```
  .  * prepare application to terminate
  .  */

  WinDestroyMsgQueue (hmq);      /* destroy main message queue */
  WinTerminate (hab);               /* de-register application */
}
```

The above example shows the necessary steps, a program has to go through to be run under OS/2 Presentation Manager.

1. *Initialization:* registration at PM, create message queue,...
2. *Message loop:* receive all messages for the application and process them
3. *Cleanup:* destroy message queue, de-register application,...

The Objective C PM class library provides a class, called `StdApp` to meet the purpose of standard initialization and message processing for every PM application. The following source code demonstrates how to use it:

```
#include <pm/pm.h>
.
.
main ()
{
  StdApp *application; /* pointer to our instance
                          of a StdApp class */

  application = [StdApp alloc]; /* create application object */

  [application init]; /* initialize application */
  .
  .
  .
  [application run];  /* process all messages */
  .
  .
  .
  [application free]; /* free application object */
}
```

As you can see, the first line of the sample includes `<pm/pm.h>`. This include file causes all include files of the PM class library to be read. After doing this, you can use all classes of the library and their methods without any restrictions.

And here a more compact version of the same part of code:

```
#include <pm/pm.h>
.
.
main ()
{
  StdApp *application = [[StdApp alloc] init];
  .
```

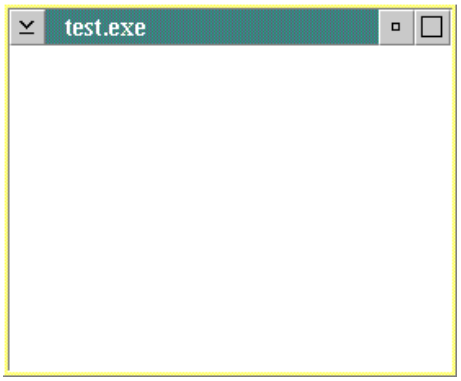

Figure 2.1: Sample application "test.exe"

```
  mainWindow = WinCreateStdWindow (HWND_DESKTOP,
                                   WS_VISIBLE,
                                   &createFlags,
                                   (PSZ) NEWCLASSNAME,
                                   (PSZ) "",
                                   0L,
                                   NULLHANDLE,
                                   1000,
                                   &clientWindow);

  while (WinGetMsg (hab,&qmsg,(HWND) NULL,0,0))
    WinDispatchMsg (hab,&qmsg);

  WinDestroyWindow (mainWindow);

  WinDestroyMsgQueue (hmq);
  WinTerminate (hab);
}
```

The following source code illustrates how much simpler the PM class library is to use than "normal" OS/2 PM API functions.

```
#include <pm/pm.h>

main ()
{
  StdApp    *application = [[StdApp alloc] init];
  StdWindow *mainWindow = [[StdWindow alloc]
                                initWithId: 1000
                                  andFlags: FCF_SIZEBORDER];

  [mainWindow makeKeyAndOrderFront: nil];
  [application run];

  [mainWindow free];
  [application free];
}
```

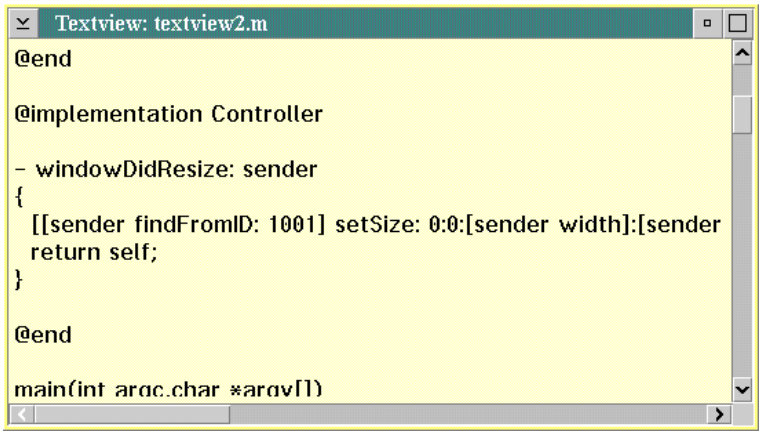

Figure 3.1: "Textview" application displaying it's own source code

```
main(int argc,char *argv[])
{
  FILE        *inputFile;
  struct stat  statbuffer;
  char        *contents;

  /*
   * check for command line arguments and
   * check given file (struct stat)
   */
  if (argc != 2) /* check for command line arguments,
                    must be exactly one */
    exit (-1);

  if (stat (argv[1],&statbuffer) < 0) /* check file */
    exit (-1);

  /*
   * open file and read contents to buffer
   */
  inputFile = fopen (argv[1],"r"); /* open text
                                      file read-only */

  contents = (char *) malloc (statbuffer.st_size + 1);
               /* allocate buffer */
  fread (contents,statbuffer.st_size,1,inputFile);
               /* read contents of file */
```

`inputFile` is a pointer to a file structure returned by `fopen`. `statbuffer` is used to retrieve information about the file using the C-Library function `stat`. Here the size of the file is stored.

After reading file information, `contents` is allocated via `malloc` and the file is opened and it's contents are read to `contents`.

