SESSION

15

Pointers to Objects

Session Checklist

- ✔ Declaring and using pointers to class objects
- ✔ Passing objects using pointers
- ✔ Allocating objects off of the heap
- ✔ Creating and manipulating linked lists
- ✔ Comparing linked list of objects to arrays of objects

30 Min.
To Go

Session 12 demonstrated how combining the array and the class structures into arrays of objects solved a number of problems. Similarly, the introduction of pointers to objects solves some problems not easily handled by arrays of class objects.

Pointers to Objects

A pointer to a programmer defined structure type works essentially the same as a pointer to an intrinsic type:

```
int* pInt;
class MyClass
```

```
{
   public:
        int  n1;
        char c2;
};
MyClass  mc;
MyClass* pMS = &mc;
```

Note

The type of `pMS` is "pointer to MyClass," which is also written `MyClass`*.

Members of such an object may be accessed as follows:

```
(*pMS).n1 = 1;
(*pMS).c2 = '\0';
```

Literally, the first expression says, "assign 1 to the member `n1` of the `MS` object pointed at by `pMS`."

Tip

The parentheses are required because "." has higher precedence than "*". The expression `*mc.pN1` means "the integer pointed at by the `pN1` member of the object `mc`.

Just as C++ defines a shortcut for use with arrays, C++ defines a more convenient operator for accessing members of an object. The `->` operator is defined as follows:

```
(*pMS).n1 is equivalent to pMS->n1
```

The arrow operator is used almost exclusively because it is easier to read; however, the two forms are completely equivalent.

Passing objects

A pointer to a class object can be passed to a function in the same way as simple pointer type.

```
// PassObjectPtr - demonstrate functions that
//                 accept an object pointer
#include <stdio.h>
```

```
#include <iostream.h>

// MyClass - a meaningless test class
class MyClass
{
    public:
        int n1;
        int n2;
};

// myFunc - pass by value version
void myFunc(MyClass mc)
{
    cout << “In myFunc(MyClass)\n”;
    mc.n1 = 1;
    mc.n2 = 2;
}

// myFunc - pass by reference
void myFunc(MyClass* pMS)
{
    cout << “In myFunc(MyClass*)\n”;
    pMS->n1 = 1;
    pMS->n2 = 2;
}

int main(int nArg, char* pszArgs[])
{
    // define a dummy object
    MyClass mc = {0, 0};
    cout << “Initial value = \n”;
    cout << “n1 = “ << mc.n1 << “\n”;

    // pass by value
    myFunc(mc);
    cout << “Result = \n”;
    cout << “n1 = “ << mc.n1 << “\n”;

    // pass by reference
    myFunc(&mc);
```

```
        cout << “Result = \n”;
        cout << “n1 = “ << mc.n1 << “\n”;
        return 0;
    }
```

The main program creates an object of class MyClass. The object is first passed to the function myFunc(MyClass) and then its address to the function myFunc (MyClass*). Both functions change the value of the object — only the changes made from within myFunc(MyClass*) “stick.”

In the call to myFunc(MyClass), C++ makes a copy of the object. Changes to mc in this function are not copied back to main(). The call to myFunc(MyClass*) passes an address to the original object in main(). The object retains any changes when control returns to main().

This copy versus original comparison is exactly analogous to a function such as fn(int) versus fn(int*).

Note

Besides retaining changes, passing a 4-byte pointer, rather than creating a copy of the entire object, may be significantly faster.

References

You can use the reference feature to let C++ perform some of the pointer manipulation:

```
    // myFunc - mc remains changed in calling function
    void myFunc(MyClass& mc)
    {
        mc.n1 = 1;
        mc.n2 = 2;
    }

    int main(int nArgs, char* pszArgs[])
    {
        MyClass mc;
        myFunc(mc);
        // ...
```

Note

You've already seen this feature. The `ClassData` example in Session 12 used a reference to the class object in the call to `getData(NameDataSet&)` in order that the data read could be returned to the caller.

Return to the heap

One must be careful not to return a reference to an object defined locally to the function:

```
MyClass* myFunc()
{
    MyClass  mc;
    MyClass* pMC = &mc;
    return pMC;
}
```

Upon return from `myFunc()`, the `mc` object goes out of scope. The pointer returned by `myFunc()` is not valid in the calling function. (See Session 13 for details.)

