Amadeus Toolbox
Programming Guidelines

Martin Hairer

March 19, 2000

Contents

Introduction

Te **Amadeus Toolbox** is a tool for **Amadeus II** users who want to develop their own external filters. It provides a collection of classes that allow you to easily construct a small user interface and to manipulate a sound. In order to do this, you have to possess a registered version of CodeWarrior or any other development environment allowing to compile C++ projects.

I will explain in detail how you are supposed to design your external filter in order to be compatible with **Amadeus II**. This version of the guidelines does not contain a detailed description of all available classes yet. I plan to do this for a future release.

This toolbox is valid for **Amadeus II v2.2**. None of the external filters developed with it will run under older versions of **Amadeus**. I will nevertheless make an effort to ensure as much backward compatibility as possible with future versions of **Amadeus**. If it nevertheless happended that I change the format of the external filters, you will be notified so that you can recompile your filters with the new version of the **Amadeus Toolbox**.

Licence limitations

In order to use the **Amadeus Toolbox**, you have to be a registered user for **Amadeus** or **Amadeus II**. This gives you the right to use any of the libraries contained in this Toolbox for **personal use**. This means that you are not allowed to sell any external filter you developed. Nevertheless, you are encouraged to distribute them and/or to send me a copy of them, so that I may include them with future releases of **Amadeus II**.

If you want to develop external filters for **Amadeus II** for **commercial purposes**, you have to find a satisfactory agreement with me. Please contact me personally at

`Martin.Hairer@math.unige.ch`.

1 First steps

Let us try to build a first example. If you own Metrowerks CodeWarrior 3 or 4, open the project “External” located in

Amadeus Toolbox:Projects:CodeWarrior 3.

If you own Metrowerks CodeWarrior 5, open the project “External” located in ...:CodeWarrior 5. Type command-M to build the example. It will create a file called “Example” in the same folder than the project. To see what “Example” does, move it somewhere into the Amadeus folder and launch **Amadeus II**. A menu item **Example** will appear in the **Effects** menu. There are two other examples that go with this release: the “Reverse” filter and the “Set Pitch...” filter. You are strongly encouraged to try to understand those examples in order to be able to write your own external files. To compile them, replace the External.c and External.rsrc files by the corresponding files in the project.

If you work with another development tool than Metrowerks CodeWarrior, you have to create a new project file. Follow the instructions contained in the Project Settings file located in the Amadeus Toolbox:Projects folder.

2 The resource file

In this section, I will explain how you are supposed to design your ressource file in order to be compatible with **Amadeus II**.

2.1 Obligatory resources

For the moment being, only two resources have to be present in an external filter.

The code. The code of the resource has to be contained in a resource of type PROC with resource ID 500. If you use one of the CodeWarrior projects included in the toolbox, this will be done automatically.

The about string. Your external filter will appear in the **About External Files** submenu of **Amadeus II**. The string you want to appear when your filter is selected has to be stored in a resource of the type STR# with resource ID 401.

2.2 Error messages

If the user makes an illegal manipulation, you may want your external filter to display error messages. These messages have to be stored in a resource of the type STR# with resource ID 401.

2.3 Menus

You may want **Amadeus II** to display a pop-up menu next to your external filter in the **Effects** menu. This is done automatically if you include a resource of the type MENU with resource ID 400.

You can also use some pop-up menus inside your dialog or modal windows. They have to have their resource ID’s bigger or equal to 500 in order not to mix up with the pop-up menus of **Amadeus II**.

2.4 Pictures

If you want to display pictures inside some of your dialogs or modal windows, they have to have their resource ID's bigger or equal to 200.

2.5 Dialogs

A dialog consists of a resource of the type DLOG and one of the type DITL. Both have to have the same resource ID, which has to be between 200 and 255, in order not to mix up with those of **Amadeus II**. The colors, as well as the window types specified in the DLOG resource are irrelevant, since **Amadeus II** overrides them anyway. But you can specify the dialog's window title in the DLOG resource.

When you design your dialog window, always keep in mind that **Amadeus** usually displays every text, button and so on in Geneva 9. You can change this from inside your code, but it is not recommended, since your user interface would not look like that of other external filters.

Since the DITL resource is interpreted by **Amadeus II**, not by the system, it has to be designed in a particular way.

Buttons. The **OK** button always has to have the item number 1. If a **Cancel** button is present, it has to have item number 2. You can also design a dialog box without any **OK** or **Cancel** button, but this is not recommended. You then have to treat it in a non-standard way inside your code.

Radio buttons/Check boxes. They are treated in a very standard way.

Editable text. If you use the standard fonts, always set the height of your edit text fields to 14 pixels. This will yield the nicest results.

Static text. The same remark as for editable text fields applies. Moreover, do not forget to set the state of those fields to **Enabled**.

Controls and pop-up menus. They have to be created from inside your code. You may want to create a **user item** to extract their exact location from your code.

