß man die Funktion `InitMainWindow()` überladen und ein eigenes Fenster einsetzen, dem man den Namen der Anwendung gibt.

Wie kann ein Fenster aktiv werden?

In der Klasse des neuen Fensters `TMyWindow` wird die virtuelle Funktion `WMLButtonDown()` überladen. diese Funktion erhält als eine BORLAND-C-Besonderheit einen Dispatcher-Index in eckigen Klammern nachgestellt der die Nummer der Botschaft enthält, die diese Funktion aktivieren soll, hier `WM_LBUTTONDOWN`. Der zusätliche Wert `WM_FIRST` ist lediglich ein Offset für alle Konstanten, der praktisch in jedem Dispatcher-Index vorkommt [`WM_LBUTTONDOWN. + WM_FIRST`].

ACHTUNG: Diese Syntax ist ungenormt, vereinfacht zwar die Sache ungemein, ist aber überhaupt nicht portabel. Bei Microsoft-C müssen die Botschaften im Event-Handler mit `switch`-Anweisungen an die richtige Funktion verteilt werden, ähnlich, wie es auch bei der TURBO-VISION-Bibliothek (Fenster-Bibliothek für DOS-Programme) der Fall ist.

In diesem Beispiel wird die aktuelle Kursorposition beim Drücken der linken Maustaste ausgegeben.

```
    HDC DC;
    char S[100];
    sprintf(S,"(%d,%d)",Msg.LP.Lo,Msg.LP.Hi);
    DC = GetDC(HWindow);
    TextOut(DC,Msg.LP.Lo,Msg.LP.Hi,S,strlen(S));
    ReleaseDC(HWindow,DC);
```

```
//OW02.CPP
#include <stdio.h>
#include <string.h>
#define WIN31
#include <owl.h>

class TMyWindow : public TWindow
{
public:
  TMyWindow
  (
    PTWindowsObject AParent,
    LPSTR ATitle
  ) : TWindow (AParent, ATitle)
  {
  }

  virtual void WMLButtonDown(RTMessage Msg)
   = [WM_FIRST + WM_LBUTTONDOWN]
  {
    HDC DC;
    char S[100];
    sprintf(S,"(%d,%d)",Msg.LP.Lo,Msg.LP.Hi);
    DC = GetDC(HWindow);
    TextOut(DC,Msg.LP.Lo,Msg.LP.Hi,S,strlen(S));
    ReleaseDC(HWindow,DC);
  }
};

class MyApp : public TApplication
// wie in OW01.CPP
```

Schreiben in das Fenster, OW03.CPP

Wie kann man in ein Fenster etwas schreiben? Dazu gibts die Funktion `Paint()` der Klasse `TWindow`, die für unseren Zweck in der Klasse `TMyWindow` überladen wird und einen neuen Begrüßungstext ausgibt.

Versuchen Sie einmal das Fenster in seiner Größe zu verändern!

Die Koordinatenangaben verschwinden alle wieder, der Begrüßungstext bleibt! Warum? Die Funktion `Paint()` wird bei jeder Veränderung des Bildschirms gerufen, die Funktion `WMLButtonDown()` dagegen nur bei jedem Mausklick.

```
// OW03.CPP
#include <stdio.h>
#include <string.h>
#define WIN31
#include <owl.h>

class TMyWindow : public TWindow
{
public:
  TMyWindow
  (
    PTWindowsObject AParent,
    LPSTR ATitle
  ) : TWindow (AParent, ATitle)
  {
  }

  virtual void WMLButtonDown(RTMessage Msg)
   = [WM_FIRST + WM_LBUTTONDOWN]
  {
    HDC DC;
    char S[100];
    sprintf(S,"(%d,%d)",Msg.LP.Lo,Msg.LP.Hi);
    DC = GetDC(HWindow);
    TextOut(DC,Msg.LP.Lo,Msg.LP.Hi,S,strlen(S));
    ReleaseDC(HWindow,DC);
  }

  virtual void Paint(HDC DC, PAINTSTRUCT& Paintinfo)
  {
    char S[100];
    sprintf(S,"Begrüßungstext");
    DC = GetDC(HWindow);
    TextOut(DC,50,50,S,strlen(S));
    ReleaseDC(HWindow,DC);
  }
};

class MyApp : public TApplication
// ... wie in OW02.CPP
```

Das Fenster weiß, was am Bildschirm steht, OW04.CPP

Was kann man also tun, daß sich der Fensterinhalt beim Verschieben mitändert? Man muß sich merken, was der Benutzer auf den Bidlschirm gemalt hat! Wie? Nichts einfacher als das! Mit den Hilfsmitteln der Container-Bibliothek schafft man das mit ein paar Zeilen, wir wollen es besonders schön machen und verwenden für einen Punkt eine eigene Struktur mit hübsch überladenen Operatoren. Die Punkte selbst speichern wir in einer Liste genau in der Reihenfolge der Eingabe:

```
  struct Pt
  {
    int x;
    int y;
    Pt(int x1, int y1) {x=x1; y=y1;}
    Pt() {x=0; y=0;}
    int operator!=(Pt p) { return !(p.x==x && p.y==y); }
    int operator==(Pt p) { return (p.x==x && p.y==y); }
  };
```

