SPECIAL CLASS DESIGN AND OBJECT BEHAVIOR

This chapter looks at special object behavior and how that behavior affects the design of individual classes and class hierarchies. You will learn about the following kinds of classes and behaviors:

- Abstract classes
- Metamorphic classes
- Extendible structures
- Disciplined classes
- Disabled classes
- Deferred binding and rebinding

Abstract classes

Metamorphic classes

Extendible structures

State-engine objects

Disabled objects

Deferred binding and rebinding

The C++ programs in this chapter are all compiled as QuickWin applications. The various .H, .CPP, and .MAK files should be located in the directory \MSVC\VCSECR.

ABSTRACT CLASSES

Abstraction is an analysis tool that removes details from an examined system. It enables you to focus on what happens to the system and to ignore how the changes happen. Object-oriented analysis and design methods use abstraction as an effective way to study the domain of a problem and examine the operations of the classes involved. In this section, I discuss abstract classes and their role in the design of a class hierarchy. You will learn about the following:

- Basic rules for declaring abstract classes
- Abstract classes as base classes in hierarchies
- Abstract classes as base classes in sub-hierarchies

Basic Rules for Abstract Classes

When you design a class hierarchy, you can specify the common operations in that hierarchy using an abstract class. The abstract class empowers you to specify what happens to the instances of the various descendant classes. The descendants themselves fill in the details on how to carry out the operations of a class.

Abstract classes fall in two categories: *purely abstract* and *partially functioning*. Purely abstract classes specify the public, protected, and private member functions common to the descendants in the class hierarchy. The definition of these member functions contains no statements. Thus, abstract classes are completely non-functional.

Partially functioning abstract classes specify the public, protected, and private member functions, along with data members that are common to all or most descendants. In addition, they implement some of the member functions that are common to all or most of the descendants.

The two types of abstract classes have the following aspects in common:

- You need to declare all of the member functions and data members as protected or private. In addition, you need to append = 0 after the

parameter list to tell the C++ compiler that you are declaring a purely abstract member function.

- The member functions not implemented by the abstract classes must be declared virtual. This kind of declaration ensures that the descendants declare these member functions also as virtual and use the same parameter list. The benefit of the consistent parameter list is the support of polymorphic behavior. The partially functioning abstract classes benefit more from this feature than the purely abstract classes.

Abstract Classes as Base Classes

The most common place for an abstract class in a class hierarchy is at the root of the hierarchy. When the hierarchy is simple, comprising a single chain of inheritance, a single abstract class suffices. When the class hierarchy has many branches, you may end up with several abstract classes. Figure 1.1 shows a schema of class hierarchy that contains several abstract classes. The `cAbsType1` class is common to all the hierarchy branches. Each branch has an additional abstract class to specify more operations that are particular to that branch.

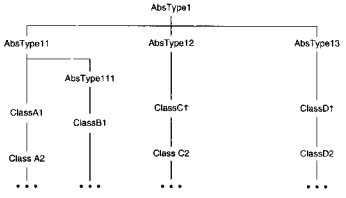

Figure 1.1. A class hierarchy that contains abstract classes.

Let's look at an example. Listing 1.1 shows the header file ABSSTACK.H and Listing 1.2 shows the source code for the ABSSTACK.CPP library file. This library implements a stack class hierarchy using an abstract class. The library unit supports stacks of strings with the following basic operations:

- Clearing a stack
- Pushing data into a stack
- Popping data off a stack

The library declares a hierarchy of three classes: `CAbsStack`, `CStrStack`, and `CVMStrStack`. The class `CAbsStack`, as the name might suggest, is an abstract class. It declares all of its data members and member functions as protected. This class is a partially functioning abstract type, since it declares data members and contains a few functioning (or implemented, if you prefer) member functions. The data members `nHeight` and `bAllocateError` maintain the stack height and data allocation status, respectively. This class declares the following member functions:

- The constructor and destructor
- The Boolean function `GetAllocateError`, which returns the value in the `bAllocateError` data member
- The Boolean function `IsEmpty`, which returns `TRUE` when the `nHeight` data member is 0; otherwise, the function yields `FALSE`
- The virtual function `Push`, which pushes a string onto the stack
- The virtual Boolean function `Pop`, which pops a string off the stack
- The virtual function `Clear`, which clears the stack

The declaration of the last three member functions includes an equal sign followed by a zero (`=0`). This syntax tells the C++ compiler that these functions are purely abstract ones.