Frames. They have to be a Picture item with resource ID 1001. You do not have to have a corresponding PICT resource. Set their state to **Enabled**.

3 The simplest code

The simplest possible external filter just produces a system beep when you select the corresponding menu item in the "Effects" menu. Its code looks like this.

```
#include "CExternalStuff.h"

#include "NewFilter.h"

pascal void main(CSoundBase* theSound, CWindowBase* theWindow, \
   CAmadBase* theApplication, CResFileBase* theFile, short theItem, long* myData)
{
   switch (theItem) {
      case kInitFilter:
      case kReleaseFilter:
```

```
        case kAboutItem:
          break;
        case kEnableItem:
          *myData = 1;
          break;
        case 0:
          SysBeep(1);
          break;
        }
    }
```

Let us explain this code in detail. The line

```
#include "CExternalStuff.h"
```

includes all the declarations needed to use the **Amadeus Toolbox**. The line

```
#include "NewFilter.h"
```

includes all the declarations needed to tell the compiler how to construct the code resource header. If you work with a tool other than CodeWarrior, you may have to change the NewFilter.h file. The next line

```
pascal void main(CSoundBase* theSound, CWindowBase* theWindow, \
  CAmadBase* theApplication, CResFileBase* theFile, short theItem, long* myData)
```

declares the routine that will be called by **Amadeus II**. It will always have to be declared this way. Let us explain the meaning of the different parameters.

theItem — The value passed in this parameter tells your routine under what circumstances it has been called. The different possible values are labelled in the following way.

kInitFilter. If theItem takes this value, it means that **Amadeus II** just started up. You can perform any initialization you want. In particular, you can set *myData to a value that will be passed to your routine in the sequel. Do **never** try to show any dialog window at this time.

kReleaseFilter. If theItem takes this value, it means that **Amadeus II** is just about to quit. If you allocated some memory at startup time, you have to release it now.

kAboutItem. The user selected your external filter in the **About External Files** menu. You have two choices. Either you set *myData=0, and **Amadeus II** will display the standard about dialog with the text contained in the STR# 401 resource. You can also set *myData=1 and show your personal about box.

kEnableItem. **Amadeus II** ask your filter if the corresponding menu item (in the **Effects** menu) has to be enabled or not. If you set *myData=1, it will be enabled, if you set *myData=0, it will be disabled.

theItem ≥ 0. Your external filter has been called by the user. If a menu is attached to your filter (if you specified a MENU resource with ID = 500), theItem contains the menu item selected by the user. If not, theItem is set to 0.

myData — If theItem is greater or equal to 0, *myData contains the value you specified when theItem was equal to kInitFilter. Typically, you may want to create a global variable and *myData contains the address for a pointer onto that variable.

theSound — This is a pointer to the CSoundBase object describing the sound contained in the frontmost window when the user calls your external filter. It contains a valid pointer only if theItem

is greater or equal to 0 or if it is equal to kEnableItem. Otherwise, it contains the nil pointer.

theWindow — This is a pointer to the CWindowBase object describing the frontmost window when the user calls your external filter. The same remarks as for theSound apply concerning the validity of the pointer.

theApplication — This is a pointer to the CAmadBase object describing the **Amadeus II** application process. This pointer is always valid.

theFile — This is a pointer to the CResFileBase object describing the resource file containing your external filter. This pointer is always valid.

Let us now turn to the body of the program. We just test the values of theItem. If it is equal to kEnableItem, we set *myData to 1, meaning that the menu item corresponding to our external filter shall always be enabled. If this item is selected, we produce a system beep by calling the SysBeep() routine.

4 Manipulating a sound object

4.1 Basic rules

Let us first state a few rules you are supposed to follow in order to produce some reliable code. A sound object is an object of the CSoundBase class. This is a virtual class, meaning that you can not create such an object by yourself. If you want to create a sound object, you have to ask **Amadeus II** to do so.

The main rule when manipulating a sound is **never try to access directly to a sound**. The reason is that you have no mean to know if a given sound is in RAM or stored on the hard drive. If you want to access to the content of a sound, you always have to ask the sound to create a valid access for you.

Another rule concerns undoing. If you want the effect of your external filter to be undoable, proceed as follows:

1. Create a new sound of the same quality as the sound you want to modify.
2. Copy the selection of the old sound into the new sound.
3. Modify the new sound.
4. Copy the content of th new sound into the selection of the old sound.

4.2 Organization of data in a sound

A sound is composed of *frames*. One frame contains a *sample* for every channel. A sample is the most fundamental constituent of a sound. The length of a sample corresponds to the recording resolution (if the sound was recorded at 16Bits, a sample is 16 Bits long).

4.3 Creation of a sound

In order to create a CSoundBase object, you have to use the

```
CSoundBase* CAmadBase::CreateSound(long theSize, SoundCar theCars)
```

method. theSize has to contain the size of the new sound (in principle, you will set it to zero). theCars contains the caracteristics of the new sound. For a description of the SoundCar structure, see Section 7.