Nicht nur, daß die Struktur `Pt` ein Koordinatenpaar `x,y` enthält, es gibt auch einen Default-Kostruktor `Pt()`, der die Punktkordinaten auf Null setzt und einen Konstruktor mit Anfangswerten. Die beiden Operatoren `==` und `!=` stellen fest, ob zwei Punkte gleich sind. Sie werden im restlichen Kode vergeblich nach der Anwendung von `==` suchen. Wer braucht `==`?

Wir werden die erfordeliche Liste natürlich nicht selbst schreiben, sondern eine fertige Liste aus der Container-Bibliothek, am besten ein solche mit Templates verwenden. Man muß bedenken, daß eine vordefinierte Liste immer etwas mehr kann als man gerade benötigt. Sie enthält also auch Vergleichsmöglichkeiten, die wir eigentlich gar nicht brauchen. Für einfache Variablentypen, für die der Vergleichsoperator implizit definiert ist, also für ganze Zahlen und Gleitkommanzahlen ist der Vergleich auf Gleichheit oder Ungleichheit kein Problem. Ist es aber schon für Strukturen. Wann sind zwei Strukturen gleich? Genau diese Frage stellt der Compiler, wenn man versucht, die Punktstruktur der vordefinierten Template-Klasse aufzudrücken. Nun, jetzt hat er sie, und in der Zeile

```
  BI_ListImp<Pt> Points;
```

wird die Liste definiert. Der erste Punkt wird mit

```
    Points.add(Pt(0,0));
```

alle weiteren Punkte mit

```
    Points.add(Pt(Msg.LP.Lo,Msg.LP.Hi));
```

in die Liste eingefügt. Der erste Punkt (0,0) dient als Markierung für das Ende der Liste. Mit diesem Behelf merken wir uns alle Benutzereingaben (beachten Sie: 3 Zeilen) und geben die so gespeicherte Liste in der Funktion `Paint()` wieder aus. Für das Auslesen der Liste wird ein zerstörungsfreier Iterator verwendet:

```
    BI_ListIteratorImp<Pt> i(Points);
    Pt p, p0;
    while ((p=i++) != p0)
    {
      sprintf(S,"(%d,%d)",p.x,p.y);
      TextOut(DC,p.x,p.y,S,strlen(S));
    }
```

Hier findet man auch die Abfrage auf das Ende der Liste mit dem Operator `!=`. Dieses Beispiel wurde dem in der Beschreibung von Object-Windows nachempfunden. Der Unterschied zur Dokumentation ist, daß dort zur Speicherung der Bildschirmpunkte ein Array verwendet wird und hier, bei der einfachen Liste der erwähnte Fehler beim Neuzeichnen auftritt.

```
//OW04.CPP
// ... wie in OW03.CPP
class TMyWindow : public TWindow
{
  struct Pt
  {
    int x;
    int y;
    Pt(int x1, int y1) {x=x1; y=y1;}
    Pt() {x=0; y=0;}
    int operator!=(Pt p) { return !(p.x==x && p.y==y); }
    int operator==(Pt p) { return (p.x==x && p.y==y); }
  };

  BI_ListImp<Pt> Points;

public:

  TMyWindow
  (
    PTWindowsObject AParent,
    LPSTR ATitle
  )
  : TWindow (AParent, ATitle)
  {
    Points.add(Pt(0,0));
  }

  virtual void WMLButtonDown(RTMessage Msg)
   = [WM_FIRST + WM_LBUTTONDOWN]
  {
    HDC DC;
    char S[10];
    sprintf(S,"(%d,%d)",Msg.LP.Lo,Msg.LP.Hi);
    DC = GetDC(HWindow);
    TextOut(DC,Msg.LP.Lo,Msg.LP.Hi,S,strlen(S));
    Points.add(Pt(Msg.LP.Lo,Msg.LP.Hi));
    ReleaseDC(HWindow,DC);
  }

  virtual void Paint(HDC DC, PAINTSTRUCT& Paintinfo)
  {
    char S[20];
    sprintf(S,"Begrüßungstext");
    DC = GetDC(HWindow);
    TextOut(DC,50,50,S,strlen(S));
    BI_ListIteratorImp<Pt> i(Points);
    Pt p, p0;
    while ((p=i++) != p0)
    {
      sprintf(S,"(%d,%d)",p.x,p.y);
      TextOut(DC,p.x,p.y,S,strlen(S));
    }
    ReleaseDC(HWindow,DC);
  }
};

class MyApp : public TApplication
// ... wie in OW03.CPP
```

Linien zeichnen, OW05.CPP

Windows ist ja aber eine grafische Benutzeroberfläche, die wir bisher nur für Texte mißbraucht haben. Dieses Beispiel ist aber gleich gut auch für Grafik verwendbar, indem wir statt der Koordinatenangabe eine Linie ziehen. Dazu müssen wir aber nicht nur den Mausklick, sondern auch die Mausbewegung berücksichtigen. Die Maus soll nur zeichnen, wenn die Maustaste gedrückt ist.

