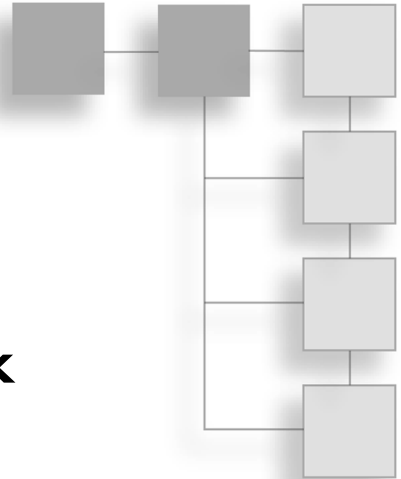


## CHAPTER 10



# INHERITANCE AND POLYMORPHISM: BLACKJACK

Classes give you the perfect way to represent game entities that have attributes and behaviors. But game entities are often related. In this chapter, you'll learn about inheritance and polymorphism, which give you ways to express those connections and can make defining and using classes even simpler and more intuitive. Specifically, you'll learn to:

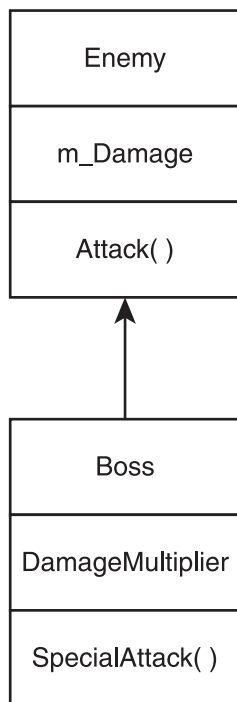
- Derive one class from another
- Use inherited data members and member functions
- Override base class member functions
- Define virtual functions to enable polymorphism
- Declare pure virtual functions to define abstract classes

## INTRODUCING INHERITANCE

One of the key elements of OOP is *inheritance*, which allows you to *derive* a new class from an existing one. When you do so, the new class automatically *inherits* (or gets) the data members and member functions of an existing class. It's like getting the work that went into the existing class free!

Inheritance is especially useful when you want to create a more specialized version of an existing class because you can add data members and member functions to the new class to extend it. For example, imagine you have a class `Enemy` that defines an enemy in a game with a member function `Attack()` and a data member `m_Damage`. You can derive a new class

Boss from `Enemy` for a boss. This means that `Boss` could automatically have `Attack()` and `m_Damage` without you having to write any code for them at all. Then, to make a boss tough, you could add a member function `SpecialAttack()` and a data member `DamageMultiplier` to the `Boss` class. Take a look at Figure 10.1, which shows the relationship between the `Enemy` and `Boss` classes.



**Figure 10.1**

`Boss` inherits `Attack()` and `m_Damage` from `Enemy` while defining `SpecialAttack()` and `m_DamageMultiplier`.

One of the many advantages of inheritance is that you can reuse classes you've already written. This reusability produces benefits that include:

- **Less work.** There's no need to redefine functionality you already have. Once you have a class that provides the base functionality for other classes, you don't have to write that code again.
- **Fewer errors.** Once you've got a bug-free class, you can reuse it without errors cropping up in it.

- **Cleaner code.** Because the functionality of base classes exists only once in a program, you don't have to wade through the same code repeatedly, which makes programs easier to understand and modify.

Most related game entities cry out for inheritance. Whether it's the series of enemies that a player faces, squadrons of military vehicles that a player commands, or an inventory of weapons that a player wields, you can use inheritance to define these groups of game entities in terms of each other, which results in faster and easier programming.

## Introducing the Simple Boss Program

The Simple Boss program demonstrates inheritance. In it, I define a class, `Enemy`, for lowly enemies. From this class, I derive a new class, `Boss`, for tough bosses that the player has to face. Then, I instantiate an `Enemy` object and call its `Attack()` member function. Next, I instantiate a `Boss` object. I'm able to call `Attack()` for the `Boss` object because it inherits the member function from `Enemy`. Finally, I call the `Boss` object's `SpecialAttack()` member function, which I defined in `Boss`, for a special attack. Since I define `SpecialAttack()` in `Boss`, only `Boss` objects have access to it. `Enemy` objects don't have this special attack at their disposal. Figure 10.2 shows the results of the program.



```
C:\Windows\system32\cmd.exe
Creating an enemy.
Attack inflicts 10 damage points!
Creating a boss.
Attack inflicts 10 damage points!
Special Attack inflicts 30 damage points!
_
```

**Figure 10.2**

The `Boss` class inherits the `Attack()` member function and then defines its own `SpecialAttack()` member function.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website ([www.cengageptr.com/downloads](http://www.cengageptr.com/downloads)). The program is in the Chapter 10 folder; the filename is `simple_boss.cpp`.

```
//Simple Boss
//Demonstrates inheritance

#include <iostream>
using namespace std;

class Enemy
{
public:
    int m_Damage;

    Enemy();
    void Attack() const;
};

Enemy::Enemy():
    m_Damage(10)
{}

void Enemy::Attack() const
{
    cout << "Attack inflicts " << m_Damage << " damage points!\n";
}

class Boss : public Enemy
{
public:
    int m_DamageMultiplier;

    Boss();
    void SpecialAttack() const;
};

Boss::Boss():
    m_DamageMultiplier(3)
{}

void Boss::SpecialAttack() const
{
    cout << "Special Attack inflicts " << (m_DamageMultiplier * m_Damage);
    cout << " damage points!\n";
}
```

```
int main()
{
    cout << "Creating an enemy.\n";
    Enemy enemy1;
    enemy1.Attack();

    cout << "\nCreating a boss.\n";
    Boss boss1;
    boss1.Attack();
    boss1.SpecialAttack();

    return 0;
}
```

## Deriving from a Base Class

I derive the Boss class from Enemy when I define Boss with the following line:

```
class Boss : public Enemy
```

Boss is based on Enemy. In fact, Enemy is called the *base class* (or *superclass*) and Boss the *derived class* (or *subclass*). This means that Boss inherits Enemy's data members and member functions, subject to access controls. In this case, Boss inherits and can directly access m\_Damage and Attack(). It's as if I defined both m\_Damage and Attack() in Boss.