The library file also declares the class `CStrStack` as a descendant of `CAbsStack`. The descendant class models a heap-based stack of strings. The actual implementation uses a dynamic linked list. The `CStrStack` class declares the pointer `pTop` to access the supporting dynamic linked list. This class declares a functioning constructor, virtual destructor, and the virtual member functions `Push`, `Pop`, and `Clear`. The statements in these member functions specify how the stack operations are implemented. The constructor initializes the stack by initializing the supporting dynamic linked list. The destructor clears the supporting linked list.

The library unit also declares the class `CVMStrStack` as another descendant of class `CAbsStack`. This descendant class models a disk-based stack of strings. The actual implementation uses a random-access file stream. The `CVMStrStack` class declares

the `szDataBuffer`, `szErrorMessage`, and `VMfile` data members. This class declares a functioning constructor, virtual destructor, and the `virtual` member functions `Push`, `Pop`, and `Clear`. The statements in these member functions specify how the stack operations are implemented—with the help of the supporting file. The constructor opens the supporting random-access stream file. The destructor assigns 0 to member `nHeight` and closes the stream `VMfile`.

LISTING 1.1. THE SOURCE CODE FOR THE ABSSTACK.H HEADER FILE.

```
#ifndef _ABSSTACK_H
#define _ABSSTACK_H

#include <fstream.h>

/*
   Implements classes of generic stacks with the following set of
   operations:

     + Push
     + Pop
     + Clear
*/


const unsigned MAX_STR = 80;
enum Boolean { false, true };

// ********************** Abstract Stack ****************
class CAbsStack
{
  protected:
    unsigned nHeight; // height of stack
    Boolean bAllocateError; // dynamic allocation error

    CAbsStack() {};
    virtual ~CAbsStack() { };

    // ****************** State Query Methods *****************
    Boolean GetAllocateError() { return bAllocateError; }
    Boolean IsEmpty() { return (nHeight == 0) ? true : false; }

    // ************** Object Manipulation Methods *************
    virtual void Push(const char* szStr) = 0;
    virtual Boolean Pop(char* szStr) = 0;
    virtual void Clear() = 0;
};
```

continues

LISTING 1.1. CONTINUED

```
struct StrStackRec {
  char szNodeData[MAX_STR+1];
  StrStackRec* pNextLink;
};

class CStrStack : public CAbsStack
{
  public:
    CStrStack();
    virtual ~CStrStack() { Clear(); }

    // ************** Object Manipulation Methods *************
    virtual void Push(const char* szStr);
    virtual Boolean Pop(char* szStr);
    virtual void Clear();

  protected:
    StrStackRec* pTop; // pointer to the top of the stack
};

class CVMStrStack : public CAbsStack
{
  public:
    CVMStrStack(const char* Filename);
    virtual ~CVMStrStack() { Clear(); }

    // ****************** State Query Methods *****************
    char* GetErrorMessage() { return szErrorMessage; }

    // ************** Object Manipulation Methods *************
    virtual void Push(const char* szStr);
    virtual Boolean Pop(char* szStr);
    virtual void Clear();

  protected:
    char szDataBuffer[MAX_STR+1]; // data buffer
    char szErrorMessage[MAX_STR+1]; // error message
    fstream VMfile; // virtual stream handle
};

#endif
```

