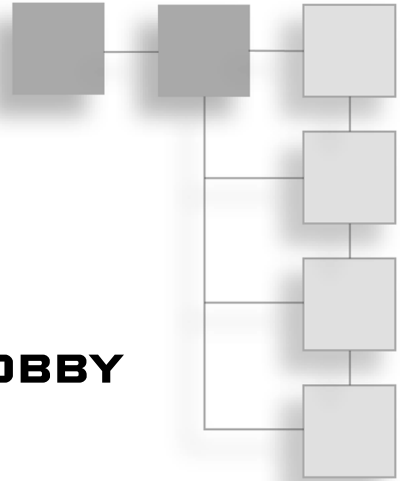


CHAPTER 9



ADVANCED CLASSES AND DYNAMIC MEMORY: GAME LOBBY

C++ gives a game programmer a high degree of control over the computer. One of the most fundamental abilities is direct control over memory. In this chapter, you'll learn about *dynamic memory*—memory that you manage yourself. But with great power comes great responsibility, so you'll also see the pitfalls of dynamic memory and how to avoid them. You'll learn a few more things about classes, too. Specifically, you'll learn to:

- Combine objects
- Use friend functions
- Overload operators
- Dynamically allocate and free memory
- Avoid memory leaks
- Produce deep copies of objects

USING AGGREGATION

Game objects are often composed of other objects. For example, in a racing game, a drag racer could be seen as a single object composed of other individual objects, such as a body, four tires, and an engine. Other times, you might see an object as a collection of related objects. In a zookeeper simulation, you might see the zoo as a collection of an arbitrary number of animals. You can mimic these kinds of relationships among objects in OOP using *aggregation*—the combining of objects so that one is part of another. For example,

you could write a `Drag_Racer` class that has an `engine` data member that's an `Engine` object. Or, you could write a `Zoo` class that has an `animals` data member that is a collection of `Animal` objects.

Introducing the Critter Farm Program

The Critter Farm program defines a new kind of critter with a name. After the program announces a new critter's name, it creates a critter farm—a collection of critters. Finally, the program performs a roll call on the farm and each critter announces its name. Figure 9.1 shows the results of the program.

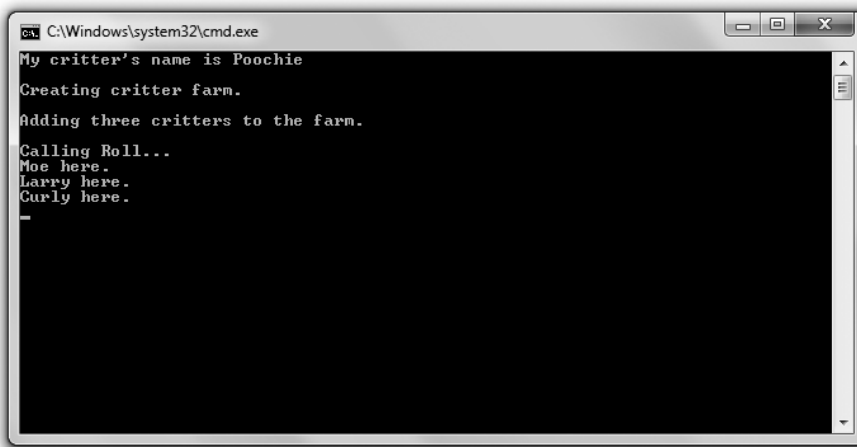


Figure 9.1

The critter farm is a collection of critters, each with a name.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 9 folder; the filename is `critter_farm.cpp`.

```
//Critter Farm
//Demonstrates object containment

#include <iostream>
#include <string>
#include <vector>
```

```

using namespace std;

class Critter
{
public:
    Critter(const string& name = "");
    string GetName() const;

private:
    string m_Name;
};

Critter::Critter(const string& name):
    m_Name(name)
{}

inline string Critter::GetName() const
{
    return m_Name;
}

class Farm
{
public:
    Farm(int spaces = 1);
    void Add(const Critter& aCritter);
    void RollCall() const;

private:
    vector<Critter> m_Critters;
};

Farm::Farm(int spaces)
{
    m_Critters.reserve(spaces);
}

void Farm::Add(const Critter& aCritter)
{
    m_Critters.push_back(aCritter);
}

void Farm::RollCall() const
{
    for (vector<Critter>::const_iterator iter = m_Critters.begin();
        iter != m_Critters.end();
        ++iter)

```

```

    {
        cout << iter->GetName() << " here.\n";
    }
}

int main()
{
    Critter crit("Poochie");
    cout << "My critter's name is " << crit.GetName() << endl;

    cout << "\nCreating critter farm.\n";
    Farm myFarm(3);

    cout << "\nAdding three critters to the farm.\n";
    myFarm.Add(Critter("Moe"));
    myFarm.Add(Critter("Larry"));
    myFarm.Add(Critter("Curly"));

    cout << "\nCalling Roll...\n";
    myFarm.RollCall();

    return 0;
}

```

Using Object Data Members

One way to use aggregation when defining a class is to declare a data member that can hold another object. That's what I did in `Critter` with the following line, which declares the data member `m_Name` to hold a `string` object.

```
string m_Name;
```

Generally, you use aggregation when an object has another object. In this case, a critter has a name. These kinds of relationships are called *has-a* relationships.

I put the declaration for the critter's name to use when I instantiate a new object with:

```
Critter crit("Poochie");
```

which calls the `Critter` constructor:

```

Critter::Critter(const string& name):
    m_Name(name)
{}

```

By passing the string literal "Poochie", the constructor is called and a `string` object for the name is instantiated, which the constructor assigns to `m_Name`. A new critter named Poochie is born.

Next, I display the critter's name with the following line:

```
cout << "My critter's name is " << crit.GetName() << endl;
```

The code `crit.GetName()` returns a copy of the `string` object for the name of the critter, which is then sent to `cout` and displayed on the screen.

Using Container Data Members

You can also use containers as data members for your objects. That's what I do when I define `Farm`. The single data member I declare for the class is simply a vector that holds `Critter` objects called `m_Critters`.

```
vector<Critter> m_Critters;
```

When I instantiate a new `Farm` object with:

```
Farm myFarm(3);
```

it calls the constructor:

```
Farm::Farm(int spaces)
{
    m_Critters.reserve(spaces);
}
```

which allocates memory for three `Critter` objects in the `Farm` object's `m_Critter` vector.

Next, I add three critters to the farm by calling the `Farm` object's `Add()` member function.

```
myFarm.Add(Critter("Moe"));
myFarm.Add(Critter("Larry"));
myFarm.Add(Critter("Curly"));
```

The following member function accepts a constant reference to a `Critter` object and adds a copy of the object to the `m_Critters` vector.

```
void Farm::Add(const Critter& aCritter)
{
    m_Critters.push_back(aCritter);
}
```

Trap

`push_back()` adds a copy of an object to a vector—this means that I create an extra copy of each `Critter` object every time I call `Add()`. This is no big deal in the `Critter Farm` program, but if I were adding many large objects, it could become a performance issue. You can reduce this overhead by using, say, a vector of pointers to objects. You'll see how to work with pointers to objects later in this chapter.

Finally, I take roll through the `Farm` object's `RollCall()` member function.

```
myFarm.RollCall();
```

This iterates through the vector, calling each `Critter` object's `GetName()` member function and getting each critter to speak up and say its name.

USING FRIEND FUNCTIONS AND OPERATOR OVERLOADING

Friend functions and operator overloading are two advanced concepts related to classes. *Friend functions* have complete access to any member of a class. *Operator overloading* allows you to define new meanings for built-in operators as they relate to objects of your own classes. As you'll see, you can use these two concepts together.

Introducing the Friend Critter Program

The Friend Critter program creates a `Critter` object. It then uses a friend function, which is able to directly access the private data member that stores the critter's name to display the critter's name. Finally, the program displays the `Critter` object by sending the object to the standard output. This is accomplished through a friend function and operator overloading. Figure 9.2 displays the results of the program.