4.4 Accessing a sound – the "CSoundAccess" class

In order to acces to a part of a sound, you have to ask the sound to create an access for you. Before you do this, you should ask the sound how big the maximal access is allowed to be. This can be done with the

long CSoundBase::MaxAccess()

method. The creation of the access can then be done with one of the

CSoundAccess* CSoundBase::GetWriteAccess(**long** pos, **long** length)

CSoundAccess* CSoundBase::GetReadAccess(**long** pos, **long** length)

methods. A sound access created with GetWriteAccess can be used to read and to write information into a sound. A sound access created with GetReadAccess should be used only to read information from a sound. If you do not intend to write into a sound, you should create the access with GetReadAccess, since this may reduce the disk accesses. In both cases, pos is the position of the access fro the beginning of the sound (in frames) and length is the length of the access. If the access could not be created successfully, both methods return a 0 pointer.

The "CSoundAccess" class provides the following methods.

CSample& GetSample(**unsigned long** pos, **short** chan)
: This method returns the sample located at pos samples **from the beginning of the access** and belonging to the channel labelled by chan.

CSample* GetSamplePtr(**unsigned long** pos, **short** chan)
: Has the same behavor than GetSample, but returns a pointer to the corresponding sample.

UpDateAccess()
: If the sound is on the hard drive, this routine makes the changes you made to the sound access effective. This method is automatically called when you destroy a sound access.

PositionAccess(**unsigned long** newpos)
: Positions the access at newpos. If you changed the sound, you should call UpDateAccess before calling PositionAccess.

ResizeAccess(**unsigned long** newsize)
: Changes the length of the access to newsize. If you changed the sound, you should call UpDateAccess before calling ResizeAccess.

unsigned long GetSize()
: Returns the length of the access.

unsigned long GetPos()
: Returns the position of the access from the beginningof the sound.

4.5 Manipulating a sound – the "CSoundBase" class

In this subsection, I will describe the most important member functions provided by the CSoundBase class that were not already described in this section.

SoundSel GetSel()
: Returns an object of the type SoundSel containing informations about the current selection of the sound. For a more detailed description of the SoundSel object, see Section 7.

SetSel(SoundSel newSel)
: Sets the selection of the sound to be equal to the selection described in newSel.

SoundCar GetCar()
: Returns an object of the type SoundCar containing informations about the quality of the sound. For a more detailed description of the SoundCar object, see Section 7.

ChangeFormat(SoundCar newCar)
: Physically changes the sound to fit the quality specified in newCar. It may happen that this routine doesn't work if the SoundManager does not recognise the quality given in newCar.

ChangeCar(SoundCar newCar)
: Changes the sound quality of the sound without physically modifying it.

long Length()
: Returns the length of the sound in frames.

unsigned long DataLength()
: Returns the length of the sound in bytes.

SelectAll()
: Sets the selection to be the whole sound.

Clear()
: Deletes the current selection.

CopyFrom(CSoundBase* source)
: Replaces the selection of the sound by the selection of source. If the sound qualities don't fit, the sound is automatically resampled. You are supposed to use this method to modify the sound **Amadeus II** passes to your routine. This will allow the user to undo your effects.

CList<CMark> *GetMarks()
: Returns a pointer to the list of marks of the sound.

SetMarks(CList<CMark>* newMarks)
: Sets the list of marks of the sound to be equal to newMarks. If you want this operation to be undoable, you should call ChangeMarks() after it.

To see an example of sound manipulation, take a look at the file Reverse.c located in the Amadeus Toolbox:Examples folder.

5 Creating windows

We will now see how it is possible to create dialog windows and modal boxes with the **Amadeus Toolbox**. Dialog windows and modal boxes are described by objects of the type CWindowBase. This is again a virtual class, so you have to ask **Amadeus II** to create objects of this type.

5.1 Controllers

A controller is an object that tells **Amadeus II** how to handle a dialog or modal window. It has to be derived from the class CController. Two controllers are defined in a standard way. An object of the type CStandardModalController allows to handle a standard modal box. An object of the type CStandardDialogController allows to handle a standard dialog window. Your own controllers are very likely to be a derived class of one of the two standard controllers.

A controller has to provide the following methods.

void OpenDialog(CWindowBase* dialog)

This method is called by **Amadeus II** when the dialog window is created. dialog is a pointer to that window. You should start this method with a call to

```
CController::OpenDialog(dialog);
```

(CStandardModalController or CStandardDialogController respectively).

void CloseDialog()

This method is called by **Amadeus II** when the dialog window is closed.

void HandleSelect(**short** Item)

This method is called by **Amadeus II** when the user selects the item number Item of your dialog window. You should terminate this method with a call to CStandardModal-Controller::HandleSelect(dialog) (CStandardDialogController respectively).

