

# 13.Filters

## ***13.1. Prerequisites and Objectives***

**Before starting this chapter you should have:**

1. A knowledge of programming in C++.
2. A knowledge of component interface design.

**After completing this chapter you will have:**

1. A knowledge of how to transform batch programs into filter components.
2. A knowledge of the principles of Hot and Cold Filters.
3. A knowledge of the Filter-Component design methodology.

## ***13.2. Introduction***

Prior to the advent of Graphical User Interface programming, virtually any program you could write was a Filter. Many of the most sophisticated programs in existence are filters. A compiler is a filter that transforms source files into object files. A link-editor is a filter that transforms object files into executable programs. A report generator is a filter that transforms database tables into printed reports. A database query engine is a filter that transforms a collection of database tables into a new table. In the area of image processing there are innumerable filters for processing images or segments of images. Filtering is one of the first things that a programmer learns. Many of the exercises in introductory programming courses are filters.

Very simply, a filter is a program that transforms an object into a different object. The new object may be of a different type, or it may be of the same type, but with more

desirable properties. For the most part, the transformation algorithms themselves are beyond the scope of this book. (See “the rest of Computer Science.”) However, most filters have similar frameworks. The purpose of this chapter is to demonstrate the principles of constructing frameworks for various types of filters. The application of these principles to other problems is straightforward.

### ***13.3. The Methodology***

With the possible exception of configuration parameters, the external interface of a Filter is quite simple. A file-to-file filter generally accepts two file names, and uses one for input and the other for output. An object filter has a mechanism for supplying objects, and a mechanism for retrieving the transformed objects. The input and output mechanisms can be implemented as properties or methods.

Once the input and output mechanisms have been chosen, the configuration parameters, if any, must be designed. Depending on their complexity, there is a wide range of methods that can be used to specify configuration parameters. Simple parameters can be specified using a set of properties. (Methods can also be used, but properties are the preferred mechanism.) More complex parameters may be supplied as configuration objects or configuration files. In some cases, the configuration parameters are so complex, that a separate program must be used to create them. (Such is the case for report generators.)

In most cases the configuration parameters are very simple or even non-existent. In any case, once the input and output mechanisms have been chosen, and the configuration parameters have been designed, then development proceeds along the same lines as an ordinary command-line program.

### **13.4. Hot Filters and Cold Filters**

The Filter category can be further broken into Hot Filters and Cold Filters. The principles of these two types of filters are quite different. A Cold Filter transforms its input in a predictable way. Its operation is frozen and unchanging. An example of such a filter is a C++ compiler. Regardless of which parameters are set, the filter transforms C++ files into object files. No change of parameters can “convince” the C++ compiler to transform Pascal files into object files.

A Hot Filter has a function that must be programmed prior to its use. An example of a Hot Filter is a report generator. Before it can generate a report, it must be given the format of the report. In many cases Hot Filters have self-defining input. This is certainly the case for a report generator, since the definition of the report will generally include the name of the data base and the table or query from which the input data must be taken. Other types of Hot Filters do not define their own input. An example of such a filter is an image processor that can apply a convolution transformation to a bit-mapped image. Before the filter can be used, it must be supplied with the definition a transformation, but once this is done, the input can come from virtually any source.

### **13.5. File Transformers**

For the purposes of this book, we will further divide filters into file transformers and object transformers. File transformers operate in file-to-file fashion, while object transformers operate in memory-to-memory fashion. (Filters that operate between files and memory are known as Serializers, and are covered in Chapter 9.) Although we feel

that the distinction between file transformers and object transformers is somewhat artificial, it is useful for our purposes.

There are many traditional programs that fit into the category of file transformers, and it is generally a simple matter to convert such programs into Filter components. In this section, we will demonstrate two file Filters that are based on UNIX/DOS programs. The first of these will transform text files between UNIX and MS Windows, while the second will analyze the words in a text document and produce a report suitable for printing. (The report will be stored in a text file.)

### **13.5.1. Unix-To-PC transformation**

In a UNIX text file, every line must end with a line-feed character. This character is a line terminator, so the file must end with a line-feed character. In a DOS file each lines must be separated from one another using a return character and a line-feed character. It is not necessary for the last line in the file to be terminated by a return/line-feed pair, but it is common to do so. (Technically, if the last two characters in a text file are a return and a line-feed, the file ends with a blank line.)

MS Windows programs do not handle UNIX text files well. Many of them do not recognize the single line-feed as an end-of-line marker, and will attempt to treat the entire file as a single line. Similarly, UNIX programs don't know what to do with the extra return characters, and will normally treat them as if they were ordinary text characters. Because C++ programs are stored in text files, porting C++ programs between MS Windows and UNIX can be a problem, the compilers may not recognize the lines properly. (It is also true that many compilers *do not* have this problem.)