LISTING 1.2. THE SOURCE CODE FOR THE ABSSTACK.CPP LIBRARY FILE.

```
#include "absstack.h"
#include <string.h>
```

```
//---------------------------------------- CStrStack ----------

CStrStack::CStrStack()
// constructor to initialize generic stack
{
  nHeight = 0;
  bAllocateError = false;
  pTop = NULL;
}

//---------------------------------------------- Push ----------

void CStrStack::Push(const char* szStr)
// push the data accessed by STRING szStr onto the stack
{
  StrStackRec* p;

  bAllocateError = false;
  if (pTop) {
    p = new StrStackRec; // allocate new stack element
    if (!p) {
      bAllocateError = true;
      return;
    }
    strcpy(p->szNodeData, szStr);
    p->pNextLink = pTop;
    pTop = p;
  }
  else {
    pTop = new StrStackRec;
    if (!pTop) {
      bAllocateError = true;
      return;
    }
    strcpy(pTop->szNodeData, szStr);
    pTop->pNextLink = NULL;
  }
  nHeight++;
}

//---------------------------------------------- Pop ----------

Boolean CStrStack::Pop(char* szStr)
/* Pops the top of the stack and returns a Boolean value.
  Function returns true if the operation was successful.  A
  false value is returned if the Pop message is sent to an empty
  stack.
*/
{
  StrStackRec* p;
```

continues

LISTING 1.2. CONTINUED

```
  if (nHeight > 0) {
    strcpy(szStr, pTop->szNodeData);
    p = pTop;
    pTop = pTop->pNextLink;
    delete p; // deallocate stack node
    nHeight--;
    return true; // return function value
  }
  else
    return false; // return function value
}

//---------------------------------------- Clear ----------

void CStrStack::Clear()
// clear the generic stack object
{
  char szStr[MAX_STR+1];

  while (Pop(szStr))
        /* do nothing */;
}

//---------------------------------------- CVMStrStack ----------

CVMStrStack::CVMStrStack(const char* Filename)
// constructor to initialize generic stack
{
  nHeight = 0;
  VMfile.open(Filename, ios::in | ios::out | ios::binary);
  if (!VMfile) {
    strcpy(szErrorMessage, "Cannot open file ");
    strcat(szErrorMessage, Filename);
    return;
  }
  else {
    bAllocateError = false;
    strcpy(szErrorMessage, "");
  }
}


//---------------------------------------- Push ----------

void CVMStrStack::Push(const char* szStr)
// push the data accessed by parameter szStr onto the stack
{
  nHeight++;
  VMfile.seekg((nHeight-1) * (MAX_STR+1));
```

```
  VMfile.write((unsigned char*) szStr, MAX_STR+1);
}

//---------------------------------------- Pop -----------

Boolean CVMStrStack::Pop(char* szStr)
// pop the top of the stack and return a Boolean value
{
  if (nHeight > 0) {
    nHeight--;
    VMfile.seekg(nHeight * (MAX_STR+1));
    VMfile.read((unsigned char*) szStr, MAX_STR+1);
    return true;
  }
  else
    return false;
}

//---------------------------------------- Clear ----------

void CVMStrStack::Clear()
// clear the generic stack object
{
  nHeight = 0;
  VMfile.close();
}
```

Let's look at a test program for the ABSSTACK.CPP library. Listing 1.3 contains this program STACK1.CPP, which tests the classes declared in Listing 1.1. This program declares objects `aStack` and `aVMStack` as the instances of classes `CStrStack` and `CVMStrStack`, respectively. This program performs the following relevant tasks:

- Instantiates the instances `aStack` and `aVMStack`. The program creates the latter object using the supporting file VS.DAT.
- Uses a `for` loop to push onto the stack `aStack` the elements of the string array `pStringArray`. The program sends a `Push` message to the object `aStack` to push each string.
- Pops the strings off the stack object `aStack`. The program sends the message `Pop` to the instance `aStack` using a `while` loop. The loop iterates as long as there is an item popped off the stack. The body of the `while` loop displays the string that is popped off the stack.
- Uses a `for` loop to push onto the stack `aVMStack` the elements of the string array `pStringArray`. The program sends the message `Push` to the object `aVMStack` to push each string.

- Pops the strings off the stack `aVMStack`. The program sends the message `Pop` to the instance `aVMStack` and uses a `while` loop. This loop iterates as long as there is an item popped off the stack. The body of the `while` loop displays the string that is popped off the stack.

