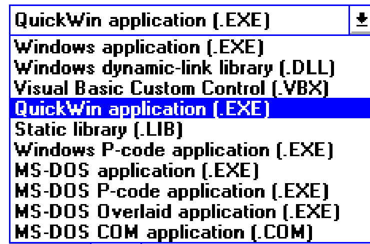
Windows-Programmierung, MICROSOFT

Franz FIALA, N, TGM DSK-404\MS

QuickWin

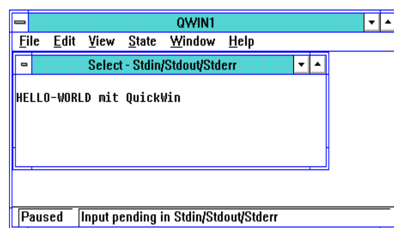
Für einen sanften Übergang von der DOS- zur Windows-Programmierung sorgt QuickWin. Was dabei allein zu tun bleibt, ist die Auswahl des richtigen Ausgangsformats im Optionen-Menü:



Hier sieht man auch gleich, welche weiteren Formate existieren.

Das ist alles! D.h. das "HELLO WORLD"-Programm mit QuickWin ist identisch mit dem "HELLO-WORLD"-Programm in DOS und schaut so aus:

```
// QWIN1.CPP
#include <iostream.h>

void main()
{
    cout << "\nHELLO-WORLD mit QuickWin\n";
    cin.get();
}
```

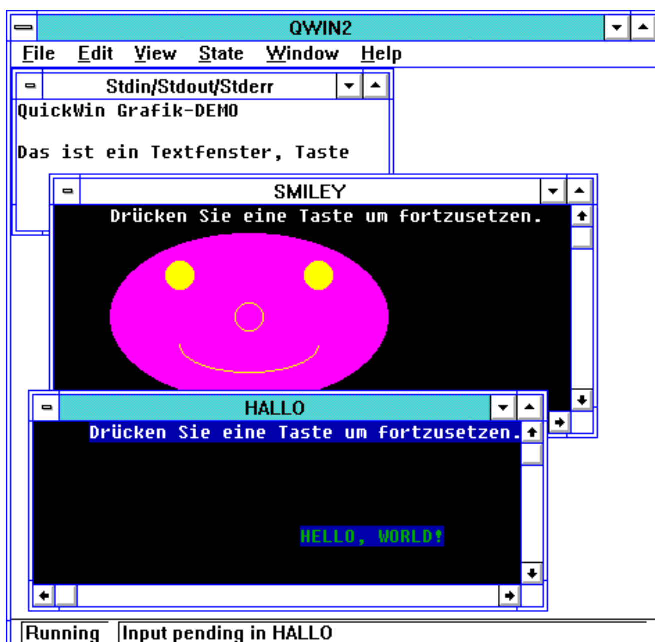


Es gibt bereits ein Elternfenster, das gleichzeitig mehrere QuickWin-Fenster verwaltet, hier nur das eine. Jede QuickWin-Applikation kann auch abgeleitete Fenster haben.

Das Schöne an QuickWin ist, daß keinerlei Änderungen der üblichen Gewohnheiten bei der DOS-Programmierung nötig sind und, daß neben einfachen Ein- und Ausgaben auch Grafik und Farbe kein Hindernis sind, wie das folgende Beispiel zeigt:

Mehrere Fenster, Grafikfenster

Dieses Programm öffnet 3 Fenster und wartet auf einen Tastendruck. Bildlaufleisten können verwendet werden. Die Bildschirme können auf volle Bildschirmgröße erweitert werden. Die darin befindlichen Texte können mit Hilfe des Menüs im Elternfenster QWIN2 kopiert werden.




```
// QWIN2.CPP öffnet mehrere Grafik-Fenster

#include <iostream.h>
#include <graph.h>

void main()
{

    int    handle;

//  Nachricht an ein Textfenster (stdin/stderr/stdout)
    cout <<  "QuickWin Grafik-DEMO\n" ;

//  Warten
    cout << "\nDas ist ein Textfenster, Taste\n" ;
    cin.get();

//  mit _setvideomode() wird ein Grafikfenster eröffnet
//  es hat den Anfangsnamen GRAFIC1.
    _setvideomode( _MAXRESMODE );
    handle = _wggetactive( ); // jedes Fenster hat ein handle
    _wgclose( handle );     // welches die Bearbeitung eben
                            // dieses Fensters ermöglicht
                            // in diesem Fall wird das Fenster
                            // geschlossen, damit ein neues
                            // mit einem eigenen Namen vergeben
                            // werden kann

//  Ein neues Fenster erstellen
    handle = _wgopen( "SMILEY" );
    _wgsetactive( handle );
    _setvideomode( _MAXRESMODE );

//  Kopf zeichnen
    _setcolor( 13 );
    _ellipse( _GFILLINTERIOR, 40, 20, 240, 140 );

//  Augen, Nase und Mund
    _setcolor( 14 );
    _ellipse( _GFILLINTERIOR, 80, 40, 100, 60 );
    _ellipse( _GFILLINTERIOR, 180, 40, 200, 60 );
    _ellipse( _GBORDER, 130, 70, 150, 90 );
    _arc( 90, 80, 190, 120, 115, 100, 165, 100 );


//  Warten auf Taste
    _settextposition( 1, 6 );
    _outtext("Drücken Sie eine Taste um fortzusetzen.");
    _inchar( );

//  Eröffnen eines Fensters im Textmodus
    handle = _wgopen( "HALLO" );
    _wgsetactive( handle );
    _setvideomode( _TEXTC80 );

//  Hintergrundfarbe und Textfarbe setzen
    _setbkcolor( 1 );
    _settextcolor( 2 );

//  Textausgeben
    _settextposition( 6, 25 );
    _outtext( "HELLO, WORLD!" );

//  Farben einstellen
    _setbkcolor( 1 );
    _settextcolor( 15 );

//  Warten auf Taste
    _settextposition( 1, 6 );
    _outtext("Drücken Sie eine Taste um fortzusetzen.");
    _inchar( );

//  Kursor ist auf der Zeile unterhalb der Nachricht
    _settextposition( 7, 1 );
}
```

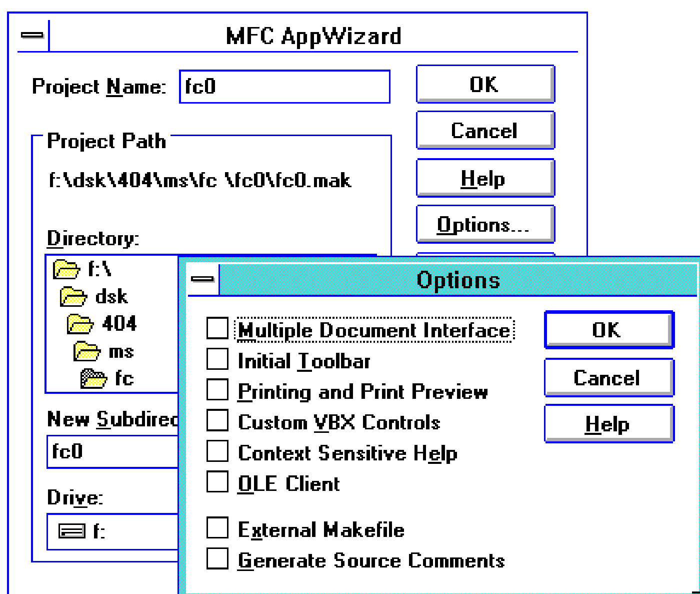
Das Programm QWIN2 (es heißt im Original SMILEY) ergibt das nebenstende Bildschirmbild, wobei die Fenster zwecks gleichzeitiger Darstellbarkeit frei arrangiert wurden. Die üblichen DOS-Grafikbefehle können mit geringen Abweichungen verwendet werden.

