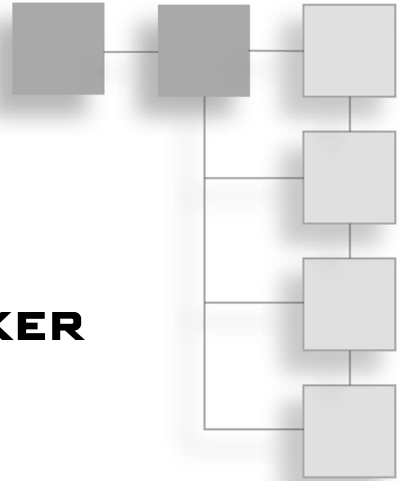


CHAPTER 8



CLASSES: CRITTER CARETAKER

Object-oriented programming (OOP) is a different way of thinking about programming. It's a modern methodology used in the creation of the vast majority of games (and other commercial software, too). In OOP, you define different types of objects with relationships to each other that allow the objects to interact. You've already worked with objects from types defined in libraries, but one of the key characteristics of OOP is the ability to make your own types from which you can create objects. In this chapter, you'll see how to define your own types and create objects from them. Specifically, you'll learn to:

- Create new types by defining classes
- Declare class data members and member functions
- Instantiate objects from classes
- Set member access levels
- Declare static data members and member functions

DEFINING NEW TYPES

Whether you're talking about alien spacecrafts, poisonous arrows, or angry mutant chickens, games are full of objects. Fortunately, C++ lets you represent game entities as software objects, complete with member functions and data members. These objects work just like the ones you've already seen, such as `string` and `vector` objects. But to use a new kind of object (say, an angry mutant chicken object), you must first define a type for it.

Introducing the Simple Critter Program

The Simple Critter program defines a brand-new type called `Critter` for creating virtual pet objects. The program uses this new type to create two `Critter` objects. Then, it gives each critter a hunger level. Finally, each critter offers a greeting and announces its hunger level to the world. Figure 8.1 shows the results of the program.



Figure 8.1

Each critter says hi and announces how hungry it is.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website. The program is in the Chapter 8 folder; the filename is `simple_critter.cpp`.

```
//Simple Critter
//Demonstrates creating a new type
#include <iostream>
using namespace std;

class Critter           // class definition -- defines a new type, Critter
{
public:
    int m_Hunger;        // data member
    void Greet();        // member function prototype
};
```

```

void Critter::Greet()    // member function definition
{
    cout << "Hi. I'm a critter. My hunger level is " << m_Hunger << ".\n";
}

int main()
{
    Critter crit1;
    Critter crit2;

    crit1.m_Hunger = 9;
    cout << "crit1's hunger level is " << crit1.m_Hunger << ".\n";
    crit2.m_Hunger = 3;
    cout << "crit2's hunger level is " << crit2.m_Hunger << ".\n\n";

    crit1.Greet();
    crit2.Greet();

    return 0;
}

```

Defining a Class

To create a new type, you can define a *class*—code that groups data members and member functions. From a class, you create individual objects that have their own copies of each data member and access to all of the member functions. A class is like a blueprint. Just as a blueprint defines the structure of a building, a class defines the structure of an object. And just as a foreman can create many houses from the same blueprint, a game programmer can create many objects from the same class. Some real code will help solidify this theory. I begin a class definition in the Simple Critter program with

```

class Critter            // class definition -- defines a new type, Critter

```

for a class named `Critter`. To define a class, start with the keyword `class`, followed by the class name. By convention, class names begin with an uppercase letter. You surround the class body with curly braces and end it with a semicolon.

Declaring Data Members

In a class definition, you can declare class data members to represent object qualities. I give the critters just one quality: hunger. I see hunger as a range that could be represented by an integer, so I declare an `int` data member `m_Hunger`.

```

    int m_Hunger;        // data member

```

This means that every `Critter` object will have its own hunger level, represented by its own data member named `m_Hunger`. Notice that I prefix the data member name with `m_`. Some game programmers follow this naming convention so that data members are instantly recognizable.

Declaring Member Functions

In a class definition, you can also declare member functions to represent object abilities. I give a critter just one—the ability to greet the world and announce its hunger level—by declaring the member function `Greet()`.

```
void Greet();           // member function prototype
```

This means that every `Critter` object will have the ability to say hi and announce its own hunger level through its member function, `Greet()`. By convention, member function names begin with an uppercase letter. At this point, I've only declared the member function `Greet()`. Don't worry, though, I'll define it outside of the class.

Hint

You might have noticed the keyword `public` in the class definition. You can ignore it for now. You'll learn more about it a bit later in this chapter, in the section, "Specifying Public and Private Access Levels."