We want our component to be able to transform files in both directions, from MS Windows to UNIX and from UNIX to MS Windows. Furthermore, if the component is asked to transform a UNIX file into a UNIX file, or an MS Windows file into an MS Windows file, it should produce the correct output, even though the input is of the wrong type.

The Filter component will be built around two functions, `MakeUnixFile`, and `MakePCFile`. Both of these functions take two arguments, the name of the input file, and the name of the output file. With minor modifications, these functions are similar to those that might be found in a typical DOS or UNIX program. We could have used the `stdio.h` package or the `iostream.h` package, but we cannot be sure that these packages handle their global variables in a manner that is consistent with the requirements of a component.

The use of global variables can cause serious problems in a component. Because two or more instances of a component can run together in the same address space, we must be very careful to make sure they don't share their global and static variables. One way to avoid problems is to move all global and static variables into the support object for the component. By doing this, we can assure ourselves that every instance of the component has its own copy of the variables.

Once global variables have been moved into the support class, the functions that access them must be made members of the class. The transformed functions must also be made members of the support object to insure that they have access to the global variables. The main difficulty that arises with this process is handling global variables with complex initializations. Since members of a class cannot have static initializers, these initializations must be moved into the constructor for the support class.

The Text File Transformer has three properties and one event. It has no methods. This simplicity in structure is typical of a file Filter. Figure 13.1 gives the property design table for the component, while Figure 13.2 gives the formal definition of the event.

Property Design Table		
Name	Type	Function
<b>InputFile</b>	String Default=Empty	Used to provide the name of the input file. Causes no other action. No restrictions.
<b>PCOutput</b>	String Default=Empty	Used to provide the name of the PC output file. When a new value is supplied, the input file is transformed into DOS/Windows format and written to the output file. Must not be the same as InputFile.
<b>UNIXOutput</b>	String Default=Empty	Used to provide the name of the UNIX output file. When a new value is supplied, the input file is transformed into UNIX format and written to the output file. Must not be the same as InputFile.

**Figure 13.1. The Properties of the Text File Transformer.**

Event Description			
Name	ErrorMessage		
Description	Reports errors in the file transformation process.		
Triggers	Open errors or I/O errors. Occurs only when assigning a new value to <i>PCOutput</i> or <i>UNIXOutput</i> . Also triggered if <i>PCOutput</i> or <i>UNIXOutput</i> is assigned a value before assigning a value to <i>InputFile</i> .		
Arguments	Name	Type	Description
	Msg	String	Verbal Description of Error

**Figure 13.2. The Text Transformer ErrorMessage Event.**

The *MakeUnixFile* function is given in Figure 13.3. In the Windows operating system, the functions to open, read and write files have become obnoxiously complicated, so we omit the details of them from Figure 13.3. The full details are available on the CD.

```

void CUnixToPCTxtCtrl::MakeUnixFile(   CString &InFileName,
                                       CString &OutFileName)
{
    ... // open input and output files, fire event and terminate if error
    // allocate buffers
    char * InBuffer = new char[10000];
    char * OutBuffer = new char[10000];
    // I/O control variables
    unsigned long BytesRead,BytesWritten;
    // output bytes used
    long OutC = 0;
    // state variables
    BOOL LastR = FALSE;
    BOOL LastN = FALSE;
    ... // Read 10,000 bytes, terminate with error event if I/O error
    while (BytesRead > 0) // scan input buffer
    {
        char * tc;
        unsigned long i;
        for (tc = InBuffer,i=0 ; i<BytesRead ; i++,tc++)
        {
            LastN = FALSE;
            if (*tc == '\r') // if \r not before \n, don't delete
            {
                LastR = TRUE;
            }
            else if (*tc == '\n')
            {
                LastR = FALSE; // if there is a preceding \n, zap it
                LastN = TRUE;
                OutBuffer[OutC] = *tc;
                OutC++;
                if (OutC >= 10000)
                {
                    ... // write output buffer
                    OutC = 0;
                }
            }
            else
            {
                if (LastR)
                {
                    OutBuffer[OutC] = '\r'; // \r not before \n keep it
                    OutC++;
                    if (OutC >= 10000)
                    {
                        ... // write output buffer
                        OutC = 0;
                    }
                }
                LastR = FALSE;
                OutBuffer[OutC] = *tc;
                OutC++;
                if (OutC >= 10000)
                {
                    ... // write output buffer
                    OutC = 0;
                }
            }
        }
    }
}

```

```
    }
  }
}
... // read 10,000 more bytes
}
if (!LastN)
{
  OutBuffer[OutC] = '\n';
  OutC++;
}
if (OutC > 0)
{
  ... // write output buffer
  OutC = 0;
}
// close files
CloseHandle(InFile);
CloseHandle(OutFile);
// delete buffers
delete [] InBuffer;
delete [] OutBuffer;
}
```

