3 Collections, CFiles, and Serializing Objects

In this chapter:

This chapter can be skipped (at least temporarily) if you are anxious to move on from console applications to MFC windows applications.

However, collections are an important part of just about all computer applications. In fact, it's difficult to have an interesting application without using some sort of collection. So while you may skip ahead, you can return here for study when you find examples that involve collections.

In section 3.1 we quickly introduce two array classes. Depending on your background, you may want to spend a good amount of time here. In the first example the array components are actual objects. There are a number of suggested exercises if you feel you need more practice in array and object concepts. The other array consists of pointers to objects. Here is a good place to brush up on pointers.

In section 3.2 we briefly introduce class templates simply because we feel you should know they exist. You may eventually want to make use of a MFC template class. This section will get you started. We make no further use of templates in this book, however.

The CFile class is discussed in the next section. This class gives us the tools to do low-level file processing.

Serializing objects refers to saving and restoring the state of objects between program runs. This is discussed in section 3.5. You must read sections 3.4 and 3.5 before reading Chapter 14. In my mind, serializing objects is one of the most important and neatest features of the MFC application framework. When you get to Chapter 14 you will be impressed with the ease in which you can save and restore your document objects.

3.1 Object and Pointer Array Collections

A bushel basket is used to hold objects such as apples. But a basket is also an object — a container object. The term "collection object" is also used, because the basket can hold a collection of things.

An array is like a container in that it also holds things. Each component in an array can be an object. When the array concept is turned into a class, arrays become much easier to manipulate.

Array components can be actual objects or pointers to objects. These two types of arrays are illustrated next. Since the two programs do the same thing, you can easily focus on the arrays and their implementation.

3.1.1 The CClient Class

Throughout this chapter, we'll be using client objects instead of apples. We'll put clients into our arrays and other data structures. Here is our declaration for this class:

```
// File Client.h

class CClient
{
private:
  CString  mName;
  char     mGender;
  int      mAge;
public:
          CClient();
  virtual ~CClient();
  void    Input();
  void    Display() const;
};
```

```
// File Client.cpp

#include "client.h"

CClient::CClient()
: mName("xxx"), mGender('X'), mAge(0)
{
}

CClient::~CClient()
{
}

void CClient::Input()
{  char  name[20];  // needed for cin

   cout << "\n Name   : "; cin >> name;
   mName = name;
   cout << " Gender : "; cin >> mGender;
   cout << " Age    : "; cin >> mAge;
   return;
}

void CClient::Display() const
{
  CString  s;
  s.Format ("%-10s  %c %d",
            mName, mGender, mAge);
  cout << s;
}
```

3.1.2 An Array Class Of Clients, Ex03a

When manipulating the components of an array — x[0]. x[1], . . x[99] — we must always keep track of (in another variable) how many components in the array are actually being used. These two variables — the array variable, x, and the count, mSize — make good member variables for an array class.

Common tasks performed on an array include adding, inserting, and removing objects from the array. These tasks become the member functions of the class.

The only tricky part of implementing this class is shifting the components up one position in the array when an insertion is performed. Similarly, components have to be shifted down when a component is deleted.

This class is implemented on the next page in program Ex03a. Notice that the collection object in the main program is named ClientS, the plural of client.

I have underlined and made boldfaced the class object in the class description. This way, you can simply do a text replacement, if you wish to have an array of different objects.

Exploring Object Concepts with a Collection Object

Once we have a collection object containing objects in a program, the programming possibilities become more interesting.

The first question that arises is this: Suppose we want to perform some task on the collection; how should that task be implemented? There are three possibilities:

```
 // Program Ex03a
#include <afxwin.h>
#include <iostream.h>
#include "client.h"
#include "AnArray.h"

int main()
{
  CAnArray   ClientS;
  CClient    A;
  A.Input(); ClientS.Add(A);
  A.Input(); ClientS.Add(A);
  A.Input(); ClientS.InsertAt(A, 1);

  cout << "\nOutput array:";
  int i;
  for (i = 0; i < ClientS.GetSize(); i++)
  {
    A = ClientS.GetAt(i);
    cout << "\n";  A.Display();
  }

  ClientS.RemoveAt(0);
  cout << "\n\nOutput array, after RemoveAt(0):";
  for (i = 0; i < ClientS.GetSize(); i++)
  {
    A = ClientS.GetAt(i);
    cout << "\n";   A.Display();
  }
  return 0;
}
```

Output for Program Ex03a

```
Name   : Al
 Gender : m
 Age    : 45

 Name   : Bob
 Gender : m
 Age    : 21

 Name   : Chas
 Gender : m
 Age    : 92

Output array:
Al            m 45
Chas          m 92
Bob           m 21
f
Output array, after RemoveAt(0):
Chas          m 92
Bob           m 21
```

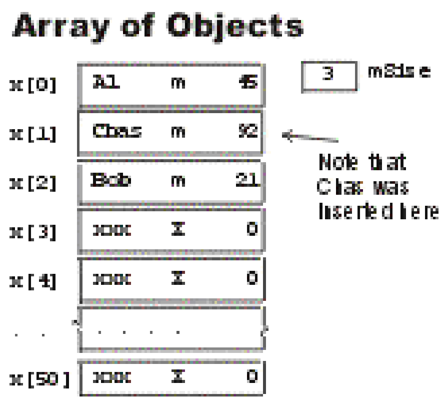


Figure 3.1 An array of objects

```
// AnArray.cpp: CAnArray implementation
#include "AnArray.h"

CAnArray::CAnArray()
:mSize(0)           {   }

CAnArray::~CAnArray() {   }

BOOL CAnArray::Add(const CClient& c)
{  x[mSize] = c;
   mSize++;
   return TRUE;
}
BOOL CAnArray::InsertAt(const CClient& c, int idx)
{
  if (mSize >= ARRAY_MAXSIZE)
    return FALSE;           // full
  if (idx < 0 || idx > mSize)
    return FALSE;           // illegal idx
  for (int i = mSize-1; i >= idx; i--)
    x[i+1] = x[i];          // shift up
  x[idx] = c;               // insert
  mSize++;
  return TRUE;
}
BOOL CAnArray::RemoveAt(int idx)
{
  if (idx < 0 || idx >= mSize)
    return FALSE;     // illegal idx
  for (int i = idx; i < mSize-1; i++)
    x[i] = x[i+1];     // shifting
  mSize--;
  return TRUE;
}
CClient CAnArray::GetAt(int idx) { return x[idx]; }

int CAnArray::GetSize() const    { return mSize;  }
```

```
// AnArray.h: CAnArray interface
#include "client.h"

#define ARRAY_MAXSIZE  500

class CAnArray
{
private:
  int       mSize;
  CClient  x[ARRAY_MAXSIZE];
public:
  CAnArray();
  virtual ~CAnArray();

  int      GetSize  () const;
  CClient GetAt    (int idx);
  BOOL     RemoveAt (int idx);
  BOOL     InsertAt (const CClient& c,
                     int idx);
  BOOL     Add      (const CClient& c);
};
```

❑ The task could be implemented as a method in the class.

❑ A class could be derived from the original collection class and the task could be implemented there.

❑ The task could be implemented as a regular C++ function where one of the arguments is the collection object.

3.1.3 Exercises

1. Before doing any of the exercises below, you should make program Ex03a interactive. Add a menu to the program so that the user can select what action is to be performed. The menu might look something like this:

```
Options: A'dd, I'nsert at, R'emove at,
   D'isplay all, E'xit program  Which ?
```

2. Adding clients to the collection is very tedious. Using the random number generator discussed in section 1.5.5, modify the CClient::Input() method so that a random name, gender, and age are generated when the Input method is called. You can generate a random name by using a code like this:

```
mName = "";
for (i = 0; i < 5; i++)
  mName = mName + (char)(65 + rand()%26);
```

The integer 65 is the ASCII code for the letter A. Of course, you might want to make the name lengths random too.