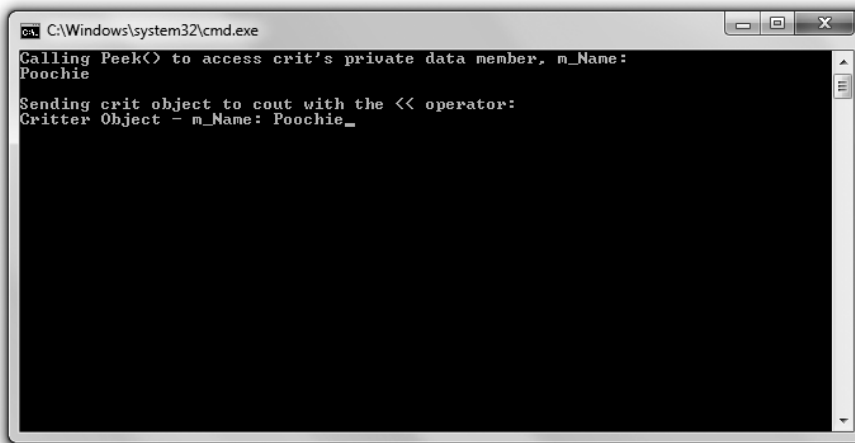


Figure 9.2

The name of the critter is displayed through a friend function, and the `Critter` object is displayed by sending it to the standard output.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 9 folder; the filename is `friend_critter.cpp`.

```
//Friend Critter
//Demonstrates friend functions and operator overloading
#include <iostream>
#include <string>
using namespace std;
class Critter
{
    //make following global functions friends of the Critter class
    friend void Peek(const Critter& aCritter);
    friend ostream& operator<<(ostream& os, const Critter& aCritter);
public:
    Critter(const string& name = "");
private:
    string m_Name;
};
Critter::Critter(const string& name):
    m_Name(name)
{}
void Peek(const Critter& aCritter);
ostream& operator<<(ostream& os, const Critter& aCritter);
int main()
{
    Critter crit("Poochie");
    cout << "Calling Peek() to access crit's private data member, m_Name: \n";
    Peek(crit);
    cout << "\nSending crit object to cout with the << operator:\n";
    cout << crit;
    return 0;
}
//global friend function that can access all of a Critter object's members
void Peek(const Critter& aCritter)
{
    cout << aCritter.m_Name << endl;
}
```

```
//global friend function that can access all of Critter object's members
//overloads the << operator so you can send a Critter object to cout
ostream& operator<<(ostream& os, const Critter& aCritter)
{
    os << "Critter Object - ";
    os << "m_Name: " << aCritter.m_Name;
    return os;
}
```

Creating Friend Functions

A friend function can access any member of a class of which it's a friend. You specify that a function is a friend of a class by listing the function prototype preceded by the keyword `friend` inside the class definition. That's what I do inside the `Critter` definition with the following line, which says that the global function `Peek()` is a friend of `Critter`.

```
friend void Peek(const Critter& aCritter);
```

This means `Peek()` can access any member of `Critter` even though it's not a member function of the class. `Peek()` takes advantage of this relationship by accessing the private data member `m_Name` to display the name of a critter passed to the function.

```
void Peek(const Critter& aCritter)
{
    cout << aCritter.m_Name << endl;
}
```

When I call `Peek()` in `main()` with the following line, the private data member `m_Name` of `crit` is displayed and Poochie appears on the screen.

```
Peek(crit);
```

Overloading Operators

Overloading operators might sound like something you want to avoid at all costs—as in, “Look out, that operator is overloaded and she’s about to blow!”—but it’s not. *Operator overloading* lets you give meaning to built-in operators used with new types that you define. For example, you could overload the `*` operator so that when it is used with two 3D matrices (objects instantiated from some class that you’ve defined), the result is the multiplication of the matrices.

To overload an operator, define a function called `operatorX`, where `X` is the operator you want to overload. That's what I do when I overload the `<<` operator; I define a function named `operator<<`.

```
ostream& operator<<(ostream& os, const Critter& aCritter)
{
    os << "Critter Object - ";
    os << "m_Name: " << aCritter.m_Name;
    return os;
}
```

The function overloads the `<<` operator so that when I send a `Critter` object with the `<<` to `cout`, the data member `m_Name` is displayed. Essentially, the function allows me to easily display `Critter` objects. The function can directly access the private data member `m_Name` of a `Critter` object because I made the function a friend of the `Critter` class with the following line in `Critter`:

```
friend ostream& operator<<(ostream& os, const Critter& aCritter);
```

This means I can simply display a `Critter` object by sending it to `cout` with the `<<` operator, which is what I do in `main()` with the following line, which displays the text `Critter Object - m_Name: Poochie`.

```
cout << crit;
```

Hint

With all the tools and debugging options available to game programmers, sometimes simply displaying the values of variables is the best way to understand what's happening in your programs. Overloading the `<<` operator can help you do that.

This function works because `cout` is of the type `ostream`, which already overloads the `<<` operator so that you can send built-in types to `cout`.

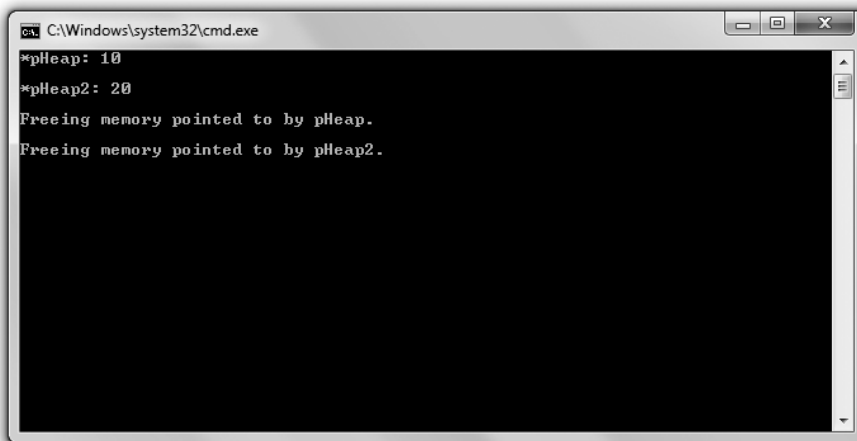
DYNAMICALLY ALLOCATING MEMORY

So far, whenever you've declared a variable, C++ has allocated the necessary memory for it. When the function that the variable was created in ended, C++ freed the memory. This memory, which is used for local variables, is called the *stack*. But there's another kind of memory that persists independent of the functions in a program. You, the programmer, are in charge of allocating and freeing this memory, collectively called the *heap* (or *free store*).

At this point, you might be thinking, “Why bother with another type of memory? The stack works just fine, thank you.” Using the dynamic memory of the heap offers great benefits that can be summed up in one word: efficiency. By using the heap, you can use only the amount of memory you need at any given time. If you have a game with a level that has 100 enemies, you can allocate the memory for the enemies at the beginning of the level and free the memory at the end. The heap also allows you to create an object in one function that you can access even after that function ends (without having to return a copy of the object). You might create a screen object in one function and return access to it. You’ll find that dynamic memory is an important tool in writing any significant game.