Foundation Classes

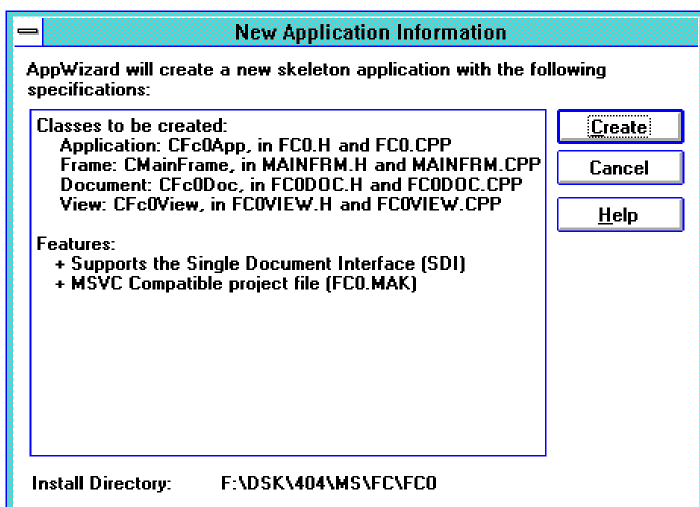
So einfach wie mit Quick-Windows geht es in C++ nicht, dafür kann man die ganze Bandbreite der Windows-Möglichkeiten nutzen. Ein Beispiel für die Leistungsfähigkeit dieser Bibliothek: Ein immerwiederkehrendes, lästiges Problem für Programmierer: Speichern und Laden von Daten. In der Windows-Oberfläche im Dateien-Menü mit den Menüpunkten Speichern, Speichern unter und Laden zu finden. Eine einzige Funktion muß für die eigenen spezifischen Daten geschrieben werden `Serialize()` in der Klasse `CDocument` und schon kann diese Funktion benutzergeführt wie mit jedem anderen Windows-Programm verwendet werden.

Aber zurück zum Anfang: Um den Einstieg zu erleichtern und um den administrativen Overhead für die jedenfalls erforderlichen Basisklassen zu erzeugen, wird gleich ein Hexenmeister, der Application Wizzard bemüht, dem - je nach Bedarf - der Class Wizzard folgt.

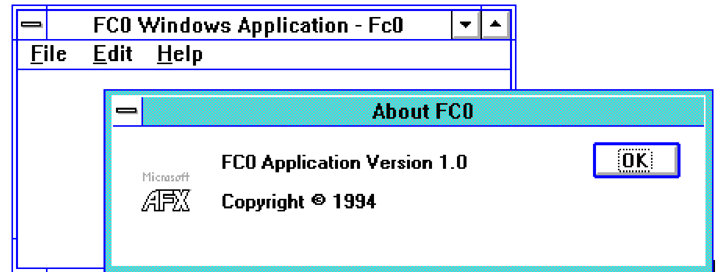
Ein Programm ist nicht mehr eine einzelne Datei sondern es sind gleich mehrere, die in einem Projekt zusammengefaßt sind. Ein Projekt wird eröffnet, indem ein Projektname vergeben wird, der gleichzeitig der Name des gleichnamigen Subdirectory ist. In den Optionen bestimmt man, welche Eigenschaften dieses Projekt haben soll. Je nach dem wird der automatisch generierte Kode länger oder kürzer ausfallen. Zum Ausprobieren wählen wir nichts von den schönen Dingen, denn wir wollen die einfachst mögliche Anwendung zeigen:



Diese Auswahl wird mit einem Zwischenfenster bestätigt:



Nachdem man mit **Project-Build** die Dateien compiliert und linkt und danach mit **Project-Execute** ausführt, meldet sich unsere Application und hat auch schon ein "About-Fenster".