For are more advanced usage, you can also define a DoAction(**short** action, **long** data) method. This allows you to send more sophisticated messages from parts of your dialog to the controller. For a more detailed explanation of the use of DoAction, see Section 6.

5.2 Creating a modal box

A modal box can be created with the

```
CWindowBase* CAmadBase::CreateModalBox(short ID, short isStandard, \
  CResFileBase* theFile = 0, CController* theController = 0, short pos = kCenter)
```

method. The parameters have the following meaning.

ID — The ID of the DITL and the DLOG resources containing the informations about the modal box. You can use one of the modal boxes of the **Amadeus II** application.

isStandard — If this parameter is set to true, the “OK” button (ID=1) of your modal box will be highlited. Moreover, the “Cancel” button (ID=2) will be called if the user types command-period. If you don’t provide a “Cancel” button, the second item in your modal box must not be a control button.

theFile — A pointer to the resource file containing the resources describing the dialog box. If you use one of the dialog boxes of Amadeus II, you have to pass 0 in this parameter.

theController — A pointer to the controller that will handle the modal box.

pos — Tells **Amadeus II** where to position the modal box. For the moment being, two values are accepted: kCenter centers the modal box, kAlert positions it within about the upper third of the screen.

A typical standard modal box will be handled in the following way.

```
CStandardModalController* theController = new CStandardModalController();
CWindowBase* theBox = theApplication->CreateModalBox(201, true, theFile, \
  theController, kAlert);
// Some initialisations...
theBox->Run();
delete theBox;
if (theController->WasOK()) {
```

```
    // Read content of modal box...
    }
  delete theController;
```

The CWindowBase::Run() method handles the modal box. It is available only if the window was created with the CAmadBase::CreateModalBox(...) method.

For more detailed informations about the methods provided by the class, see Subsection 5.4. For more informations about the various objects you can place in a window, see Section 6.

5.3 Creating a dialog window

Creating a dialog window works in the same fashion as creating a modal box, but you have to use the CAmadBase::CreateDialogBox(...) method instead. The argument list is exactly the same. The difference is that in the case of a modal box, you know that after the control flow quits the CWindowBase::Run() method, the user clicked either “OK” or “Cancel”. In the case of a dialog box, your window becomes a part of the **Amadeus II** interface. You know that the user wants to close the dialog box only inside the controller.

Typically, a dialog box can be handled in the following way.

```
CMyController* theController = new CMyController(theSound, theApplication, theFile);
CWindowBase* theBox = theApplication->CreateDialog(201, true, theFile, theController);
theWindow->AddDaughter(theBox);
```

The theWindow->AddDaughter(theBox) command makes sure that if the user closes the sound window containing the sound you are supposed to handle before he closes your dialog window, the dialog window closes automatically.

In this example, the class CMyController is user defined, and could look something like

```
class CMyController: public CStandardDialogController {
  CSoundBase* theSound;
  CAmadBase* theApplication;
  CResFileBase* theFile;
public:
  CMyController(CSoundBase* s, CAmadBase* app, CResFileBase* fil) {
    theSound = s;
    theApplication = app;
    theFile = fil;
    }
   CMyController() {}

  void HandleOK(); // Has to be defined
  virtual void OpenDialog(CWindowBase*);
  virtual void HandleSelect(short);
  };

void CMyController::OpenDialog(CWindowBase* theWin)
{
  CStandardDialogController::OpenDialog(theWin);
  // Initialization of the dialog box
```

```
}

void CMyController::HandleSelect(short SelectID)
{
  switch (SelectID) {
    case 1:
      HandleOK();
      break;
    }
  CStandardDialogController::HandleSelect(SelectID);
}
```

If you want to delete the dialog window from within your controller, never use the **delete** command. Always use the _delete macro instead (see Subsection 7.9).

5.4 The "CWindowBase" class

In this section, we will give a more detailed explanation of the methods provided by the CWindowBase class.

GiveFocus(CPart* thePart, **short** isTab)
: Sets the part thePart to be currently focused. Focusing only makes sense if thePart is of the type CEditText or CListPart. If isTab is true, the focusing takes place as if the user pressed the tabulation key. If it is false, it acts as if the user clicked into the part.

AdvanceFocus(**short** isTab)
: Gives the current focus to the next focusable part in the window's part list.

RGBColor GetBackColor()
: Returns the current color of the window's background. You should draw your user defined parts accordingly.

MySetPort()
: Sets the current graphics port to be the one of the window. A call to MySetPort() should **always** be balanced by a call to RestoreWorld().

AddDaughter(CWindowBase* newDaughter)
: Links the window newDaughter to the current window in a way so that if the current window is destroyed, newDaughter is destroyed too.

RemoveItems(**short** deb, **short** end)
: Destroys the idialog items deb to **end**. If end is not specified, destroys all the dialog items from deb to the last one.