Defining Member Functions

You can define member functions outside of a class definition. Outside of the `Critter` class definition, I define the `Critter` member function `Greet()`, which says hi and displays the critter's hunger level.

```
void Critter::Greet()    // member function definition
{
    cout << "Hi. I'm a critter. My hunger level is " << m_Hunger << ".\n";
}
```

The definition looks like any other function definition you've seen, except for one thing—I prefix the function name with `Critter::`. When you define a member function outside of its class, you need to qualify it with the class name and scope resolution operator so the compiler knows that the definition belongs to the class.

In the member function, I send `m_Hunger` to `cout`. This means that `Greet()` displays the value of `m_Hunger` for the specific object through which the function is called. This simply means that the member function displays the critter's hunger level. You can access the data members and member functions of an object in any member function simply by using the member's name.

Instantiating Objects

When you create an object, you *instantiate* it from a class. In fact, specific objects are called *instances* of the class. In `main()`, I instantiate two instances of `Critter`.

```
Critter crit1;
Critter crit2;
```

As a result, I have two `Critter` objects: `crit1` and `crit2`.

Accessing Data Members

It's time to put these critters to work. Next, I give my first critter a hunger level.

```
crit1.m_Hunger = 9;
```

The preceding code assigns 9 to `crit1`'s data member `m_Hunger`. Just like when you access an available member function of an object, you can access an available data member of an object using the member selection operator.

To prove that the assignment worked, I display the critter's hunger level.

```
cout << "crit1's hunger level is " << crit1.m_Hunger << ".\n";
```

The preceding code displays `crit1`'s data member `m_Hunger` and correctly shows 9. Just like when you assign a value to an available data member, you can get the value of an available data member through the member selection operator.

Next, I show that the same process works for another `Critter` object.

```
crit2.m_Hunger = 3;
cout << "crit2's hunger level is " << crit2.m_Hunger << ".\n\n";
```

This time, I assign 3 to `crit2`'s data member `m_Hunger` and display it.

So, `crit1` and `crit2` are both instances of `Critter`, yet each exists independently and each has its own identity. Also, each has its own `m_Hunger` data member with its own value.

Calling Member Functions

Next, I again put the critters through their paces. I get the first critter to give a greeting.

```
crit1.Greet();
```

The preceding code calls `crit1`'s `Greet()` member function. The function accesses the calling object's `m_Hunger` data member to form the greeting it displays. Because `crit1`'s `m_Hunger` data member is 9, the function displays the text: Hi. I'm a critter. My hunger level is 9.

Finally, I get the second critter to speak up.

```
crit2.Greet();
```

The preceding code calls `crit2`'s `Greet()` member function. This function accesses the calling object's `m_Hunger` data member to form the greeting it displays. Because `crit2`'s `m_Hunger` data member is 3, the function displays the text: `Hi. I'm a critter. My hunger level is 3.`

USING CONSTRUCTORS

When you instantiate objects, you often want to do some initialization—usually assigning values to data members. Luckily, a class has a special member function known as a *constructor* that is called automatically every time a new object is instantiated. This is a big convenience because you can use a constructor to perform initialization of the new object.

Introducing the Constructor Critter Program

The Constructor Critter program demonstrates constructors. The program instantiates a new `Critter` object, which automatically invokes its constructor. First, the constructor announces that a new critter has been born. Then, it assigns the value passed to it to the critter's hunger level. Finally, the program calls the critter's greeting member function, which displays the critter's hunger level, proving that the constructor did in fact initialize the critter. Figure 8.2 shows the results of the program.



Figure 8.2

The Critter constructor initializes a new object's hunger level automatically.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website. The program is in the Chapter 8 folder; the filename is `constructor_critter.cpp`.

```
//Constructor Critter
//Demonstrates constructors

#include <iostream>

using namespace std;

class Critter
{
public:
    int m_Hunger;

    Critter(int hunger = 0);      // constructor prototype
    void Greet();

};

Critter::Critter(int hunger)    // constructor definition
{
    cout << "A new critter has been born!" << endl;
    m_Hunger = hunger;
}

void Critter::Greet()
{
    cout << "Hi. I'm a critter. My hunger level is " << m_Hunger << ".\n\n";
}

int main()
{
    Critter crit(7);
    crit.Greet();

    return 0;
}
```

Declaring and Defining a Constructor

I declare a constructor in `Critter` with the following code:

```
Critter(int hunger = 0);      // constructor prototype
```

As you can see from the declaration, the constructor has no return type. It can't; it's illegal to specify a return type for a constructor. Also, you have no flexibility when naming a constructor. You have to give it the same name as the class itself.

Hint

A *default constructor* requires no arguments. If you don't define a default constructor, the compiler defines a minimal one for you that simply calls the default constructors of any data members of the class. If you write your own constructor, then the compiler won't provide a default constructor for you. It's usually a good idea to have a default constructor, so you should make sure to supply your own when necessary. One way to accomplish this is to supply default arguments for all parameters in a constructor definition.