**Figure 13.3. The MakeUnixFile Function.**

The *MakePCFile* function is similar to the *MakeUnixFile* function. Characters are scanned one at a time, and when a line-feed character is encountered, the function checks to see if the preceding character was a return, and if not it inserts a return character into the output file. The *MakePCFile* function is shown in Figure 13.4.

```

void CUnixToPCTxtCtrl::MakePCFile(CString &InFileName,
                                   CString &OutFileName)
{
    ... // open input and output files.
    ... // Report any errors using ErrorMessage Event.
    // allocate buffers
    char * InBuffer = new char[10000];
    char * OutBuffer = new char[10000];
    // read/write control variables
    unsigned long BytesRead,BytesWritten;
    // output character count
    long OutC = 0;
    // State variable
    BOOL LastR = FALSE;
    ... // Read 10,000 bytes
    while (BytesRead > 0) // while not EOF
    {
        char * tc;
        unsigned long i;
        // scan each byte
        for (tc = InBuffer,i=0 ; i<BytesRead ; i++,tc++)
        {
            if (*tc == '\r')
            {
                // record the presence of a return character
                LastR = TRUE;
                OutBuffer[OutC] = *tc;
                OutC++;
                if (OutC >= 10000)
                {
                    ... // write output buffer
                    OutC = 0;
                }
            }
            else if (*tc == '\n')
            {
                // if the last character wasn't a return, add one
                if (!LastR)
                {
                    OutBuffer[OutC] = '\r';
                    OutC++;
                    if (OutC >=10000)
                    {
                        // write output buffer
                        OutC = 0;
                    }
                }
                LastR = FALSE;
                OutBuffer[OutC] = *tc;
                OutC++;
                if (OutC >= 10000)
                {
                    ... // write output buffer
                    OutC = 0;
                }
            }
        }
    }
    else

```

```

    {
        // ordinary character
        LastR = FALSE;
        OutBuffer[OutC] = *tc;
        OutC++;
        if (OutC >= 10000)
        {
            ... // write output buffer
            OutC = 0;
        }
    }
}
... // read 10,000 more bytes
}
if (OutC > 0)
{
    ... // write output buffer
    OutC = 0;
}
// close files and delete buffers
CloseHandle(InFile);
CloseHandle(OutFile);
delete [] InBuffer;
delete [] OutBuffer;
}

```

**Figure 13.4. The MakePCFile Function.**

Note that the functions given in Figure 13.3 and Figure 13.4 are essentially the same sort of functions that one would find in a typical batch program.

### 13.5.2. The Word Counter

Next we illustrate a typical read-file-and-print operation. However, instead of printing the output, we will store the output in a text file. The resultant component is a file-transformation component. The Word Counter component will read a text file, and count the number of times each word occurs in the text. The result will be an alphabetical list of unique words with a count of the number of times each word is used, and a similar list sorted by frequency of use.

The heart of this component is the **CWCount** class, which encapsulates the functionality needed to count word frequencies. Each unique word is represented by an

object of the class **CWCItem**. Figure 13.5 gives the definition of **CWCItem**. Each object contains the word in lower case, and a count of the number of times it occurs. These objects will be stored in a hash table which is implemented as a collection of singly linked lists.

```
class CWCount;
class CWCItem
{
    friend CWCount; // container class
protected:
    CWCItem * Next;
    CString Word; // the word in lower case
    long Occur; // occurrences
public:
    CWCItem();
    CWCItem(LPCTSTR NewWord); // sets Occur to one
    virtual ~CWCItem();
};
```

**Figure 13.5. The CWCItem Class.**

The **CWCount** class is given in Figure 13.6. The user of the **CWCount** object must break the input text into words, and call the *Word* function for each word. The **CWCount** object will keep track of unique words and count the occurrences of each. A hash table is used to keep track of the words. For the hash table to work properly, each entry in the array must be initialized to the NULL pointer by the class constructor. Each entry in the hash table is treated as the head of a stack. The hash-table key is computed by treating the characters of the word as unsigned 8-bit integers, computing their sum, and extracting the low-order eight bits of the result. The *Clear* function is used at the beginning of each operation to empty the hash table for a new operation. Once the input text has been processed, the usage report can be obtained by calling the three functions *GetTotalsString*, *GetUsageString*, and *GetAlphaString*. The function *GetTotalsString* produces a multi-line text message giving the total number of unique words and the total