### Hint

---

You might have noticed that I made all of the members of the classes public, including their data members. I did this because it makes for the simplest first example of a base and derived class. You also might have noticed that I used the keyword public when deriving Boss from Enemy. For now, don't worry about this. I'll cover it all in the next example program, Simple Boss 2.0.

---

To derive classes of your own, follow my example. After the class name in a class definition, put a colon followed by an access modifier (such as public), followed by the name of the base class. It's perfectly acceptable to derive a new class from a derived class, and sometimes it makes perfect sense to do so. However, to keep things simple, I'll deal with only one level of inheritance in this example.

A few base class member functions are not inherited by derived classes. They are as follows:

- Constructors
- Copy constructors

- Destructors
- Overloaded assignment operators

You have to write your own versions of these in the derived class.

## Instantiating Objects from a Derived Class

In `main()`, I instantiate an `Enemy` object and then call its `Attack()` member function. This works just as you'd expect. The interesting part of the program begins next, when I instantiate a `Boss` object.

```
Boss boss1;
```

After this line of code, I have a `Boss` object with an `m_Damage` data member equal to 10 and an `m_DamageMultiplier` data member equal to 3. How did this happen? Although constructors and destructors are not inherited from a base class, they are called when an instance is created or destroyed. In fact, a base class constructor is called before the derived class constructor to create its part of the final object.

In this case, when a `Boss` object is instantiated, the default `Enemy` constructor is automatically called and the object gets an `m_Damage` data member with a value of 10 (just like any `Enemy` object would). Then, the `Boss` constructor is called and finishes off the object by giving it an `m_DamageMultiplier` data member with a value of 3. The reverse happens when a `Boss` object is destroyed at the end of the program. First, the `Boss` class destructor is called for the object, and then the `Enemy` class destructor is called. Because I didn't define destructors in this program, nothing special happens before the `Boss` object ceases to exist.

### Hint

---

The fact that base class destructors are called for objects of derived classes ensures that each class gets its chance to clean up any part of the object that needs to be taken care of, such as memory on the heap.

---

## Using Inherited Members

Next, I call an inherited member function of the `Boss` object, which displays the exact same message as `enemy1.Attack()`.

```
boss1.Attack();
```

That makes perfect sense because the same code is being executed and both objects have an `m_Damage` data member equal to 10. Notice that the function call looks the same as it did for `enemy1`. The fact that `Boss` inherited the member function from `Enemy` makes no difference in how the function is called.

Next, I get `Boss` to pull out its special attack, which displays the message `Special Attack` inflicts 30 damage points!

```
boss1.SpecialAttack();
```

The thing to notice about this is that `SpecialAttack()`, declared as a part of `Boss`, uses the data member `m_Damage`, declared in `Enemy`. That's perfectly fine. `Boss` inherits `m_Damage` from `Enemy` and, in this example, the data member works like any other data member in the `Boss` class.

## CONTROLLING ACCESS UNDER INHERITANCE

When you derive one class from another, you can control how much access the derived class has to the base class' members. For the same reasons that you want to provide only as much access as is necessary to a class' members to the rest of your program, you want to provide only as much access as is necessary to a class' members to a derived class. Not coincidentally, you use the same access modifiers that you've seen before: `public`, `protected`, and `private`. (Okay, you haven't seen `protected` before, but I'll explain that modifier in the "Using Access Modifiers with Class Members" section.)

### Introducing the Simple Boss 2.0 Program

The Simple Boss 2.0 program is another version of the Simple Boss program from earlier in this chapter. The new version, Simple Boss 2.0, looks exactly the same to the user, but the code is a little different because I put some restrictions on base class members. If you want to see what the program does, take a look back at Figure 10.2.

You can download the code for this program from the Cengage Learning website ([www.cengageptr.com/downloads](http://www.cengageptr.com/downloads)). The program is in the Chapter 10 folder; the filename is `simple_boss2.cpp`.

```
//Simple Boss 2.0
//Demonstrates access control under inheritance

#include <iostream>
using namespace std;

class Enemy
{
public:
    Enemy();
    void Attack() const;
```

```

protected:
    int m_Damage;
};

Enemy::Enemy():
    m_Damage(10)
{}

void Enemy::Attack() const
{
    cout << "Attack inflicts " << m_Damage << " damage points!\n";
}

class Boss : public Enemy
{
public:
    Boss();
    void SpecialAttack() const;

private:
    int m_DamageMultiplier;
};

Boss::Boss():
    m_DamageMultiplier(3)
{}

void Boss::SpecialAttack() const
{
    cout << "Special Attack inflicts " << (m_DamageMultiplier * m_Damage);
    cout << " damage points!\n";
}

int main()
{
    cout << "Creating an enemy.\n";
    Enemy enemy1;
    enemy1.Attack();

    cout << "\nCreating a boss.\n";
    Boss boss1;
    boss1.Attack();
    boss1.SpecialAttack();

    return 0;
}

```



## Using Access Modifiers with Class Members

You've seen the access modifiers `public` and `private` used with class members before, but there's a third modifier you can use with members of a class—`protected`. That's what I use with the data member of `Enemy`.