Introducing the Heap Program

The Heap program demonstrates dynamic memory. The program dynamically allocates memory on the heap for an integer variable, assigns it a value, and then displays it. Next, the program calls a function that dynamically allocates memory on the heap for another integer variable, assigns it a value, and returns a pointer to it. The program takes the returned pointer, uses it to display the value, and then frees the allocated memory on the heap. Finally, the program contains two functions that demonstrate the misuse of dynamic memory. I don’t call these functions, but I use them to illustrate what *not* to do with dynamic memory. Figure 9.3 shows the program.



```
C:\Windows\system32\cmd.exe
*Heap: 10
*Heap2: 20
Freeing memory pointed to by pHeap.
Freeing memory pointed to by pHeap2.
```

Figure 9.3

The two `int` values are stored on the heap.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 9 folder; the filename is heap.cpp.

```
// Heap
// Demonstrates dynamically allocating memory
#include <iostream>

using namespace std;

int* intOnHeap(); //returns an int on the heap
void leak1();      //creates a memory leak
void leak2();      //creates another memory leak

int main()
{
    int* pHeap = new int;
    *pHeap = 10;
    cout << "*pHeap: " << *pHeap << "\n\n";

    int* pHeap2 = intOnHeap();
    cout << "*pHeap2: " << *pHeap2 << "\n\n";

    cout << "Freeing memory pointed to by pHeap.\n\n";
    delete pHeap;

    cout << "Freeing memory pointed to by pHeap2.\n\n";
    delete pHeap2;

    //get rid of dangling pointers
    pHeap = 0;
    pHeap2 = 0;

    return 0;
}

int* intOnHeap()
{
    int* pTemp = new int(20);
    return pTemp;
}

void leak1()
{
    int* drip1 = new int(30);
}
```

```
void leak2()
{
    int* drip2 = new int(50);
    drip2 = new int(100);
    delete drip2;
}
```

Using the new Operator

The `new` operator allocates memory on the heap and returns its address. You use `new` followed by the type of value you want to reserve space for. That's what I do in the first line of `main()`.

```
int* pHeap = new int;
```

The `new int` part of the statement allocates enough memory on the heap for one `int` and returns the address on the heap for that chunk of memory. The other part of the statement, `int* pHeap`, declares a local pointer, `pHeap`, which points to the newly allocated chunk of memory on the heap.

By using `pHeap`, I can manipulate the chunk of memory on the heap reserved for an integer. That's what I do next; I assign 10 to the chunk of memory and then I display that value stored on the heap, using `pHeap`, as I would any other pointer to `int`. The only difference is that `pHeap` points to a piece of memory on the heap, not the stack.

Hint

You can initialize memory on the heap at the same time you allocate it by placing a value, surrounded by parentheses, after the type. This is even easier than it sounds. For example, the following line allocates a chunk of memory on the heap for an `int` variable and assigns 10 to it. The statement then assigns the address of that chunk of memory to `pHeap`.

```
int* pHeap = new int(10);
```

One of the major advantages of memory on the heap is that it can persist beyond the function in which it was allocated, meaning that you can create an object on the heap in one function and return a pointer or reference to it. That's what I demonstrate with the following line:

```
int* pHeap2 = intOnHeap();
```

The statement calls the function `intOnHeap()`, which allocates a chunk of memory on the heap for an `int` and assigns 20 to it.

```
int* intOnHeap()
{
    int* pTemp = new int(20);
    return pTemp;
}
```

Then, the function returns a pointer to this chunk of memory. Back in `main()`, the assignment statement assigns the address of the chunk of memory on the heap to `pHeap2`. Next, I use the returned pointer to display the value.

```
cout << "pHeap2: " << *pHeap2 << "\n\n";
```

Hint

Up until now, if you wanted to return a value created in a function, you had to return a copy of the value. But by using dynamic memory, you can create an object on the heap in a function and return a pointer to the new object.

Using the delete Operator

Unlike storage for local variables on the stack, memory that you've allocated on the heap must be explicitly freed. When you're finished with memory that you've allocated with `new`, you should free it with `delete`. That's what I do with the following line, which frees the memory on the heap that stored 10.

```
delete pHeap;
```

That memory is returned to the heap for future use. The data that was stored in it is no longer available. Next, I free some more memory, which frees the memory on the heap that stored 20.

```
delete pHeap2;
```

That memory is returned to the heap for future use, and the data that was stored in it is no longer available. Notice that there's no difference, as far as `delete` is concerned, regarding where in the program I allocated the memory on the heap that I'm deleting.

Trick

Because you need to free memory that you've allocated once you're finished with it, a good rule of thumb is that every `new` should have a corresponding `delete`. In fact, some programmers write the `delete` statement just after writing the `new` statement whenever possible, so they don't forget it.

An important point to understand here is that the two previous statements free the memory on the heap, but they do not directly affect the local variables `pHeap` and `pHeap2`.

This creates a potential problem because `pHeap` and `pHeap2` now point to memory that has been returned to the heap, meaning that they point to memory that the computer can use in some other way at any given time. Pointers like this are called *dangling pointers*, and they are quite dangerous. You should never attempt to dereference a dangling pointer. One way to deal with dangling pointers is to assign 0 to them, and that's what I do with the following lines, which reassign both dangling pointers so they no longer point to some memory to which they should not point.

```
pHeap = 0;
pHeap2 = 0;
```

Another good way to deal with a dangling pointer is to assign a valid memory address to it.

Trap

Using `delete` on a dangling pointer can cause your program to crash. Be sure to set a dangling pointer to 0 or reassign it to point to a new, valid chunk of memory.

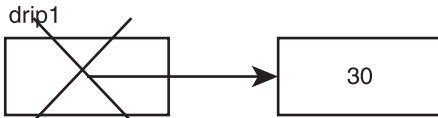
Avoiding Memory Leaks

One problem with allowing a programmer to allocate and free memory is that he might allocate memory and lose any way to get at it, thus losing any way to ever free it. When memory is lost like this, it's called a *memory leak*. Given a large enough leak, a program might run out of memory and crash. As a game programmer, it's your responsibility to avoid memory leaks.

I've written two functions in the `Heap` program that purposely create memory leaks in order to show you what *not* to do when using dynamic memory. The first function is `leak1()`, which simply allocates a chunk of memory on the heap for an `int` value and then ends.

```
void leak1()
{
    int* drip1 = new int(30);
}
```

If I were to call this function, memory would be lost forever. (Okay, it would be lost until the program ended.) The problem is that `drip1`, which is the only connection to the newly acquired chunk of memory on the heap, is a local variable and ceases to exist when the function `leak1()` ends. So, there's no way to free the allocated memory. Take a look at Figure 9.4 for a visual representation of how the leak occurs.

**Figure 9.4**

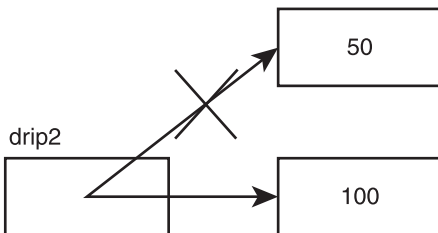
The memory that stores 30 can no longer be accessed to be freed, so it has leaked out of the system.

To avoid this memory leak, I could do one of two things. I could use `delete` to free the memory in `leak1()`, or I could return a copy of the pointer `drip1`. If I choose the second option, I have to make sure to free this memory in some other part of the program.

The second function that creates a memory leak is `leak2()`.

```
void leak2()
{
    int* drip2 = new int(50);
    drip2 = new int(100);
    delete drip2;
}
```

The memory leak is a little more subtle, but there is still a leak. The first line in the function body, `int* drip2 = new int(50);`, allocates a new piece of memory on the heap, assigns 50 to it, and has `drip2` point to that piece of memory. So far, so good. The second line, `drip2 = new int(100);`, points `drip2` to a new piece of memory on the heap, which stores the 100. The problem is that the memory on the heap that stores 50 now has nothing pointing to it, so there is no way for the program to free that memory. As a result, that piece of memory has essentially leaked out of the system. Check out Figure 9.5 for a visual representation of how the leak occurs.

**Figure 9.5**

By changing `drip2` so that it points to the memory that stores 100, the memory that stores 50 is no longer accessible and has leaked out of the system.

The last statement of the function, `delete drip2;`, frees the memory that stores 100, so this won't be the source of another memory leak. But remember, the memory on the heap that stores 50 has still leaked out of the system. Also, I don't worry about `drip2`, which technically has become a dangling pointer, because it will cease to exist when the function ends.