AppendItems(**short** ID, **short** dh, **short** dv)
: Appends the dialog items described in the DITL resource with identification number ID to the dialog window. The variables dh and dv allow to make a vertical and/or horizontal offset of the items.

Rect GetItemRect(**short** item)
: Returns the bounding rectangle of the dialog item number item.

InstallItem(**short** item, CRectPart* newPart)
: Replaces the dialog item number item by the window part specified in newPart. Typically, this is useful if you want to include pop-up menus or parts like that into your dialog window.

CRectPart* GetItem(**short** item)
: Returns a pointer to the dialog item specified by item.

CTextItem* GetTextItem(**short** item)
: Works like GetItem, but returns 0 if the specified item is not of the type CTextItem. For the moment being, static text fields, editable text fields and pop-up menus are of that type.

SetCtlValue(**short** item, **short** value)
: In order to use this method, the item labelled by item has to be of the type CControlPart. It sets the value of the control to value. Objects of the type CControlPart are typically control buttons, radio buttons and check boxes.

short GetCtlValue(**short** item)
: Returns the value of the control labelled by item.

StopDrawing()
: Sets the clip of the window to an emty rectangle, so all subsequent drawing in the window has no effect.

ResumeDrawing()
: Sets the clip of the window to the whole window.

double ReadBox(**short** item)
: In order to use this method, the item labelled by item has to be of the type CTextPart. It looks for a floating point number in the string corresponding to the item labelled by item and returns that number.

long ReadBoxInt(**short** item)
: Behaves like ReadBox with integers instead of floating point numbers.

string ReadBoxString(**short** item)
: Behaves like ReadBox with strings instead of floating point numbers.

WriteItem(**short** item, **char*** Msg,...)
: Behaves exactly like the C routine “printf”, but the output is shown in the item labelled by item.

ReadItem(**short** item, **char*** Msg,...)
: Behaves exactly like the C routine “scanf”, but the input is taken from the item labelled by item.

If the window is a modal box, there is a method CloseBox() avilable that closes the modal box. It should be called from inside the controller when the user wants to close the modal box. The controller of a modal box should never try to destroy the modal box with the delete function.

6 Window parts

In this section, we will describe the various parts you can install in a dialog window. A part can be a text field, a control, a picture or anything else in the window. If the user clicks into a part, the part will analyze

the click and send an Action event to your controller. If the part is a standard dialog part, this will call the HandleSelect() method of your controller. If the part is created by the external filter, the DoAction(action, data) method will typically be called. The action field will contain a number that you can associate to your part (do always use numbers bigger or equal to 1000). The data method will contain some piece of information the part communicates to the controller.

Every window part is a derived class of the CRectPart class, which is itself derived from the generic CPart class. We will first of all describe the methods provided by the CRectPart class.

short InPart(Point pt)
: Returns whether the point pt is contained in the part or not. The point has to be given in local coordinates.

Rect GetPartRect()
: returns the rectangle delimiting the part.

GetPartRect(Rect newRect)
: Sets the rectangle delimiting the part to newRect.

InvalPart() Sends a system event such that the part will be redrawn at the next window update.

Enable() Enables the part.

Disable() Disables the part.

If you want to define your own window part, you have to define the following methods.

_Enable() Enable the part and draw it accordingly.

_Disable() Disable the part and draw it accordingly.

short WantFocus()
: Returns **true** or **false** whether the window part can be focused or not. If WantFocus() returns **true**, you should provide the methods GiveFocus() and TakeFocus().

short TrackCursor()
: Returns **true** or **false** whether the window part has a special behavior when the cursor is located in it or not. If TrackCursor() returns **true**, you should provide the methods EnterPart() and LeavePart().

If you want to, you can optionally redefine one of the following methods.

short DoClick(EventRecord* theEvent)
: Handles a click into the part by the user. The parameter theEvent contains a pointer to the event record returned by the system.

short DoKey(EventRecord* theEvent)
: Handles a key pressure. The parameter theEvent contains a pointer to the event record returned by the system.

DoIdle() Provides an idle routine that is called by **Amadeus II** at system time.

void GiveFocus(**short** isTab)
: If your window part can be focused, this method is called by **Amadeus II** when you get the focus. You should redraw the part accordingly. The state of isTab tells you if the user focused your part by clicking into it or by pressing the “tab” key.

TakeFocus() If your window part can be focused, this method is called by **Amadeus II** when you loose the focus. You should redraw the part accordingly.

EnterPart() This method is called by **Amadeus II** when the mouse enters the part.

LeavePart() This method is called by **Amadeus II** when the mouse leaves the part.

There is a number of window parts that are already defined by **Amadeus II**. The most important ones will be described shortly in the following subsections.