Following this part of code, the initialization of the used PM classes takes place:

Just add some more variable declarations to the first section of the code:

After initializing this, the main window is shown and the size of the MLE window is adapted to the size of the main window, to fill it's complete interior:

```
/*
 * show window, set MLE size and  display contents of file
 */
[window makeKeyAndOrderFront: nil]; /* show window */
[mle setSize: 0:0:[window width]:
                  [window height]]; /* set MLE size */
[mle setText: contents]; /* display contents of file */
```

This code also sets the text displayed in the MLE window to be the buffer area `contents`.

3.2 Message loop

The main message loop is started by calling `[application run]`. As mentioned before, this method terminates, when the main window gets closed.

3.3 Cleanup

After the window was closed, all objects are destroyed and the previously allocated buffer area is freed again:

```
/*
 * free all resources
 */
free (contents); /* free contents buffer */
fclose (inputFile); /* close file */

[application free]; /* free application */
[window free]; /* free window */
```

Note, that `[window free]` automatically destroys all it's child windows, in our case, the MLE window.

3.4 Compilation

To compile this application, store the code shown in the following subsection to the file `textview.m` (it can be found in `\usr\samples\textview`) and type:

```
gcc -c textview.m
gcc -o textview.exe textview.o -lobjcpm -lobjc
emxbind -ep textview.exe
```

```
            initWithId: 1001
              andFlags: (WS_VISIBLE | MLS_READONLY |
                         MLS_HSCROLL | MLS_VSCROLL)
                    in: window];
  [window insertChild: mle]; /* insert MLE into window */

  /*
   * calculate title of window and set it
   */
  title = (char *) malloc (11 +  /* allocate buffer for title */
   strlen (argv[1]));
  sprintf (title,"Textview: %s",argv[1]); /* fill title buffer */

  [window setTitle: title]; /* set window title */

  free (title); /* free title buffer */

  /*
   * show window, set MLE size and  display contents of file
   */
  [window makeKeyAndOrderFront: nil]; /* show window */
  [mle setSize: 0:0:[window width]:
                   [window height]]; /* set MLE size */
  [mle setText: contents]; /* display contents of file */

  /*
   * run application
   */
  [application run];

  /*
   * free all resources
   */
  free (contents); /* free contents buffer */
  fclose (inputFile); /* close file */

  [application free]; /* free application */
  [window free]; /* free window */
}
```

If you compile this program you will see, that the main window is resizable, but the MLE window inside the window remains the same size, whatever size it's parent window is.

The rest of this chapter shows how an object can be automatically notified, when the main window resizes, to adapt the size of the MLE window.

3.5 Delegate objects

One of the main advantages of Objective C compared to most other object-oriented programming languages is the possibility to check at runtime, if an object implements a specific method. This provides a simple way for objects to send messages to other objects, if these messages can be processed, to notify of some special occurrence.

By calling `[sender findFromID: 1001]` the method queries a pointer to an instance of `Window` or one of it's subclasses. This window must be a child window of `sender` and have the OS/2 PM identifier `1001`.

Using this method returns a pointer to the MLE window's associated Window object. This method is sent a `setSize::::` message to adapt it's size to the size of the sending window.

`setSize::::` takes the coordinates of the lower left corner of the window (the first and second parameters) relative to it's parent's lower left corner. The last two parameters represent the *width* and *height*, the window should be resized to.

The lower left corner of the MLE window should be the same as the lower left corner of it's parent, `(0/0)`. The width and height of the MLE window is queried from the sender by using the appropriate methods `width` and `height`.

As this method has a return type of `id`[1], `self` is returned on successful completion of the method.

The following section shows the modified source code of `textview.m`, which is stored in the directory `\usr\samples\textview` under the name `textview2.m`.

3.6.1 Modified version of Textview: "textview2.m"

```
#include <pm/pm.h>
#include <io.h>
#include <sys/types.h>
#include <sys/stat.h>

@interface Controller : Object
{
}

- windowDidResize: sender;

@end

@implementation Controller

- windowDidResize: sender
{
  [[sender findFromID: 1001] setSize:
         0:0:[sender width]:[sender height]];
  return self;
}

@end

main(int argc,char *argv[])
{
  StdApp      *application;
  StdWindow   *window;
  Window      *mle;
  Controller  *controller;
  FILE        *inputFile;
  struct stat  statbuffer;
```

[1] `id` is a pointer to a generic Objective C object

```
  free (title); /* free title buffer */

  /*
   * show window and display contents of file
   */
  [mle setText: contents]; /* display contents of file */
  [window makeKeyAndOrderFront: nil]; /* show window */

  /*
   * run application
   */
  [application run];

  /*
   * free all resources
   */
  free (contents); /* free contents buffer */
  fclose (inputFile); /* close file */

  [application free]; /* free application */
  [window free]; /* free window */
  [controller free]; /* free controller */
}
```

3.7 Sample makefiles

In the directory `\usr\samples\makefile` you can find a sample `makefile` together with the used make-include file `makefile.preamble`.

To use this makefile, just copy `makefile` and `makefile.preamble` to your application directory and fill in the correct places in `makefile`.