To make data entry even easier, you might further modify Input() so that when the user wants to add someone, you can have the method ask how many people are to be added.

3. Suppose we have two arrays of CClients.

```
CAnArray ClientList;
CAnArray SpecialList;
```

Which of the following actions are legal?

a) Assign one array object to the other:

```
SpecialList = ClientList;
```

b) Invoke a function on the array object. Can the parameters be either value or reference parameters?

```
DisplayYoungClients (ClientList);
```

c) Another example of the above:

```
RemoveDuplicates (SpecialList, ClientList)
```

4. An array of CPoints. Modify the CAnArray class so that it holds CPoint objects. Randomly generate 100 points and add them to the array.

a) Allow the user to enter a point. Then have the program find the CPoint in the array that is closest to the entered point.

b) Find the two points that are closest together. Use the TaxiCabDistance—see page 117—to simplify the calculations. This function just add the lengths of the horizontal and vertical legs of the triangle formed by joining the two points with a line.

5. An array of CRects: Suppose you have an array of randomly generated rectangles.

a) Allow the user to enter a point. Then have the program find the smallest rectangle that contains the point.

6. In the main program, is the following statement legal? Explain. If not legal, can it be made to be legal. Would you want to make it legal?

```
ClientS.x[2].Display();
```

3.1.4 An Array Class of Pointers to Clients

Instead of holding the actual client objects in the array of the "array collection object," we could put pointers to the client objects into the array. This is illustrated in program Ex03b on the next page.

Why might we want to hold pointers to objects rather than the actual objects?

❑ First, this could result in considerable space savings if we typically make use of fewer than half the components in the array.

❑ Second, if we wanted to rearrange the values in the array, we would only have to move around 32-bit pointers instead of entire CClient objects.

The diagrams of these two classes concisely illustrate how they differ.

3.1.5 Exercises

1. Compare the RemoveAt() member function for the two classes. How are they similar? How do they differ?

2. Write a member function named Exchange(idx1, idx2) for the two classes that exchanges the values at the two component positions.

3. Assume the following: a) A client object requires 24 bytes of storage, b) A pointer requires 4 bytes, and c) each of the arrays—the array of objects and the array of pointers to objects—have 100 components. If we have 50 objects how much space is con-

```
// Program Ex03b
#include <afxwin.h>
#include <iostream.h>
#include "client.h"
#include "AptrArray.h"

int main()
{
  CAptrArray  ClientS;
  CClient     *pClient;

  pClient = new CClient;
  pClient->Input();   ClientS.Add(pClient);
  pClient = new CClient;
  pClient->Input();   ClientS.Add(pClient);
  pClient = new CClient;
  pClient->Input();   ClientS.InsertAt(pClient,1);

  cout << "\n\nOutput array:";
  int i;
  for (i = 0; i < ClientS.GetSize(); i++)
  {
    pClient = ClientS.GetAt(i);
    cout << "\n";   pClient->Display();
  }

  ClientS.RemoveAt(0);
  cout << "\n\nOutput array, after RemoveAt(0):";

  for (i = 0; i < ClientS.GetSize(); i++)
  {
    pClient = ClientS.GetAt(i);
    cout << "\n";   pClient->Display();
  }
  return 0;
}
```

Output from Program Ex03b

```
Name   : Al
Gender : m
Age    : 45

Name   : Bob
Gender : m
Age    : 21

Name   : Chas
Gender : m
Age    : 92

Output array:
Al          m 45
Chas        m 92
Bob         m 21

Output array, after RemoveAt(0):
Chas        m 92
Bob         m 21
```

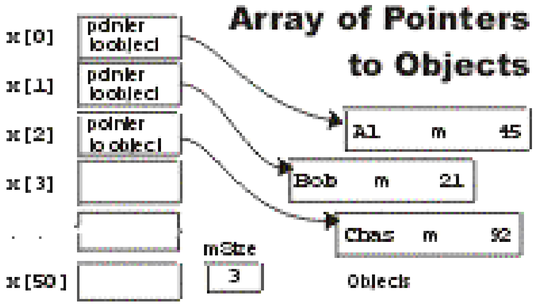


Figure 3.2 An array of pointers to objects

```
// AptrArray1.cpp: CAptrArray implementation
#include "AptrArray1.h"

CAptrArray::CAptrArray():mSize(0) { }

CAptrArray::~CAptrArray()
{  int i;
   for (i = 0; i < mSize; i++)
     delete x[i];
}
BOOL CAptrArray::Add (CClient *pc)
{ x[mSize] = pc;
  mSize++;
  return TRUE;
}
BOOL CAptrArray::InsertAt (CClient *pc, int idx)
{
  if (mSize >= ARRAY_MAXSIZE)
    return FALSE;           // full
  if (idx < 0 || idx > mSize)
    return FALSE;           // illegal idx
  for (int i = mSize-1; i >= idx; i--)
    x[i+1] = x[i];          // shift up
  x[idx] = pc;              // insert
  mSize++;
  return TRUE;
}
CClient& CAptrArray::GetAt(int idx)
 {   return x[idx]; }

int CAptrArray::GetSize() const
 {   return mSize; }
//          continued on right - >
```

```
// AptrArray1.h: CAptrArray interface

#include "client.h"
#define ARRAY_MAXSIZE  50

class CAptrArray
{
private:
  int       mSize;
  CClient   *x[ARRAY_MAXSIZE];
public:
  CAptrArray();
  virtual ~CAptrArray();

  int       GetSize  () const;
  CClient* GetAt     (int idx);
  BOOL      RemoveAt (int idx);
  BOOL      InsertAt (CClient *pc, int idx);
  BOOL      Add       (CClient *pc);
};
```

```
BOOL CAptrArray::RemoveAt(int idx)
{
  if (idx < 0 || idx >= mSize)
    return FALSE;    // illegal idx

  delete x[i];       // destruct object

  for (int i = idx; i < mSize-1; i++)
    x[i] = x[i+1];

  mSize--;
  return TRUE;
}
```

sumed by each array?

4. If you want to iterate through a collection of objects it is sometimes easier to use functions like GetFirst(), GetNext(). If these functions return a BOOL value, you can easily detect when you are at the end of the collection. Implement these two functions in both examples.

3.2 Class Templates for Array Collections

A class template is a prescription for creating a class where one or more characteristics of the class can be parameterized. This is the only section in the book that we'll use or discuss templates.

Class templates became a part of the C++ standard around 1994. A Standard Template Library (STL) now exists for C++. It contains templates for many "collection" classes. I think it takes a good deal of practice to use STL effectively. MFC contains a number of templates for collection classes as well. One, the CArray, is discussed in section 3.2.2.

3.2.1 Program Ex03c: Creating and Using a Class Template

In program Ex03a we used an "array" class. The data held in the array are CClient objects. Suppose we wanted the array to hold another type of object, a CCar object, for example.

We could do this by using the search and replace capability of an editor to replace all words "CClient" with "CCar" thereby creating new header and implementation files. Notice that I have underlined the words in AnArray.h and AnArray.cpp in Ex03a (on page 41) that pertain to the specific data type. As you can see, not too much has to be changed.

Another way to get an "array" class for another data type is to make the data type a parameter. This is what class templates allow us to do. There are two important aspects to keep in mind when dealing with templates.