The STACK1.MAK file should contain the ABSSTACK.CPP and STACK1.CPP files and should be located in the directory \MSVC\VCSECR.

LISTING 1.3. THE SOURCE CODE FOR THE STACK1.CPP PROGRAM FILE, WHICH TESTS THE CLASSES IN ABSSTACK.H.

```
/*
    Program to test stacks of strings
*/

#include "absstack.h"
#include <iostream.h>

main()
{
  const unsigned MAX_STRINGS = 10;
  char* Filename = "aVMStack.DAT";
  char* pStringArray[MAX_STRINGS] =
              { "California", "Virginia", "Michigan",
                "New York", "Washington", "Nevada",
                "Alabama", "Alaska", "Florida", "Maine"};
  char chAKey;
  char szString[MAX_STR+1];
  CStrStack aStack;
  CVMStrStack aVMStack(Filename);

  cout << "Testing heap-based stacks objects\n\n";
  for (int i = 0; i < MAX_STRINGS; i++) {
    cout << "Pushing ";
    cout.width(12);
    cout << pStringArray[i] << " into the stack\n";
    aStack.Push(pStringArray[i]);
  }
  cout << "\nEnter any character to continue... ";
  cin >> chAKey;
  cout << "\n";

  while (aStack.Pop(szString)) {
    cout << "Popping off ";
    cout.width(12);
    cout << szString << " from the stack\n";
  };
  cout << "\nEnter any character to continue... ";
```

```
  cin >> chAKey;
  cout << "\a\n\n\n";

  cout << "Testing virtual stacks objects\n\n";
  for (i = 0; i < MAX_STRINGS; i++) {
    cout << "Pushing ";
    cout.width(12);
    cout << pStringArray[i] << " into the stack\n";
    aVMStack.Push(pStringArray[i]);
  }
  cout << "\nEnter any character to continue... ";
  cin >>  chAKey;
  cout << "\n";

  while (aVMStack.Pop(szString)) {
    cout << "Popping off ";
    cout.width(12);
    cout << szString << " from the stack\n";
  }

  return 0;
}
```

Here is a sample output for the program in Listing 1.3:

```
Testing heap-based stacks objects

Pushing   California into the stack
Pushing     Virginia into the stack
Pushing     Michigan into the stack
Pushing     New York into the stack
Pushing   Washington into the stack
Pushing       Nevada into the stack
Pushing      Alabama into the stack
Pushing       Alaska into the stack
Pushing      Florida into the stack
Pushing        Maine into the stack

Enter any character to continue... c

Popping off        Maine from the stack
Popping off      Florida from the stack
Popping off       Alaska from the stack
Popping off      Alabama from the stack
Popping off       Nevada from the stack
Popping off   Washington from the stack
Popping off     New York from the stack
Popping off     Michigan from the stack
Popping off     Virginia from the stack
Popping off   California from the stack
```

```
Enter any character to continue... c



Testing virtual stacks objects

Pushing   California into the stack
Pushing     Virginia into the stack
Pushing     Michigan into the stack
Pushing     New York into the stack
Pushing   Washington into the stack
Pushing       Nevada into the stack
Pushing      Alabama into the stack
Pushing       Alaska into the stack
Pushing      Florida into the stack
Pushing        Maine into the stack

Enter any character to continue... c

Popping off        Maine from the stack
Popping off      Florida from the stack
Popping off       Alaska from the stack
Popping off      Alabama from the stack
Popping off       Nevada from the stack
Popping off   Washington from the stack
Popping off     New York from the stack
Popping off     Michigan from the stack
Popping off     Virginia from the stack
Popping off   California from the stack
```

Abstract Objects in Sub-Hierarchies

You can use abstract classes as the base classes in sub-hierarchies. In other words, such abstract classes have non-abstract class parents. This kind of abstract class occurs more often in sophisticated class hierarchies, such as Turbo Vision. Figure 1.2 shows a class hierarchy that contains internal abstract classes.