1. Add the name of your application file to the line containing `APPLICATION =` (including the suffix `.exe`).
2. Add the names of your object files to the line containing `OBJECTS =`.
3. Add all OS/2 resource files (the files with extenstion `.res`) to the line containing the statement `RESOURCES =`.

This makefile was written for *GNU make*. Possible targets are:

- no target $\cdots$ this automatically compiles and links the application program
- *dep* or *depend* $\cdots$ check all files for dependencies and create a `.depend` file, which is automatically included.
- *clean* $\cdots$ removes all temporary files (compiled resources, application program, object files, core dump file, ...)

```
# Makefile for PM programs using Objective C class library

include Makefile.preamble
```

The menu *File* has id `2000`, the menu items *Open...* and *Exit* the ids `2001` respectively `2002`.

Between the two menu items *Open...* and *Exit* a separator item should be inserted.

The resulting menu is shown in figure 4.1.

To load this menu, just create a resource definition file, type in the menu declaration and use `RC.EXE` to produce a binary resource file. When linking the application, don't forget to specify the name of the binary resource file (just like any other object file).

When creating the main window of Textview, binary or `FCF_MENU` with the given flag `FCF_SIZEBORDER`. When creating the window, the menu resource will be loaded and displayed in the window's actionbar. Which menu will be loaded depends on the OS/2 PM identifier of the window, which you specify at creation. It must be the same as the identifier specified in the resource definition for the menu (in our case, it's `1000`).

4.2 Dialogs

Using a dialog editor, you can easily create dialog windows and either store a resource definition file or a binary resource file to disk.

Just like normal windows, dialog windows are created by the application using the appropriate dialog window class `StdDialog`. In addition to creating the window object, the contents of the dialog are loaded from the main resource file linked to the application.

After creation, dialog windows can be displayed using `makeKeyAndOrderFront:`. In addition to normally displaying the dialog windows, which causes the dialog to run non-modal, you can also run a dialog modal for a given parent window. Using `runModalFor:` the dialog window is displayed, but working with it's parent window, which it runs modal for, is not possible untill the dialog window gets closed again (*dismissed*.

4.3 Command bindings

After a menu bar has been created, or a dialog window was loaded from a resource file, some of the menu items or window objects in the dialog send command messages to their owner. By processing these messages, the program can react to user actions.

Using the classes provided by this library, you can bind command messages to designated methods of an object. When a special command message was sent to a window, the appropriate method of an object gets called.

All methods, which can be bound to command messages must be of the form `nameOfMethod: sender`. The parameter `sender` stores a pointer to the sending instance of a `StdWindow` or a `StdDialog`, which calls the method.

Command messages can be bound to objects and appropriate methods using `bindCommand: withObject: selector:`. The first parameter of this method is the identifier of the PM object, which posts command messages, the second is a pointer to the Objective C object, which implements the method to be called, the third and last is the *selector* of the method to be called. The selector of a method can be queried using `@selector(nameOfMethod)`.

To bind the command message sent by the menu item *Exit*, which has an OS/2 PM id of `2002` to the `performClose:` method of the window object, just insert

```
[window bindCommand: 2002
         withObject: window
        andSelector: @selector(performClose:)];
```

```
  StdApp     *application = [[StdApp alloc] init];
  StdDialog  *mainDialog = [[StdDialog alloc]
                                initWithId: IDD_MAIN];
  Controller *controller = [[Controller alloc] init];

  [mainDialog createObjects];

  [mainDialog bindCommand: DID_OK withObject: controller
                selector: @selector(plot:)];

  [mainDialog runModalFor: nil];

  [controller free];
  [mainDialog free];
  [application free];
}
```

`[[StdDialog alloc] initWithId: IDD_MAIN]` creates a dialog object and loads it's binary resource template from the main binary resource file. The dialog id is `IDD_MAIN`.

`[mainDialog bindCommand: ...` binds the command message sent by the pushbutton, which has id `DID_OK` to the `plot:` method of the object `controller`.

`[mainDialog runModalFor: nil]` runs a modal dialog. Normally, this dialog is run modal for a certain window, but when `nil` is specified, this only causes the method to wait for termination of the dialog window.

The class `Controller` itself has to load the program `gnuplot.exe` and send it appropriate commands to plot the given function.

The class implements one instance variable, `gnuplot` to store a file handle to the gnuplot program, and three methods, `init` to open the plotting program, `free` to close it at the end and `plot:`, which does the plotting work. The following interface declarations is stored as `controller.h` in `\usr\samples\gnuplot`.

4.4.2 "controller.h", Gnuplot PM interface

```
#include <pm/pm.h>
#include <stdio.h>

@interface Controller : Object
{
  FILE *gnuplot;
}

- init;
- free;
- plot: sender;

@end
```

The implementation uses some of the unix-like features of the *emx C-Library*.

```
- init
{
```

```
  return self;
}
```

The following section shows the complete source code of the implementation of the class `Controller`.