word count. The *GetUsageString* produces a multi-line text string containing the list of unique words, along with a count of the number of times that word occurs in the text. Each word is on a separate line. The list is sorted in descending order by the number of occurrences. The *GetAlphaString* function produces a string identical to that of the *GetUsageString* function, except the words are sorted into ascending sequence alphabetically. For both functions the words are listed in lower-case letters.

```
class CWCount
{
protected:
    long UniqueWords; // count of unique words
    long WordCount; // count of total words
    // Hash key is obtained by converting word to lower case
    // and adding up the letters as unsigned 8-bit integers.
    // The last 8 bits are trimmed off and used as a key.
    CWCItem * HashTable[256];
public:
    BOOL IsEmpty(void);
    // construction, destruction
    CWCount();
    virtual ~CWCount();
    // input functions
    // add word to hash table, or increment count
    void Word(LPCTSTR InWord);
    // delete all words from hash table & set counts to zero
    void Clear(void);
    // output functions
    // produce the totals line in text format
    void GetTotalsString(CString &Out);
    // produce the text of the list sorted by usage
    void GetUsageString(CString &Out);
    // produce the text of the list sorted alphabetically
    void GetAlphaString(CString &Out);
protected:
    // utilities
    // array of words in usage order
    CWCItem ** GetUsageArray(void);
    // array of words in alphabetical order
    CWCItem ** GetSortedArray(void);
    // compute hash-table key
    long Hash(LPCTSTR InWord);
};
```

Figure 13.6. The CWCount Class.

The *Word* function, which performs most of the real work, is given in Figure 13.7.

Figure 13.8 shows the computation of the hash-table index.

```
void CWCount::Word(LPCTSTR InWord)
{
    WordCount++; // count total number of words
    CString Work = InWord; // place word in a convenient wrapper
    Work.MakeLower(); // convert to lower case
    long HIndex = Hash(Work); // compute hash index
    // search linked list for this hash table entry
    for (CWCItem * Temp = HashTable[HIndex];
        Temp != NULL && Temp->Word != Work;
        Temp=Temp->Next);
    if (Temp != NULL)
    {
        // if word already in table, increment occurrences
        Temp->Occur++;
    }
    else
    {
        // if word not in table, create new entry and link into table
        UniqueWords++; // count unique words
        Temp = new CWCItem(Work); // sets occurrences to one
        Temp->Next = HashTable[HIndex];
        HashTable[HIndex] = Temp;
    }
}
```

**Figure 13.7. The Word Function.**

```
long CWCount::Hash(LPCTSTR InWord)
{
    long rv = 0;
    for (const char * tc = InWord ; *tc != '\0' ; tc++)
    {
        rv += (unsigned char)(*tc);
        rv &= 0xff; // trim each time to avoid overflow
    }
    return rv;
}
```

**Figure 13.8. Computing the Hash-Table Index.**

The remaining functions of the **CWCount** object are straightforward. For interested readers, they can be found on the accompanying CD.

The Word Counter component is quite simple, having two properties and one event.

These are documented in Figure 13.9.

Property Design Table		
Name	Type	Function
<b>InputFile</b>	String Default=Empty	Used to provide the name of the input file. File will be read and processed immediately upon assignment to this property. <b>Restrictions:</b> Write Only, Run-Time Only
<b>OutputFile</b>	String Default=Empty	Used to provide the name of the PC output file. When a new value is supplied, the accumulated totals are formatted and written to this file. <b>Restrictions:</b> Write Only, Run-Time Only

Event Description			
Name	ErrorMessage		
Description	Reports errors in the file transformation process.		
Triggers	Open errors I/O errors, assignment to OutputFile with no prior assignment to InputFile.		
Arguments	Name	Type	Description
	Msg	String	Verbal Description of Error

**Figure 13.9. The Word Counter Interface.**

The entire component is implemented within the access routines for the properties *InputFile* and *OutputFile*. (This is typical for a file-transformation component.) The routines for these properties are lengthy, but straightforward. The code can be found on the accompanying CD.

### **13.6. Object Transformers**

Object transformers are similar to file-transformers in that they generally implement standard, non-graphical algorithms. The primary difference is that the input to an object transformer is already in a form that is readily usable by an algorithm. In most cases the input will be a pointer to a complex object obtained from another component. The object-

transformer either transforms the existing object to “improve” it in some way, or creates a new object from the existing object. The most interesting examples of such components are far too complicated to be presented here, so we will concentrate instead on a few simple examples.

In all of our examples, it is necessary to work around a fundamental weakness in the ActiveX technology, namely that it is impossible to define a property with an object type or an object pointer type. For that reason we will coerce all object pointers into long integers, and pass the addresses as integers. In general, this is a poor programming technique, because long integers and addresses are not guaranteed to be the same length on all future systems. In fact, the memory sizes of many existing computers far exceed the limitations of a 32-bit address. There are other difficulties with passing object addresses from component to component, but we will save that discussion for Chapter 19.