I define the constructor outside of the class with the following code:

```
Critter::Critter(int hunger)           // constructor definition
{
    cout << "A new critter has been born!" << endl;
    m_Hunger = hunger;
}
```

The constructor displays a message saying that a new critter has been born and initializes the object's `m_Hunger` data member with the argument value passed to the constructor. If no value is passed, then the constructor uses the default argument value of 0.

Trick

You can use *member initializers* as a shorthand way to assign values to data members in a constructor. To write a member initializer, start with a colon after the constructor's parameter list. Then type the name of the data member you want to initialize, followed by the expression you want to assign to the data member, surrounded by parentheses. If you have multiple initializers, separate them with commas. This is much simpler than it sounds (and it's really useful, too). Here's an example that assigns hunger to `m_Hunger` and boredom to `m_Boredom`. Member initializers are especially useful when you have many data members to initialize.

```
Critter::Critter(int hunger = 0, int boredom = 0):
    m_Hunger(hunger),
    m_Boredom(boredom)
{} // empty constructor body
```

Calling a Constructor Automatically

You don't explicitly call a constructor; however, whenever you instantiate a new object, its constructor is automatically called. In `main()`, I put my constructor into action with the following code:

```
Critter crit(7);
```


When `crit` is instantiated, its constructor is automatically called and the message `A new critter has been born!` is displayed. Then, the constructor assigns 7 to the object's `m_Hunger` data member.

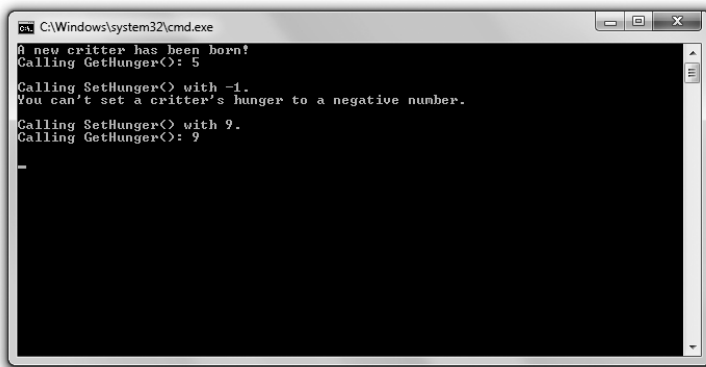
To prove that the constructor worked, back in `main()`, I call the object's `Greet()` member function and sure enough, it displays `Hi. I'm a critter. My hunger level is 7.`

SETTING MEMBER ACCESS LEVELS

Like functions, you should treat objects as encapsulated entities. This means that, in general, you should avoid directly altering or accessing an object's data members. Instead, you should call an object's member functions, allowing the object to maintain its own data members and ensure their integrity. Fortunately, you can enforce data member restrictions when you define a class by setting member access levels.

Introducing the Private Critter Program

The Private Critter program demonstrates class member access levels by declaring a class for critters that restricts direct access to an object's data member for its hunger level. The class provides two member functions—one that allows access to the data member and one that allows changes to the data member. The program creates a new critter and indirectly accesses and changes the critter's hunger level through these member functions. However, when the program attempts to change the critter's hunger level to an illegal value, the member function that allows the changes catches the illegal value and doesn't make the change. Finally, the program uses the hunger-level-setting member function with a legal value, which works like a charm. Figure 8.3 shows the results of the program.



```
C:\Windows\system32\cmd.exe
A new critter has been born!
Calling GetHunger(): 5
Calling SetHunger() with -1.
You can't set a critter's hunger to a negative number.
Calling SetHunger() with 9.
Calling GetHunger(): 9
```

Figure 8.3

By using a `Critter` object's `GetHunger()` and `SetHunger()` member functions, the program indirectly accesses an object's `m_Hunger` data member.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website. The program is in the Chapter 8 folder; the filename is `private_critter.cpp`.

```
//Private Critter
//Demonstrates setting member access levels
#include <iostream>
using namespace std;
class Critter
{
public:          // begin public section
    Critter(int hunger = 0);
    int GetHunger() const;
    void SetHunger(int hunger);
private:       // begin private section
    int m_Hunger;
};
Critter::Critter(int hunger):
    m_Hunger(hunger)
{
    cout << "A new critter has been born!" << endl;
}
int Critter::GetHunger() const
{
    return m_Hunger;
}
void Critter::SetHunger(int hunger)
{
    if (hunger < 0)
    {
        cout << "You can't set a critter's hunger to a negative number.\n\n";
    }
    else
    {
        m_Hunger = hunger;
    }
}
```

```

int main()
{
    Critter crit(5);
    //cout << crit.m_Hunger; //illegal, m_Hunger is private!
    cout << "Calling GetHunger(): " << crit.GetHunger() << "\n\n";

    cout << "Calling SetHunger() with -1.\n";
    crit.SetHunger(-1);

    cout << "Calling SetHunger() with 9.\n";
    crit.SetHunger(9);
    cout << "Calling GetHunger(): " << crit.GetHunger() << "\n\n";

    return 0;
}

```

Specifying Public and Private Access Levels

Every class data member and member function has an access level, which determines where in your program you can access it. So far, I've always specified class members to have public access levels using the keyword `public`. Again, in `Critter`, I start a public section with the following line:

```
public:           // begin public section
```

By using `public:`, I'm saying that any data member or member function that follows (until another access level specifier) will be public. This means that any part of the program can access them. Because I declare all of the member functions in this section, it means that any part of my code can call any member function through a `Critter` object.