4.4.3 "controller.m", Gnuplot PM interface

```
#include "Controller.h"
#include "gnuplot.h"

@implementation Controller

- init
{
  [super init];
  gnuplot = popen ("gnuplot.exe","w");
  return self;
}

- free
{
  pclose (gnuplot);
  return [super free];
}

- plot: sender
{
  char   *string;
  char   *leftX,*rightX;
  double  left,right;

  string = [[sender findFromID: IDD_PLOTSTRING] text: NULL];

  if ([[sender findFromID: IDD_RANGECHECK] checked]) {
    leftX = [[sender findFromID: IDD_LEFTX] text: NULL];
    rightX = [[sender findFromID: IDD_RIGHTX] text: NULL];

    if ((sscanf (leftX,"%lf",&left) == 1) &&
        (sscanf (rightX,"%lf",&right) == 1) &&
        (right > left)) {
      fprintf (gnuplot,"plot [%lf:%lf] %s\n",left,right,string);
    } else
      fprintf (gnuplot,"plot %s\n",string);

    free (leftX);
    free (rightX);
  } else
    fprintf (gnuplot,"plot %s\n",string);

  fflush (gnuplot);
  free (string);
```

```
    CONTROL "",
            IDD_RANGECHECK, 179, 30, 10, 10, WC_BUTTON,
            BS_AUTOCHECKBOX | WS_TABSTOP | WS_VISIBLE
    CONTROL "Plot",
            DID_OK, 145, 10, 40, 14, WC_BUTTON,
            BS_PUSHBUTTON | BS_DEFAULT | WS_TABSTOP | WS_VISIBLE
  }
}
```

`gnuplot.dlg` defines a dialog template for dialog `IDD_MAIN`. This template is normally written by a dialog editor.

```
#define IDD_MAIN             3000
#define IDD_PLOTSTRING       3001
#define IDD_PLOT             3002
#define IDD_LEFTX            3003
#define IDD_RIGHTX           3004
#define IDD_RANGECHECK       3005
```

The include file `gnuplot.h` is also normally created by the used dialog editor. It contains definitions for the constants used in the resource definition file.

The binary resource file can be created using `RC.EXE` by typing the command sequence `rc -r gnuplot.rc` at an OS/2 command line. This creates the binary resource file `gnuplot.res`, which can be linked to the application as the main resource file.

Compare the following `makefile` to the makefile template described in section 3.7 at page 24 to realized, how to fill in these templates.

```
# Makefile for PM programs using Objective C class library

include Makefile.preamble

ifeq (.depend,$(wildcard .depend))
include .depend
endif

APPLICATION = plot.exe
OBJECTS = plot.o controller.o
RESOURCES = gnuplot.res

all: $(APPLICATION)

depend dep:
$(CPP) -MM *.m > .depend

$(APPLICATION): $(OBJECTS) $(RESOURCES)
$(CC) -o $(APPLICATION) $(OBJECTS) $(RESOURCES) \
                -lobjcpm -lobjc
emxbind -ep $(APPLICATION)
$(STRIP) $(APPLICATION)

clean:
rm -rf $(OBJECTS) $(RESOURCES) $(APPLICATION) core *~
```

```
  myDBFile = [[DBFile alloc]     // allocate and initialize
              init: "test.dbf"]; // data file "test.dbf"

  .  /*
  .   * here the database file can be used, records can be
  .   * read, modified or written back.
  .   */

  [myDBFile free]; // close data file and free resources
}
```

Access to a `DBFile` is record-oriented. Each `DBFile`-object contains a buffer large enough to store exactly one record.

This *record buffer* can be filled with a record already stored in the database file and can be written back to disk. Additionally the record can be modified by the application program using the database library.

The program fragment shown above allocates and initializes a `DBFile` object and opens a database file calles *test.dbf* for reading and writing.

This file can be found in `\usr\samples\dbtest` after installing the sample files.

It defines a simple database file with two fields. The first field is called *NAME*. It can hold a string with a maximum length of 30 characters, the second is called *PHONE*, and is able to store a string with a maximum length of 20 characters.

The following records are already stored in the database file:

Nr.	NAME	PHONE	remark
0	Joe	23987-3	
1	Frank	6334589	
2	Sue	523593	
3	Michael	9845-43	deleted
4	Kurt	2543	

As you can see, the database file contains five records. The fourth is deleted. In the following, we will see, how access to this file is achieved.

`DBFile` provides a method to read in a record from the associated database file. This method is called `readRecord:`. This method takes one parameter, the position, where in the file the record is stored. After reading the record, the data is copied into the record buffer, where it can be accessed by the application program.

The access to the data is accomplished by using methods of objects of type `DBField` or one of its subclasses. `DBField` implements the methods `string` to read the data stored in the record buffer and `setString:` to write data into the record buffer.

When initializing a `DBFile` object, all needed `DBField` objects are automatically created and can be used thereafter by the application program.