Class Templates are Processed by the C++ Preprocessor

Since the C++ compiler's preprocessor processes templates, the preprocessor transforms — by text substitution — a class template into the desired class definition.

Consequently, all template statements (the class definition and member function definitions) are put into a header file. (See the AnTArray.h file in program Ex03c on page 45.)

Template Syntax

The template syntax is similar to that of a function. The line preceding the class definition and member function definitions looks like this:

```
template < type Param1, type Param2, … >
```

Since our class template has two parameters:

❏ the data type stored in the components, and
❏ the maximum number of components in the array,

the lines preceding our definitions looks like this:

```
template <class DataType, int MAXSIZE>
```

By carefully comparing the corresponding lines in program files Ex03a and Ex03c, you will get the basic idea of the template syntax.

Notice how the template is used in main() to specify my "array" collection class:

```
CAnTArray<CClient, 25>  ClientS;
```

```
// Program Ex03c  Using an Array Template

#include <afxwin.h>      // This main program is the
#include <iostream.h>    // same as in Ex03a except for
#include "client.h"      // two lines:
#include "AnTArray.h"    //    * this line *

int main()
{
  CAnTArray<CClient, 25> ClientS; // * this line *
  CClient   A;

  A.Input();    ClientS.Add(A);
  A.Input();    ClientS.Add(A);
  A.Input();    ClientS.InsertAt(A, 1);

  cout << "\nOutput array:";
  int i;
  for (i = 0; i < ClientS.GetSize(); i++)
  {
    A = ClientS.GetAt(i); cout << "\n"; A.Display();
  }

  ClientS.RemoveAt(0);
  cout << "\n\nOutput array, after RemoveAt(0):";
  for (i = 0; i < ClientS.GetSize(); i++)
  {
    A = ClientS.GetAt(i); cout << "\n"; A.Display();
  }
  return 0;
}
```

Output Program Ex03c

```
Name   : Alice
Gender : f
Age    : 56

Name   : Barbara
Gender : f
Age    : 23

Name   : Carol
Gender : f
Age    : 82

Output array:
Alice        f 56
Carol        f 82
Barbara      f 23

Output array, after RemoveAt(0):
Carol        f 82
Barbara      f 23
```

```
// AnTArray.h CAnTArry Template class
#include <afxwin.h>

template <class DataType, int MAXSIZE>
class CAnTArray
{
private:
  int        mSize;
  DataType  x[MAXSIZE];
public:
  CAnTArray();
  virtual ~CAnTArray();

  int       GetSize  () const;
  DataType GetAt    (int idx);
  BOOL      RemoveAt (int idx);
  BOOL      InsertAt (const DataType& c,
                               int idx);
  BOOL      Add      (const DataType& c);
};

template <class DataType, int MAXSIZE>
CAnTArray<DataType, MAXSIZE>::CAnTArray()
:mSize(0)
{   }

template <class DataType, int MAXSIZE>
CAnTArray<DataType, MAXSIZE>::~CAnTArray()
{   }

template <class DataType, int MAXSIZE>
BOOL CAnTArray<DataType, MAXSIZE>::Add
                       (const DataType& c)
{
  x[mSize] = c;
  mSize++;
  return TRUE;
}
```

```
template <class DataType, int MAXSIZE>
BOOL CAnTArray<DataType, MAXSIZE>::InsertAt
                         (const DataType& c, int idx)
{
  if (mSize >= MAXSIZE)
    return FALSE;           // full
  if (idx < 0 || idx > mSize)
    return FALSE;           // illegal idx
  for (int i = mSize-1; i >= idx; i--)
    x[i+1] = x[i];          // shift up

  x[idx] = c;               // insert
  mSize++;
  return TRUE;
}
template <class DataType, int MAXSIZE>
BOOL CAnTArray<DataType, MAXSIZE>::RemoveAt(int idx)
{
  if (idx < 0 || idx >= mSize)
    return FALSE;    // illegal idx
  for (int i = idx; i < mSize-1; i++)
    x[i] = x[i+1];
  mSize--;
  return TRUE;
}
template <class DataType, int MAXSIZE>
DataType CAnTArray<DataType, MAXSIZE>::GetAt(int idx)
{
   return x[idx];
}

template <class DataType, int MAXSIZE>
int CAnTArray<DataType, MAXSIZE>::GetSize() const
{
  return mSize;
}
```

3.2.2 Program Ex03d: The MFC Class Template CArray

MFC has six template-based collection classes. In the first three classes — CArray, CList, and CMap — the actual objects are held in the components of the data structures, like in program Ex03a.

In the remaining three collection classes — CTypedPtrArray, CTypedPtrList, and CTypedPtrMap — the components hold pointers to the actual objects, similar to the collection class in Ex03b.

Using the MFC CArray Template

The MFC CArray template allows us to create a dynamic array where the components can be any type. A CArray object is called dynamic because it can grow or shrink in size. The fact that you can delete elements or insert elements into the middle of the array is very handy.

Program Ex03d on page 47 illustrates several member functions. The names and uses of these functions are like those used in program Ex03a.

Notice how we create our ClientS array with the use of the template:

```
CArray<CClient, CClient&> ClientS;
```

CClient is the object type to be stored in the components. CClient& is the argument type to be used in the member functions.

CArray Member Functions

Suppose we have declared “array” to be a CArray object. The following are some of the member functions in the CArray class.

```
array.SetSize(n);
idx = array.GetSize();

idx = array.Add (object);
array.InsertAt (idx, object);
array.SetAt (idx, object);
array.SetAtGrow (idx, object);

object = array.GetAt(idx);
objectRef = array.ElementAt(idx);

array.RemoveAt (idx);
array.RemoveAll();

idx = array.Append (const CArray& src);
array.Copy (const CArray& src);
```

Below is the output from a run of program Ex03d.

Output from Program Ex03d

```
Options: A'dd, I'nsertAt, R'emoveAt
  G'etAt, L'ist, Q'uit   Which ? a

 Name   : Bill
 Gender : m
 Age    : 76

Options: A'dd, I'nsertAt, R'emoveAt
  G'etAt, L'ist, Q'uit   Which ? A

 Name   : Sally
 Gender : f
 Age    : 35

Options: A'dd, I'nsertAt, R'emoveAt
  G'etAt, L'ist, Q'uit   Which ? L

Number of clients: 2
0: Bill         m 76
1: Sally        f 35

Options: A'dd, I'nsertAt, R'emoveAt
  G'etAt, L'ist, Q'uit   Which ?
```

Exercises:

1. Take one of the exercises you did in one of the previous sections and implement it using the MFC CArray class. You may find that all you have to do is copy some of your existing code.

```
// Program Ex03d
// Using the MFC CArray template class

#include <afxwin.h>
#include <iostream.h>
#include "client.h"    // need this

#include <afxtempl.h>  // templates defined here

int main()
{
  CArray<CClient, CClient&> ClientS;

  ClientS.SetSize(0); // Initial size but grows

  CClient   A;
  char      resp;
  int       idx, i, size;

  do
  {
    cout << "\nOptions: A'dd, I'nsertAt,"
            " R'emoveAt\n  G'etAt"
            ", L'ist, Q'uit   Which ? ";
    cin >> resp;
    resp = toupper(resp);
    if (resp == 'Q') break;

    switch (resp)
    {
    case 'A':
      A.Input();
      ClientS.Add(A);
      break;
    case 'I':
      A.Input();
      cout << "\n  At what position ? ";
      cin >> idx;
      ClientS.InsertAt (idx, A);
      break;
    case 'R':
     cout << "\n  Remove client at what position ? ";
      cin >> idx;
      ClientS.RemoveAt (idx);
      break;
    case 'G':
      cout << "\n  Get client at what position ? ";
      cin >> idx;
      A = ClientS.GetAt(idx);
      A.Display (); cout << "\n";
      break;
    case 'L':
      size = ClientS.GetSize();
      cout << "\nNumber of clients: " << size;
      for (i = 0; i < size; i++)
      {
        A = ClientS.GetAt(i);
        cout << "\n" << i << ": "; A.Display();
      }
      cout << "\n";
      break;
    default: break;
    }
  }while (true);

  return 0;
}
```

3.3 CObList, A List Collection

In addition to the CList and CTypedPtrList template list classes, MFC has a very useful non-template list class named CObList. This collection class is used in a number of examples later in the book.