Next, I specify a private section with the following line:

```
private:         // begin private section
```

By using `private:`, I'm saying that any data member or member function that follows (until another access level specifier) will be private. This means that only code in the `Critter` class can directly access it. Since I declare `m_Hunger` in this section, it means that only the code in `Critter` can directly access an object's `m_Hunger` data member. Therefore, I can't directly access an object's `m_Hunger` data member through the object in `main()` as I've done in previous programs. So the following line in `main()`, if uncommented, would be an illegal statement:

```
//cout << crit.m_Hunger; //illegal, m_Hunger is private!
```

Because `m_Hunger` is `private`, I can't access it from code that is not part of the `Critter` class. Again, only code that's part of `Critter` can directly access the data member.

I've only shown you how to make data members `private`, but you can make member functions `private`, too. Also, you can repeat access modifiers. So if you want, you could have a `private` section, followed by a `public` section, followed by another `private` section in a class. Finally, member access is `private` by default. Until you specify an access modifier, any class members you declare will be `private`.

Defining Accessor Member Functions

An *accessor member function* allows indirect access to a data member. Because `m_Hunger` is `private`, I wrote an accessor member function, `GetHunger()`, to return the value of the data member. (For now, you can ignore the keyword `const`.)

```
int Critter::GetHunger() const
{
    return m_Hunger;
}
```

I put the member function to work in `main()` with the following line:

```
cout << "Calling GetHunger(): " << crit.GetHunger() << "\n\n";
```

In the preceding code, `crit.GetHunger()` simply returns the value of `crit`'s `m_Hunger` data member, which is 5.

Trick

Just as you can with regular functions, you can inline member functions. One way to inline a member function is to define it right inside of the class definition, where you'd normally only declare the member function. If you include a member function definition in a class, then of course you don't need to define it outside of the class.

An exception to this rule is that when you define a member function in a class definition using the keyword `virtual`, the member function is not automatically inlined. You'll learn about virtual functions in Chapter 10, "Inheritance and Polymorphism: Blackjack."

At this point, you might be wondering why you'd go to the trouble of making a data member `private` only to grant full access to it through accessor functions. The answer is that you don't generally grant full access. For example, take a look at the accessor member function I defined for setting an object's `m_Hunger` data member, `SetHunger()`:

```
void Critter::SetHunger(int hunger)
{
    if (hunger < 0)
```

```

{
    cout << "You can't set a critter's hunger to a negative number.\n\n";
}
else
{
    m_Hunger = hunger;
}
}

```

In this accessor member function, I first check to make sure that the value passed to the member function is greater than zero. If it's not, it's an illegal value, and I display a message, leaving the data member unchanged. If the value is greater than zero, then I make the change. This way, `SetHunger()` protects the integrity of `m_Hunger`, ensuring that it can't be set to a negative number. Just as I've done here, most game programmers begin their accessor member function names with `Get` or `Set`.

Defining Constant Member Functions

A *constant member function* can't modify a data member of its class or call a non-constant member function of its class. Why restrict what a member function can do? Again, it goes back to the tenet of asking only for what you need. If you don't need to change any data members in a member function, then it's a good idea to declare that member function to be constant. It protects you from accidentally altering a data member in the member function, and it makes your intentions clear to other programmers.

Trap

Okay, I lied a little. A constant member function can alter a static data member. You'll learn about static data members a bit later in this chapter, in the "Declaring and Initializing Static Data Members" section. Also, if you qualify a data member with the `mutable` keyword, then even a constant member function can modify it. For now, though, don't worry about either of these exceptions.

You can declare a constant member function by putting the keyword `const` at the end of the function header. That's what I do in `Critter` with the following line, which declares `GetHunger()` to be a constant member function.

```
int GetHunger() const;
```

This means that `GetHunger()` can't change the value of any non-static data member declared in the `Critter` class, nor can it call any non-constant `Critter` member function. I made `GetHunger()` constant because it only returns a value and doesn't need to modify any data member. Generally, `Get` member functions can be defined as constant.