Allocating the object off of the heap solves the problem:

```
MyClass* myFunc()
{
    MyClass* pMC = new MyClass;
    return pMC;
}
```

Tip

The heap is used to allocate objects in a number of different situations.

20 Min.
To Go

The Array Data Structure

As a container of objects the array has a number of advantages including the capability to access a particular entry quickly and efficiently:

```
MyClass mc[100];          // allocate room for 100 entries
mc[n];                    // access the n'th ms entry
```

Weighed against that are a number of disadvantages:

Arrays are of fixed length. You can calculate the number of array entries to allocate at run time, but once created the size of the array can not be changed:

```
void fn(int nSize)
{
    // allocate an array to hold n number of
    // MyClass objects
    MyClass* pMC = new MyClass[n];

     // size of the array is now fixed and cannot
     // be changed

    // ...
}
```

In addition, each entry in the array must be of exactly the same type. It is not possible to mix objects of class `MyClass` and `YourClass` in the same array.

Finally, it is difficult to add an object to the middle of an array. To add or remove an object, the program must copy each of the adjoining elements up or down in order to make or remove a gap.

There are alternatives to the array that do not suffer from these limitations. The most well-known of these is the linked list.

Linked Lists

The linked list uses the same principle as the “holding hands to cross the street” exercise when you were a child. Each object contains a link to the next object in the chain. The “teacher,” otherwise known as the head pointer, points to the first element in the list.

A linkable class is declared as follows:

```
class LinkableClass
{
    public:
        LinkableClass* pNext;

        // other members of the class
};
```

Here pNext points to the next entry in the list. This is shown in Figure 15-1.

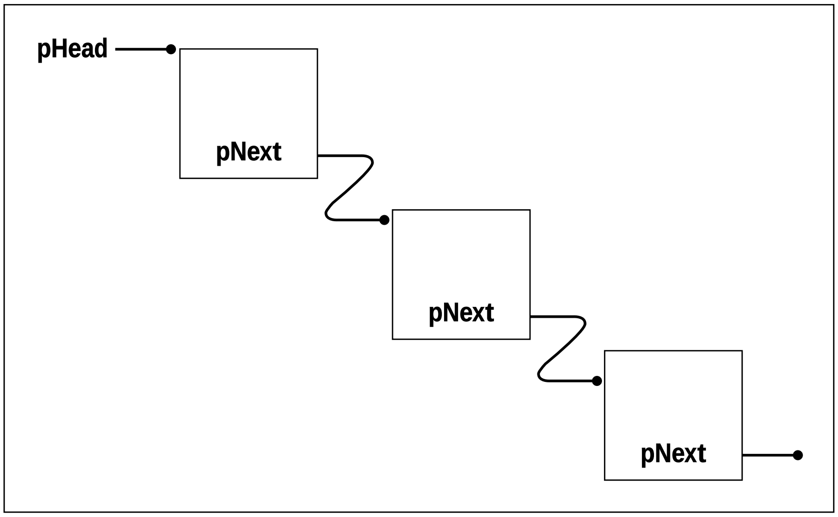

Figure 15-1
A linked list consists of a number of objects, each of which points to the next element in the list.

The head pointer is simply a pointer of type `LinkableClass*`:

```
LinkableClass* pHead = (LinkableClass*)0;
```

Note **Always initialize any pointer to 0. Zero, generally known as null when used in the context of pointers, is universally known as the "nonpointer." In any case, referring to address 0 will always cause the program to halt immediately.**

Tip **The cast from the `int` 0 to `LinkableClass*` is not necessary. C++ understands 0 to be of all types, sort of the "universal pointer." However, I find it a good practice.**

Adding to the head of a linked list

To see how linked lists work in practice consider the following simple function which adds the argument passed it to the beginning of the list:

```
void addHead(LinkableClass* pLC)
{
    pLC->pNext = pHead;
    pHead = pLC;
}
```

The process is shown graphically in Figure 15-2. After the first line, the *pLC object points to the first object in the list (the same one pointed at by pHead), shown here as step A. After the second statement, the head pointer points to the object passed, *pLC shown in step B.