Festhalten: Im Subdirectory `FC0` sind 31 Dateien mit 2,78MB entstanden sowie ein weiteres Subdirectory `RES`, das das verwendete Symbol für die Anwendung enthält:

Text im Fenster

Eine Anwendung besteht aus mindestens einer `CDocument`- und einer `CView`-Klasse beziehungsweise davon abgeleiteten Klassen. Diese Ableitung besorgt der Application-Wizzard für uns, denn wir erhalten zwei Dateien `FC0DOC.CPP` und `FC0VIEW.CPP` und auch die zugehörigen Header-Dateien und darin die abgeleieteten Klassen `CFc0Doc` und `CFc0View`. Der für diese Anwednunge gewählte Name `FC0` wird in die Bezeichnungen der Klassen unverwechselbar mitaufgenommen.

Was sind nun CDocument und CView?

Ganz grob kann man sagen, daß `CDocument` die Daten verwaltet und `CView` die Darstellung und die Kommunikation mit dem Benutzer. Ein Programm, das nur rechnet und mit Dateien kommuniziert, benötigt im Prinzip keine `CView`-Klasse (das nur zur Illustration).

Immer wenn der Bildschirm neu gezeichnet wird, wird die Funktion `OnDraw()` der Klasse `CView` aufgerufen. Will man daher etwas darstellen, dann muß diese Funktion das besorgen. Übergibt man das zu Zeichnende dem Konstruktor von `CView`, dann wird es zwar einmal gezeichnet aber kaum bewegt man das Fenster, verschwindet der Text oder die Zeichnung, da die Klasse `CView` nur einmal gezeichnet wird und dann nur mehr die Funktion `OnDraw()`.

Für das Schreiben eines Textes in den Bildschirm benötigen wir einen sogenannten **Display-Context**, der mit der `OnDraw()`-Funktion gleich mitgeliefert wird. Alle Ausgaben an den Bildschirm erfolgen über einen solchen Display-Context:

```
void CFc0View::OnDraw(CDC* pDC)
{
    pDC->TextOut(100,100,"HALLO");
    pDC->MoveTo(100,100);
    pDC->LineTo(50,50);
}
```

Der zwischen den Klammern stehende Kode ist alles, das man selbst schreiben muß, um den HALLO-Text und eine Linie von 100,100 nach 50,50 zu ziehen.

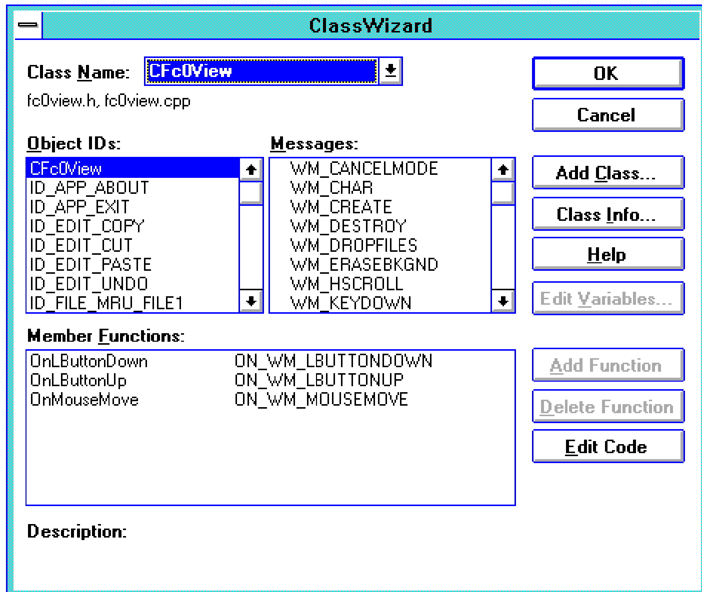
Unser Ziel ist es, mit der Maus einen schönen Begrüßungstext zu zeichnen, wie es sich bei einer grafischen Benutzeroberfläche gehört.

Der Kursor soll beim Klicken der linken Taste einen Punkt zeichnen, und danach beim Ziehen der Maus eine Linie. Das Loslassen der Maus unterbricht die Linie. Ein wichtiges Problem ist daher die Kommunikation mit dem Benutzer.