Occasionally, when debugging a program, you can find yourself looking at MFC code. I've seen COblist being used by MFC code a number of times. Frankly, I think this is a terrific class and I highly recommend using it.

3.3.1 CObList: Some Important Points

A CObList object, illustrated in program Ex03e on the page 49, is a two-way linked list. The data in each node of the linked list is a pointer to an object, like a CClient object. Notice the "object pointers" and the "objects" in Figure 3.3 on page 49.

Objects Pointed to Must be Derived from CObject

The objects that are being pointed to — like the CClient objects — must be derived from the CObject base class.

Only one change in our CClient class is needed to satisfy this requirement. Publicly derive CClient from CObject with the following line modification.

```
class CClient: public CObject
```

The CObject class adds considerable functionality to our CClient objects.

The Division of Responsibilities

It is important to keep in mind the division of responsibilities.

❏ The CObList class member functions — AddTail(), InsertBefore(), RemoveAt() — take care of adding and deleting nodes in the linked list. That is, they manage the list nodes.

❏ Our responsibility is to take care of the object pointers in each node, and the objects they point to.

To illustrate these responsibilities, let's suppose we want to delete a client like Carol. We must first delete that client object—Carol. Then we use CObList::RemoveAt() to remove the node in the linked list.

The POSITION Object

To move around in the list, we need a POSITION object.

```
POSITION  pos;  // it is like a pointer
```

Here is how to move to the first node in the list:

```
pos = List.GetHeadPosition();
```

Here, we move to the next node:

```
List.GetNext(pos);
```

Once at a node, we'll want to access the object pointed to by the object pointer at that node.

```
pObjPtr = List.GetAt(pos); // Now you have a
                           // pointer to the object.
pObjPtr->SomeMethod(); // Do something with it
```

3.3.2 CObList Member Functions

The member functions can be put in various categories.

Construction

You can construct a CObList object or a pointer to a CObList object:

```
CObList   List;
CObList*  pList;
```

Status

We can determine how many nodes are in the list:

```
count = List.GetCount();
```

and determine if it is empty:

```
if (List.IsEmpty()) { appropriate code }
```

Adding / Inserting

There are a variety of ways to add objects to the list. But first you need the actual object and a pointer to the object. See the program case 'A' for an example. In general, it is done like this:

```
pObjPtr = new CSomeClass;
pObjPtr->Input();    // Give object a value
List.AddTail(pObjPtr);
```

Here are the various functions for adding and inserting:

```
List.AddTail (objPtr);
List.AddHead (objPtr);
InsertBefore (pos,obPtr);
InsertAfter (pos,obPtr);
```

We can actually add the nodes of another list to this list:

```
List.AddHead (CObList*);
```

Retrieving object pointers

When an object pointer is retrieved, you must cast the pointer to be of the appropriate type. This is illustrated in the program.

```
objPtr = (cast type) list.GetAt(pos);
objPtr = (cast type) list.GetHead();
objPtr = (cast type) list.GetTail();
```

Node Removal

Note: Before removing the node, you must first remove the object being pointed to!

```
List.RemoveAt(pos);
List.RemoveHead();
List.RemoveTail();
List.RemoveAll();
```

Iteration

There are five functions for moving around in the list. That is, five ways to move the position. You can test variable pos for the NULL value.

```
pos    = List.GetHeadPosition();
pos    = List.GetTailPosition();
objPtr = (cast type) List.GetNext(pos);
objPtr = (cast type) List.GetPrev(pos);
pos = list.FindIndex(index); // see next line!
      // 0<= index and index < List.GetCount()
```

The FindIndex method, takes an index value like 4 and will return a POSITION object that you can use to access that node.

The GetNext() and GetPrev() Member Functions

In addition to moving the position, GetNext() and GetPrev() also return the object pointer at the position. So, we usually use the following form of these functions:

```
objPtr = (cast pointer) list.GetNext(pos);
```

```
objPtr = (cast pointer) list.GetPrev(pos);
```

It is critical that you understand the sequence of the two tasks performed by GetNext() and GetPrev(). Consider this statement

```
objPtr = list.GetNext(pos);
```

❏ First, the object pointer is returned.
❏ Second, pos is adjusted so that it points to the next node in the list.

The program code for case “list” (page 49) illustrates the GetNext() function as it is typically used.

Modification

You can easily change what a node in CObList points to with this function:

```
List.SetAt(pos,objPtr);
```

Searching

To use the following, first read about it in Help. Personally, I have not found the following function to be particularly useful.

```
pos = List.Find(CObject*, StartingPosition);
```

```
// Ex03e_CObList
#include <afxwin.h>
#include <iostream.h>
#include "client.h"

int main()
{
  CObList      myClientList;
  POSITION     pos;
  CClient       *pClient;
  int           i;
  char          resp;
  int           nbrClients;
  do
  {
    cout << "\nOPTIONS: A'ddTail, I'nsertBefore,";
    cout << "\n G'etAt, L'istAll, R'emoveAt, Q'uit Which? ";
    cin >> resp;  resp = toupper(resp);
    if (resp == 'Q') break;  // exit loop here

    switch (resp)
    {
    case 'A':
      pClient = new CClient; // Construct an object
      pClient->Input();       // Give it a value
      myClientList.AddTail(pClient);
      break;
    case 'I':
      cout << "\n  Before what position ? ";
      cin >> i;
      if (i >= myClientList.GetCount())
      { cout << "\n Impossible to insert here";
        break;
      }
      pClient = new CClient;
      pClient->Input();
      pos = myClientList.FindIndex (i);
      myClientList.InsertBefore (pos, pClient);
      break;
    case 'G':
      cout << "  Get client at position (0,1,...) ? ";
      cin >> i;
      nbrClients = myClientList.GetCount();
      pos = myClientList.FindIndex (i);
      pClient = (CClient*) myClientList.GetAt(pos);
      cout << "  ";
      pClient->Display();
      break;
    case 'L':
      nbrClients = myClientList.GetCount();
      pos = myClientList.GetHeadPosition();
      for (i = 0; i < nbrClients; i++)
      {
        pClient = (CClient*)myClientList.GetNext(pos);
        cout << "\n  " << i << ": ";
        pClient->Display();
      }
      cout << "\n";
      break;
    case 'R':
      cout << "\n  Exercise.\n";
      break;
    default:
      break;
    }
  } while (true);
// continued on right - - >
```