6.1 The "cicnButton" class

This class allows you easily to create buttons defined by three cicn resources. Those cicn resources must have consecutive identification numbers and correspond to the following states:

1. The unpressed active button.
2. The pressed active button.
3. The inactive button.

The constructor has the following syntax:

```
cicnButton(short ID, Rect prect, CWindowBase* owner, \
  short action, CResFileBase* theFile);
```

The parameters hav the folowing meaning.

ID The resource ID of the first cicn resource.

prect The rectangle delimiting the button. For nice results, this rectangle should have the same size as the icon rectangle.

owner A pointer to the window object in which you want to install the button.

action The identification number of the action sent to your window controller when the user presses the button. This number should be set greater or equal to 1000. If you don't want any message to be sent to the controller, you can set 0 in this field.

theFile A pointer to the resource file containing the cicn resources. If this field is set to zero, the **Amadeus II** resource file will be used.

The cicnButton provides the following methods in addition to those provided by the CRectPart class.

SetIcon(**short** ID)
: Changes the identification number of the icon series.

SetRelease(**short** r)
: Tells the button whether it should return to its initial state when the user clicked on it (r = **true**) or it should remain down (r = **false**).

SetPassive(**short** p)
: Tells the button whether it should be clickable or not. Non-clickable buttons are mainly intended for "decorative" uses.

SetDown(**short** d)
: Allows to press down or release the button from inside the program.

6.2 The "CTextPart" class

This is a virtual class that can not be created by its own. It describes generically all the window parts that are supposed to display some text. The methods provided by the classes derived from CTextPart are the following.

SetText(string newText)
: Changes the text of the part to newText and redraws the part accordingly.

sring GetText()
: Returns the text contained in the part.

SetFont(**short** newFont)
: Sets the font in which the text is displayed to newFont. The default value is kFontIDGeneva.

SetSize(**short** newSize)
: Sets the size at which the text is displayed to newSize.

SetFace(**short** newFace)
: Sets the face at which the text is displayed to newFace (*e.g.* bold, italic and so on).

SetJustify(**short** newJust)
: Sets the justification type of the text to newFace (*e.g.* teCenter, teFlushRight, teFlushLeft).

WriteData(**char*** Msg,...)
: Behaves like the C routine printf.

ReadData(**char*** Msg,...)
: Behaves like the C routine scanf.

There are several classes that are derived from the CTextPart class. The CEditText and the CStaticText classes are created by **Amadeus II** when you create a dialog window or a modal box from a resource where editable and/or static text fields are defined. The CControlPart class also derives from CTextPart. It allows to create controls (like *e.g.* the little arrows you can see in the example dialog box and/or the "Interpolate..." modal box). Its contructor has the following syntax.

```
CControlPart(short Type, string Title, Rect theRect, CWindowBase* owner, \
  short cMin, short cMax, short action)
```

This is the meaning of the various arguments

Type
: ID of the procedure describing the control (*e.g.* kControlLittleArrowsProc for the little arrows).

Title
: string containing the title of the control.

theRect
: rectangle delimiting the control.

owner
: Pointer to the window object in which the control has to be displayed.

cMin
: Minimal value of the control.

cMax
: Maximal value of the control.

action
: The identification number of the action sent to your window controller when the user uses the control. The CControlPart class will put the ID of the part in which the user clicked in the data field of the DoAction() method of your controller.

Here is a short description of the methods provided by the CControlPart class.

SetValue(**short** newVal)
: Sets the value of the control to newVal.

short GetValue()
: Returns the value of the control.

SetMin(**short** newMin)
: Sets the minimal value of the control to newMin.

SetMax(**short** newMax)
: Sets the maximal value of the control to newMax.

SetHilite(**short** newHilite)
: Sets the hilite state of the control to newHilite.

6.3 The "CMenuButton" class

This window part allows you to create pop-up menus in your dialog windows. The syntax of the constructor is the following.

```
CMenuButton(short MenuID, short theChoice, Rect theRect, \
  CWindowBase *owner, CResFileBase* theFile)
```

This is the meaning of the various arguments.

MenuID	The identification number of the MENU resource containing the menu.
theChoice	The number of the menu item to be selected.
theRect	The rectangle delimiting the pop-up menu.
owner	The window owning the pop-up menu.
theFile	A pointer to the resource file in which the MENU resource is defined.

Here is a short description of the methods provided by the CMenuButton class.

short GetChoice()
: Returns the menu item selected by the user.

PutChoice(**short** newChoice)
: Selects the menu item newChoice.

6.4 The "CSimpleIndicator" class

This window part allows you to create a standard indicator bar (like in the "Memory" floating palette). The Creator has the syntax

```
CSimpleIndicator(Rect theRect, CWindowBase* owner)
```

whith evident meanings of the arguments. The values of the indicator are floating point numbers reaching from 0 to 1. The SetValue(**short** newVal) method allows to set this value.