WORKING WITH DATA MEMBERS AND THE HEAP

You've seen how you can use aggregation to declare data members that store objects, but you can also declare data members that are pointers to values on the heap. You might use a data member that points to a value on the heap for some of the same reasons you would use pointers in other situations. For example, you might want to declare a data member for a large 3D scene; however, you might only have access to the 3D scene through a pointer. Unfortunately, problems can arise when you use a data member that points to a value on the heap because of the way that some default object behaviors work. But you can avoid these issues by writing member functions to change these default behaviors.

Introducing the Heap Data Member Program

The Heap Data Member program defines a new type of critter with a data member that is a pointer, which points to an object stored on the heap. The class defines a few new member functions to handle situations in which an object is destroyed, copied, or assigned to another object. The program destroys, copies, and assigns objects to show that the objects behave as you'd expect, even with data members pointing to values on the heap. Figure 9.6 shows the results of the Heap Data Member program.

```

C:\Windows\system32\cmd.exe
Constructor called
I'm Rover and I'm 3 years old. &m_pName: 73F2ED48003AF644
Destructor called

Constructor called
I'm Poochie and I'm 5 years old. &m_pName: 73F2ED48003AF78C
Copy Constructor called
I'm Poochie and I'm 5 years old. &m_pName: 73F2ED48003AF660
Destructor called
I'm Poochie and I'm 5 years old. &m_pName: 73F2ED48003AF78C

Constructor called
Constructor called
Overloaded Assignment Operator called
I'm crit2 and I'm 9 years old. &m_pName: 73F2ED48003AF644
I'm crit2 and I'm 9 years old. &m_pName: 73F2ED48003AF634

Constructor called
Overloaded Assignment Operator called
I'm crit and I'm 11 years old. &m_pName: 73F2ED48003AF624
Destructor called
Destructor called
Destructor called
  
```

Figure 9.6

Objects, each with a data member that points to a value on the heap, are instantiated, destroyed, and copied.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 9 folder; the filename is `heap_data_member.cpp`.

```
//Heap Data Member
//Demonstrates an object with a dynamically allocated data member

#include <iostream>
#include <string>

using namespace std;

class Critter
{
public:
    Critter(const string& name = "", int age = 0);
    ~Critter();           //destructor prototype
    Critter(const Critter& c); //copy constructor prototype
    Critter& Critter::operator=(const Critter& c); //overloaded assignment op
    void Greet() const;

private:
    string* m_pName;
    int m_Age;
};

Critter::Critter(const string& name, int age)
{
    cout << "Constructor called\n";
    m_pName = new string(name);
    m_Age = age;
}

Critter::~~Critter()           //destructor definition
{
    cout << "Destructor called\n";
    delete m_pName;
}

Critter::Critter(const Critter& c) //copy constructor definition
{
    cout << "Copy Constructor called\n";
    m_pName = new string(*(c.m_pName));
    m_Age = c.m_Age;
}
```

```

Critic& Critter::operator=(const Critter& c) //overloaded assignment op def
{
    cout << "Overloaded Assignment Operator called\n";
    if (this != &c)
    {
        delete m_pName;
        m_pName = new string(*(c.m_pName));
        m_Age = c.m_Age;
    }
    return *this;
}

void Critter::Greet() const
{
    cout << "I'm " << *m_pName << " and I'm " << m_Age << " years old. ";
    cout << "&m_pName: " << &m_pName << endl;
}

void testDestructor();
void testCopyConstructor(Critter aCopy);
void testAssignmentOp();

int main()
{
    testDestructor();
    cout << endl;

    Critter crit("Poochie", 5);
    crit.Greet();
    testCopyConstructor(crit);
    crit.Greet();
    cout << endl;

    testAssignmentOp();

    return 0;
}

void testDestructor()
{
    Critter toDestroy("Rover", 3);
    toDestroy.Greet();
}

void testCopyConstructor(Critter aCopy)
{
    aCopy.Greet();
}

```

```
void testAssignmentOp()
{
    Critter crit1("crit1", 7);
    Critter crit2("crit2", 9);
    crit1 = crit2;
    crit1.Greet();
    crit2.Greet();
    cout << endl;

    Critter crit3("crit", 11);
    crit3 = crit3;
    crit3.Greet();
}
```

Declaring Data Members that Point to Values on the Heap

To declare a data member that points to a value on the heap, you first need to declare a data member that's a pointer. That's just what I do in `Critter` with the following line, which declares `m_pName` as a pointer to a string object.

```
string* m_pName;
```

In the class constructor, you can allocate memory on the heap, assign a value to the memory, and then point a pointer data member to the memory. That's what I do in the constructor definition with the following line, which allocates memory for a string object, assigns `name` to it, and points `m_pName` to that chunk of memory on the heap.

```
m_pName = new string(name);
```

I also declare a data member that is not a pointer:

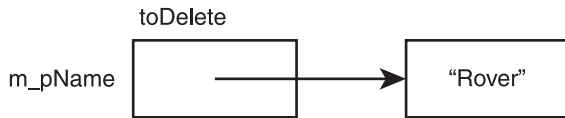
```
int m_Age;
```

This data member gets its value in the constructor the way you've seen before, with a simple assignment statement:

```
m_Age = age;
```

You'll see how each of these data members is treated differently as `Critter` objects are destroyed, copied, and assigned to each other.

Now, the first object with a data member on the heap is created when `main()` calls `testDestructor()`. The object, `toDestroy`, has an `m_pName` data member that points to a string object equal to "Rover" that's stored on the heap. Figure 9.7 provides a visual representation of the `Critter` object. Note that the image is abstract because the name of the critter is actually stored as a string object, not a string literal.

**Figure 9.7**

A representation of a `Critter` object. The string object equal to "Rover" is stored on the heap.

Declaring and Defining Destructors

One problem that can occur when a data member of an object points to a value on the heap is a memory leak. That's because when the object is deleted, the pointer to the heap value disappears along with it. If the heap value remains, it produces a memory leak. To avoid a memory leak, the object should clean up after itself before it is destroyed by deleting its associated heap value. Fortunately, there's a member function, the *destructor*, that's called just before an object is destroyed, which can be used to perform the necessary cleanup.

A default destructor, which is created for you by the compiler if you don't write your own, doesn't attempt to free any memory on the heap that a data member might point to. This behavior is usually fine for simple classes, but when you have a class with data members that point to values on the heap, you should write your own destructor so you can free the memory on the heap associated with an object before the object disappears, avoiding a memory leak. That's what I do in the `Critter` class. First, inside the class definition, I declare the destructor. Notice that a destructor has the name of the class preceded by ~ (the tilde character) and does not have any parameters or return a value.

```
Critter::~Critter()                //destructor definition
{
    cout << "Destructor called\n";
    delete m_pName;
}
```

In `main()`, I put the destructor to the test when I call `testDestructor()`. The function creates a `Critter` object, `toDestroy`, and invokes its `Greet()` method, which displays `I'm Rover` and `I'm 3 years old`. `&m_pName: 73F2ED48003AF644`. The message provides a way to see the values of the object's `m_Age` data member and the string pointed to by its `m_pName` data member. But it also displays the address of the string on the heap stored in the pointer `m_pName`. The important thing to note is that after the `Greet()` message is displayed, the function ends and `toDestroy` is ready to be destroyed. Fortunately, `toDestroy`'s destructor

is automatically called just before this happens. The destructor displays `Destructor` called and deletes the `string` object equal to "Rover" that's on the heap, cleaning up after itself and leaking no memory. The destructor doesn't do anything with the `m_Age` data member. That's perfectly fine, since `m_Age` isn't on the heap, but is part of `toDestroy` and will be properly disposed of right along with the rest of the `Critter` object.