Input/Output for Program Ex03e

```
Options:  A'ddTail,  I'nsertBefore,
 G'etAt,  L'istAll,  R'emoveAt,  Q'uit  Which ? a

 Name   : Alice
 Gender : f
 Age    : 41

Options:  A'ddTail,  I'nsertBefore,
 G'etAt,  L'istAll,  R'emoveAt,  Q'uit  Which ? A

 Name   : Jim
 Gender : m
 Age    : 36

Options:  A'ddTail,  I'nsertBefore,
 G'etAt,  L'istAll,  R'emoveAt,  Q'uit  Which ? L

  0: Alice       f 41
  1: Jim         m 36

Options:  A'ddTail,  I'nsertBefore,
 G'etAt,  L'istAll,  R'emoveAt,  Q'uit  Which ? I

  Before what position ? 1

 Name   : Carol
 Gender : f
 Age    : 32

Options:  A'ddTail,  I'nsertBefore,
 G'etAt,  L'istAll,  R'emoveAt,  Q'uit  Which ? l

  0: Alice       f 41
  1: Carol       f 32
  2: Jim         m 36

Options:  A'ddTail,  I'nsertBefore,
 G'etAt,  L'istAll,  R'emoveAt,  Q'uit  Which ?
```

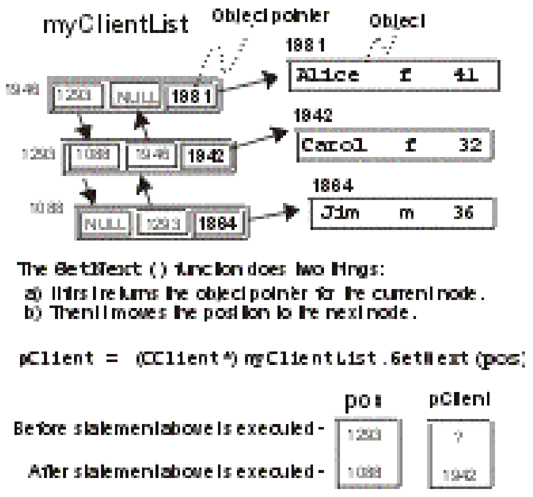

myClientList
Object pointer
Object
1881
1946
1293
NULL
1881
Alice f 41
1942
1293
1088
1946
1942
Carol f 32
1884
1088
NULL
1293
1884
Jim m 36
The GetNext () function does two things:
a) It first returns the object pointer for the current node.
b) Then it moves the position to the next node.
pClient = (CClient*) myClientList.GetNext (pos);
pos
pClient
Before statement above is executed -
After statement above is executed -
1293
1088
?
1942

Figure 3.3 A CObList example

```
  // Must remove all objects
  pos = myClientList.GetHeadPosition();
  do
  {
    if (pos == NULL) break;
    pClient = (CClient*)myClientList.GetNext(pos);
    delete pClient;  // delete object
  } while (true);
  // Also remove nodes here. Destructor would do it.
  myClientList.RemoveAll();

  return 0;
}
```

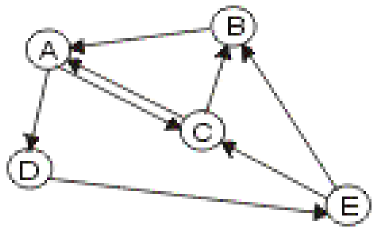
3.3.3 Exercises

1. Implement the RemoveAt option in the program.

2. Data structures, such as stacks, queues, and priority queues can be easily built when you use a CObList object as a data member that holds the linked data.

a) Write a stack class built around a CObList object.
b) Write a queue class built around a CObList object.

3. One of the more involved data structures is a Directed Graph (see below). This can be implemented in a number of different ways with CObList objects.



3.4 CFile, A File Object

The CFile class lets us deal with files as objects. CFile is actually the base class for all other file classes. For example, the class CStdioFile, derived from CFile, gives us the ability to manipulated buffered text files.

3.4.1 CFile: Some Important Points

Think of a disk file as a sequence of bytes on a disk. There is a byte 0, byte 1, and so on. Consequently, we can talk about a byte position within the file — the byte at position 17, for example. See Figure 3.4 on page 51.

For each file there is a “file pointer.” The value of the file pointer is the “byte position” in the file where the next read or write begins. In other words, the file pointer value tells us where the disk “read/write” head is located.

The data in a file usually has some sort of specific organization. For example, the data might consist of a series of temperature readings at specific times. So the organization might be:

```
| time temp | time temp | time temp | etc
```

You have to be aware of the organization used to write the data into the file, so you can read it back in the same way. In fact, in this case you would write to the file and read back in these “time, temperature” chunks.

3.4.2 Program Ex03f

Program Ex03f reads client data from the keyboard and then writes it to the file, one CClient chunk at a time. The program allows us to move to any client (e.g., the 5th client) in the file, read it back into the computer memory, and then display it.

To be able to read one client it is necessary that each client object be exactly the same length. Since a CString can be any length, the type of our mName member variable in the CClient class has been changed to a character array of length 12. Here is how our class definition starts out. After this it is unchanged from CClient in Ex03a.

```
class CClient
{
private:
  char      mName[12]; // not CString
  char      mGender;
  int       mAge;
```

By the way, we can use the C language sizeof() function to see that each CClient object is 24 bytes long. You might think this size is too big. But remember, this class has a virtual destructor. Consequently, each object carries a pointer to the vTable.

3.4.3 CFile Member Functions

Construction / Initialization / Destruction

As illustrated below, we usually use the default constructor for CFile objects. The reason for this is that we'll want to have a dialog with the user so he/she can tell us what to do. Consequently, the Open() member function is usually used to 'initialize' the CFile object.

```
CFile   myFile;
if (myFile.Open ("My.dat",
                 CFile::modeReadWrite))
    ...
```

The left parameter of the Open function is the disk file name that gets associated with the CFile object. The file name, "My.dat", is a file in the project folder. You can specify a path if you wish. Also, you can use a CString variable here.

The right parameter specifies the processing modes/styles for the file. Here are various styles:

`modeCreate`	If file exists, truncate to 0 length
`modeRead`	Opens for reading only
`modeWrite`	Opens file writing only
`modeReadWrite`	Can read and write

```
// Ex03f_CFile
#include <afxwin.h>
#include <iostream.h>
#include "client.h"

int main()
{
  CFile      myFile;
  CString     myFileName = "My.dat";
  int         i;
  char        resp;
  CClient     aClient;        // acts as buffer
  int clSize = sizeof(aClient);
  int         nbrClients;

  cout << "\nFile name: " << myFileName;
  cout << "\nOptions: R)  Read the existing file or";
  cout << "\n         D)  Delete, start with new file";
  cout << "\nWhich (R or D) ? ";
  cin  >> resp;  resp = toupper (resp);

  if (resp == 'R')
  {
     if (myFile.Open (myFileName, CFile::modeReadWrite))
        cout << "\nExisting file opened";
    else
    {
      cout << "\nProblem opening Existing file";
      return 0;
    }
  } else {
    if ( myFile.Open (myFileName,
         CFile::modeCreate | CFile::modeReadWrite)
        )
        cout << "\nNew file opened";
    else
    { cout << "\nProblem opening a New file";
      return 0;
    }
  } // end if

  do
  {
    cout << "\nOptions: A'dd,  R'etrieve,  L'ist all";
    cout << "\n         Q'uit,  Which ? ";
    cin >> resp;  resp = toupper(resp);
    if (resp == 'Q') break;
    switch (resp)
    {
    case 'A':
      aClient.Input();
      myFile.SeekToEnd();
      myFile.Write(&aClient, clSize);
      break;
    case 'R':
      nbrClients = myFile.GetLength()/clSize;
      cout << "  Retrieve which client (0.."
            << nbrClients-1 << ") ? ";
      cin >> i;
      if (i <= nbrClients)
      {
        myFile.Seek (i * clSize, CFile::begin);
        myFile.Read (&aClient, clSize);
        cout << "  Client " << i << ": ";
        aClient.Display();
        cout << "\n";
      }
      break;
//              continued on right - - >
```