`protected:`

```
int m_Damage;
```

Members that are specified as `protected` are not accessible outside of the class, except in some cases of inheritance. As a refresher, here are the three levels of member access:

- `public` members are accessible to all code in a program.
- `protected` members are accessible only in their own class and certain derived classes, depending upon the access level used in inheritance.
- `private` members are only accessible in their own class, which means they are not directly accessible in any derived class.

## Using Access Modifiers when Deriving Classes

When you derive a class from an existing one, you can use an access modifier, such as `public`, which I used in deriving `Boss`.

```
class Boss : public Enemy
```

Using `public` derivation means that `public` members in the base class become `public` members in the derived class, `protected` members in the base class become `protected` members in the derived class, and `private` members in the base class are inaccessible in the derived class.

---

### Trick

Even if base data members are `private`, you can still use them indirectly through base class member functions. You can even get and set their values if the base class has accessor member functions.

---

Because `Boss` inherits from `Enemy` using the keyword `public`, `Boss` inherits `Enemy`'s `public` member functions as `public` member functions. It also means that `Boss` inherits `m_Damage` as a `protected` data member. The class essentially acts as if I simply copied and pasted the code for these two `Enemy` class members right into the `Boss` definition. But through the beauty of inheritance, I didn't have to do this. The upshot is that the `Boss` class can access `Attack()` and `m_Damage()`.

**Hint**

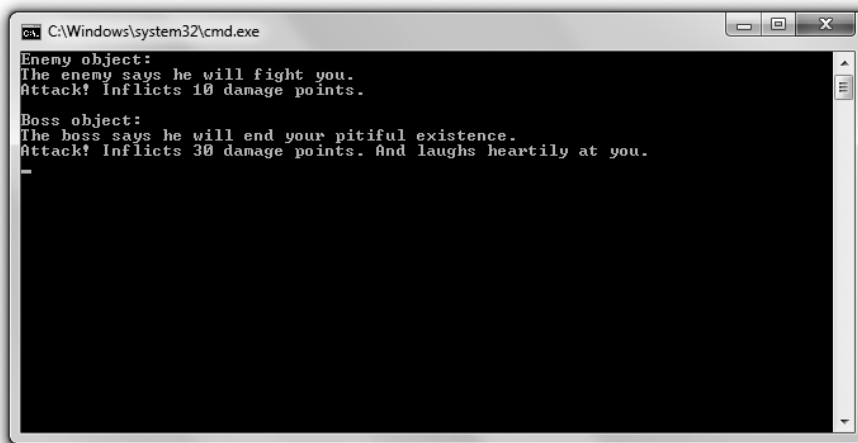
You can derive a new class with the `protected` and `private` keywords, but they're rarely used and are beyond the scope of this book.

## CALLING AND OVERRIDING BASE CLASS MEMBER FUNCTIONS

You're not stuck with every base class member function you inherit in a derived class as is. You have options that allow you to customize how those inherited member functions work in your derived class. You can override them by giving them new definitions in your derived class. You can also explicitly call a base class member function from any member function of your derived class.

### Introducing the Overriding Boss Program

The Overriding Boss program demonstrates calling and overriding base class member functions in a derived class. The program creates an enemy that taunts the player and then attacks him. Next, the program creates a boss from a derived class. The boss also taunts the player and attacks him, but the interesting thing is that the inherited behaviors of taunting and attacking are changed for the boss (who is a bit cockier than the enemy). These changes are accomplished through function overriding and calling a base class member function. Figure 10.3 shows the results of the program.



```
C:\Windows\system32\cmd.exe
Enemy object:
The enemy says he will fight you.
Attack! Inflicts 10 damage points.

Boss object:
The boss says he will end your pitiful existence.
Attack! Inflicts 30 damage points. And laughs heartily at you.
```

**Figure 10.3**

The `Boss` class inherits and overrides the base class member functions `Taunt()` and `Attack()`, creating new behaviors for the functions in `Boss`.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website ([www.cengageptr.com/downloads](http://www.cengageptr.com/downloads)). The program is in the Chapter 10 folder; the filename is `overriding_boss.cpp`.

```
//Overriding Boss
//Demonstrates calling and overriding base member functions
#include <iostream>

using namespace std;

class Enemy
{
public:
    Enemy(int damage = 10);
    void virtual Taunt() const;    //made virtual to be overridden
    void virtual Attack() const;  //made virtual to be overridden

private:
    int m_Damage;
};

Enemy::Enemy(int damage):
    m_Damage(damage)
{}

void Enemy::Taunt() const
{
    cout << "The enemy says he will fight you.\n";
}

void Enemy::Attack() const
{
    cout << "Attack! Inflicts " << m_Damage << " damage points.";
}

class Boss : public Enemy
{
public:
    Boss(int damage = 30);
    void virtual Taunt() const;    //optional use of keyword virtual
    void virtual Attack() const;  //optional use of keyword virtual
};

Boss::Boss(int damage):
    Enemy(damage)                //call base class constructor with argument
{}

```

```

void Boss::Taunt() const      //override base class member function
{
    cout << "The boss says he will end your pitiful existence.\n";
}

void Boss::Attack() const    //override base class member function
{
    Enemy::Attack();         //call base class member function
    cout << " And laughs heartily at you.\n";
}

int main()
{
    cout << "Enemy object:\n";
    Enemy anEnemy;
    anEnemy.Taunt();
    anEnemy.Attack();

    cout << "\n\nBoss object:\n";
    Boss aBoss;
    aBoss.Taunt();
    aBoss.Attack();

    return 0;
}

```

## Calling Base Class Constructors

As you've seen, the constructor for a base class is automatically called when an object of a derived class is instantiated, but you can also explicitly call a base class constructor from a derived class constructor. The syntax for this is a lot like the syntax for a member initialization list. To call a base class constructor from a derived class constructor, after the derived constructor's parameter list, type a colon followed by the name of the base class, followed by a set of parentheses containing whatever parameters the base class constructor you're calling needs. I do this in the `Boss` constructor, which says to explicitly call the `Enemy` constructor and pass it damage.

```

Boss::Boss(int damage):
    Enemy(damage)          //call base class constructor with argument
{}

```

This allows me to pass the `Enemy` constructor the value that gets assigned to `m_Damage`, rather than just accepting its default value.