Kommunikation mit dem Benutzer

Gemäß dieser Aufgabenstellung müßte jeweils beim Betätigen der Maus etwas in den Datenstrukturen verändert werden. Daher benötigen wir je eine Funktion für jede der Mausaktivitäten.

Wir haben uns gemerkt, daß alle Kommunikation mit dem Benutzer über eine `CView`-Klasse abläuft. Normalerweise müßte man jetzt in der Header-Datei den entsprechenden Prototyp und in der CPP-Datei den dazugehörigen Kode einfügen. Nicht bei den Foundation-Classes: Man bemüht einfach einen neuen Hexenmeister, den Class-Wizzard im Menü Browse und erhält folgendes Bild.



Im linken Rahmen sieht man einzelnen Objekte und im Kasten daneben, die Windows-Nachrichten (WM_..), die von dem betreffenden Objekt bearbeitet werden können. Aus dieser Nachrichtenliste wählt man jene Nachrichten aus, die durch das Programm bearbeitet werden sollen; in unserem Fall sind das WM_LBUTTONDOWN, WM_LBUTTONUP und WM_MOUSEMOVE. Wenn wir sie anklicken, generiert der Wizzard zugehörige Member-Funktionen, die man im unteren Rahmen aufgezählt sieht, der Wizzard erledigt das aber auch in den zugehörigen Header-Dateien und auch in der Code-Datei. Nichts mehr zu tun. Es entstehen die Funktionen OnLButtonDown(), OnLButtonUp() und OnMouseMove().

Als erste Versuch wollen wir lediglich die aktuelle Kursorposition bei jedem Mausklick anzeigen lassen:

```
void CFcOView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CClientDC dc(this);

    char S[100];
    sprintf(S,"(%d,%d)",point.x,point.y);
    dc.TextOut(point.x,point.y,S);
}
```

Die Klasse CClientDC ist ein solcher Display-Context, der diesen auf unser Fenster beschränkt. Alle Ausgaben (hier TextOut) erfolgen über diesen Display-Context.

Gezeichnetes Hello

Jetzt wollen wir zeichnen, ohne das Gezeichnete zu merken (Das hat die wichtige Konsequenz, daß die Zeichnung beim Verändern des Fensters verschwindet).

Wir merken uns in einer Variablen ButtonDown, ob die Maustaste gerade gedrückt ist, außerdem merken wir uns die aktuelle Position des Kursors in oldpoint. Diese Variablen werden in der Headerdatei unseres View, FCOVIEW.H als private-Variable eingetragen:

```
    BOOL ButtonDown;
    CPoint oldpoint;
```

Der nun folgende Kodeteil ist das, was man einfügen muß:

```
void CFcOView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CClientDC dc(this);

    char S[100];
    sprintf(S,"(%d,%d)",point.x,point.y);
    dc.TextOut(point.x,point.y,S);
    if (!ButtonDown)
    {
      ButtonDown = TRUE;
      oldpoint=point;
    }
}

void CFcOView::OnLButtonUp(UINT nFlags, CPoint point)
{
    if (ButtonDown)
    {
      ButtonDown = FALSE;
    }
}

void CFcOView::OnMouseMove(UINT nFlags, CPoint point)
{
    if (ButtonDown)
    {
      CClientDC dc(this);
      dc.MoveTo(oldpoint);
      dc.LineTo(point);
      oldpoint=point;
    }
}
```

Immer wenn die linke Maustaste gedrückt wird, erfolgt die Ausgabe der Mausposition; Außerdem wird die Position in oldpoint festgehalten. Bei Mausbewegung wird die Funktion OnMouseMove() aktiviert und eine Linie von oldpoint zum aktuellen Punkt gezogen aber nur, wenn eben die Taste gedrückt ist: if (ButtonDown).

Man kann den gewünschten Begrüßungstext schon zeichnen! Versuchen Sie aber jetzt während des zeichnens das Fenster zu verändern, etwa indem Sie es am Rand ergreifen und vergrößern. Die Zeichnung verschwindet, da das Fenster neu gezeichnet wird und die Folge der Punkte nirgendwo festgehalten wurde. Lediglich der Begrüßungstext "HALLO" und die Probelinie der Funktion OnDraw() bleiben erhalten.