In the first example, we will show how to embed a simple algorithm in a Filter component. In the second example, we will show how to create a Hot Filter that is configurable at run-time. For the third example, we will discuss a text-to-object transformer.

### **13.6.1. The Integer Sorter**

Unlike file-transformers, which can operate as stand-alone components, object transformers must be coupled with other components to provide the input and process the output of the object transformer. In the Integer Sorter, we will embed a simple insertion sort algorithm in its own component. However, to test the component, we must create two other components, one that will provide the input to the Integer Sorter, and another that will display the output of the Integer Sorter in human-readable form. Figure

13.10 shows the relationship of these three components. This sort of structure is typical for most object transformers.



**Figure 13.10. Component Relationships for Object Transformers.**

In Figure 13.10 the communication between the three components is via an object called a Dynarray. This object is a wrapper for a dynamic array of integers. The definition of the **CDynarray** class is given in Figure 13.11. The implementation of this class is straightforward.

```
class CDynarray
{
public:
    // dynamic array implementation
    long * Data;
    long Count;
    // construction and destruction
    CDynarray(long NewCount, long * NewData);
    CDynarray(const CDynarray &x);
    CDynarray();
    virtual ~CDynarray();
};
```

**Figure 13.11. The CDynarray Class.**

The interface of the integer sorter is quite simple, consisting of just two properties, *Sorted*, and *Unsorted*. The formal description of these properties is given in Figure 13.12.

Property Design Table		
Name	Type	Function
<b>Unsorted</b>	Long/Pointer Default=NULL	Supplies the address of the CDynarray object to be sorted. No action is taken as a result of the assignment. The input object will not be modified by this component, however the object is owned by the component, and may be deleted by the component. <b>Restrictions:</b> Write Only, Run-Time Only
<b>Sorted</b>	Long/Pointer Default=NULL	Provides the address of the sorted CDynarray object. This is a new object created by the Integer Sorter component. The sort operation is performed as part of the implementation of this property. The component does not own this new object, and will not delete it. <b>Restrictions:</b> Read Only, Run-Time Only

**Figure 13.12. The Integer Sorter Interface.**

The implementations of the *Sorted* and *Unsorted* properties are given in Figure 13.13 and Figure 13.14. The *SetUnsorted* function receives a pointer to a **CDynarray** object in the form of a long integer. The component assumes ownership of the object, in the sense that is permitted to delete the object when it no longer needs it. The sorting operation is performed when the sorted object is requested through an access to the *Sorted* property. Instead of sorting the existing **CDynarray** object, a copy of the object is created, sorted, and returned to the requester. The requester then assumes ownership of the returned object.

```

void CIntSorterCtrl::SetUnsorted(long nNewValue)
{
    if (SortItem != NULL)
    {
        delete SortItem;
    }
    SortItem = (CDynarray *)nNewValue;
    SetModifiedFlag();
}

```

**Figure 13.13. The SetUnsorted Function.**

```
long CIntSorterCtrl::GetSorted()
{
    if (SortItem == NULL)
    {
        // if nothing to sort, return NULL
        return (long)SortItem;
    }
    if (SortItem->Count == 0)
    {
        // if object is empty, create empty object and return it
        CDynarray * EmptyItem = new CDynarray;
        return (long)EmptyItem;
    }
    // create duplicate object, sort it and return pointer to it
    CDynarray * NewSort = new CDynarray(*SortItem);
    InsertSort(NewSort);
    return (long)NewSort;
}
```

**Figure 13.14. The GetSorted Function.**

The *GetSorted* function calls the *InsertSort* routine to sort the **CDynarray** object. This routine, which is illustrated in Figure 13.15 is essentially the standard text-book version of the Insert Sort algorithm. It is obvious that virtually any sort algorithm could be embedded in this component in the same fashion as the Insert Sort algorithm, illustrating the manner in which standard algorithms can be embedded in filter components.

```

void CIntSorterCtrl::InsertSort(CDynarray *Item)
{
    if (Item == NULL)
    {
        return;
    }
    if (Item->Count == 0)
    {
        return;
    }
    if (Item->Data == NULL)
    {
        return;
    }
    for (long i=1 ; i<Item->Count ; i++)
    {
        long x = Item->Data[i];
        for (long j = i-1 ; j >= 0 && Item->Data[j] > x ; j--)
        {
            Item->Data[j+1] = Item->Data[j];
        }
        j++;
        Item->Data[j] = x;
    }
}

```

**Figure 13.15. The InsertSort Function.**

Although the Integer Sorter is a stand-alone component, it requires some source for the **CDynarray** objects. By the same token, if the output of the component is to be useful, it requires some consumer of the sorted **CDynarray** objects it creates. In an actual application, there would be natural producers and consumers of these objects. However since we are dealing with this component in isolation, we must create our own producers and consumers specifically for the purpose of testing. Thus we have created two additional components which will serve as the producer and consumer of the **CDynarray** objects. Since these two components are artificial, we will not give complete details, but merely give the definition of their interfaces. The **DynarrayMaker** component has two properties, *NewValue* and *Unsorted*, and one method, *Clear*. *NewValue* is used to add a value to the **CDynarray** object being created, while *Unsorted* is used to retrieve the address of the newly created **CDynarray** object. The *Clear* method is used to begin a

new **CDynarray** object. The property descriptions are given in Figure 13.16, and the method description is given in Figure 13.17. When the *Unsorted* property of the DynarrayMaker is accessed, a new copy of the **CDynarray** object is created. The requester then becomes the owner of the newly created object.