When I first instantiate `aBoss` in `main()`, the `Enemy` constructor is called and passed the value 30, which gets assigned to `m_Damage`. Then, the `Boss` constructor runs (which doesn't do much of anything) and the object is completed.

## Hint

---

Being able to call a base class constructor is useful when you want to pass specific values to it.

---

## Declaring Virtual Base Class Member Functions

Any inherited base class member function that you expect to be overridden in a derived class should be declared as `virtual`, using the keyword `virtual`. When you declare a member function `virtual`, you provide a way for overridden versions of the member function to work as expected with pointers and references to objects. Since I know that I'll override `Taunt()` in the derived class, `Boss`, I declare `Taunt()` `virtual` in my base class, `Enemy`.

```
void virtual Taunt() const;    //made virtual to be overridden
```

## Trap

---

Although you can override non-virtual member functions, this can lead to behavior you might not expect. A good rule of thumb is to declare any base class member function to be overridden as `virtual`.

---

Outside the `Enemy` class definition, I define `Taunt()`:

```
void Enemy::Taunt() const
{
    cout << "The enemy says he will fight you.\n";
}
```

Notice that I didn't use the keyword `virtual` in the definition. You don't use `virtual` in the definition of a member function, only in its declaration.

Once a member function has been declared as `virtual`, it's `virtual` in any derived class. This means you don't have to use the keyword `virtual` in a declaration when you override a virtual member function, but you should use it anyway because it will remind you that the function is indeed `virtual`. So, when I override `Taunt()` in `Boss`, I explicitly declare it as `virtual`, even though I don't have to:

```
void virtual Taunt() const;    //optional use of keyword virtual
```

## Overriding Virtual Base Class Member Functions

The next step in overriding is to give the member function a new definition in the derived class. That's what I do for the `Boss` class with:

```
void Boss::Taunt() const    //override base class member function
{
    cout << "The boss says he will end your pitiful existence.\n";
}
```

This new definition is executed when I call the member function through any `Boss` object. It replaces the definition of `Taunt()` inherited from `Enemy` for all `Boss` objects. When I call the member function in `main()` with the following line, the message `The boss says he will end your pitiful existence.` is displayed.

```
aBoss.Taunt();
```

Overriding member functions is useful when you want to change or extend the behavior of base class member functions in derived classes.

### Trap

---

Don't confuse override with overload. When you override a member function, you provide a new definition of it in a derived class. When you overload a function, you create multiple versions of it with different signatures.

---

### Trap

---

When you override an overloaded base class member function, you hide all of the other overloaded versions of the base class member function—meaning that the only way to access the other versions of the member function is to explicitly call the base class member function. So if you override an overloaded member function, it's a good idea to override every version of the overloaded function.

---

## Calling Base Class Member Functions

You can directly call a base class member function from any function in a derived class. All you have to do is prefix the class name to the member function name with the scope resolution operator. That's what I do when I define the overridden version of `Attack()` for the `Boss` class.

```
void Boss::Attack() const    //override base class member function
{
    Enemy::Attack();          //call base class member function
    cout << " And laughs heartily at you.\n";
}
```

The code `Enemy::Attack()`; explicitly calls the `Attack()` member function of `Enemy`. Because the `Attack()` definition in `Boss` overrides the class' inherited version, it's as if I've extended the definition of what it means for a boss to attack. What I'm essentially saying is that when a boss attacks, the boss does exactly what an enemy does and then adds a laugh. When I call the member function for a `Boss` object in `main()` with the following line, `Boss' Attack()` member function is called because I've overloaded `Attack()`.

```
aBoss.Attack();
```

The first thing that `Boss' Attack()` member function does is explicitly call `Enemy's Attack()` member function, which displays the message `Attack! Inflicts 30 damage points`. Then, `Boss' Attack()` member function displays the message `And laughs heartily at you`.

### Trick

---

You can extend the way a member function of a base class works in a derived class by overriding the base class method and then explicitly calling the base class member function from this new definition in the derived class and adding some functionality.

---

## USING OVERLOADED ASSIGNMENT OPERATORS AND COPY CONSTRUCTORS IN DERIVED CLASSES

You already know how to write an overloaded assignment operator and a copy constructor for a class. However, writing them for a derived class requires a little bit more work because they aren't inherited from a base class.

When you overload the assignment operator in a derived class, you usually want to call the assignment operator member function from the base class, which you can explicitly call using the base class name as a prefix. If `Boss` is derived from `Enemy`, the overloaded assignment operator member function defined in `Boss` could start:

```
Boss& operator=(const Boss& b)
{
    Enemy::operator=(b);    //handles the data members inherited from Enemy
    //now take care of data members defined in Boss
```

The explicit call to `Enemy's` assignment operator member function handles the data members inherited from `Enemy`. The rest of the member function would take care of the data members defined in `Boss`.