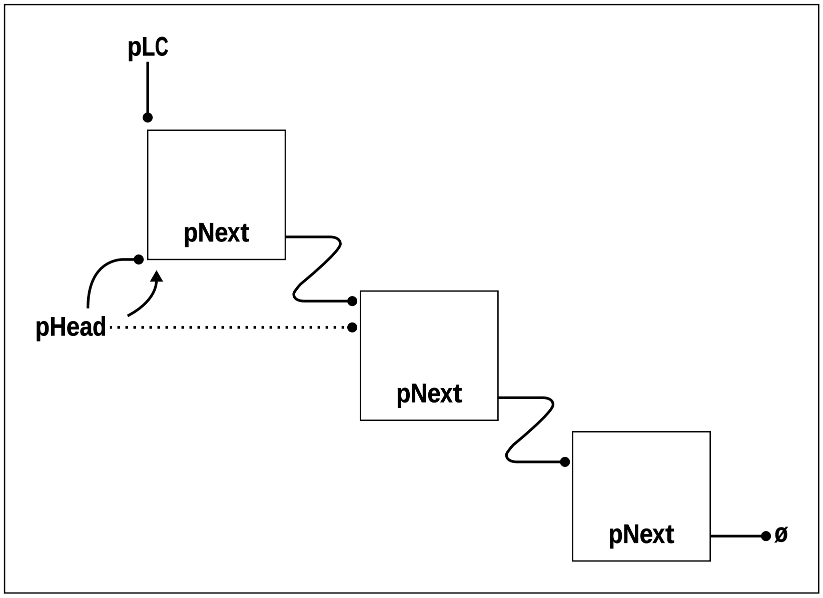

Figure 15-2
Adding an object to the head of a linked list is a two-step process.

Other operations on a linked list

Adding an object to the head of a list is the simplest of the operations on a linked list. Adding an element to the end of the list is a bit trickier:

```
void addTail(LinkableClass* pLC)
{
    // start with a pointer to the beginning
    // of the linked list
    LinkableClass* pCurrent = pHead;

    // iterate through the list until we find
    // the last object in the list - this will
    // be the one with the null next pointer
    while(pCurrent->pNext != (LinkableClass*)0)
    {
        // move pCurrent over to the next entry
        pCurrent = pCurrent->pNext;
    }

    // now make that object point to LC
    pCurrent->pNext = pLC;

    // make sure that LC's next pointer is null
    // thereby marking it as the last element in
    // the list
    pLC->pNext = (LinkableClass*)0;
}
```

The `addTail()` function begins by iterating through the loop looking for the entry who's `pNext` pointer is null — this is the last entry in the list. With that in hand, `addTail()` links the `*pLC` object to the end.

(Actually, as written `addTail()` has a bug. A special test must be added for `pHead` itself being null, indicating that the list was previously empty.)

A `remove()` function is similar. This function removes the specified object from the list and returns a 1 if successful or a 0 if not.

```
int remove(LinkableClass* pLC)
{
    LinkableClass* pCurrent = pHead;
```

```
        // if the list is empty, then obviously
        // we couldn't find *pLC in the list
        if (pCurrent == (LinkableClass*)0)
        {
            return 0;
        }

        // iterate through the loop looking for the
        // specified entry rather than the end of
        // the list
        while(pCurrent->pNext)
        {
            // if the next entry is the *pLC object...
            if (pLC == pCurrent->pNext)
            {
                // ...then point the current entry at
                // the next entry instead
                pCurrent->pNext = pLC->pNext;

                // not abolutely necessary, but remove
                // the next object from *pLC so as not
                // to get confused
                pLC->pNext = (LinkableClass*)0;
                return 1;
            }
        }
        return 0;
    }
```

The `remove()` function first checks to make sure that the list is not empty — if it is, `remove()` returns a fail indicator because obviously the `*pLC` object is not present if the list is empty. If the list is not empty, `remove()` iterates through each member until it finds the object that points to `*pLC`. If it finds that object, `remove()` moves the `pCurrent->pNext` pointer around `*pLC`. This process is shown graphically in Figure 15-3.