Hint

When you have a class that allocates memory on the heap, you should write a destructor that cleans up and frees that memory.

Declaring and Defining Copy Constructors

Sometimes an object is copied automatically for you. This occurs when an object is:

- Passed by value to a function
- Returned from a function
- Initialized to another object through an initializer
- Provided as a single argument to the object's constructor

The copying is done by a special member function called the *copy constructor*. Like constructors and destructors, a default copy constructor is supplied for you if you don't write one of your own. The default copy constructor simply copies the value of each data member to data members of the same name in the new object—a *member-wise copy*.

For simple classes, the default copy constructor is usually fine. However, when you have a class with a data member that points to a value on the heap, you should consider writing your own copy constructor. Why? Imagine a `Critter` object that has a data member that's a pointer to a `string` object on the heap. With only a default copy constructor, the automatic copying of the object would result in a new object that points to the same single `string` on the heap because the pointer of the new object would simply get a copy of the address stored in the pointer of the original object. This member-wise copying produces a *shallow copy*, in which the pointer data members of the copy point to the same chunks of memory as the pointer data members in the original object.

Let me give you a specific example. If I hadn't written my own copy constructor in the Heap Data Member program, when I passed a `Critter` object by value with the following

function call, the program would have automatically made a shallow copy of `crit` called `aCopy` that existed in `testCopyConstructor()`.

```
testCopyConstructor(crit);
```

`aCopy`'s `m_pName` data member would point to the exact same string object on the heap as `crit`'s `m_pName` data member does. Figure 9.8 shows you what I mean. Note that the image is abstract since the name of the critter is actually stored as a string object, not a string literal.

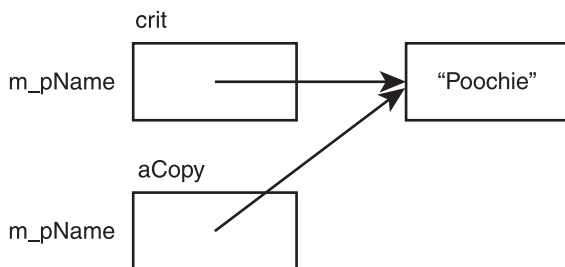


Figure 9.8

If a shallow copy of `crit` were made, both `aCopy` and `crit` would have a data member that points to the same chunk of memory on the heap.

Why is this a problem? Once `testCopyConstructor()` ends, `aCopy`'s destructor is called, freeing the memory on the heap pointed to by `aCopy`'s `m_pName` data member. Because of this, `crit`'s `m_pName` data member would point to memory that has been freed, which would mean that `crit`'s `m_pName` data member would be a dangling pointer! Figure 9.9 provides you with a visual representation of this. Note that the image is abstract since the name of the critter is actually stored as a string object, not a string literal.

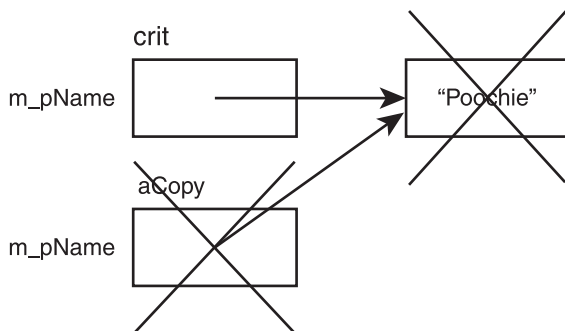


Figure 9.9

If the shallow copy of the `Critter` object were destroyed, the memory on the heap that it shared with the original object would be freed. As a result, the original object would have a dangling pointer.

What you really need is a copy constructor that produces a new object with its own chunk of memory on the heap for each data member that points to a heap object—a *deep copy*. That's what I do when I define a copy constructor for the class, which replaces the default one provided by the compiler. First, inside the class definition, I declare the copy constructor:

```
Critter(const Critter& c);    //copy constructor prototype
```

Next, outside the class definition, I define the copy constructor:

```
Critter::Critter(const Critter& c)        //copy constructor definition
{
    cout << "Copy Constructor called\n";
    m_pName = new string(*(c.m_pName));
    m_Age = c.m_Age;
}
```

Just like this one, a copy constructor must have the same name as the class. It returns no value, but accepts a reference to an object of the class—the object that needs to be copied. The reference should be made a constant reference to protect the original object from being changed during the copy process.

The job of a copy constructor is to copy any data members from the original object to the copy object. If a data member of the original object is a pointer to a value on the heap, the copy constructor should request memory from the heap, copy the original heap value to this new chunk of memory, and then point the appropriate copy object data member to this new memory.

When I call `testCopyConstructor()` by passing `crit` to the function by value, the copy constructor I wrote is automatically called. You can tell this because the text `Copy Constructor called.` appears on the screen. My copy constructor creates a new `Critter` object (the copy) and accepts a reference to the original in `c`. With the line `m_pName = new string(*(c.m_pName));`, my copy constructor allocates a new chunk of memory on the heap, gets a copy of the `string` pointed to by the original object, copies it to the new memory, and points the `m_pName` data member of the copy to this memory. The next line, `m_Age = c.m_Age;` simply copies the value of the original's `m_Age` to the copy's `m_Age` data member. As a result, a deep copy of `crit` is made, and that's what gets used in `testCopyConstructor()` as `aCopy`.

You can see that the copy constructor worked when I called `aCopy's Greet()` member function. In my sample run, the member function displayed a message, part of which was `I'm Poochie and I'm 5 years old`. This part of the message shows that `aCopy` correctly got a copy of the values of the data members from the object `crit`. The second part of the

message, `&m_pName: 73F2ED48003AF660`, shows that the `string` object pointed to by the data member `m_pName` of `aCopy` is stored in a different chunk of memory than the string pointed to by the data member `m_pName` of `crit`, which is stored at memory location `73F2ED48003AF78C`, proving that a deep copy was made. Remember that the memory addresses displayed in my sample run may be different from the ones displayed when the program is run again. However, the key here is that the addresses stored in `crit`'s `m_pName` and `aCopy`'s `m_pName` are different from each other.

When `testCopyConstructor()` ends, the copy of the `Critter` object used in the function, stored in the variable `aCopy`, is destroyed. The destructor frees the chunk of memory on the heap associated with the copy, leaving the original `Critter` object, `crit`, created in `main()`, unaffected. Figure 9.10 shows the results. Note that the image is abstract since the name of the critter is actually stored as a `string` object, not a `string` literal.

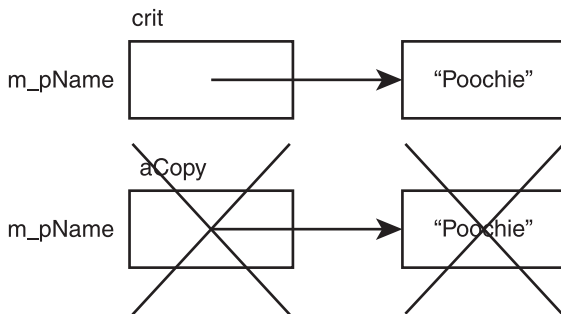


Figure 9.10

With a proper copy constructor, the original and the copy each point to their own chunk of memory on the heap. Then, when the copy is destroyed, the original is unaffected.

Hint

When you have a class with data members that point to memory on the heap, you should consider writing a copy constructor that allocates memory for a new object and creates a deep copy.

Overloading the Assignment Operator

When both sides of an assignment statement are objects of the same class, the class' assignment operator member function is called. Like a default copy constructor, a default assignment operator member function is supplied for you if you don't write one of your own. Also like the default copy constructor, the default assignment operator provides only member-wise duplication.