Conceptually, both kinds of abstract classes are similar. The kind of abstract classes that I presented in the last section are concentrated at the root of the hierarchy. The kind of abstract classes that I present here are located well inside the class hierarchy. In addition, this genre of class tends to be partially functioning. Of course, declaring all of the data members and member functions as private ensures that client programs don't use this abstract type accidentally.

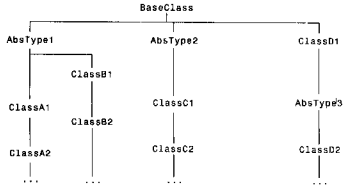

Figure 1.2. A class hierarchy that contains internal abstract classes.

Let's look at a program that illustrates an abstract class at the root of a sub-hierarchy. Listings 1.4 and 1.5 show the source code for the header file ABSARRAY.H and the library file ABSARRAY.CPP. This program defines the following classes:

- The `CArray` class models an unordered dynamic array of strings.
- The abstract class `CAbsSortArray` is a descendant of `CArray` that defines operations for the next classes.
- The `CSortArray` class is a descendant of `CAbsSortArray` that models ordered arrays. This class supports case-sensitive sorting and searching.
- The `CNocaseSortArray` class is another descendant of `CAbsSortArray` that models ordered arrays. This class supports case-insensitive sorting and searching.

The classes `CSortArray` and `CNocaseSortArray` offer two sample variations of ordered arrays. You can add more sibling classes, for example, to sort and search by specific portions of a string, or to arrange arrays in descending order.

The *CArray* Class

After introducing the classes in the hierarchy of dynamic string arrays, let me explain each class in more detail. The `CArray` class has a constructor, a destructor, a set of data members, and a group of member functions.

This class declares the following data members:

- The `pDataPtr` data member is the pointer to the dynamic array of strings.
- The `nMaxSize` data member stores the number of elements in the dynamic array.
- The `nWorkSize` data member contains the number of elements with meaningful information. Values for `nWorkSize` range from `0` to `nMaxSize`.
- The Boolean data member `bAllocateError` stores the dynamic-allocation error status.

The constructor allocates the dynamic space specified by the parameter `nArraySize`. This constructor also assigns the argument for this parameter to the `nMaxSize` data member, sets the `nWorkSize` field to zero, and assigns empty strings to the elements of the dynamic array. The destructor regains the dynamic memory allocated to the array.

The `CArray` class declares the following member functions:

- The function `GetMaxSize` returns the value in the data member `nMaxSize`.
- The function `GetWorkSize` returns the value in the data member `nWorkSize`.
- The Boolean function `GetAllocateError` returns the value in the data member `bAllocateError`. Use this function to determine whether an instance was successfully created.
- The virtual Boolean function `Store` saves the string `szStr` in the array element number `nIndex`. This function returns `TRUE` if the argument for `Index` is valid (that is, in the range of `0` to `nMaxSize - 1`). Otherwise, this function yields `FALSE`.
- The virtual Boolean function `Recall` retrieves the string from array element number `nIndex`. This function returns `TRUE` if the argument for `nIndex` is valid (that is, in the range of `0` to `nWorkSize - 1`). Otherwise, the function yields `FALSE`. The parameter `szStr` passes the retrieved string when the argument for parameter `nIndex` is valid.
- The virtual function `Search` returns the index of the array element that matches the search string `szKey`. If there is no match, this function yields the constant `NOT_FOUND` (that is, `0xffff`). Since the class `CArray` models an unordered array of strings, the `Search` member function performs a linear search.

The CAbsSortArray Class

The `CAbsSortArray` class is a descendant of `CArray` that models an abstract class for sorted arrays. This class declares its data member `bInOrder`, a constructor, and member functions. The `bInOrder` data member stores the sort order status of an instance. This class declares the following member functions:

- The virtual Boolean function `Store`. This member function invokes the inherited `Store` member function and assigns `FALSE` to the data member `bInOrder` if the inherited member function returns `TRUE`.
- The virtual function `Search` is an empty shell that specifies the declaration of the `Search` member functions in descendant classes. These member functions would conduct binary searches on the ordered array.
- The virtual function `Sort` is an empty shell that specifies the declaration of the `Sort` member functions in the descendant classes. This function assigns `false` to the member `bInOrder`.