7 Utility classes and macros

7.1 The “CList” template

This template allows you to create easily lists of any type of object. If you want to create a list of objects of type “T”, you have to make a declaration like

```
CList<T> myList;
```

I will not give a detailed explanation of all the mthods available to manipulate CList<T> objects, but only of the most useful ones.

T* **operator**[](**long index**)
: Returns a pointer to the element number index of the list and sets the current element to be this one. The first element is labelled by 0.

T* GetFirst()
: Returns a pointer to the first element of the list and sets the current element to be the first one.

T* GetNext()
: Returns a pointer to the element following the current element and then increments the position of the current element.

T* GetLast()
: Returns a pointer to the last element of the list and sets the current element to be the last one.

T* GetCurrent()
: Returns a pointer to the current element of the list.

Add(T* newElement)
: Adds the object newElement at the end of the list.

Remove(T* oldElement)
: Removes oldElement from the list without destroying it.

DestroyData(T* oldElement)
: Removes oldElement from the list and destroys it.

Destroy()
: Destroy all elements of the list.

If you delete an object of type CList<T>, its elements are only removed, but not destroyed.

7.2 The “SoundCar” structure

This structure describes the caracteristics of a sound. It has the following data members.

mSampleRate
: a 32Bit integer describing the sampling rate at which the sound was recorded.

mSampleSize
: a 16Bit integer describing the resolution of the sound in Bytes. If the sound was recorded at 16Bit resolution, mSampleSize = 2.

mNumChans
: a 16Bit integer containing the number of channels

mFixedRate
: an unsigned 32Bit integer containing the sample rate as a fixed point number.

7.3 The "SoundSel" structure

This structure describes the caracteristics of the current selection of a sound. The available data members are the following.

mSelStart — a 32Bit integer containing the position of the beginning of the selection. The position is computed in frames, meaning that if mSelStart is equal to mSampleRate, the beginning of the selection is at 1 second.

mSelEnd — a 32Bit integer containing the position of the end of the selection.

mSelMode — a 32Bit integer containing information about which channels are selected. Channels are labelled from 1 to n. If the kth channel is selected, the expression

(mSelMode>>k)%2

is true. Otherwise, it is false.

Moreover, the SoundSel structure has two member functions.

Length() — Returns the value mSelEnd-mSelStart.

NChansSel() — Returns the number of selected channels.

7.4 The "CSample" class

This class allows to change the values of a sound sample. In order to get a value, you have to use one of the get8(), get16() or get24() methods, depending of the quality of the sound. All of them return a 32Bit integer between -8388608 and +8388607. You can modify this value and put it back into the sample using the put8(), put16() or put24() methods. If your value may exceed the -8388608...+8388607 range, you can use the putTest8(), putTest16() or putTest24() methods, so your value is automatically truncated, but you loose a little bit of execution speed.

7.5 The "CEasySample" class

This class allows to change the values of a sound sample in an easier way. You do not have to worry about wether the samples are coded on 8, 16 or 24 Bits. The way the CEasySample class is used is illustrated in this short example that multiplies the 50 first samples of the sound access myAccess by 2.

```
CEasySample* mySample;
NewEasy(mySound, mySample);
mySample->SetData(myAccess->GetSamplePtr(0,0));
for (short i=0; i<50; i++) {
  mySample->put(mySample->get()*2);
  mySample->Advance(1);
  }
delete mySample;
```

The NewEasy(sound,sample) macro initializes the sample. You have to put in the first argument a pointer to the sound of which sample will be a sample. The sound is required in order to initialize the CEasySample class the right way.

The Advance(**long** n) method advances the sample by n times the length of the sample (8, 16 or 24 Bits). If you want to advance by one frame, you have to pass the number of channels in n. The only disadvantage of the CEasySample class is that you may loose a little bit of execution spped.

The SetData(CSample *aSample) method allows to tell the class on which sample to point.

7.6 The "CMark" structure

MarkPos — Nosition of the mark (in frames).

MarkName — A string containing the name of the mark.

MarkColor — Color of the mark. The allowed values are blackColor, yellowColor, magentaColor, redColor, cyanColor, greenColor and blueColor.

7.7 The "CSpec" class

This class is provided so that you can easily make Fourier transforms to parts of a sound. A spectrum is a list of S complex numbers (data points) indexed from 0 to $S-1$.

The constructor has the syntax

CSpec(**short** ℓ)

where l is the basis 2 logarithm of the size of the spectrum, *i.e.*

$$S = 2^{\ell}\ , \quad s = 2^{\ell+1}$$

where S is the size of the spectrum and s is the number of data points that will be read from the sound. The methods provided by CSpec are the following.

CopyFromSound(CSoundBase* source, **long** pos, **short** chan, **short** j)
: This method copies the data points from the sound source into the spectrum. The parameter pos contains the position (in frames) of the first data point. The parameter chan contains the channel ou want to read data from. If the parameter j is given, the methods reads ony every jth data point.