Property Design Table		
Name	Type	Function
<b>NewValue</b>	Long Integer Default=None Maximum=None Minimum=None	Used to add a new value to the CDynarray object being created. All values assigned to this property will be accumulated into a CDynarray object. <b>Restrictions:</b> Write Only, Run-Time Only
<b>Unsorted</b>	Long Integer Default=NULL	Contains the address of a newly created CDynarray object. The CDynarray object will contain all integers assigned to the NewValue property since the last call to the Clear method. <b>Restrictions:</b> Read Only, Run-Time Only

**Figure 13.16. Property Table for CDynarrayMaker.**

Method Description			
<b>Name</b>	Clear		
<b>Return Value</b>	Void		
<b>Description</b>	Deletes all elements from the queue.		
<b>Arguments</b>	<b>Name</b>	<b>Type</b>	<b>Description</b>
Void			

**Figure 13.17. Clear Method of CDynarrayMaker.**

The DynarrayViewer component subclasses a multi-line text box, and presents the list of sorted integers in ASCII form. This component has a single property, *Sorted*. When the address of a **CDynarray** object is assigned to this property, it will be displayed in the subclassed text box. Despite the property name, the **CDynarray** object does not have to be sorted. The DynarrayViewer assumes ownership of the objects assigned to it, in the

sense that it is permitted to delete the object when it is no longer required. The description of the *Sorted* property is given in Figure 13.18.

Property Design Table		
Name	Type	Function
<b>Sorted</b>	Long Integer Default=NULL	Must contain the address of a CDynarray object. The CDynarray object will be displayed in ASCII form. <b>Restrictions:</b> Write Only, Run-Time Only

**Figure 13.18. Property Table for CDynarray Viewer.**

### 13.6.2. Polynomial Evaluator

The Polynomial Evaluator is an example of a hot filter. The purpose of this component is to filter floating point numbers using the rule  $y=P(x)$ . The filter is passed a stream of  $x$  values, and produces a stream of  $y$  values. The coefficients of the polynomial  $P$  determine the nature of the transformation. Both the coefficients and the degree of  $P$  can be changed at run time, but are assumed to be somewhat stable once they have been assigned.

Although this component can give the flavor of a hot filter, the more interesting hot filters are much more complicated than this. For example, we have created several circuit simulator filters in our research.. The purpose of these filters is to transform a sequence of binary inputs into a sequence of binary outputs. Before feeding input vectors to the simulator, it is necessary to configure the filter by passing it the description of a digital circuit. The digital circuit description is itself created by a series of filters. Although the Polynomial Evaluator is nowhere near this level of complexity, its basic principles are similar to those of the circuit simulator.

In a hot filter the properties and methods are divided into two distinct categories, those used for configuration, and those used for filtering. The design processes for the two sets are more or less independent of one another. The Polynomial Evaluator has two properties that are used for configuration, *Degree*, and *Coefficient*. Both properties are read-write in nature. *Coefficient* is an array of floating point numbers which is indexed from zero through the value of the *Degree* property. The *Degree* property must be assigned a value before the coefficients can be supplied. The description of these two properties is given in Figure 13.19.

Property Design Table		
Name	Type	Function
<b>Degree</b>	Long Integer Default=0 Minimum=0 Maximum=None	Sets the degree of the polynomial. The coefficient array is destroyed when a new value is assigned to this property. This property must be assigned a value before the coefficients of a new polynomial can be supplied. <b>Restrictions:</b> Run-Time Only.
<b>Coefficient</b>	Floating Point Array Default=0.0 Minimum=None Maximum=None	An array that contains the coefficients of the current polynomial. The maximum index is equal to the degree of the polynomial, and the minimum index is zero. <b>Restrictions:</b> Run-Time Only.