The CSortArray Class

The `CSortArray` class is a descendant of `CAbsSortArray`. This class inherits the following from the parent and ancestor classes:

- The data members from the `CArray` and `CAbsSortArray` classes
- The `Recall` member function from class `CArray`
- The `Store` member function from class `CAbsSortArray`

`CSortArray` declares a constructor, the member function `Search`, and the member function `Sort`. These two member functions are fully functioning. The `Search` member function performs a case-sensitive binary search for the parameter `szKey` in the dynamic array. This member function first examines the data member `bInOrder` to determine whether the array needs to be sorted before performing the binary search. The `Sort` member function performs case-sensitive sorting using the Comb sort method.

The CNocaseSortArray Class

The `CNocaseSortArray` class is another descendant of `CAbsSortArray`. This class inherits the same items from its parent and ancestor class as `CSortArray`. The `CNocaseSortArray` declares a constructor, the member function `Search`, and the member function `Sort`. These two member functions are fully functioning.

The `Search` member function performs a case-insensitive binary search for the parameter `szKey` in the dynamic array. This member function first examines the data member `bInOrder` to determine whether the array needs to be sorted before proceeding with the binary search. The `Sort` member function carries out case-insensitive sorting using the `Comb` sort method.

LISTING 1.4. THE SOURCE CODE FOR THE ABSARRAY.H HEADER FILE.

```
// file that declares arrays with an abstract object type

#ifndef _ABSARRAY_H
#define _ABSARRAY_H

const unsigned NOT_FOUND = 0xffff;
const unsigned DEFAULT_SIZE = 10;
const unsigned NIL = 0;

enum Boolean { false, true };

class CArray
{
  public:
    CArray(unsigned nArraySize = DEFAULT_SIZE);
    virtual ~CArray();

    unsigned GetMaxSize() { return nMaxSize; }
    unsigned GetWorkSize() { return nWorkSize; }
    Boolean GetAllocateError() { return bAllocateError; }

    virtual Boolean Store(const char* szStr, unsigned nIndex);
    virtual Boolean Recall(char* szStr, unsigned nIndex);
    virtual unsigned Search(const char* szKey);

  protected:
    char** pDataPtr;
    unsigned nMaxSize;
    unsigned nWorkSize;
    Boolean bAllocateError;
};

class CAbsSortArray : public CArray
{
 public:
   CAbsSortArray(unsigned nArraySize = DEFAULT_SIZE)
     : CArray(nArraySize) { bInOrder = false; }
   virtual Boolean Store(const char* szStr, unsigned nIndex);
   virtual unsigned Search(const char* szKey)
```

```
    { return NOT_FOUND; };
  virtual void Sort() { bInOrder = false; };

protected:
  Boolean bInOrder;
};

class CSortArray : public CAbsSortArray
{
  public:
    CSortArray(unsigned nArraySize = DEFAULT_SIZE)
     : CAbsSortArray(nArraySize) {}
    virtual unsigned Search(const char* szKey);
    virtual void Sort();
};

class CNocaseSortArray : public CAbsSortArray
{
  public:
    CNocaseSortArray(unsigned nArraySize = DEFAULT_SIZE)
      : CAbsSortArray(nArraySize) {}
    virtual unsigned Search(const char* szKey);
    virtual void Sort();
};

#endif
```

LISTING 1.5. THE SOURCE CODE FOR THE ABSARRAY.CPP LIBRARY FILE.

```
// library that implements arrays with an abstract object type

#include "absarray.h"
#include <string.h>

//---------------------------------------- CArray ----------

CArray::CArray(unsigned nArraySize)
// construct instance of CArray
{
  nMaxSize = (nArraySize == 0) ? DEFAULT_SIZE : nArraySize;
  nWorkSize = 0;
  pDataPtr = new char*[nMaxSize];
  bAllocateError = (pDataPtr == NIL) ? true : false;
  if (!bAllocateError)
    for (unsigned i = 0; i < nMaxSize; i++)
      pDataPtr[i] = NIL;
}
```

CONTINUED