Input/Output Ex03F

```
File name: My.dat
Options: R)  Read the existing file or
         D)  Delete, start with new file
Which (R or D) ? R

Existing file opened
Options: A'dd,   R'etrieve,  L'ist all
         Q'uit,  Which ? L
  Number of clients:  3
  0: Joe          m 45
  1: Kim          f 31
  2: Bob          m 65

Options: A'dd,   R'etrieve,  L'ist all
         Q'uit,  Which ? A

 Name   : Marcia
 Gender : f
 Age    : 41

Options: A'dd,   R'etrieve,  L'ist all
         Q'uit,  Which ? L
  Number of clients:  4
  0: Joe          m 45
  1: Kim          f 31
  2: Bob          m 65
  3: Marcia       f 41

Options: A'dd,   R'etrieve,  L'ist all
         Q'uit,  Which ?
```

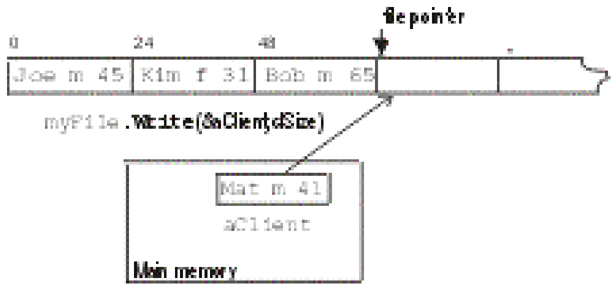


Figure 3.4 Reading and writing a file

```
    case 'L':
      nbrClients = myFile.GetLength()/clSize;
      cout << "  Number of clients:  " << nbrClients;
      myFile.SeekToBegin();
      for (i = 0; i < nbrClients; i++)
      {
        myFile.Read (&aClient, clSize);
        cout << "\n  " << i << ": ";
        aClient.Display();
      }
      cout << "\n";
      break;
    default:
      break;
    } // end switch

  } while (true);

  myFile.Close();
  return 0;
}
```

Notice that we "bitwise OR" two CFile modes together in the program:

```
myFile.Open (myFileName,
        CFile::modeCreate | CFile::modeReadWrite)
```

Input/Output

The word "buffer" refers to an area in memory that contains data to be written to a file. It also refers to the area that receives the data when a file is read. In program Ex03f, our buffer is a CClient object with the name, aClient. The parameter on the left is the address of the buffer, &aClient in this case. The right parameter is the number of bytes to write or read.

```
myFile.Write (&aClient, sizeof(aClient));
n = myFile.Read (&aClient, sizeof(aClient));
```

The Read method returns the actual number of bytes read. If it is less than what we expect to read, we are at the end of the file! You can use this fact to test for the end of a file.

Moving the File Pointer

The file pointer contains the position within the file where the next Read() or Write() begins. Executing a Read() or Write() moves the file pointer by the number of bytes read or written. Also, there are three "seek" functions that directly change the position of the file pointer.

```
myFile.SeekToBegin();
myFile.SeekToEnd();
n = myFile.Seek(BytesToMove, FromWhere);
```

In the Seek() function above, "FromWhere" must be one of the following modes:

```
CFile::begin, CFile::current, CFile::end
```

The return value "n" is the current position relative to the beginning of the file. That is, it is the value in the "file pointer."

Status of the CFile Object

It is handy to know the length of the file.

```
n = myFile.GetLength();
n = myFile.GetPosition();
myFile.SetLength(newLength);  // used to shorten file
```

Notice how we determine the number of clients in the file:

```
nClients = myFile.GetLength()/sizeof(aClient);
```

3.4.4 Exercises

1. Compare the "file pointer" in CFile with the POSITION object used with CObList objects. From a syntactic and logical perspective, how are they similar and different?

2. Modify the CObList program Ex03e so that after the user has finished manipulating the data in the list, the data is saved in a file. Also, when that program begins, ask the user if he/she wants to load the list with data from the file.

3.5 Serializing Objects

One of the most elegant aspects of MFC is the way it saves a “document object“ to disk when you “close“ it. Of course, when you open a “document object“, MFC then restores (recreates) the object in main memory. And it does all this work with just a little help from us! I think you will be amazed when you see how smoothly it works in Chapter 14. In this section we’ll look at some of the details.

Serialization is the process of making objects persistent. A persistent object is one that can save its own state when a program terminates and then can restore its state when the program begins execution again. In this section we illustrate how to serialize three different kinds of objects.

You can definitely skip this section and proceed directly to Chapter 4 where we begin writing programs that actually look like windows programs. However, you must read this section before Chapter 14. We are including this topic discussion here simply because serialization is more easily discussed with a console application. Also, serialization uses CFiles, which were just discussed in the last section.

In a way, some of the programs we write in Chapter 14 are actually simpler than the ones in this section. The reason is that in this section we have to write code that in Chapter 14 is done automatically by MFC. Those programs in Chapter 14 support the MFC's “document/view“ architecture which provides us with a number of powerful features.

You should know that serialization is different from simply saving data to a file, like we did in the last section. With a little experience, you will know which of these ways to use when saving information.

3.5.1 A Strategy for Saving An Object

In the last section we saved a number of CClient objects to file. It wasn't difficult. But we did have to know a fair amount about the CFile object.

What if our object were a little more complicated than a CClient object? For example, how might we save a CTruck or a CCar object? How might we save a fleet (perhaps represented as a linked list) of CTrucks, CCars, and CVehicle objects?

Since objects can get very complicated, it's hard to imagine a strategy that will save and restore objects in general. Fortunately, MFC has a strategy. There are a few important parts to this strategy.

Read and Write to a CArchive Object

One part of the strategy is to have the programmer deal with a simpler type of storage object. Instead of dealing with a CFile object, we'll save our objects to a CArchive object. To save something to a CArchive object is, in most situations, very easy. It's like using the equivalent of “cin” and “cout”. In fact, we do it with statements like this:

```
ar << memberData; saving
ar >> memberData; restoring
```

The variable ar is a CArchive object.

A CArchive object has considerable built-in capability. Once the CArchive object gets the data, it writes it into the CFile object. We do not have to worry about any details. CArchive takes care of them.

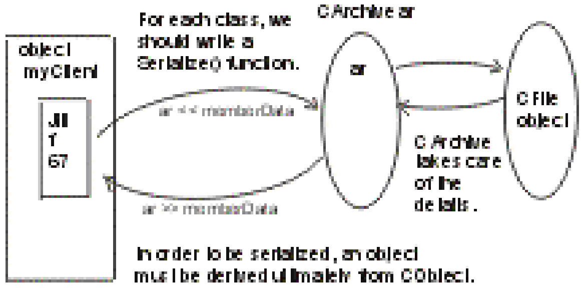

object myClient
Jill
f
67
For each class, we should write a Serialize() function.
CArchive ar
ar
ar << memberData
ar >> memberData
CFile object
CArchive takes care of the details.
In order to be serialized, an object must be derived ultimately from CObject.

Figure 3.5 Serializing data

A Serialize() Function for each Class

Another part of MFC's strategy for implementing persistence is to have each object save and restore itself. This is done in the Serialize() member function. Serialize() is a virtual function in the CObject class.

If we want a class object to be persistent,

❑ the class must be derived from CObject and
❑ the class must override COject's Serialize() function.

Here is the Serialize() function for the class CClient:

```
void CClient::Serialize(CArchive& ar)
{
  CObject::Serialize (ar); // First, serialize the object
                           // we're derived from
  if (ar.IsStoring())
    ar << mName << mGender << mAge;
  else
    ar >> mName >> mGender >> mAge;
}
```

Notice that, if the class is derived from another class, we serialize the base object first. Then, we serialize our own data members. This is better illustrated in program Ex03h on page 57.