**Figure 13.19. Property Table for the Polynomial Evaluator.**

The implementation of the *Degree* property is given in Figure 13.20, while the implementation of the *Coefficient* property is given in Figure 13.21. The implementations are straightforward.

```

void CPolyEvalCtrl::OnDegreeChanged()
{
    if (m_degree < 0)
    {
        m_degree = 0; // Enforce minimum value
    }
    if (A != NULL)
    {
        delete [] A; // delete existing coefficient array
    }
    A = new float [m_degree+1]; // create new coefficient array
    for (long i = 1 ; i <= m_degree ; i++)
    {
        A[i] = 0.0; // initialize all coefficients to zero
    }
    A[0] = 1.0; // set constant coefficient to 1
}

```

**Figure 13.20. The Implementation of the Degree Property.**

```

float CPolyEvalCtrl::GetCoefficient(long Idx)
{
    if (Idx <= m_degree)
    {
        return A[Idx];
    }
    return 0.0f;
}

void CPolyEvalCtrl::SetCoefficient(long Idx, float newValue)
{
    if (Idx <= m_degree)
    {
        A[Idx] = newValue;
    }
}

```

**Figure 13.21. The Implementation of the Coefficient Property.**

In most hot filters, the properties and methods used for configuration will be more complex than those used for filtering. The polynomial evaluator is no exception. For filtering we have one method, named  $p$ , that will be used to evaluate the current polynomial on a floating point operand. The method description is given in Figure 13.22, and the implementation, which is the standard Newtonian algorithm, is given in Figure 13.23.

Method Description			
<b>Name</b>	p		
<b>Return Value</b>	Floating Point		
<b>Description</b>	Evaluates the current polynomial on its argument and returns the result.		
Arguments	Name	Type	Description
	x	Floating Point	Value for polynomial evaluation.

**Figure 13.22. Description of the p Method.**

```

float CPolyEvalCtrl::p(float x)
{
    float rv = A[m_degree];
    for (long i = m_degree - 1 ; i >= 0 ; i--)
    {
        rv = rv*x + A[i];
    }
    return rv;
}

```

**Figure 13.23. Implementation of the p method.**

### 13.6.3. The Simple Graphical Editor Scriptor Control

For our final example, we will return to the Simple Graphical Editor first introduced in Chapter 7. Suppose you have created a similar editor, and wish to create a series of web-pages explaining how to use your component. You may wish to illustrate some of the drawing features of your component by embedding the component directly in the web page and drawing the objects in response to button clicks or other user input. This would, of course, require that you have some sort of background editor to create the drawings programmatically. You could certainly use a background editor similar to the SGE Background editor described in Chapter 8, but if you want to illustrate complex drawings, this would require a large number of function calls, and some prior experimentation with the parameters to make sure that you got the drawing just right. A

simpler alternative is to use a scripting component, such as that described in this section. The scripting component is a two-way filter that can transform a **CGraphicList** object into an ASCII script, or transform an ASCII script into a **CGraphicList** object. The scripts can be used incrementally to create a drawing in stages. This component is based on the serialization components described in Chapter 9.

You would use this component by first creating a drawing program in Visual Basic or some other language, and adding a scripting component to it. You would first create a drawing, and then use the scripting component to transform the drawing into an ASCII script. The script could then be displayed in a multi-line text box, and cut and pasted into your web page. Your web page would also have a scripting component, which would be used to transform the script back into a drawing. It is possible to separate these two functions into two different components, but for simplicity, we have integrated them into a single component.

Like the other SGE components, the SGEScriptor component has a set of model manipulation properties and methods. These include the *ModelHandle* property, the *ReleaseModel* method and the *DeleteModel* Method. Because these are identical to the properties and methods of the other SGE components, their descriptions are omitted. In addition to these, the SGEScriptor component has a *New* method to create a new drawing. When a new script is processed by the component, all drawing objects in the script are added to the current drawing. The *New* method is used to terminate the existing drawing and start a new one. The description of this method can be found in Figure 13.24. It is not necessary to call this method before adding scripts.

Method Description			
<b>Name</b>	New		
<b>Return Value</b>	Void		
<b>Description</b>	Deletes the existing model and creates a new empty model. Use <i>ReleaseModel</i> prior to calling this method to prevent deletion of the existing mode.		
<b>Arguments</b>	<b>Name</b>	<b>Type</b>	<b>Description</b>
Void			

**Figure 13.24. Description of the New Method.**

The SGEsScriptor is both a hot and a cold filter. When transforming drawings into scripts, it acts as a cold filter, but when transforming scripts into drawings, it acts as a hot filter with the model manipulation features playing the configuration role. The filtering itself is done using the *Script* property.

The implementation of the *Script* property is quite complex. When the value of the property is read, the existing model is serialized, and the resultant string is returned as the value of the property. (An empty string is returned if there is no model.) When the value of the property is written, the script is parsed into lines, and each line is passed to the deserialization function used by the serialization components of Chapter 9. The resultant drawing objects are added to the current drawing. If there is no current drawing, then a new empty drawing is created. The description of the *Script* property is given in Figure 13.25.