USING STATIC DATA MEMBERS AND MEMBER FUNCTIONS

Objects are great because each instance stores its own set of data, giving it a unique identity. But what if you want to store some information about an entire class, such as the total number of instances that exist? You might want to do this if you've created a bunch of enemies and you want them to fight the player based on their total number. For example, if their total number is below a certain threshold, you might want the enemies to run away. You could store the total number of instances in each object, but that would be a waste of storage space. Plus, it would be cumbersome to update all of the objects as the total changes. Instead, what you really want is a way to store a single value for an entire class. You can do this with a static data member.

Introducing the Static Critter Program

The Static Critter program declares a new kind of critter with a static data member that stores the total number of critters that have been created. It also defines a static member function that displays the total. Before the program instantiates any new `Critter` objects, it displays the total number of critters by directly accessing the static data member that holds the total. Next, the program instantiates three new critters. Then it displays the total number of critters by calling a static member function that accesses the static data member. Figure 8.4 shows the results of the program.



```
C:\Windows\system32\cmd.exe
The total number of critters is: 0
A critter has been born!
A critter has been born!
A critter has been born!
The total number of critters is: 3
-
```

Figure 8.4

The program stores the total number of `Critter` objects in the static data member `s_Total` and accesses that data member in two different ways.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website. The program is in the Chapter 8 folder; the filename is `static_critter.cpp`.

```
//Static Critter
//Demonstrates static member variables and functions
#include <iostream>
using namespace std;
class Critter
{
public:
    static int s_Total;    //static member variable declaration
                        //total number of Critter objects in existence

    Critter(int hunger = 0);
    static int GetTotal(); //static member function prototype
private:
    int m_Hunger;
};

int Critter::s_Total = 0; //static member variable initialization
Critter::Critter(int hunger):
    m_Hunger(hunger)
{
    cout << "A critter has been born!" << endl;
    ++s_Total;
}

int Critter::GetTotal()    //static member function definition
{
    return s_Total;
}

int main()
{
    cout << "The total number of critters is: ";
    cout << Critter::s_Total << "\n\n";
    Critter crit1, crit2, crit3;

    cout << "\nThe total number of critters is: ";
    cout << Critter::GetTotal() << "\n";

    return 0;
}
```

Declaring and Initializing Static Data Members

A *static data member* is a single data member that exists for the entire class. In the class definition, I declare a static data member `s_Total` to store the number of `Critter` objects that have been instantiated.

```
static int s_Total;    //static member variable declaration
```

You can declare your own static data members just as I did, by starting the declaration with the `static` keyword. I prefixed the variable name with `s_` so it would be instantly recognizable as a static data member.

Outside of the class definition, I initialize the static data member to 0.

```
int Critter::s_Total = 0;    //static member variable initialization
```

Notice that I qualified the data member name with `Critter::`. Outside of its class definition, you must qualify a static data member with its class name. After the previous line of code executes, there is a single value associated with the `Critter` class, stored in its static data member `s_Total` with a value of 0.

Hint

You can declare a static variable in non-class functions, too. The static variable maintains its value between function calls.

Accessing Static Data Members

You can access a public static data member anywhere in your program. In `main()`, I access `Critter::s_Total` with the following line, which displays 0, the value of the static data member and the total number of `Critter` objects that have been instantiated.

```
cout << Critter::s_Total << "\n\n";
```

Hint

You can also access a static data member through any object of the class. Assuming that `crit1` is a `Critter` object, I could display the total number of critters with the following line:

```
cout << crit1.s_Total << "\n\n";
```

I also access this static data member in the `Critter` constructor with the following line, which increments `s_Total`.

```
++s_Total;
```


This means that every time a new object is instantiated, `s_Total` is incremented. Notice that I didn't qualify `s_Total` with `Critter::`. Just as with non-static data members, you don't have to qualify a static data member with its class name inside its class.

Although I made my static data member public, you can make a static data member private—but then, like any other data member, you can only access it in a class member function.

Declaring and Defining Static Member Functions

A *static member function* exists for the entire class. I declare a static member function in `Critter` with the following line:

```
static int GetTotal(); //static member function prototype
```

You can declare your own static member function as I did, by starting the declaration with the keyword `static`. Static member functions are often written to work with static data members.

I define the static member function `GetTotal()` that returns the value of the static data member `s_Total`.

```
int Critter::GetTotal()    //static member function definition
{
    return s_Total;
}
```

A static member function definition is much like the non-static member function definitions you've seen so far. The major difference is that a static member function cannot access non-static data members. This is because a static member function exists for the entire class and is not associated with any particular instance of the class.

Calling Static Member Functions

After I instantiate three `Critter` objects in `main()`, I reveal the total number of critters again with the following line, which displays 3.

```
cout << Critter::GetTotal() << "\n\n";
```

To properly identify the static member function, I had to qualify it with `Critter::`. To call a static member function from outside of its class, you must qualify it with its class name.