Daher ist es unser abschließendes Ziel, die Punktfolge in geeigneter Form zu speichern. Wir benutzen dazu eine Liste der Klasse CObList. [Auf die Problematik typenreiner Datenstrukturen am Beispiel einer Liste wird im Sonderdruck **SON-3** besonders eingegangen. Hier sei nur soviel wiederholt, daß mit der Version 2.0 der Foundation-Classes entweder typenrichtige Datenstrukturen gebildet werden können, wenn sie von der Klasse CObject abgeleitet wurden, wie das auf unsere Punkte vom Typ Cpoint zutrifft. Handelt es sich um andere Datentypen, muß statt CObList CPtrList verwendet werden, die aber lediglich void-Pointer verwaltet und beim Rücklesen aus der Liste einen cast-Operator benötigt. Ja, und void-Pointer sind gerade das, was man mit den sogenannten template-s vermeiden kann, etwas, was auch in den Foundation-Classes, d.h. besser gesagt in Visual-C++ wünschenswert wäre.]

Speichern eine Folge von Ereignissen in einer Liste

Wir wollen Punkte des Typs CPoint speichern. Die Klasse CPoint ist aber nicht von CObject abgeleitet, daher kann eine typenrichtige Liste mit CPoint-Objekten nicht unmittelbar gebildet werden. Dazu gibt es zwei Wege: 1. Erzeugen eines neuen Typs, z.B. myPoint, der von CObject abgeleitet ist und einen CPoint enthält und Benutzung der Liste CObList oder 2. Benutzung der Liste CPtrList und Verwendung des cast-Operators beim Lesen der Elemente.

Der Einfachheit halber benutzen wir die zweite Möglichkeit. Zunächst deklarieren wir eine Liste des Typs CPtrList in der Klassendefinition von CFcOView:

```
    CPtrList *liste;
```

und generieren die Liste im Konstruktor

```
CFcOView::CFcOView()
{
  ButtonDown = FALSE;
  liste = new CPtrList(sizeof(CPoint));
}
```

und vernichten sie im Destruktor

```
CFcOView::~CFcOView()
{
  delete liste;
}
```

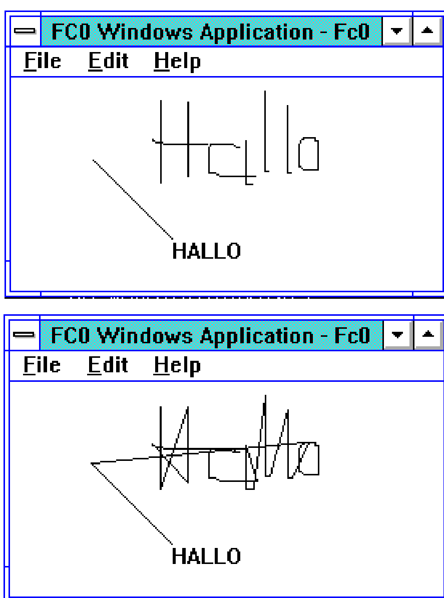
Der Kode für die Funktionen für die Mausbetätigung ändern wir wie folgt. Das Wiederholen der Zeichnung besorgt wieder die Funktion OnDraw(), in der die Liste ausgelesen wird.

```
void CFcOView::OnDraw(CDC* pDC)
{
    pDC->TextOut(100,100,"HALLO");
    pDC->MoveTo(100,100);
    pDC->LineTo(50,50);
```

```
    if (!liste->IsEmpty())
    {
      POSITION pos;
      for( pos = liste->GetHeadPosition(); pos != NULL; )
      {
        CPoint *p=(CPoint*)liste->GetNext( pos );
        pDC->LineTo(*p);
      }
    }
}

void CFcOView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CClientDC dc(this);

    // char S[100];
    // sprintf(S,"(%d,%d)",point.x,point.y);
    // dc.TextOut(point.x,point.y,S);

    if (!ButtonDown)
    {
      ButtonDown = TRUE;
      oldpoint=point;
      liste->AddHead(new CPoint(point));
    }
}

void CFcOView::OnLButtonUp(UINT nFlags, CPoint point)
{
    if (ButtonDown)
    {
      ButtonDown = FALSE;
    }
}

void CFcOView::OnMouseMove(UINT nFlags, CPoint point)
{
    if (ButtonDown)
    {
      CClientDC dc(this);
      dc.MoveTo(oldpoint);
      dc.LineTo(point);
      oldpoint=point;
      liste->AddHead(new CPoint(point));
    }
}
```