Notice the placement of the macro `DECLARE_SERIAL(CClient)` in the header file, and `IMPLEMENT_SERIAL(CClient, CObject, 0)` in the cpp file.

3.5.2 Program Ex03g, Serializing a Simple Object

Notice the three steps in the main() function on page 55. The discussion below focuses on these steps.

First, you should know that when we get to Chapter 14, the code in step 1 and step 3 will be supplied automatically by MFC. In any event, in both steps, only the object name changes. I've underlined it.

Second, you'll think this program seems too involved to simply serialize one CClient object. But, you'll shortly see how this technique generalizes. It is very easy to save a collection of CClients, as we do in program Ex03i on page 58.

Program Step 1

The code in step 1 creates a CFile object. Then a CArchive object is created and “linked“ to the CFile object. It then calls our Serialize() function, which reads the object data from CArchive.

Notice the program output labeled “First run.” No archive existed, so the default constructor was called for the object. When displayed, it has the default values xxx, X, and 0.

Program Step 2

In step 2, we let the user modify the CClient object, which the user does as you can see in the screen shot “First run.”

Program Step 3

In step 3, our object gets serialized again. In this case, it gets written to the file.

In the CClient::Serialize() function, notice the statement if (ar.IsStoring()). The CArchive object knows when it is storing or retrieving because the CArchive object was constructed with the style “load“ or the style “store”. See the CArchive constructor in Steps 1 and 2.

Second Program Run

Inspect the screen shot for the “Second run.” When the object is displayed, we see it as it was last modified in the “First run.”

```
// Ex03g_Serialize  A Simple Object
#include <afx.h>
#include <iostream.h>
#include "client.h"

int main()
{
  CClient      myClient;
  char          resp;
  CString  myArFileName = "myArc.arc";

  // Step 1: = = = = = = = = = = = = = = = = = = = = = = =
  //  MFC will automatically do this for you when
  //  your project uses the document/view architecture.
  CFile  myArFile;
  if (myArFile.Open (myArFileName, CFile::modeRead))
  {
    CArchive ar (&myArFile, CArchive::load); //construct
    myClient.Serialize (ar);
    ar.Close();                              //destruct
    myArFile.Close();
  } // = = = = = = = = = = = = = = = = = = = = = = = = =

  // Step 2: Do some processing of your object
  cout << "Object myClient is: ";
  myClient.Display();

  cout << "\n\nWant change myClient (Y/N) ?";
  cin >> resp;
  if (resp == 'Y' || resp == 'y')
    myClient.Input();

  cout << "\n  ... exiting program ...\n";

  // Step 3: = = = = = = = = = = = = = = = = = = = = = = =
  //  MFC will automatically do this for you later on
  if (myArFile.Open (myArFileName,
                 CFile::modeCreate | CFile::modeWrite))
  {
    CArchive ar (&myArFile, CArchive::store); //constr
    myClient.Serialize (ar);
    ar.Close();
    myArFile.Close();
  } // = = = = = = = = = = = = = = = = = = = = = = = = =

  return 0;
}
```

Output Ex03g First run

```
Object myClient is: xxx      X 0

Want change myClient (Y/N) ?y

 Name   : Jill
 Gender : f
 Age    : 67

  ... exiting program ...
Press any key to continue
```

Output Ex03g Second run

```
Object myClient is: Jill     f 67

Want change myClient (Y/N) ?y

 Name   : Joan
 Gender : f
 Age    : 48

  ... exiting program ...
Press any key to continue
```

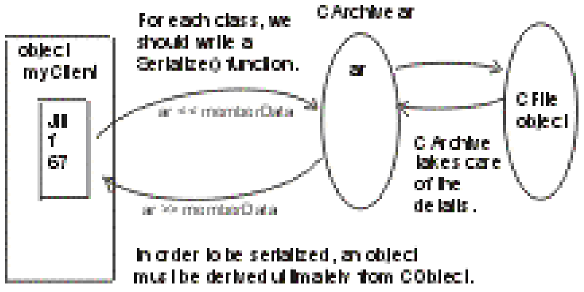

For each class, we should write a Serialize() function.
C Archive ar
object myClient
Jill
f
67
ar << memberData
ar
ar >> memberData
C File object
C Archive takes care of the details.
In order to be serialized, an object must be derived ultimately from CObject.

Figure 3.6 Serializing data

```
// Client.cpp
#include <iostream.h>
#include "Client.h"

IMPLEMENT_SERIAL(CClient, CObject, 0)

void CClient::Serialize(CArchive& ar)
{
  CObject::Serialize (ar); // call serialize function
                           // in class derived from
  if (ar.IsStoring())
    ar << mName << mGender << mAge;
  else
    ar >> mName >> mGender >> mAge;
}

// All the remaining member functions are
// the same as in CClient in Ex03a
//CClient::CClient()
//CClient::~CClient()
//void CClient::Input()
//void CClient::Display() const
```

```
// Client.h
#include <afxwin.h>

class CClient : public CObject
{
DECLARE_SERIAL(CClient)
private:
  CString  mName;
  char      mGender;
  int       mAge;

public:
            CClient();
  virtual ~CClient();
  void     Input();
  void     Display() const;
  void     Serialize (CArchive& ar);
};
```

3.5.3 Program Ex03h, Serializing a More Complex Object

This program illustrates two things:

❏ How to serialize an object that is derived from another object.

❏ How to serialize an object that has a subobject.

A CTruck object is a good example to illustrate these points. Here is the data involved in a CTruck object.

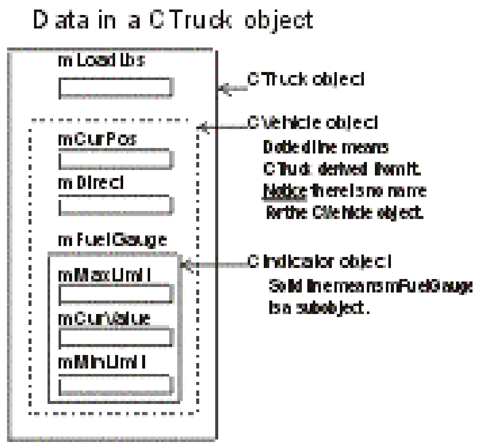


Figure 3.7

Three Classes Need Modification

There are three classes involved in serializing a CTruck object. CTruck is derived from CVehicle, and CVehicle has a subobject, CIndicator.

The header files for CIndicator and CVehicle must be modified. Both classes must be derived from CObject.

Since CTruck is derived from CVehicle, it ultimately has CObject as its base class, which is a requirement for serializing an object. Each header file must contain a `DECLARE_SERIAL(className)` macro and a prototype for the serialize function.

In the Truck.cpp file, notice the line `IMPLEMENT_SERIAL(CTruck, CVehicle)`. Here we are giving the compiler information about the current class and the class from which it is derived.

In the Vehicle.cpp file, notice that we had to apply Serialize() to the mFuelGauge object because we have not overloaded the insertion and extraction operators.

In the Vehicle.h file, we changed the type of mDirect. For some unknown reason to me, the compiler didn't want to see an enumerated type.

3.5.4 Program Ex03i, Serializing a Collection

In section 3.3 we discussed a program, Ex03e, which creates and modifies a linked list of pointers to CClient objects.