For the copy constructor, you also usually want to call the copy constructor from a base class, which you can call just like any base class constructor. If `Boss` is derived from `Enemy`, the copy constructor defined in `Boss` could start:

```
Boss (const Boss& b): Enemy(b)    //handles the data members inherited from Enemy
{
    //now take care of data members defined in Boss
```

By calling `Enemy`'s copy constructor with `Enemy(b)`, you copy `Enemy`'s data members into the new `Boss` object. In the remainder of `Boss`' copy constructor, you can take care of copying the data members declared in `Boss` into the new object.

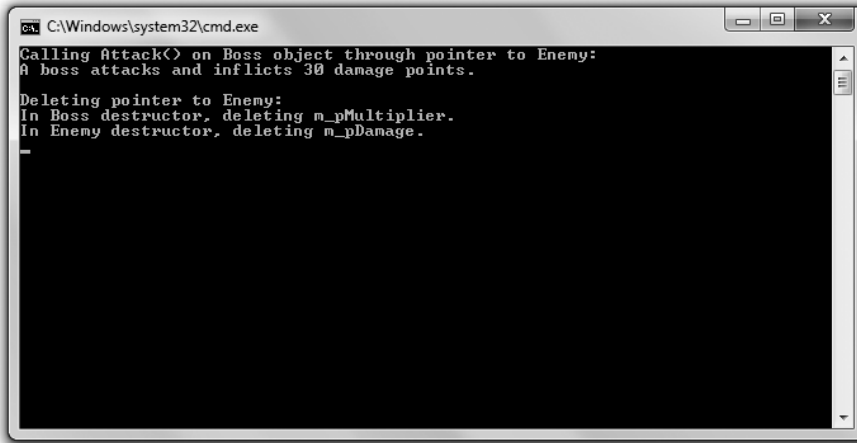
## INTRODUCING POLYMORPHISM

One of the pillars of OOP is *polymorphism*, which means that a member function will produce different results depending on the type of object for which it is being called. For example, suppose you have a group of bad guys that the player is facing, and the group is made of objects of different types that are related through inheritance, such as enemies and bosses. Through the magic of polymorphism, you could call the same member function for each bad guy in the group, say, to attack the player, and the type of each object would determine the exact results. The call for the enemy objects could produce one result, such as a weak attack, while the call for bosses could produce a different result, such as a powerful attack. This might sound a lot like overriding, but polymorphism is different because the effect of the function call is dynamic and is determined at run time, depending on the object type. But the best way to understand this isn't through theoretical discussion; it is through a concrete example.

## Introducing the Polymorphic Bad Guy Program

The Polymorphic Bad Guy program demonstrates how to achieve polymorphic behavior. It shows what happens when you use a pointer to a base class to call inherited virtual member functions. It also shows how using virtual destructors ensures that the correct destructors are called for objects pointed to by pointers to a base class. Figure 10.4 shows the results of the program.





**Figure 10.4**

Through polymorphism, the correct member functions and destructors are called for objects pointed to by pointers to a base class.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website ([www.cengageptr.com/downloads](http://www.cengageptr.com/downloads)). The program is in the Chapter 10 folder; the filename is `polymorphic_bad_guy.cpp`.

```
//Polymorphic Bad Guy
//Demonstrates calling member functions dynamically
#include <iostream>
using namespace std;
class Enemy
{
public:
    Enemy(int damage = 10);
    virtual ~Enemy();
    void virtual Attack() const;
protected:
    int* m_pDamage;
};
Enemy::Enemy(int damage)
{
    m_pDamage = new int(damage);
}
```

```

Enemy::~~Enemy()
{
    cout << "In Enemy destructor, deleting m_pDamage.\n";
    delete m_pDamage;
    m_pDamage = 0;
}

void Enemy::Attack() const
{
    cout << "An enemy attacks and inflicts " << *m_pDamage << " damage points.";
}

class Boss : public Enemy
{
public:
    Boss(int multiplier = 3);
    virtual ~Boss();
    void virtual Attack() const;

protected:
    int* m_pMultiplier;
};

Boss::Boss(int multiplier)
{
    m_pMultiplier = new int(multiplier);
}

Boss::~~Boss()
{
    cout << "In Boss destructor, deleting m_pMultiplier.\n";
    delete m_pMultiplier;
    m_pMultiplier = 0;
}

void Boss::Attack() const
{
    cout << "A boss attacks and inflicts " << (*m_pDamage) * (*m_pMultiplier)
        << " damage points.";
}

int main()
{
    cout << "Calling Attack() on Boss object through pointer to Enemy:\n";
    Enemy* pBadGuy = new Boss();
    pBadGuy->Attack();
}

```

```

cout << "\n\nDeleting pointer to Enemy:\n";
delete pBadGuy;
pBadGuy = 0;
return 0;
}

```

## Using Base Class Pointers to Derived Class Objects

An object of a derived class is also a member of the base class. For example, in the Polymorphic Bad Guy program, a `Boss` object is an `Enemy` object, too. That makes sense because a boss is really only a specialized kind of enemy. It also makes sense because a `Boss` object has all of the members of an `Enemy` object. Okay, so what? Well, because an object of a derived class is also a member of the base class, you can use a pointer to the base class to point to an object of the derived class. That's what I do in `main()` with the following line, which instantiates a `Boss` object on the heap and creates a pointer to `Enemy`, `pBadGuy`, that points to the `Boss` object.

```
Enemy* pBadGuy = new Boss();
```

Why in the world would you want to do this? It's useful because it allows you to deal with objects without requiring that you know their exact type. For example, you could have a function that accepts a pointer to `Enemy` that could work with either an `Enemy` or a `Boss` object. The function wouldn't have to know the exact type of object being passed to it; it could work with the object to produce different results depending on the object's type, as long as derived member functions were declared virtual. Because `Attack()` is virtual, the correct version of the member function will be called (based on the type of object) and will not be fixed by the type of pointer.