Access to a special `DBField` object is provided through the `field:` method of `DBFile`. So, if you want to print the *NAME* field of the currently loaded record, you could do this:

```
printf ("Name: %s\n",[[myDBFile field:0] string])
```

Now, here's a simple program (it is stored as `dbtest1.m` in `\usr\samples\dbtest`) which reads all records and prints the name and the phone number of each entry:

```
#include <db/db.h>

main ()
{
  DBFile *myDBFile = [[DBFile alloc] init: "test.dbf"];

  if ([myDBFile findFirst]) {
    printf ("NAME                          PHONE              \n");
    printf ("=================================================\n");

    do {
      printf ("%-30s %s\n",[[myDBFile field:0] string],
                   [[myDBFile field:1] string]);
    } while ([myDBFile findNext]);
  }
  [myDBFile free];
}
```

This program checks first, if an active record exists, and only if this condition is met, the title lines are printed. Thereafter the record itself is printed and the next record is read. This procedure of printing a record and reading a new record is continued until `findNext` notifies the program that no more active records exist.

Using the `delete` method, you can mark a record as deleted. But take care. The changes are not written to the database file automatically. The record is only marked as deleted in the record buffer. After modifying a record, you have to write it to disk again using `replace`.

Deleting the second record, the name and phone number of Frank, would look like this:

```
  [myDBFile readRecord:1]; // read record
  [myDBFile delete];       // mark record as deleted
  [myDBFile replace];      // write changes to database
```

As simple as deleting a record, you can mark an already deleted record as active again. Just use `undelete`. The following three lines mark the fourth record, Michaels name and phone number, as active again:

```
  [myDBFile readRecord:3]; // read record
  [myDBFile undelete];     // mark record as active
  [myDBFile replace];      // write changes to database
```

`menu` shall print all active records stored in the database and display a simple menu, where the user of the application can choose, what he wants to do.

`printInfo` is used to print a list of all active records in the database file.

`deleteRecord` and `undeleteRecord` will ask the user, which record he wants to mark as deleted or active, and then commit the task of deleting or activating the specified record.

`addRecord` prompts the user for the necessary data (NAME, PHONE), which is to be stored in a new record and then appends this newly created record to the database file.

`modifyRecord` on the other hand allows the user to change the data of a record already stored in the database file.

The program will look like this:

```
#include <db/db.h>

@interface AddressDatabase : Object {
 .
 .
 .
@end

@implementation AddressDatabase
 .
 .
 .
@end

main ()
{
  AddressDatabase *mydb = [[AddressDatabase alloc] init];
  int chosen;

  while ((chosen = [mydb menu]) != 5) {
    switch (chosen) {
    case 1:
      [mydb addRecord];
      break;
    case 2:
      [mydb modifyRecord];
      break;
    case 3:
      [mydb deleteRecord];
      break;
    case 4:
      [mydb undeleteRecord];
    default: ;
    }
  }

  [mydb free];
}
```

As you can see, the main function only creates and initializes the object `mydb` and then displays

6.3 Displaying the menu

To provide an easy to use user-interface – which is by no means as elegant as a PM application program – a simple menu is printed, to let the user choose, what actions he wants to do on the database file.

This user-interface is implemented in the method `menu`, which displays all active records and a menu, and then lets the user choose, which program function he would like to execute. `menu` returns the number of the chosen program function, which are shown in the following table:

Nr.	Program function
1	add new record
2	modify existing record
3	delete record
4	mark record as active
5	exit program

This method – just a simple sequence of C statements, without using some of the methods provided by the database library – is implemented like this:

```
- (int) menu
{
  int chosen;

  printf ("\n             Address Database\n\n");

  [self printInfo];

  printf ("\n    (1) ... add Record     (2) ... modify Record\n");
  printf ("    (3) ... delete Record  (4) ... restore Record\n");
  printf ("    (5) ... quit Program\n");
  printf ("\n         What shall I do? ");
  scanf ("%d",&chosen);
  printf ("\n");

  return chosen;
}
```

6.4 Deleting a record

```
- deleteRecord
{
  long recNumber;

  printf ("\nWhich record shall I delete? ");
  scanf ("%d",&recNumber);

  [database readRecord: recNumber];

  if ([database deleted]) {
    printf ("\nThis record is already deleted!\n");
    return nil;
```

6.6 Adding a new record

```
- addRecord
{
  char nameField[31];
  char phoneField[21];

  printf ("\nName:  ");
  scanf ("%s",nameField);

  printf ("Phone: ");
  scanf ("%s",phoneField);

  [database clear];
  [[database field: 0] setString: nameField];
  [[database field: 1] setString: phoneField];
  [database append];

  return self;
}
```

6.7 Modifying an existing record

```
- modifyRecord
{
  long recNumber;
  char nameField[31];
  char phoneField[21];

  printf ("\nWhich record would you like to modify? ");
  scanf ("%d",&recNumber);

  [database readRecord: recNumber];

  if ([database deleted]) {
    printf ("\nThis record is deleted! You can't modify it!\n");
    return nil;
  }

  printf ("\nName:  ");
  scanf ("%s",nameField);

  printf ("Phone: ");
  scanf ("%s",phoneField);

  [database clear];
  [[database field: 0] setString: nameField];
  [[database field: 1] setString: phoneField];

  [database replace];

  return self;
```