Property Design Table		
Name	Type	Function
<b>Script</b>	String Default=Empty	When read, transforms the existing drawing into a script and returns the script. When written, deserializes the input script and adds the drawing objects to the existing drawing. If there is no existing drawing an empty drawing is created before adding objects. <b>Restrictions:</b> Run-Time Only.

**Figure 13.25. Property Table for the Script Property.**

Figure 13.26 contains the implementation of the read portion of the *Script* property.

The majority of the work is done by the *SSerialize* function of the **CGraphicList** object.

```
// serialize model and return resultant string
BSTR CSGEScriptorCtrl::GetScript()
{
    CString Buffer;
    Buffer.Empty();
    if (Model != NULL) // Normally Supplied by ModelHandle
    {
        Model->SSerialize(Buffer);
    }
    // still empty if model is null
    return Buffer.AllocSysString(); // convert CString to BSTR
}
```

**Figure 13.26. Read Portion of the Script Property.**

The majority of the write portion of the *Script* property is devoted to parsing the input script into lines. When the new objects are added to the drawing, they will be added as part of the selection. It will be possible to undo the addition, and the dirty flag will of the drawing will be set. The implementation of the write portion of the *Script* property is given in Figure 13.27.

```

// deserialize new Script property value, and add to current model
// create new model if necessary
void CSGEScriptorCtrl::SetScript(LPCTSTR lpszNewValue)
{
    if (Model == NULL) // Create model if there isn't one
    {
        Model = new CGraphicList;
    }
    // deselect everything so we can add new script to selection
    Model->ClearSelect();
    long BP = 0;
    long BufferBytes = strlen(lpszNewValue);
    CString S;
    S.Empty();
    // standard before adding a new UNDO
    Model->ClearRedo();
    // we will permit script to be undone
    Model->NewUndoCommand();
    // scan each byte of the new script
    // when an end of line is found, pass the line to
    // SDeserialize (global function, part of model)
    while (BP < BufferBytes)
    {
        // accumulate current byte
        S += lpszNewValue[BP];
        if (lpszNewValue[BP] == '\n')
        {
            // EOL found, deserialize S, and erase it
            CGraphicObject * Obj = SDeserialize(S);
            S.Empty();
            // if format OK, add new object to model,
            // with UNDO, with Dirty bit, added to selection
            if (Obj != NULL)
            {
                Model->InsertObject(Obj);
            }
        }
        // Next byte
        BP++;
    }
    if (!S.IsEmpty())
    {
        // If there are bytes left over in S, it is probably an
        // unterminated line, so go ahead and process it.
        CGraphicObject * Obj = SDeserialize(S);
        S.Empty();
        // if format OK, add to model
        if (Obj != NULL)
        {
            Model->InsertObject(Obj);
        }
    }
}
}

```

Figure 13.27. The Write Portion of the Script Property.

### **13.7. A Review of the Methodology**

Since most of the complexity of a Filter lies in its internal implementation, one must begin with a careful design of this portion of the component. Except for the simplest Filters, this will entail a detailed object-oriented design of the component internals. The component interface is used only for input and output.

The main interface decision is whether to use file-level input and output or object-level input and output. Some combination of the two can also be used. In the case of object-level input and output, the input can be supplied in a single block (as it is in our examples) or it can be streamed. In general, file-level input and output permits larger amounts of data than object-level input and output. Streaming the input and output can provide a compromise between the two. In streamed input, a method call or property assignment is used to start the input process. When input becomes exhausted, the component issues an event requesting more input. Streamed output works the same way, with the component issuing events when a portion of the output is available.

Once the design of the input and output interfaces has been completed, the rest of the development proceeds as if the component were an ordinary program.

### **13.8. Conclusion**

Because they contain general algorithms of arbitrary complexity, Filters are the mainstay of heavy-duty computing. Virtually any algorithm can be embedded in a Filter, allowing Filters to act as a bridge between program-level design and component-level design. Many traditional sorts of programs can be turned into components by embedding them in a File Filter wrapper.

Despite these advantages, there are few Filters on the market today. The most commonly available filters are those that provide some sort of image processing. For experienced programmers who are not familiar with component-level design, Filters are the best way to break into component level programming. In the future, we should expect to see Filters for many different applications appearing on the market.

### **13.9. Exercises**

1. Select your favorite algorithm from your algorithms course and implement it as a filter. Create the required generator and display components to test your filter.
2. Take a project that you completed for another course, and embed it in one or more filter components.
3. Create a text filter that converts text strings to upper case.
4. Create a hot filter that accepts a list of words as configuration data. The filter will filter strings searching for the words in the list. When one is found, it will be replaced by the string “(deleted)”. (Without the quotes.)
5. Create a filter that reverses the characters of a string.