I prove that the behavior will be polymorphic in `main()`. Remember that `pBadGuy` is a pointer to `Enemy` that points to a `Boss` object. So, the following line calls the `Attack()` member function of a `Boss` object through a pointer to `Enemy`, which correctly results in the `Attack()` member function defined in `Boss` being called and the text `A boss attacks and inflicts 30 damage points.` being displayed on the screen.

```
pBadGuy->Attack();
```

### Hint

---

Virtual functions produce polymorphic behavior through references as well as through pointers.

---

**Trap**


---

If you override a non-virtual member function in a derived class and call that member function on a derived class object through a pointer to a base class, you'll get the results of the base class member function and not the derived class member function definition. This is easier to understand with an example. If in the Polymorphic Bad Guy program I hadn't declared `Attack()` as virtual, then when I invoked the member function through a pointer to `Enemy` on a `Boss` object with `pBadGuy->Attack();`, I would have gotten the message `An enemy attacks and inflicts 10 damage points`. This would have happened as a result of *early binding*, in which the exact member function is bound based on the pointer type—in this case, `Enemy`. But because `Attack()` is declared as virtual, the member function call is based on the type of object being pointed to at run time, `Boss` in this case, not fixed by pointer type. I achieve this polymorphic behavior as the result of *late binding* because `Attack()` is virtual. The moral of the story is that you should only override virtual member functions.

---

**Trap**


---

The benefits of virtual functions aren't free; there is a performance cost associated with the overhead. Therefore, you should use virtual functions only when you need them.

---

## Defining Virtual Destructors

When you use a pointer to a base class to point to an object of a derived class, you have a potential problem. When you delete the pointer, only the base class' destructor will be called for the object. This could lead to disastrous results because the derived class' destructor might need to free memory (as the destructor for `Boss` does). The solution, as you might have guessed, is to make the base class' destructor virtual. That way, the derived class' destructor is called, which (as always) leads to the calling of the base class' destructor, giving every class the chance to clean up after itself.

I put this theory into action when I declare `Enemy`'s destructor virtual.

```
virtual ~Enemy();
```

In `main()`, when I delete the pointer pointing to the `Boss` object with the following line, the `Boss` object's destructor is called, which frees the memory on the heap that `m_pDamageMultiplier` points to and displays the message `In Boss destructor, deleting m_pMultiplier`.

```
delete pBadGuy;
```

Then, `Enemy`'s destructor is called, which frees the memory on the heap that `m_pDamage` points to and displays the message `In Enemy destructor, deleting m_pDamage`. The object is destroyed, and all memory associated with the object is freed.

## Trick

A good rule of thumb is that if you have any virtual member functions in a class, you should make the destructor virtual, too.

## USING ABSTRACT CLASSES

At times, you might want to define a class to act as a base for other classes, but it doesn't make sense to instantiate objects from this class because it's so generic. For example, suppose you have a game with a bunch of types of creatures running around in it. Although you have a wide variety of creatures, they all have two things in common: They have a health value, and they can offer a greeting. So, you could define a class, `Creature`, as a base from which to derive other classes, such as `Pixie`, `Dragon`, `Orc`, and so on. Although `Creature` is helpful, it doesn't really make sense to instantiate a `Creature` object. It would be great if there were a way to indicate that `Creature` is a base class only, and not meant for instantiating objects. Well, C++ lets you define a kind of class just like this, called an *abstract class*.

## Introducing the Abstract Creature Program

The Abstract Creature program demonstrates abstract classes. In the program, I define an abstract class, `Creature`, which can be used as a base class for specific creature classes. I define one such class, `Orc`. Then, I instantiate an `Orc` object and call a member function to get the orc to grunt hello and another member function to display the orc's health. Figure 10.5 shows the results of the program.



**Figure 10.5**

The orc is an object instantiated from a class derived from an abstract class for all creatures.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website ([www.cengageptr.com/downloads](http://www.cengageptr.com/downloads)). The program is in the Chapter 10 folder; the filename is `abstract_creature.cpp`.

```
//Abstract Creature
//Demonstrates abstract classes

#include <iostream>
using namespace std;

class Creature    //abstract class
{
public:
    Creature(int health = 100);
    virtual void Greet() const = 0;    //pure virtual member function
    virtual void DisplayHealth() const;

protected:
    int m_Health;
};

Creature::Creature(int health):
    m_Health(health)
{}

void Creature::DisplayHealth() const
{
    cout << "Health: " << m_Health << endl;
}

class Orc : public Creature
{
public:
    Orc(int health = 120);
    virtual void Greet() const;
};

Orc::Orc(int health):
    Creature(health)
{}

void Orc::Greet() const
{
    cout << "The orc grunts hello.\n";
}
```

```
int main()
{
    Creature* pCreature = new Orc();
    pCreature->Greet();
    pCreature->DisplayHealth();

    return 0;
}
```

## Declaring Pure Virtual Functions

A *pure virtual function* is one to which you don't need to give a definition. The logic behind this is that there might not be a good definition in the class for the member function. For example, I don't think it makes sense to define the `Greet()` function in my `Creature` class because a greeting really depends on the specific type of creature—a pixie twinkles, a dragon blows a puff of smoke, and an orc grunts.