For simple classes, the default assignment operator is usually fine. However, when you have a class with a data member that points to a value on the heap, you should consider writing an overloaded assignment operator of your own. If you don't, you'll end up with shallow copies of objects when you assign one object to another. To avoid this problem, I overloaded the assignment operator for `Critter`. First, inside the class definition, I write the declaration:

```
Critter& Critter::operator=(const Critter& c); //overloaded assignment op
```

Next, outside the class definition, I write the member function definition:

```
Critter& Critter::operator=(const Critter& c) //overloaded assignment op def
{
    cout << "Overloaded Assignment Operator called\n";
    if (this != &c)
    {
        delete m_pName;
        m_pName = new string(*(c.m_pName));
        m_Age = c.m_Age;
    }
    return *this;
}
```

Notice that the member function returns a reference to a `Critter` object. For robust assignment operation, return a reference from the overloaded assignment operator member function.

In `main()`, I call a function that tests the overloaded assignment operator for this class.

```
testAssignmentOp();
```

The `testAssignmentOp()` creates two objects and assigns one to the other.

```
Critter crit1("crit1", 7);
Critter crit2("crit2", 9);
crit1 = crit2;
```

The preceding assignment statement, `crit1 = crit2;`, calls the assignment operator member function—`operator=()`—for `crit1`. In the `operator=()` function, `c` is a constant reference to `crit2`. The goal of the member function is to assign the values of all of the data members of `crit2` to `crit1` while making sure each `Critter` object has its own chunks of memory on the heap for any pointer data members.

After `operator=()` displays a message that the overloaded assignment operator has been called, it uses the `this` pointer. What's the `this` pointer? It's a pointer that all non-static

member functions automatically have, which points to the object that was used to call the function. In this case, `this` points to `crit1`, the object being assigned to.

The next line, `if (this != &c)`, checks to see whether the address of `crit1` is not equal to the address of `crit2`—that is, it tests if the object isn't being assigned to itself. Because it's not, the block associated with the `if` statement executes.

Inside the `if` block, `delete m_pName`; frees the memory on the heap that `crit1`'s `m_pName` data member pointed to. The line `m_pName = new string(*(c.m_pName));` allocates a new chunk of memory on the heap, gets a copy of the string pointed to by the `m_pName` data member of `crit2`, copies the string object to the new heap memory, and points the `m_pName` data member of `crit1` to this memory. You should follow this logic for all data members that point to memory on the heap.

The last line in the block, `m_Age = c.m_Age`; simply copies the value of the `crit2`'s `m_Age` to `crit1`'s `m_Age` data member. You should follow this simple member-wise copying for all data members that are not pointers to memory on the heap.

Finally, the member function returns a copy of the new `crit1` by returning `*this`. You should do the same for any overloaded assignment operator member function you write.

Back in `testAssignmentOp()`, I prove that the assignment worked by calling `crit1.Greet()` and `crit2.Greet()`. `crit1` displays the message `I'm crit2 and I'm 9 years old. &m_pName: 73F2ED48003AF644`, while `crit2` displays the message `I'm crit2 and I'm 9 years old. &m_pName: 73F2ED48003AF634`. The first part of each message, `I'm crit2 and I'm 9 years old.`, is the same and shows that the copying of values worked. The second part of each message is different and shows that each object points to different chunks of memory on the heap, which demonstrates that I avoided shallow copies and have truly independent objects after the assignment.

In the last test of the overloaded assignment operator, I demonstrate what happens when you assign an object to itself. That's what I do next in the function with the following lines:

```
Critter crit3("crit", 11);
crit3 = crit3;
```

The preceding assignment statement, `crit3 = crit3`;, calls the assignment operator member function—`operator=()`—for `crit3`. The `if` statement checks to see whether `crit3` is being assigned to itself. Because it is, the member function simply returns a reference to the object through `return *this`. You should follow this logic in your own overloaded assignment operator because of potential problems that can arise from only one object being involved in an assignment.

Hint

When you have a class with a data member that points to memory on the heap, you should consider overloading the assignment operator for the class.

INTRODUCING THE GAME LOBBY PROGRAM

The Game Lobby program simulates a game lobby—a waiting area for players, usually in an online game. The program doesn't actually involve an online component. It creates a single line in which players can wait. The user of the program runs the simulation and has four choices. He can add a person to the lobby, remove a person from the lobby (the first person in line is the first to leave), clear out the lobby, or quit the simulation. Figure 9.11 shows the program in action.

```

C:\Windows\system32\cmd.exe

Here's who's in the game lobby:
Mike
Steve
Larry

GAME LOBBY
0 - Exit the program.
1 - Add a player to the lobby.
2 - Remove a player from the lobby.
3 - Clear the lobby.

Enter choice: 2

Here's who's in the game lobby:
Steve
Larry

GAME LOBBY
0 - Exit the program.
1 - Add a player to the lobby.
2 - Remove a player from the lobby.
3 - Clear the lobby.

Enter choice: 1

```

Figure 9.11

The lobby holds players who are removed in the order in which they were added.

Used with permission from Microsoft.

The Player Class

The first thing I do is create a `Player` class to represent the players who are waiting in the game lobby. Because I don't know how many players I'll have in my lobby at one time, it makes sense to use a dynamic data structure. Normally, I'd go to my toolbox of containers from the STL. But I decided to take a different approach in this program and create my own kind of container using dynamically allocated memory that I manage. I didn't do this because it's a better programming choice (always see whether you can leverage good work

done by other programmers, like the STL) but because it makes for a better game programming example. It's a great way to really see dynamic memory in action.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 9 folder; the filename is `game_lobby.cpp`. Here's the beginning of the program, which includes the `Player` class:

```
//Game Lobby
//Simulates a game lobby where players wait

#include <iostream>
#include <string>

using namespace std;

class Player
{
public:
    Player(const string& name = "");
    string GetName() const;
    Player* GetNext() const;
    void SetNext(Player* next);

private:
    string m_Name;
    Player* m_pNext; //Pointer to next player in list
};

Player::Player(const string& name):
    m_Name(name),
    m_pNext(0)
{}

string Player::GetName() const
{
    return m_Name;
}

Player* Player::GetNext() const
{
    return m_pNext;
}

void Player::SetNext(Player* next)
{
    m_pNext = next;
}
```

The `m_Name` data member holds the name of a player. That's straightforward, but you might be wondering about the other data member, `m_pNext`. It's a pointer to a `Player` object, which means that each `Player` object can hold a name and point to another `Player` object. You'll get the point of all this when I talk about the `Lobby` class. Figure 9.12 provides a visual representation of a `Player` object.



Figure 9.12

A `Player` object can hold a name and point to another `Player` object.

The class has a `Get` accessor method for `m_Name` and `Get` and `Set` accessor member functions for `m_pNext`. Finally, the constructor is pretty simple. It initializes `m_Name` to a string object based on what's passed to the constructor. It also sets `m_pNext` to 0, making it a null pointer.

The Lobby Class

The `Lobby` class represents the lobby or line in which players wait. Here's the class definition:

```
class Lobby
{
    friend ostream& operator<<(ostream& os, const Lobby& aLobby);

public:
    Lobby();
    ~Lobby();
    void AddPlayer();
    void RemovePlayer();
    void Clear();

private:
    Player* m_pHead;
};
```

The data member `m_pHead` is a pointer that points to a `Player` object, which represents the first person in line. `m_pHead` represents the head of the line.

Because each `Player` object has an `m_pNext` data member, you can link a bunch of `Player` objects in a *linked list*. Individual elements of linked lists are often called *nodes*. Figure 9.13 provides a visual representation of a game lobby—a series of player nodes linked with one player at the head of the line.