Zusammenfassung

Es ist nicht allzuviel des Kode, den man für ein einfaches Fenster schreiben muß; man ist aber auch noch weit davon entfernt, die Zusammenhänge wirklich zu verstehen.



□

C++-Beiträge in den PC-NEWS

PC-NEWS 20, S.22..25: Zeigt den Übergang von Strukturen zu Klassen an Hand eines Beispiels mit einer Struktur eines Gehaltsempfängers, bestehend aus einem Namen (String) und einem Gehalt.

PC-NEWS 21, S.21..23: Ableitung, Vererbung, virtuelle Funktionen, abstrakte Basisklassen

PC-NEWS 22, S.41: Zur Schreibweise in C, S.41..42: Friends, S 42..44: Überladen, S.44: Referenzvariable, S.45: Neues in C++, S.45..48: OOP mit C++, S.49: Wie lernt man am besten C++? S.49..51: C++ im Unterricht, Ein Beitrag zur Sprachendiskussion.

PC-NEWS 23, S.26..30: Materialien für den C-Unterricht, S.59..62: Entwicklung einer Klasse von Parametersubstitution über Parameterübergabe, Polymorphie, S.63..64: Erweiterungen von C in C++.

PC-NEWS 25, S.33..36: Eine kleine Grafikbibliothek, zeigt insbesonderen zum Abschluß den Begriff Polymorphie.

PC-NEWS 30: S.47..64: Eine IO-Klasse für hardwarenahes Programmieren.

PC-NEWS 35: S.40..44. ADIM-Skripten im Vergleich, darunter auch C++-Skriptum, sowie Fehlerkorrektur dazu.

PC-NEWS 35: S.52..53: const und static in C und C++.

PC-NEWS 35: S.54..68: Hauptspeicherzugriffe mit eigenen Klassen.

Weitere Hinweise für C-Programmierung

ADIM-Bände 40 (C), 50 (C++)
PC-NEWS-31, Von PASCAL zu C
Der schnelle Weg zum hohen C, PCN-SON-003

Real programmers programs never work right the first time. But if you throw them on the machine they can be patched into working in only a few 30-hours debugging sessions.

Real programmers don't use Fortran. Fortran is for wimpy engineers who wear white socks, pipe stress freaks, and crystallography weenies. They get excited over finite state analysis and nuclear reactor simulation.

Real programmers don't use COBOL. COBOL is for wimpy application programmers.

Real programmers never work 9 to 5. If any **real programmers** are around at 9 am, it's because they were up all night.

Real programmers don't write in BASIC. Actually, no programmers write in BASIC, after the age of 12.

Real programmers don't document. Documentation is for simps who can't read the listings or the object deck.

Real programmers don't write in Pascal, or Bliss, or Ada, or any of those pinko computer science languages. Strong typing is for people with weak memories.

Real programmers know better than the users what they need.

Real programmers think structured programming is a communist plot.

Real programmers don't use schedules. Schedules are for manager's toadies. Real programmers like to keep their manager in suspense.

Real programmers think better when playing adventure.

Real programmers don't use PL/I. PL/I is for insecure momma's boys who can't choose between COBOL and Fortran.

Real programmers don't use APL, unless the whole program can be written on one line.