You specify a pure virtual function by placing an equal sign and a zero at the end of the function header. That's what I did in `Creature` with the following line:

```
virtual void Greet() const = 0;    //pure virtual member function
```

When a class contains at least one pure virtual function, it's an abstract class. Therefore, `Creature` is an abstract class. I can use it as the base class for other classes, but I can't instantiate objects from it.

An abstract class can have data members and virtual functions that are not pure virtual. In `Creature`, I declare a data member `m_Health` and a virtual member function `DisplayHealth()`.

## Deriving a Class from an Abstract Class

When you derive a new class from an abstract class, you can override its pure virtual functions. If you override all of its pure virtual functions, then the new class is not abstract and you can instantiate objects from it. When I derive `Orc` from `Creature`, I override `Creature`'s one pure virtual function with the following lines:

```
void Orc::Greet() const
{
    cout << "The orc grunts hello.\n";
}
```

This means I can instantiate an object from `Orc`, which is what I do in `main()` with the following line:

```
Creature* pCreature = new Orc();
```

The code instantiates a new `Orc` object on the heap and assigns the memory location of the object to `pCreature`, a pointer to `Creature`. Even though I can't instantiate an object from `Creature`, it's perfectly fine to declare a pointer using the class. Like all base class pointers, a pointer to `Creature` can point to any object of a class derived from `Creature`, like `Orc`.

Next, I call `Greet()`, the pure virtual function that I override in `Orc` with the following line:

```
pCreature->Greet();
```

The correct greeting, The orc grunts hello., is displayed.

Finally, I call `DisplayHealth()`, which I define in `Creature`.

```
pCreature->DisplayHealth();
```

It also displays the proper message, Health: 120.

## INTRODUCING THE BLACKJACK GAME

The final project for this chapter is a simplified version of the casino card game Blackjack (tacky green felt not included). The game works like this: Players are dealt cards with point values. Each player tries to reach a total of 21 without exceeding that amount. Numbered cards count as their face value. An ace counts as either 1 or 11 (whichever is best for the player), and any jack, queen, or king counts as 10.

The computer is the house (the casino), and it competes against one to seven players. At the beginning of the round, all participants (including the house) are dealt two cards. Players can see all of their cards, along with their total. However, one of house's cards is hidden for the time being.

Next, each player gets the chance to take one additional card at a time for as long as he likes. If a player's total exceeds 21 (known as *busting*), the player loses. After all players have had the chance to take additional cards, the house reveals its hidden card. The house must then take additional cards as long as its total is 16 or less. If the house busts, all players who have not busted win. Otherwise, each remaining player's total is compared to the house's total. If the player's total is greater than the house's, he wins. If the player's total is less than the house's, he loses. If the two totals are the same, the player ties the house (also known as *pushing*). Figure 10.6 shows the game.



```

C:\Windows\system32\cmd.exe
Welcome to Blackjack!
How many players? (1 - 7): 2
Enter player name: Mike
Enter player name: Ariella

Mike: 2c 9c (11)
Ariella: 9s 9h (18)
House: XX Jh

Mike, do you want a hit? (Y/N): y
Mike: 2c 9c 2d (13)
Mike, do you want a hit? (Y/N): y
Mike: 2c 9c 2d 9c (23)
Mike busts.

Ariella, do you want a hit? (Y/N): n
House: 5c Jh (15)
House: 5c Jh 7s (22)
House busts.
Ariella wins.

Do you want to play again? (Y/N): _

```

**Figure 10.6**

One player wins; the other is not so lucky.

Used with permission from Microsoft.

## Designing the Classes

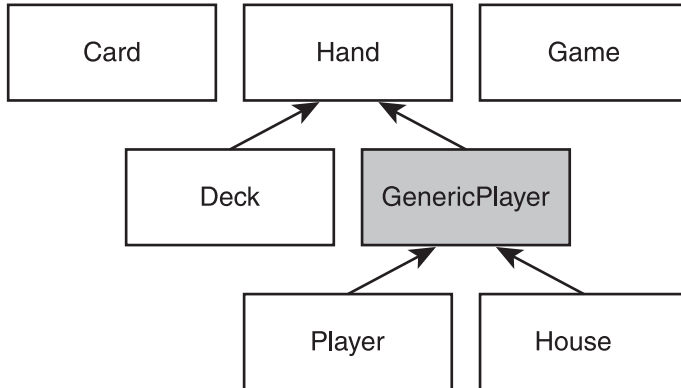
Before you start coding a project with multiple classes, it is helpful to map them out on paper. You might make a list and include a brief description of each class. Table 10.1 shows my first pass at such a list for the Blackjack game.

**Table 10.1** Blackjack Classes

Class	Base Class	Description
Card	None	A Blackjack playing card.
Hand	None	A Blackjack hand. A collection of Card objects.
Deck	Hand	A Blackjack deck. Has extra functionality that Hand doesn't, such as shuffling and dealing.
GenericPlayer	Hand	A generic Blackjack player. Not a full player, but the common elements of a human player and the computer player.
Player	GenericPlayer	A human Blackjack player.
House	GenericPlayer	The computer player (the house).
Game	None	A Blackjack game.

To keep things simple, all member functions will be public and all data members will be protected. Also, I'll use only public inheritance, which means that each derived class will inherit all of its base class members.

In addition to describing your classes in words, it helps to draw a family tree of sorts to visualize how your classes are related. That's what I did in Figure 10.7.



**Figure 10.7**

Inheritance hierarchy of classes for the Blackjack game. `GenericPlayer` is shaded because it turns out to be an abstract class.

Next, it's a good idea to get more specific. Ask yourself about the classes. What exactly will they represent? What will they be able to do? How will they work with the other classes?