CopyFromPtr(**long** addr, SoundCar theCar, **short** j)
: Has essentially the same behavior than CopyFromSound, but you have to provide a valid address to the first **sample** you want to put into the spectrum. theCar has to contain the caracteristics of the sound.

FFT() — Makes a Fourier transform of the spectrum.

InvFFT() — Makes a inverse Fourier transform of the spectrum.

CopyToSound(CSoundBase* dest, **long** p, **short** chan, **long** ℓ_1, **long** ℓ_2)
: Allows to copy the data from the spectrum to the sound dest. p is the position of the first frame you want to modify and chan the channel you want to write into. This routine modifies the frames number p to $p+S$. If we denote by F_k the value of the relevant sample in the kth frame of the sound and by D_k the value of the kth data point, CopyToSound perfoms the transfomation

$$F_{p+j} \mapsto \begin{cases} F_{p+j} + D_j & \text{if} \quad 0 \le j < \ell_1 \\ D_j & \text{if} \quad \ell_1 \le j < S - \ell_2 \\ F_{p+j} + D_j & \text{if} \quad S - \ell_2 \le j < S \end{cases}$$

I apologize for the rather mathematical notations in this subsection, but I didn't find another straightforward way to write it.

MakeOverlap(**short** start, **long** ℓ, **short** type, **long** t)

This method allows to cut down in a smooth way the ends of the data points via a cutoff function $f(x)$ The value of f depends on type in the following way:

$$f(x) = \begin{cases} x & \text{if type = 0} \\ \frac{1-\cos(\pi x)}{2} & \text{if type = 1} \end{cases} .$$

If type = 2, the cutoff function f is equal to

$$f(x) = \begin{cases} 0 & \text{if} \quad 0 \le x \le 1/3 \\ \frac{1+\cos(3\pi x)}{2} & \text{if} \quad 1/3 < x < 2/3 \\ 1 & \text{if} \quad 2/3 \le x \le 1 \end{cases} .$$

Let us take again the notations from above. If start is set to **true**, the method MakeOverlap performs the transform

$$D_j \mapsto \begin{cases} D_j & \text{if} \quad 0 \le j < t \\ f\left(\frac{j-x}{\ell}\right) D_j & \text{if} \quad t \le j < t+\ell \\ D_j & \text{if} \quad t+\ell \le j < S \end{cases}$$

If start is set to **false**, it performs the transform

$$D_j \mapsto \begin{cases} D_j & \text{if} \quad 0 \le j < S-\ell-t \\ f\left(\frac{S-t-j}{\ell}\right) D_j & \text{if} \quad S-\ell-t \le j < S-t \\ D_j & \text{if} \quad S-t \le j < S \end{cases}$$

PutZeros(**long** first, **long** last)

Sets all the data points located between first and last (included both of them) to 0.

double MaxValue()

Returns the maximal value of the squares of the norms of the data points.

7.8 The "_exterror" macro

This macro allows to generate error messages. You should use it like _exterror(ID, appl, file). The error message has to be stored in a resource of type STR# and resource ID 401. The parameters have the following meaning:

ID — The number of the error string inside the STR# resource.

appl — The pointer to the application **Amadeus II**.

file — The pointer to your resource file.

7.9 The "_delete" macro

This macro should be used to delete a dialog window. Do never destroy a dialog window by simply calling **delete** myWin. Always call _delete(myWin) instead.

7.10 String conversion routines

There are two routines that are provided to convert C string into Pascal strings and vice-versa.

The routine String2Pas(**const** string& A, Str255& Res) puts the string A into the Str255 object Res.

The routine Pas2String(**const** Str255& A, string& Res) puts the Str255 object A into the string Res.

7.11 The "CMenu" class

This class allows to handle a menu object. Do not create such an object by yourself. Always ask **Amadeus II** to return one. The methods provided by this class are.

MenuHandle GetMenu()
: Returns the MenuHandle associated to the menu.

short NItems() Returns the number of menu items in the menu.

EnDis(**short** item, **short** enable)
: Enables/disables the menu item number item.

SetMark(**short** item, **short** mark)
: Sets a mark (*e.g.* a diamond) next to the menu item number item.

7.12 The "CAmadBase" class

This is a virtual class describing the **Amadeus II** application process. It is derived from the generic CApplicationBase class. The most common methods provided by this class have already been described (how to create a dialog window, a sound, etc). Here are nevertheless a few other useful methods.

short ReserveMem(**unsigned long** size)
: Tells the application to try to free size bytes of memory in the application heap. This will usually be achieved by putting some sound objects onto the hard drive. The return value can be **true** or **false** whether the application was successful or not.

CMenu* GetMenu(**short ID**, CResFileBase* theFile)
: Returns a pointer to the CMenu object corresponding to the MENU resource with identification number ID of the file theFile.

short WasError()
: Returns **true** if an error is present in the event queue.