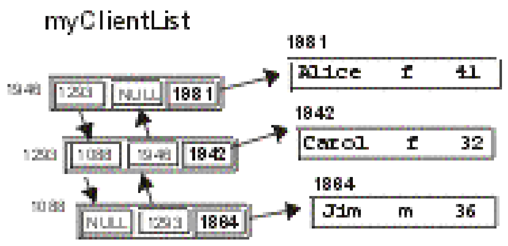


Figure 3.8

The list is held in a variable, declared like this:

```
CObList  myClientList;
```

Serializing a CObList Object

So how do we serialize the myClientList object? Answer: MFC created to class CObList. It knows all about its internal structure. Since CObList is an MFC class, it is MFC's obligation to provide a Serialize() function, which it did!

Consequently, all we must do is put the step 1 code at the beginning of the program, and the step 3 code near the end and change myClient.Serialize(ar) to myClientList.Serialize(ar)!

This is illustrated in program Ex03i on page 58.

```
// Ex03h_Serialize CTruck,a more complex object
// some code omitted

int main()//  very similar to previous main()
{
  CTruck   T;
  char     resp;
  int      load;
  // Step 1: = = Like previous. = = = = = =
  . . .
  T.Serialize (ar);  // This is the only difference
  . . .

  // Step 2: Possibly modify the object
  T.Display("\nTruck T: ");
  cout << "\n\nWant to change Truck (Y/N) ?";
  cin >> resp;
  if (resp == 'Y' || resp == 'y')
  {
    cout << "\nNew load in lbs ? ";
    cin >> load;
    T.SetLoadLbs(load);
  }
  // Step 3: = = Like previous. = = = = = =
  . . .
  T.Serialize (ar);  // This is the only difference
  . . .
  return 0;
}
```

Output Ex03h First Run

```
Truck T:  at(  0,  0) heading:N
             fuel:10.00 Load: 0

Want to change Truck (Y/N) ?y

New load in lbs ? 1000

  ... exiting program ...
Press any key to continue
```

Output Ex03h Second Run

```
Truck T:  at(  0,  0) heading:N
             fuel:10.00 Load:1000

Want to change Truck (Y/N) ?y

New load in lbs ? 2500

  ... exiting program ...
Press any key to continue
```

```
// Truck.cpp
//  The following code was added to file

IMPLEMENT_SERIAL(CTruck, CVehicle, 0)

void CTruck::Serialize (CArchive& ar)
{
  CVehicle::Serialize (ar);

  if (ar.IsStoring())
    ar << mLoadLbs;
  else
    ar >> mLoadLbs;
}
  . . . other code the same . . .
```

```
// Indicator.cpp
//  The following code was added to file

IMPLEMENT_SERIAL(CIndicator, CObject, 0)

void CIndicator::Serialize (CArchive& ar)
{
  CObject::Serialize (ar);

  if (ar.IsStoring())
    ar << mMaxLimit << mCurValue << mMinLimit;
  else
    ar >> mMaxLimit >> mCurValue >> mMinLimit;
}
. . . other code the same . . .
```

```
// Vehicle.cpp
//  The following code was added to file.

IMPLEMENT_SERIAL(CVehicle, CObject, 0)

void CVehicle::Serialize (CArchive& ar)
{
  CObject::Serialize (ar);

  if (ar.IsStoring())
  {
    ar << mCurPos << mDirect;
    mFuelGauge.Serialize(ar);
  }
  else
  {
    ar >> mCurPos >> mDirect;
    mFuelGauge.Serialize(ar);
  }
}
. . . other code the same . . .
```

```
// Vehicle.h

// Note: Compiler didn't want to
// serialize my enumerated type Direction.
// So I changed it as shown below, in bold

enum Direction {NORTH,EAST,SOUTH,WEST};

class CVehicle : public CObject
{
DECLARE_SERIAL(CVehicle)
public:
  CPoint      mCurPos;
private:
  UINT        mDirect;  // changed this
  CIndicator  mFuelGauge;
  . . . etc
```

```
// Ex03i_Serialize CObList
#include <afxwin.h>
#include <iostream.h>
#include "client.h"  // With serialize function

int main()
{
  CObList    myClientList;
  POSITION   pos;
  CClient    *pClient;
  int         i;
  char        resp;
  int         nbrClients;
  CString    myArFileName = "myArc.arc";

  // Step 1: = = = = = = = = = = = = = = = = = = = = = //
  //  MFC will automatically insert code like when  //
  //  project uses the document/view architecture.  //
  CFile  myArFile;                                   //
  if (myArFile.Open (myArFileName, CFile::modeRead)) //
  {                                                  //
    CArchive ar (&myArFile, CArchive::load);         //
    myClientList.Serialize (ar);                     //
    ar.Close();                                     //
    myArFile.Close();                                //
  } // = = = = = = = = = = = = = = = = = = = = = = = =//

  do // Step 2: Modify object.  Code from Ex03e
  {
    cout << "\nOPTIONS: A'ddTail, I'nsertBefore, G'etAt";
    cout << "\n L'istAll, R'emoveAt, Q'uit Which? ";
    cin >> resp;  resp = toupper(resp);
    if (resp == 'Q') break;

    switch (resp)
    {
    case 'A':
      pClient = new CClient; // Construct an object
      pClient->Input();      // Give it a value
      myClientList.AddTail(pClient);
      break;
                 // code from Ex03e goes here
     default:
      break;
    }
  } while (true);

  // Step 3: = = = = = = = = = = = = = = = = = = = =  //
  //  MFC will automatically add code like this       //
  if (myArFile.Open (myArFileName,                   //
            CFile::modeCreate | CFile::modeWrite))    //
  {                                                   //
    CArchive ar (&myArFile, CArchive::store);         //
    myClientList.Serialize (ar);                      //
    ar.Close();                                      //
    myArFile.Close();                                 //
  } // = = = = = = = = = = = = = = = = = = = = = = =  //

  // Must remove all objects from memory
  pos = myClientList.GetHeadPosition();
  do
  {
    if (pos == NULL) break;
    pClient = (CClient*)myClientList.GetNext(pos);
    delete pClient;
  } while (true);
  myClientList.RemoveAll();

  return 0;
}
```

Input/Output Program Ex03i: First run

```
OPTIONS: A'ddTail, I'nsertBefore, G'etAt
    L'istAll, R'emoveAt, Q'uit Which? L

OPTIONS: A'ddTail, I'nsertBefore, G'etAt
    L'istAll, R'emoveAt, Q'uit Which? A

 Name   : Joe
 Gender : m
 Age    : 87

OPTIONS: A'ddTail, I'nsertBefore, G'etAt
    L'istAll, R'emoveAt, Q'uit Which? A

 Name   : Alice
 Gender : f
 Age    : 52

OPTIONS: A'ddTail, I'nsertBefore, G'etAt
    L'istAll, R'emoveAt, Q'uit Which? L

  0: Joe          m 87
  1: Alice        f 52

OPTIONS: A'ddTail, I'nsertBefore, G'etAt
    L'istAll, R'emoveAt, Q'uit Which? Q
```

Input/Output Program Ex03i: Second run

```
OPTIONS: A'ddTail, I'nsertBefore, G'etAt
   L'istAll, R'emoveAt, Q'uit Which? L

  0: Joe          m 87
  1: Alice        f 52

OPTIONS: A'ddTail, I'nsertBefore, G'etAt
   L'istAll, R'emoveAt, Q'uit Which?
```

```
// Client.h

#include <afxwin.h>

class CClient : public CObject
{
DECLARE_SERIAL(CClient)

private:
  CString  mName;
  char      mGender;
  int       mAge;

public:
          CClient();
  virtual ~CClient();
  void    Input();
  void    Display() const;
  void    Serialize (CArchive& ar);
};
```