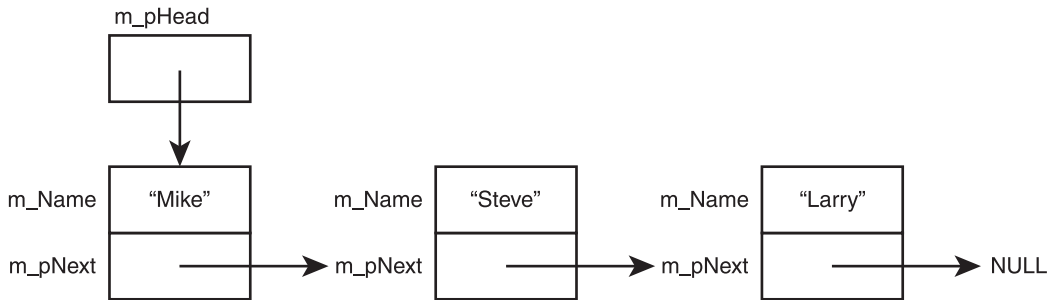


Figure 9.13

Each node holds a name and a pointer to the next player in the list. The first player in line is at the head.

One way to think about the player nodes is as a group of train cars that carry cargo and are connected. In this case, the train cars carry a name as cargo and are linked through a pointer data member, `m_pNext`. The `Lobby` class allocates memory on the heap for each `Player` object in the list. The `Lobby` data member `m_pHead` provides access to the first `Player` object at the head of the list.

The constructor is very simple. It simply initializes the data member `m_pHead` to 0, making it a null pointer.

```
Lobby::Lobby():
    m_pHead(0)
{ }
```

The destructor simply calls `Clear()`, which removes all the `Player` objects from the list, freeing the allocated memory.

```
Lobby::~~Lobby()
{
    Clear();
}
```

`AddPlayer()` instantiates a `Player` object on the heap and adds it to the end of the list. `RemovePlayer()` removes the first `Player` object in the list, freeing the allocated memory.

I declare the function operator `<<()` a friend of `Lobby` so that I can send a `Lobby` object to `cout` using the `<<` operator.

Trap

The Lobby class has a data member, `m_pHead`, which points to `Player` objects on the heap. Because of this, I included a destructor that frees all of the memory occupied by the `Player` objects on the heap instantiated by a Lobby object to avoid any memory leaks when a Lobby object is destroyed. However, I didn't define a copy constructor or overload the assignment operator in the class. For the Game Lobby program, this isn't necessary. But if I wanted a more robust Lobby class, I would have defined these member functions.

The Lobby::AddPlayer() Member Function

The `Lobby::AddPlayer()` member function adds a player to the end of the line in the lobby.

```
void Lobby::AddPlayer()
{
    //create a new player node
    cout << "Please enter the name of the new player: ";
    string name;
    cin >> name;
    Player* pNewPlayer = new Player(name);

    //if list is empty, make head of list this new player
    if (m_pHead == 0)
    {
        m_pHead = pNewPlayer;
    }
    //otherwise find the end of the list and add the player there
    else
    {
        Player* pIter = m_pHead;
        while (pIter->GetNext() != 0)
        {
            pIter = pIter->GetNext();
        }
        pIter->SetNext(pNewPlayer);
    }
}
```

First, the function gets the new player's name from the user and uses it to instantiate a new `Player` object on the heap. Then it sets the object's pointer data member to the null pointer.

Next, the function checks to see whether the lobby is empty. If the Lobby object's data member `m_pHead` is 0, then there's no one in line. If so, the new `Player` object becomes the head of the line and `m_pHead` is set to point to a new `Player` object on the heap.

If the lobby isn't empty, the player is added to the end of the line. The function accomplishes this by moving through the list one node at a time, using `pIter`'s `GetNext()`

member function, until it reaches a `Player` object whose `GetNext()` returns 0, meaning that it's the last node in the list. Then, the function makes that node point to the new `Player` object on the heap, which has the effect of adding the new object to the end of the list. Figure 9.14 illustrates this process.

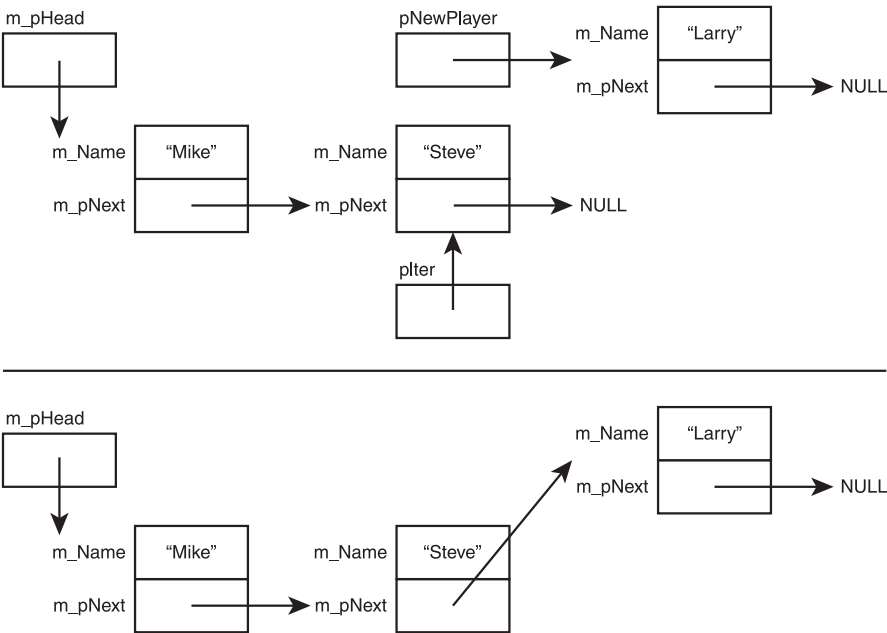


Figure 9.14
The list of players just before and just after a new player node is added.

Trap

`Lobby::AddPlayer()` marches through the entire list of `Player` objects every time it's called. For small lists, this isn't a problem, but with large lists, this inefficient process can become unwieldy. There are more efficient ways to do what this function does. In one of the chapter exercises, your job will be to implement one of these more efficient methods.

The Lobby::RemovePlayer() Member Function

The `Lobby::RemovePlayer()` member function removes the player at the head of the line.

```
void Lobby::RemovePlayer()
{
    if (m_pHead == 0)
    {
        cout << "The game lobby is empty. No one to remove!\n";
    }
}
```



```

else
{
    Player* pTemp = m_pHead;
    m_pHead = m_pHead->GetNext();
    delete pTemp;
}
}

```

The function tests `m_pHead`. If it's 0, then the lobby is empty and the function displays a message that says so. Otherwise, the first player object in the list is removed. The function accomplishes this by creating a pointer, `pTemp`, and pointing it to the first `Player` object in the list. Then the function sets `m_pHead` to the next thing in the list—either the next `Player` object or 0. Finally, the function destroys the `Player` object pointed to by `pTemp`. Check out Figure 9.15 for a visual representation of how this works.