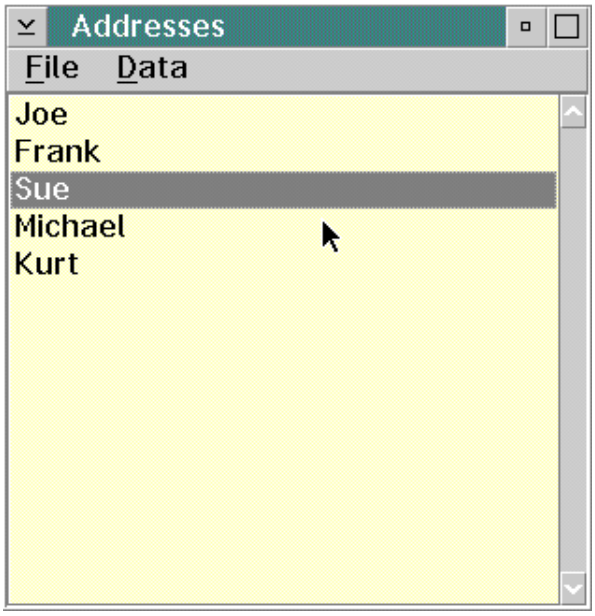

Figure 7.1: Main window of address database

The second menu used by this program is called *Data*. Here all actions concerning data manipulation, such as adding new records, deleting records etc. can be found.

The menu items found here are *New...* to add a new record to the end of the database, *Edit...* to change the data stored for the selected record in the listbox, *Delete* to delete a record from the database and *Info...* to display the information associated with the selected record in the listbox control.

This program will only display the records which are not marked as deleted in the database file. Deleting a record only marks it as deleted, but by means of this program, the record can not be undeleted again.

7.3 Database file

The program automatically opens a database file called `ADDRESS.DBF` in the current directory at startup and writes the changes to this file when exiting the application.

An empty database template file is provided in `\usr\samples\address` with the name `EMPTY.DBF`.

7.4 Classes used in the application

In addition to the standard PM and database classes, only one class is created anew. This class is called `Controller`. This class implements all necessary methods used by the program, as *record insertion* etc.

The methods defined here can be divided into three parts. The *initializing* methods are used to open the database file and read in all records at program start. The *database access* methods will be called directly from the appropriate menu items and all display dialog windows to change or view the data stored. At last, the *delegate* methods are implemented, which are used to react to some user actions, such as resizing the window and exiting the program.

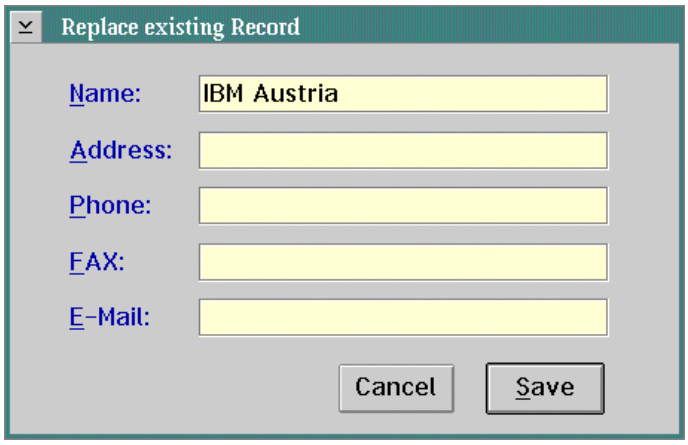

Figure 7.3: Dialog window for editing already existing records

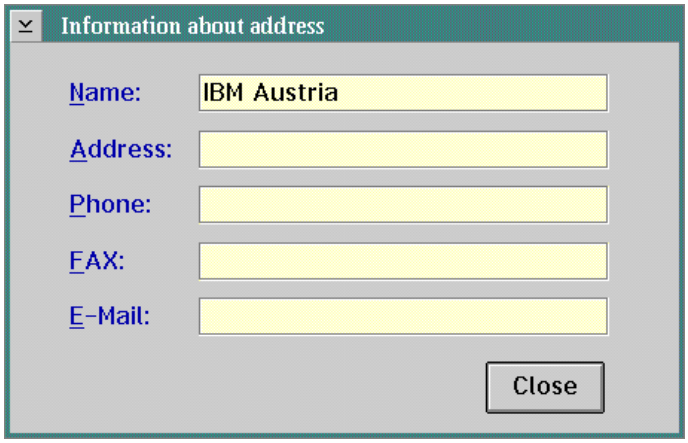

Figure 7.4: Dialog window used to display all information stored in a record

is shown in figure 7.3) and for displaying the information associated with a record (see figure 7.4).

The instance variables defined as `ids` are initialized to provide easy and fast access to the dialog items themselves. they are initialized at dialog creation time.

`database` is a pointer to a `DBFile` object which is used to retrieve and save records from and to the database file itself.

`recordList` is a list object to enable the application to hold more than one record at a time in memory. This provides faster and easier access to the single records. When the program is started, all record information is retrieved into this object and at exit time, all modified or new records are written to the database again.

7.4.2 Initializing methods

The initializing methods are used to initialize the object itself and all objects used by this one. Additionally, the only destructor method, `free`, is also described here.

- `init` is used to create all dialog windows in memory. Here all instance variables of this object are initialized. Additionally the database file is opened and the record list object is

Last, the database object is created and initialized and the database file `address.dbf` is opened. The record list used to hold all records in memory is created and the method returns to the caller.