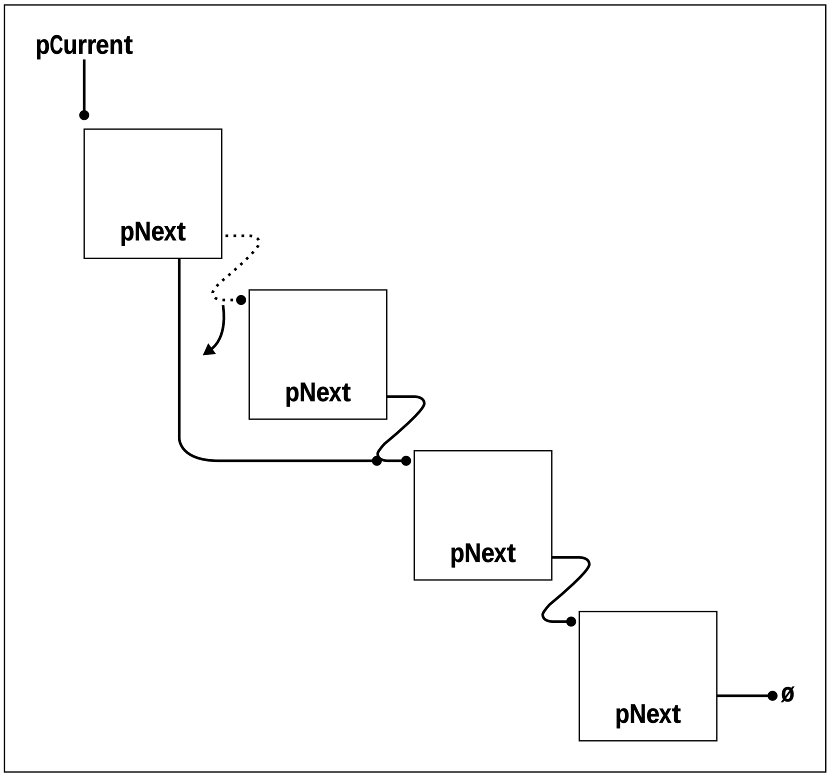

Figure 15-3
"Wire around" an entry to remove it from a linked list.

Properties of linked lists

Linked lists are everything that arrays are not. Linked lists can expand and contract at will as entries are added and removed. Inserting an object in the middle of a linked list is quick and simple — existing members do not need to be copied about. Similarly, sorting elements in a linked list is much quicker than the same process on the elements of an array.

On the negative side of the ledger, finding a member in a linked list is not nearly as quick as referencing an element in an array. Array elements are directly accessible via the index — no similar feature is available for the linked list. Programs must search sometimes the entire list to find any given entry.

A Linked NameData Program

10 Min. To Go

The `LinkedListData` program shown here is a linked-list version of the array-based `ClassData` program from Session 12.

```
// LinkedListData - store name data in
//                  a linked list of objects
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// NameDataSet - stores name and social security
//               information
class NameDataSet
{
    public:
        char szFirstName[128];
        char szLastName [128];
        int  nSocialSecurity;

        // the link to the next entry in the list
        NameDataSet* pNext;
};

// the pointer to the first entry
// in the list
NameDataSet* pHead = 0;

// addTail - add a new member to the linked list
void addTail(NameDataSet* pNDS)
{
    // make sure that our list pointer is NULL
    // since we are now the last element in the list
    pNDS->pNext = 0;

    // if the list is empty,
    // then just point the head pointer to the
    // current entry and quit
    if (pHead == 0)
    {
        pHead = pNDS;
        return;
    }

    // otherwise find the last element in the list
    NameDataSet* pCurrent = pHead;
    while(pCurrent->pNext)
    {
        pCurrent = pCurrent->pNext;
    }

    // now add the current entry onto the end of that
    pCurrent->pNext = pNDS;
}

// getData - read a name and social security
//           number; return null if no more to
//           read
NameDataSet* getData()
{
    // get a new entry to fill
    NameDataSet* pNDS = new NameDataSet;

    // read the first name
    cout << “\nEnter first name:”;
    cin  > pNDS->szFirstName;

    // if the name entered is ‘exit’...
    if ((strcmp(pNDS->szFirstName, “exit”) == 0)
        ||
        (strcmp(pNDS->szFirstName, “EXIT”) == 0))
    {
        // ...delete the still empty object...
        delete pNDS;

        // ...return a null to terminate input
        return 0;
    }

    // read the remaining members
    cout << “Enter last name:”;
    cin  > pNDS->szLastName;

    cout << “Enter social security number:”;
    cin  > pNDS->nSocialSecurity;

    // zero the pointer to the next entry
    pNDS->pNext = 0;

    // return the address of the object created
    return pNDS;
}

// displayData - output the index’th data set
void displayData(NameDataSet* pNDS)
{
    cout << pNDS->szFirstName
         << “ “
         << pNDS->szLastName
         << “/”
         << pNDS->nSocialSecurity
         << “\n”;
}

int main(int nArg, char* pszArgs[])
{
    cout << “Read name/social security information\n”
         << “Enter ‘exit’ for first name to exit\n”;

    // create (another) NameDataSet object
    NameDataSet* pNDS;
    while (pNDS = getData())
    {
        // add it onto the end of the list of
        // NameDataSet objects
        addTail(pNDS);
    }

    // to display the objects, iterate through the
    // list (stop when the next address is NULL)
```

```
        cout << "Entries:\n";
        pNDS = pHead;
        while(pNDS)
        {
            // display current entry
            displayData(pNDS);

            // get the next entry
            pNDS = pNDS->pNext;
        }
        return 0;
    }
```