Innerhalb der Fensterklasse `TMyWindow` muß man sich daher zwei Dinge merken

1. Den aktuellen Display-Kontext `DragDC`, der beim Drücken der linken Maustaste gesetzt wird. und
2. den aktuellen Zustand der linken Maustaste in `ButtonDown`, da die Maus auch ohne Taste bewegt wird.

Die beiden neuen Funktionen `WMMouseMove()` und `WMLButtonUp()` und `WMLButtonDown()` sind neu gestaltete Funktion.

Das Zeichnen gelingt ja ganz gut aber beim Erneuern mit `Paint()` werden noch die alten Punktkoordinaten gesetzt.

Das Programm `OW05.CPP` ist hier nicht abgebildet sondern nur die endgültige Version `OW06.CPP`:

```
//OW06.CPP
#include <stdio.h>
#include <string.h>
#define WIN31
#include <owl.h>
#include <listimp.h>

class TMyWindow : public TWindow
{
  HDC DragDC;
  BOOL ButtonDown;
  struct Pt
  {
    int x;
    int y;
    Pt(int x1, int y1) {x=x1; y=y1;}
    Pt() {x=0; y=0;}
    int operator!=(Pt p) { return !(p.x==x && p.y==y); }
    int operator==(Pt p) { return (p.x==x && p.y==y); }
  };
  BI_ListImp<Pt> Points;

public:

  TMyWindow
  (
    PTWindowsObject AParent,
    LPSTR ATitle
  )
  : TWindow (AParent, ATitle)
  {
    Points.add(Pt(0,0));
    ButtonDown = FALSE;
  }

  virtual void WMMouseMove(RTMessage Msg)
   = [WM_FIRST + WM_MOUSEMOVE]
  {
    if (ButtonDown)
    {
      LineTo(DragDC,Msg.LP.Lo,Msg.LP.Hi);
      Points.add(Pt(Msg.LP.Lo,Msg.LP.Hi));
    }
  }

  virtual void WMLButtonDown(RTMessage Msg)
   = [WM_FIRST + WM_LBUTTONDOWN]
  {
    if (!ButtonDown)
    {
      ButtonDown = TRUE;
      DragDC = GetDC(HWindow);
      MoveTo(DragDC,Msg.LP.Lo,Msg.LP.Hi);
      Points.add(Pt(Msg.LP.Lo,Msg.LP.Hi));
    }
  }

  virtual void WMLButtonUp(RTMessage Msg)
   = [WM_FIRST + WM_LBUTTONUP]
  {
    if (ButtonDown)
    {
      ButtonDown = FALSE;
      ReleaseDC(HWindow,DragDC);
    }
  }

  virtual void Paint(HDC DC, PAINTSTRUCT& Paintinfo)
  {
    char S[20];
    sprintf(S,"Begrüßungstext");
    DC = GetDC(HWindow);
    TextOut(DC,50,50,S,strlen(S));
    BI_ListIteratorImp<Pt> i(Points);
    Pt p, p0;
    while ((p=i++) != p0)
    {
      LineTo(DC,p.x,p.y);
    }
    ReleaseDC(HWindow,DC);
  }
};

class MyApp : public TApplication
{

public:

  MyApp
  (
    LPSTR AName,
    HANDLE hInstance,
    HANDLE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
  ) : TApplication
      (
        AName,
        hInstance,
        hPrevInstance,
        lpCmdLine,
        nCmdShow
      )
  {
  }

  virtual void InitMainWindow()
  {
    MainWindow = new TMyWindow (NULL,Name);
  }
  BOOL CanClose()
  {
    return MessageBox
    (
      MainWindow->HWindow,
      "Wollen Sie wirklich aufhören?",
      "Verabschiedung",
      MB_YESNO | MB_ICONQUESTION
    ) == IDYES;
  }
};

int PASCAL WinMain
(
  HANDLE hInstance,
  HANDLE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow)
{
  MyApp *a= new MyApp
  (
    "Hello",
    hInstance,
    hPrevInstance,
    lpCmdLine,
    nCmdShow
  );
  a->Run();
  return a->Status;
}
```

□