Hint

You can also access a static member function through any object of the class. Assuming that `crit1` is a `Critter` object, I could display the total number of critters with the following line:

```
cout << crit1.GetTotal() << "\n\n";
```

Because static member functions exist for the entire class, you can call a static member function without any instances of the class in existence. And just as with private static data members, private static member functions can only be accessed by other member functions of the same class.

INTRODUCING THE CRITTER CARETAKER GAME

The Critter Caretaker game puts the player in charge of his own virtual pet. The player is completely responsible for keeping the critter happy, which is no small task. He can feed and play with the critter to keep it in a good mood. He can also listen to the critter to learn how the critter is feeling, which can range from happy to mad. Figure 8.5 shows off the game.

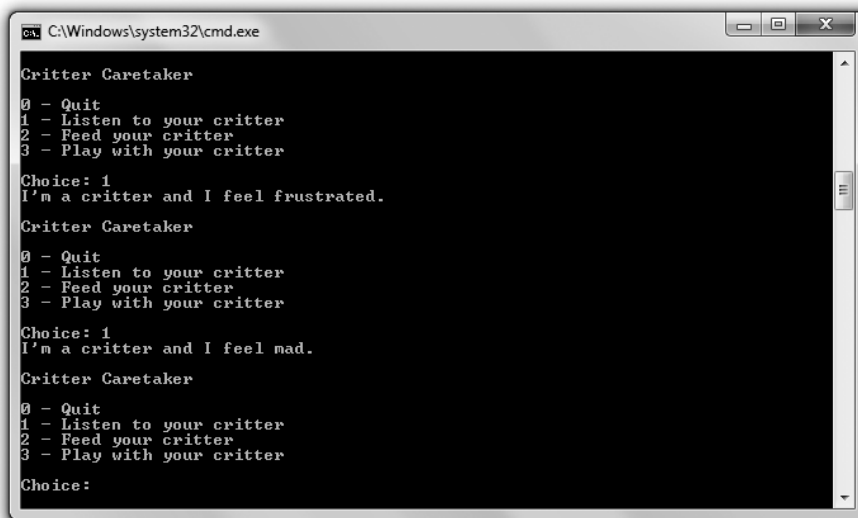


Figure 8.5

If you fail to feed or entertain your critter, it will have a mood change for the worse. (But don't worry—with the proper care, your critter can return to a sunny mood.)

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website. The program is in the Chapter 8 folder; the filename is `critter_caretaker.cpp`.

Planning the Game

The core of the game is the critter itself. Therefore, I first plan my `Critter` class. Because I want the critter to have independent hunger and boredom levels, I know that the class will have private data members for those.

- `m_Hunger`
- `m_Boredom`

The critter should also have a mood, directly based on its hunger and boredom levels. My first thought was to have a private data member, but a critter's mood is really a calculated value based on its hunger and boredom. Instead, I decided to have a private member function that calculates a critter's mood on the fly, based on its current hunger and boredom levels:

- `GetMood()`

Next, I think about public member functions. I want the critter to be able to tell the player how it's feeling. I also want the player to be able to feed and play with the critter to reduce its hunger and boredom levels. I need three public member functions to accomplish each of these tasks.

- `Talk()`
- `Eat()`
- `Play()`

Finally, I want another member function that simulates the passage of time, to make the critter a little more hungry and bored:

- `PassTime()`

I see this member function as private because it will only be called by other member functions, such as `Talk()`, `Eat()`, or `Play()`.

The class will also have a constructor to initialize data members. Take a look at Figure 8.6, which models the `Critter` class. I preface each data member and member function with a symbol to indicate its access level; I use `+` for public and `-` for private.

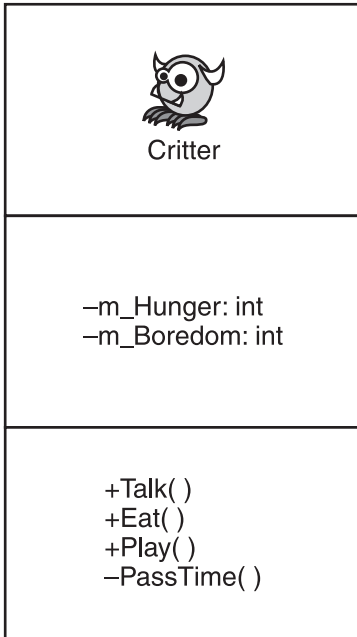


Figure 8.6
Model for the Critter class

Planning the Pseudocode

The rest of the program will be pretty simple. It'll basically be a game loop that asks the player whether he wants to listen to, feed, or play with the critter, or quit the game. Here's the pseudocode I came up with:

Create a critter

While the player doesn't want to quit the game

 Present a menu of choices to the player

 If the player wants to listen to the critter

 Make the critter talk

 If the player wants to feed the critter

 Make the critter eat

 If the player wants to play with the critter

 Make the critter play

The Critter Class

The Critter class is the blueprint for the object that represents the player's critter. The class isn't complicated, and most of it should look familiar, but it's long enough that it makes sense to attack it in pieces.