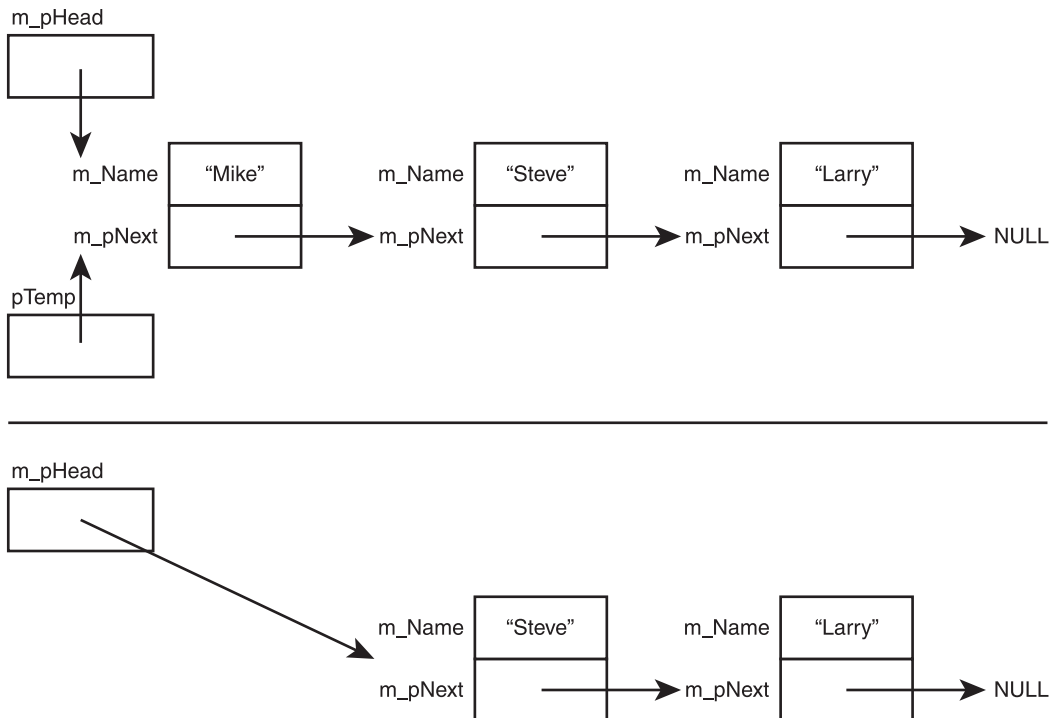


Figure 9.15
The list of players just before and just after a player node is removed.

The Lobby::Clear() Member Function

The `Lobby::Clear()` member function removes all of the players from the lobby.

```
void Lobby::Clear()
{
    while (m_pHead != 0)
    {
        RemovePlayer();
    }
}
```

If the list is empty, the loop isn't entered and the function ends. Otherwise, the loop is entered and the function keeps removing the first `Player` object in the list by calling `RemovePlayer()` until there are no more `Player` objects.

The operator<<() Member Function

The `operator<<()` member function overloads the `<<` operator so I can display a `Lobby` object by sending it to `cout`.

```
ostream& operator<<(ostream& os, const Lobby& aLobby)
{
    Player* pIter = aLobby.m_pHead;
    os << "\nHere's who's in the game lobby:\n";
    if (pIter == 0)
    {
        os << "The lobby is empty.\n";
    }
    else
    {
        while (pIter != 0)
        {
            os << pIter->GetName() << endl;
            pIter = pIter->GetNext();
        }
    }
    return os;
}
```

If the lobby is empty, the appropriate message is sent to the output stream. Otherwise, the function cycles through all of the players in the list, sending their names to the output stream, using `pIter` to move through the list.

The main() Function

The `main()` function displays the players in the lobby, presents the user with a menu of choices, and performs the requested action.

```
int main()
{
    Lobby myLobby;
    int choice;

    do
    {
        cout << myLobby;
        cout << "\nGAME LOBBY\n";
        cout << "0 - Exit the program.\n";
        cout << "1 - Add a player to the lobby.\n";
        cout << "2 - Remove a player from the lobby.\n";
        cout << "3 - Clear the lobby.\n";
        cout << endl << "Enter choice: ";
        cin >> choice;

        switch (choice)
        {
            case 0: cout << "Good-bye.\n"; break;
            case 1: myLobby.AddPlayer(); break;
            case 2: myLobby.RemovePlayer(); break;
            case 3: myLobby.Clear(); break;
            default: cout << "That was not a valid choice.\n";
        }
    }
    while (choice != 0);

    return 0;
}
```

The function first instantiates a new `Lobby` object, and then it enters a loop that presents a menu and gets the user's choice. Then it calls the corresponding `Lobby` object's member function. If the user enters an invalid choice, he or she is told so. The loop continues until the user enters 0.

SUMMARY

In this chapter, you should have learned the following concepts:

- Aggregation is the combining of objects so that one is part of another.
- Friend functions have complete access to any member of a class.

- Operator overloading allows you to define new meanings for built-in operators as they relate to objects of your own classes.
- The stack is an area of memory that is automatically managed for you and is used for local variables.
- The heap (or free store) is an area of memory that you, the programmer, can use to allocate and free memory.
- The `new` operator allocates memory on the heap and returns its address.
- The `delete` operator frees memory on the heap that was previously allocated.
- A dangling pointer points to an invalid memory location. Dereferencing or deleting a dangling pointer can cause your program to crash.
- A memory leak is an error in which memory that has been allocated becomes inaccessible and can no longer be freed. Given a large enough leak, a program might run out of memory and crash.
- A destructor is a member function that's called just before an object is destroyed. If you don't write a destructor of your own, the compiler will supply a default destructor for you.
- The copy constructor is a member function that's invoked when an automatic copy of an object is made. A default copy constructor is supplied for a class if you don't write one of your own.
- The default copy constructor simply copies the value of each data member to data members with the same names in the copy, producing a member-wise copy.
- Member-wise copying can produce a shallow copy of an object, in which the pointer data members of the copy point to the same chunks of memory as the pointers in the original object.
- A deep copy is a copy of an object that has no chunks of memory in common with the original.
- A default assignment operator member function, which provides only member-wise duplication, is supplied for you if you don't write one of your own.
- The `this` pointer is a pointer that all non-static member functions automatically have; it points to the object that was used to call the function.

QUESTIONS AND ANSWERS

Q: Why should you use aggregation?

A: To create more complex objects from other objects.

Q: What is composition?

A: A form of aggregation in which the composite object is responsible for the creation and destruction of its object parts. Composition is often called a *uses-a* relationship.

Q: When should I use a friend function?

A: When you need a function to have access to the non-public members of a class.

Q: What is a friend member function?

A: A member function of one class that can access all of the members of another class.

Q: What is a friend class?

A: A class that can access all of the members of another class.

Q: Can't operator overloading become confusing?

A: Yes. Giving too many meanings or unintuitive meanings to operators can lead to code that's difficult to understand.

Q: What happens when I instantiate a new object on the heap?

A: All of the data members will occupy memory on the heap and not on the stack.

Q: Can I access an object through a constant pointer?

A: Sure. But you can only access constant member functions through a constant pointer.

Q: What's wrong with shallow copies?

A: Because shallow copies share references to the same chunks of memory, a change to one object will be reflected in another object.

Q: What is a linked list?

A: A dynamic data structure that consists of a sequence of linked nodes.

Q: How is a linked list different from a vector?

A: Linked lists permit insertion and removal of nodes at any point in the list but do not allow random access, like vectors. However, the insertion and deletion of nodes in the middle of the list can be more efficient than the insertion and deletion of elements in the middle of vectors.

Q: Is there a container class from the STL that serves as a linked list?

A: Yes, the `list` class.

Q: Is the data structure used in the Game Lobby program a linked list?

A: It shares similarities to a linked list, but it is really a queue.

Q: What's a queue?

A: A data structure in which elements are removed in the same order in which they were entered. This process is often called first in, first out (FIFO).

Q: Is there a kind of container from the STL that serves as a queue?

A: Yes, the `queue` container adaptor.

DISCUSSION QUESTIONS

1. What types of game entities could you create with aggregation?
2. Do friend functions undermine encapsulation in OOP?
3. What advantages does dynamic memory offer to game programs?
4. Why are memory leaks difficult errors to track down?
5. Should objects that allocate memory on the heap always be required to free it?

EXERCISES

1. Improve the `Lobby` class from the Game Lobby program by writing a friend function of the `Player` class that allows a `Player` object to be sent to `cout`. Next, update the function that allows a `Lobby` object to be sent to `cout` so that it uses your new function for sending a `Player` object to `cout`.
2. The `Lobby::AddPlayer()` member function from the Game Lobby program is inefficient because it iterates through all of the player nodes to add a new player to the end of the line. Add an `m_pTail` pointer data member to the `Lobby` class that always points to the last player node in the line and use it to more efficiently add a player.
3. What's wrong with the following code?

```
#include <iostream>
using namespace std;

int main()
{
    int* pScore = new int;
    *pScore = 500;
    pScore = new int(1000);
    delete pScore;
    pScore = 0;

    return 0;
}
```