Although somewhat lengthy, the LinkedListData program is relatively simple. The main() function begins by calling getData() to fetch another NameDataSet entry from the user. If the user enters exit, then getData() returns a null. main() calls addTail() to add the entry returned from getData() to the end of the linked list.

After there are no more NameDataSet objects forthcoming from the user, main() iterates through the list, displaying each using the displayData() function.

The getData() function first allocates an empty NameDataSet object from the heap. getData() continues by reading the first name of the entry to add. If the user enters a first name of exit or EXIT, the function deletes the object and returns a null to the caller. getData() continues by reading the last name and social security number. Finally, getData() zeroes out the pNext pointer before returning.

Tip

Never leave link pointers uninitialized. Use the old programmer's wives tale: "Zero them out when in doubt." (All wives of old programmers say that.)

The addTail() function appearing here is similar to the addTail() function demonstrated earlier in the chapter. Unlike that earlier version, this addTail() checks whether the list is empty before starting. If pHead is null, then addTail() points it at the current entry and terminates.

The displayData() function is a pointer-based version of the earlier displayData() functions.

Other Containers

A *container* is a structure designed to contain objects. Arrays and linked lists are specific instances of containers. The heap is also a form of container; it contains a separate block of memory that is available to your program.

Done!

You may have heard of other types of containers including the FIFO (first-in-first-out) and the LIFO (last-in-first-out), also known as the stack. These provide two functions, one to add and the other to remove objects. FIFO removes the oldest object, while LIFO removes the most recent object.

REVIEW

Creating pointers to class objects makes it possible to modify the value of class objects from within a function. Passing a reference to a class object is also considerably more efficient than passing a class object by reference. However, adding a pointer data member to a class introduces the possibility of linking objects from one object to another into a linked list. The linked list data structure offers certain advantages over arrays while giving up other efficiencies.

- Pointers to class objects work in essentially the same way as pointers to other data types. This includes the capability to pass objects by reference to functions.
- A pointer to a local object has no meeting once control passes from the function. Objects allocated off the heap do not have scope limitations and, therefore, can be passed from function to function. However, it is incumbent upon the programmer to remember to return objects back to the heap or face fatal and difficult-to-trace memory leaks.
- Objects can be strung together in linked lists if the class includes a pointer to an object of its own type. It is easy to remove and add objects to linked lists. Although not demonstrated here, sorting the objects in a linked list is also easier than sorting an array. Object pointers are also useful in the creation of other types of containers not demonstrated here.

QUIZ YOURSELF

1. Given the following:

   ```
   class MyClass
   {
       int n;
   }
   MyClass* pM;
   ```

   how would you reference the data member `m` from the pointer `pM`? (See "Pointers to Objects.")
2. What is a container? (See "Other Containers.")
3. What is a linked list? (See "Linked Lists.")
4. What is a head pointer? (See "Linked Lists.")