The Class Definition

After some initial comments and statements, I begin the Critter class.

```
//Critter Caretaker
//Simulates caring for a virtual pet

#include <iostream>

using namespace std;

class Critter
{
public:
    Critter(int hunger = 0, int boredom = 0);
    void Talk();
    void Eat(int food = 4);
    void Play(int fun = 4);

private:
    int m_Hunger;
    int m_Boredom;

    int GetMood() const;
    void PassTime(int time = 1);
};
```

m_Hunger is a private data member that represents the critter's hunger level, while m_Boredom is a private data member that represents its boredom level. I'll go through each member function in its own section.

The Class Constructor

The constructor takes two arguments: hunger and boredom. The arguments each have a default value of zero, which I specified in the constructor prototype back in the class definition. I use hunger to initialize m_Hunger and boredom to initialize m_Boredom.

```
Critter::Critter(int hunger, int boredom):
    m_Hunger(hunger),
    m_Boredom(boredom)
{ }
```

The GetMood() Member Function

Next, I define GetMood().

```
inline int Critter::GetMood() const
{
    return (m_Hunger + m_Boredom);
}
```

The return value of this inlined member function represents a critter's mood. As the sum of a critter's hunger and boredom levels, a critter's mood gets worse as the number increases. I made this member function private because it should only be invoked by another member function of the class. I made it constant since it won't result in any changes to data members.

The PassTime() Member Function

PassTime() is a private member function that increases a critter's hunger and boredom levels. It's invoked at the end of each member function where the critter does something (eats, plays, or talks) to simulate the passage of time. I made this member function private because it should only be invoked by another member function of the class.

```
void Critter::PassTime(int time)
{
    m_Hunger += time;
    m_Boredom += time;
}
```

You can pass the member function the amount of time that has passed; otherwise, time gets the default argument value of 1, which I specify in the member function prototype in the Critter class definition.

The Talk() Member Function

The Talk() member function announces the critter's mood, which can be happy, okay, frustrated, or mad. Talk() calls GetMood() and, based on the return value, displays the appropriate message to indicate the critter's mood. Finally, Talk() calls PassTime() to simulate the passage of time.

```
void Critter::Talk()
{
    cout << "I'm a critter and I feel ";
    int mood = GetMood();
```

```

if (mood > 15)
{
    cout << "mad.\n";
}
else if (mood > 10)
{
    cout << "frustrated.\n";
}
else if (mood > 5)
{
    cout << "okay.\n";
}
else
{
    cout << "happy.\n";
}
PassTime();
}

```

The Eat() Member Function

Eat() reduces a critter's hunger level by the amount passed to the parameter food. If no value is passed, food gets the default argument value of 4. The critter's hunger level is kept in check and is not allowed to go below zero. Finally, PassTime() is called to simulate the passage of time.

```

void Critter::Eat(int food)
{
    cout << "Brruppp.\n";
    m_Hunger -= food;
    if (m_Hunger < 0)
    {
        m_Hunger = 0;
    }
    PassTime();
}

```

The Play() Member Function

Play() reduces a critter's boredom level by the amount passed to the parameter fun. If no value is passed, fun gets the default argument value of 4. The critter's boredom level is kept in check and is not allowed to go below zero. Finally, PassTime() is called to simulate the passage of time.

```

void Critter::Play(int fun)
{
    cout << "Wheee!\n";
    m_Boredom -= fun;
    if (m_Boredom < 0)
    {
        m_Boredom = 0;
    }
    PassTime();
}

```

The main() Function

In `main()`, I instantiate a new `Critter` object. Because I don't supply values for `m_Hunger` or `m_Boredom`, the data members start out at 0, and the critter begins life happy and content. Next, I create a menu system. If the player enters 0, the program ends. If the player enters 1, the program calls the object's `Talk()` member function. If the player enters 2, the program calls the object's `Eat()` member function. If the player enters 3, the program calls the object's `Play()` member function. If the player enters anything else, he is told that the choice is invalid.

```

int main()
{
    Critter crit;
    crit.Talk();

    int choice;
    do
    {
        cout << "\nCritter Caretaker\n\n";
        cout << "0 - Quit\n";
        cout << "1 - Listen to your critter\n";
        cout << "2 - Feed your critter\n";
        cout << "3 - Play with your critter\n\n";

        cout << "Choice: ";
        cin >> choice;

        switch (choice)
        {
            case 0:
                cout << "Good-bye.\n";
                break;

```



```

        case 1:
            crit.Talk();
            break;
        case 2:
            crit.Eat();
            break;
        case 3:
            crit.Play();
            break;
        default:
            cout << "\nSorry, but " << choice << " isn't a valid choice.\n";
    }
} while (choice != 0);
return 0;
}