`free` only frees all objects associated or created with/by this object. The simple source code for this method looks like this:

```
- free
{
  [database free];
  [insertRecord free];
  [replaceRecord free];
  [infoRecord free];

  return [super free];
}
```

The last one of the initializer methods, called `readList:` is used to fill the record list with all data in the database. This can simply be accomplished by calling `[recordList fetchAllRecords]`.

Then the first data field of each record is extracted from the list and displayed in the listbox in the main window.

```
- readList: sender;
{
  ListBox *nameListBox = [sender findFromID: IDD_PUSHBUTTON1];
  int i;

  [recordList fetchAllRecords];

  for (i = 0;i < [recordList count];i++) {
    [[recordList findRecordAt: i] copyToDB];
    [nameListBox insertItem: LIT_END text: [[database field: 0] string]];
  }

  return self;
}
```

You can see that a simple `for` loop is used to visit all records in the list. The line `[[recordList findRecordAt: i] copyToDB]` copies the record information stored in the record list of record `i` to the database buffer. Then the string stored in the first data field is appended to the listbox as a new listbox item.

7.4.3 Database access methods

To modify the information stored in the database file and displayed in the listbox, four methods, called `insert:`, `replace:`, `info:` and `delete:` have been implemented. These methods are called directly as response to user actions. Therefor every method has exactly one parameter, called `sender`.

The most simple method is `delete:`. This method shall display a message box querying the user, if he really wants to delete the selected record. If no record was selected, `delete:` has to return `nil` without further processing.

```
  .

  [[recordList findRecordAt: selected] copyToDB];

  [infoName setText: [[database field: 0] string]];
  [infoAddress setText: [[database field: 1] string]];
  [infoPhone setText: [[database field: 2] string]];
  [infoFax setText: [[database field: 3] string]];
  [infoEMail setText: [[database field: 4] string]];

  [infoRecord runModalFor: sender];

  return self;
}
```

This copies the data associated with the selected record from the record list to the database buffer. Afterwards the text in the data fields is displayed in the entry field objects, and the dialog window is run modal.

`replace:` is used to edit the data for an already existing record. In the beginning, the corresponding entry fields must be filled with the strings stored in the data fields of the selected record, as shown previously.

After running the dialog modal, the data must be copied back from the entry fields to the database buffer and then into the record list again.

```
- replace: sender
{
  .
  .
  .
  [replaceRecord runModalFor: sender];

  if ([replaceRecord result] == DID_OK) {
    nameBuffer = [replaceName text: NULL];
    addressBuffer = [replaceAddress text: NULL];
    phoneBuffer = [replacePhone text: NULL];
    faxBuffer = [replaceFax text: NULL];
    emailBuffer = [replaceEMail text: NULL];

    [[database field: 0] setString: nameBuffer];
    [[database field: 1] setString: addressBuffer];
    [[database field: 2] setString: phoneBuffer];
    [[database field: 3] setString: faxBuffer];
    [[database field: 4] setString: emailBuffer];

    [[recordList findRecordAt: selected] copyFromDB];
    [[recordList findRecordAt: selected] replace];

    [nameListBox deleteItem: selected];
    [nameListBox insertItem: selected text: nameBuffer];

    free (nameBuffer);
    free (addressBuffer);
    free (phoneBuffer);
```

For a more complete description of the methods used here and others provided by the database library, see the reference manual.

7.5 The main function of the application

The main function of the application can be found in the file `addresses.m`. It only performs the standard actions, as creating and initializing the objects, and after creating all necessary objects but before starting execution of the application, the record list is read in by calling `[controller readList: mainWindow]`. The complete source code is shown below.

Here you can see that all menu items which are used are bound directly to the previously described methods of the class `Controller`.

```
main()
{
  StdApp *addresses = [[StdApp alloc] init];
  MainWindow *mainWindow = [[MainWindow alloc] initWithId: IDD_MAIN
                                andFlags: (FCF_MENU | FCF_ACCELTABLE |
                                           FCF_SIZEBORDER)];
  Controller *controller = [[Controller alloc] init];

  [mainWindow setTitle: "Addresses"];

  [mainWindow setDelegate: controller];
  [mainWindow bindCommand: IDM_EXIT withObject: controller
                selector: @selector(closeApp:)];
  [mainWindow bindCommand: IDM_NEWAD withObject: controller
                selector: @selector(insert:)];
  [mainWindow bindCommand: IDM_EDITAD withObject: controller
                selector: @selector(replace:)];
  [mainWindow bindCommand: IDM_INFOAD withObject: controller
                selector: @selector(info:)];
  [mainWindow bindCommand: IDM_DELETEAD withObject: controller
                selector: @selector(delete:)];

  [mainWindow createObjects];

  [mainWindow insertChild: [[ListBox alloc] initWithId: IDD_PUSHBUTTON1
                andFlags: WS_VISIBLE | WS_TABSTOP
                      in: mainWindow]];

  [controller readList: mainWindow];

  [mainWindow makeKeyAndOrderFront: nil];

  [addresses run];

  [mainWindow free];
  [addresses free];
}
```

```
  // => here goes the container stuff

  [application run];

  [mainwindow free];
  [application free];
}
```

8.1.2 Step one: Creating the container

The first thing, we will have to do, is to declare a new variable of type `Container *`, which we will use to access the container control.