```

SUMMARY

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

- Object-oriented programming (OOP) is a way of thinking about programming in which you define different types of objects with relationships that interact with each other.
- You can create a new type by defining a class.
- A class is a blueprint for an object.
- In a class, you can declare data members and member functions.
- When you define a member function outside of a class definition, you need to qualify it with the class name and scope resolution operator (::).
- You can inline a member function by defining it directly in the class definition.
- You can access data members and member functions of objects through the member selection operator (.).
- Every class has a constructor—a special member function that’s automatically called every time a new object is instantiated. Constructors are often used to initialize data members.
- A default constructor requires no arguments. If you don’t provide a constructor definition in your class, the compiler will create a default constructor for you.
- Member initializers provide shorthand to assign values to data members in a constructor.

- You can set member access levels in a class by using the keywords `public`, `private`, and `protected`. (You'll learn about `protected` in Chapter 9, "Advanced Classes and Dynamic Memory: Game Lobby.")
- A public member can be accessed by any part of your code through an object.
- A private member can be accessed only by a member function of that class.
- An accessor member function allows indirect access to a data member.
- A static data member exists for the entire class.
- A static member function exists for the entire class.
- Some game programmers prefix private data member names with `m_` and static data member names with `s_` so that they're instantly recognizable.
- A constant member function can't modify non-static data members or call non-constant member functions of its class.

QUESTIONS AND ANSWERS

Q: What is procedural programming?

A: A paradigm where tasks are broken down into a series of smaller tasks and implemented in manageable chunks of code, such as functions. In procedural programming, functions and data are separate.

Q: What is an object?

A: An entity that combines data and functions.

Q: Why create objects?

A: Because the world—and especially game worlds—are full of objects. By creating your own types, you can represent objects and their relationships to other objects more directly and intuitively than you might be able to otherwise.

Q: What is object-oriented programming?

A: A paradigm where work is accomplished through objects. It allows programmers to define their own types of objects. The objects usually have relationships to each other and can interact.

Q: Is C++ an object-oriented programming language or a procedural programming language?

A: C++ is a multi-paradigm programming language. It allows a game programmer to write games in a procedural way or an object-oriented way—or through a combination of both (to name just a few options).

Q: Should I always try to write object-oriented game programs?

A: Although object-oriented programming is used in almost every commercial game on the market, you don't have to write games using this paradigm. C++ lets you use one of several programming paradigms. In general, though, large game projects will almost surely benefit from an object-oriented approach.

Q: Why not make all class members public?

A: Because it goes against the idea of encapsulation.

Q: What is encapsulation?

A: The quality of being self-contained. In the world of OOP, encapsulation prevents client code from directly accessing the internals of an object. Instead, it encourages client code to use a defined interface to the object.

Q: What are the benefits of encapsulation?

A: In the world of OOP, encapsulation protects the integrity of an object. For example, you might have a spaceship object with a fuel data member. By preventing direct access to this data member, you can guarantee that it never becomes an illegal value (such as a negative number).

Q: Should I provide access to data members through accessor member functions?

A: Some game programmers say you should never provide access to data members through accessor member functions because even though this kind of access is indirect, it goes against the idea of encapsulation. Instead, they say you should write classes with member functions that provide the client with all of the functionality it could need, eliminating the client's need to access a specific data member.

Q: What are mutable data members?

A: Data members that can be modified even by constant member functions. You create a mutable data member using the keyword `mutable`. You can also modify a mutable data member of a constant object.

Q: Why is it useful to have a default constructor?

A: Because there might be times when objects are automatically created without any argument values passed to a constructor—for example, when you create an array of objects.

Q: What is a structure?

A: A structure is very similar to a class. The only real difference is that the default access level for structures is public. You define a structure by using the keyword `struct`.

Q: Why does C++ have both structures and classes?

A: So that C++ retains backward compatibility with C.

Q: When should I use structures?

A: Some game programmers use structures to group only data together, without functions (because that's how C structures work). But it's probably best to avoid structures whenever possible and use classes instead.

DISCUSSION QUESTIONS

1. What are the advantages and disadvantages of procedural programming?
2. What are the advantages and disadvantages of object-oriented programming?
3. Are accessor member functions a sign of poor class design? Explain.
4. How are constant member functions helpful to a game programmer?
5. When is it a good idea to calculate an object's attribute on the fly rather than storing it as a data member?

EXERCISES

1. Improve the Critter Caretaker program so that you can enter an unlisted menu choice that reveals the exact values of the critter's hunger and boredom levels.
2. Change the Critter Caretaker program so that the critter is more expressive about its needs by hinting at how hungry and bored it is.
3. What design problem does the following program have?

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

class Critter
{
public:
    int GetHunger() const {return m_Hunger;}
private:
    int m_Hunger;
};

int main()
{
    Critter crit;
    cout << crit.GetHunger() << endl;
    return 0;
}
```