So, add `Container *container` to the declaration section of the main function.

Creation of the object itself is just as simple, as creating a Button object. Using `initWithId: andFlags: in:` a new object, previously allocated with `alloc`, will be initialized.

```
  container = [[Container alloc] initWithId: 1001
                                  andFlags: (CCS_MINIRECORDCORE |
                                             WS_VISIBLE)
                                       in: mainwindow];
  [mainwindow insertChild: container];
  [container setSize: 0:0:[mainwindow width]:[mainwindow height]];
```

The container object is created using the flags `CCS_MINIRECORDCORE` and `WS_VISIBLE`. The second one only tells the container to be visible. `CCS_MINIRECORDCORE` is used to specify, which kind of data will be stored in the container. At the moment, only this style is supported. Don't use the flag `CCS_RECORDCORE`.

Then the container set a child window of `mainwindow`, and the size of the newly created control is set according to the size of it's parent window.

8.1.3 Step two: Inserting data

After creation, it's possible to insert data into the container. As stated before, all kinds of Objective C objects can be stored in a container. For simplicity, only instances of `Object` are used here.

Insertion itself can be performed using three distinct methods, `insertObject:`, `insertObject: withTitle:` and `insertObject: withTitle: andIcon:`. If you are going to display the data stored in the container as icons, you will normally use the third method shown above, because it let's you set all information, which will be displayed later.

- `insertObject:` ··· inserts a new object into the container. In Icon View, no Icon text will be displayed. The Icon used for this object will be the clock-mouse pointer.
- `insertObject: withTitle:` ··· analogous to `insertObject:` an object is inserted into the container, but here, a title text can be specified.
- `insertObject: withTitle: andIcon:` ··· here all data, which will be displayed in Icon View, can be specified. The Icon parameter must be a valid Icon resource.

We will use all three methods described above to insert three different objects into the container. Here goes the code:

```
  [container setSize: 0:0:[mainwindow width]:[mainwindow height]];

  [container insertObject: [[Object alloc] init]]; // first object

  [container insertObject: [[Object alloc] init]
              withTitle: "Title of object"];     // second object

  [container insertObject: [[Object alloc] init]
              withTitle: "Another object"
                andIcon: WinQuerySysPointer (HWND_DESKTOP,
                                             SPTR_APPICON,
                                             FALSE)]; // third object
  [container arrange];

  [application run];

  [mainwindow free];
  [application free];
}
```

Just compile and link this program:

```
gcc -o cont1.exe cont1.m -lobjcpm -lobjc
emxbind -ep cont1.exe
```

Figure 8.1 on the page before shows the output produced by this application.

8.2 Using the Details View

The detail's view of a container object looks like a multi-column Listbox. In addition to the features of a simple Listbox control, the container class makes it possible to edit the data displayed by simply selecting it and entering the new data.

Container objects in detail's view are widely used across the user interface provided by the Workplace Shell. Each folder object can be opened in detail's view to provide access to all data associated with the data or program objects stored inside.

As mentioned before, a Container object in detail's view looks like a multi-column Listbox. So – in contrast to all other views of Container objects – the data is always some kind of sorted. Every record (item) stored in a Container is displayed in one line of the Container. Every data field of the items is displayed in one column.

8.2.1 Creating columns

So first we have to specify the number of columns to be displayed in the Container object. After allocation and initialization using `alloc` and `initWithId: andFlags: in:` the program must create all columns needed for display. This is done using `addColumn:`.

`addColumn:` takes one parameter of type `(char *)`, which is the column title. This column title normally is displayed at the head of the column, in the topmost line of the display area of the Container object.

Adding columns at the moment is done one column at a time. So creating more than one column results in calling `addColumn:` for each of them. Take care that the created column is always

```
}

- init;

- setFieldData: (ULONG) field withString: (char *) aString;
- (char *) fieldData: (ULONG) field;

@end
```

As no data will be allocated dynamically, the class does not have to implement a `free` method.

The implementation is just as simple.

```
@implementation ContainerObject
- init
{
  [super init];

  fields[0][0] = 0x0;
  fields[1][0] = 0x0;
  fields[2][0] = 0x0;
  fields[3][0] = 0x0;

  return self;
}

- setFieldData: (ULONG) field withString: (char *) aString
{
  if (field > 3)  // field number not valid ?
    return nil;

  strncpy (fields[field],aString,29); // copy at most 29 chars
  fields[field][29] = 0x0;            // last character is NULL

  return self;
}

- (char *) fieldData: (ULONG) field
{
  if (field > 3)  // field number not valid ?
    return NULL;
  else
    return fields[field];
}

@end
```

Note that the method `init` is only implemented to initialize the four strings, so the data displayed is always valid.

`setFieldData: withString:` just copies at most 29 characters into the buffer area for the string. As `strncpy ()` does not append a `NULL` character if the source string is longer than the specified maximum size, the 30^{th} character is set to `0x0`.

`fieldData:` just returns a pointer to the internal buffer area of the field data referenced by `field`.