I see `Card` objects as real-life cards. You don't copy a card when you deal it from the deck to a hand; you move it. For me, that means `Hand` will have a data member that is a vector of pointers to `Card` objects, which will exist on the heap. When a card moves from one `Hand` to another, it's really pointers that are being copied and destroyed.

I see players (the human players and the computer) as Blackjack hands with names. That's why I derive `Player` and `House` (indirectly) from `Hand`. (Another equally valid view is that players have a hand. If I had gone this route, `Player` and `House` would have had `Hand` data members instead of being derived from `Hand`.)

I define `GenericPlayer` to house the functionality that `Player` and `House` share, as opposed to duplicating this functionality in both classes.

Also, I see the deck as separate from the house. The deck will deal cards to the human players and the computer-controlled house in the same way. This means that `Deck` will

have a member function to deal cards that is polymorphic and will work with either a `Player` or a `House` object.

To really flesh things out, you can list the data members and member functions that you think the classes will have, along with a brief description of each. That's what I do next in Tables 10.2 through 10.8. For each class, I list only the members I define in it. Several classes will, of course, be inherited members from base classes.

**Table 10.2 Card Class**

Member	Description
<code>rank m_Rank</code>	Rank of the card (ace, 2, 3, and so on). <code>rank</code> is an enumeration for all 13 ranks.
<code>suit m_Suit</code>	Suit of the card (clubs, diamonds, hearts, or spades). <code>suit</code> is an enumeration for the four possible suits.
<code>bool m_IsFaceUp</code>	Indicates whether the card is face up. Affects how the card is displayed and the value it has.
<code>int GetValue()</code>	Returns the value of the card.
<code>void Flip()</code>	Flips a card. Face up becomes face down, and face down becomes face up.

**Table 10.3 Hand Class**

Member	Description
<code>vector&lt;Card*&gt; m_Cards</code>	Collection of cards. Stores pointers to <code>Card</code> objects.
<code>void Add(Card* pCard)</code>	Adds a card to the hand. Adds a pointer to <code>Card</code> to the vector <code>m_Cards</code> .
<code>void Clear()</code>	Clears all cards from the hand. Removes all pointers in the vector <code>m_Cards</code> , deleting all associated <code>Card</code> objects on the heap.
<code>int GetTotal() const</code>	Returns the total value of the hand.

**Table 10.4 GenericPlayer Class (Abstract)**

Member	Description
string m_Name	Generic player's name.
virtual bool IsHitting() const = 0	Indicates whether the generic player wants another hit. Pure virtual function.
bool IsBusted() const	Indicates whether the generic player is busted.
void Bust() const	Announces that the generic player busts.

**Table 10.5 Player Class**

Member	Description
virtual bool IsHitting() const	Indicates whether the player wants another hit.
void Win() const	Announces that the player wins.
void Lose() const	Announces that the player loses.
void Push() const	Announces that the player pushes.

**Table 10.6 House Class**

Member	Description
virtual bool IsHitting() const	Indicates whether the house is taking another hit.
void FlipFirstCard()	Flips over the first card.

**Table 10.7 Deck Class**

Member	Description
void Populate()	Creates a standard deck of 52 cards.
void Shuffle()	Shuffles cards.
void Deal(Hand& aHand)	Deals one card to a hand.
void AdditionalCards (GenericPlayer& aGenericPlayer)	Gives additional cards to a generic player for as long as the generic player can and wants to hit.

**Table 10.8 Game Class**

Member	Description
Deck m_Deck	A deck of cards.
House m_House	The casino's hand, the house.
vector<Player> m_Players	Collection of human players. A vector of Player objects.
void Play()	Plays a round of Blackjack.

## Planning the Game Logic

The last part of my planning is to map out the basic flow of one round of the game. I wrote some pseudocode for the Game class' Play() member function. Here's what I came up with:

```

Deal players and the house two initial cards
Hide the house's first card
Display players' and house's hands
Deal additional cards to players
Reveal house's first card
Deal additional cards to house
If house is busted
    Everyone who is not busted wins
Otherwise
    For each player
        If player isn't busted
            If player's total is greater than the house's total
                Player wins
            Otherwise if player's total is less than house's total
                Player loses
            Otherwise
                Player pushes
Remove everyone's cards

```

At this point, you know a lot about the Blackjack program and you haven't even seen a single line of code yet! But that's a good thing. Planning can be as important as coding (if not more so). Because I've spent so much time describing the classes, I won't describe every part of the code. I'll just point out significant or new ideas. You can download the code for this program from the Cengage Learning website ([www.cengageptr.com/downloads](http://www.cengageptr.com/downloads)). The program is in the Chapter 10 folder; the filename is `blackjack.cpp`.

**Hint**


---

The `blackjack.cpp` file contains seven classes. In C++ programming, it's common to break up files like this into multiple files, based on individual classes. However, the topic of writing a single program using multiple files is beyond the scope of this book.

---

**The Card Class**

After some initial statements, I define the `Card` class for an individual playing card.

```
//Blackjack
//Plays a simple version of the casino game of blackjack; for 1 - 7 players

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>

using namespace std;

class Card
{
public:
    enum rank {ACE = 1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
               JACK, QUEEN, KING};
    enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};

    //overloading << operator so can send Card object to standard output
    friend ostream& operator<<(ostream& os, const Card& aCard);

    Card(rank r = ACE, suit s = SPADES, bool ifu = true);

    //returns the value of a card, 1 - 11
    int GetValue() const;

    //flips a card; if face up, becomes face down and vice versa
    void Flip();

private:
    rank m_Rank;
    suit m_Suit;
    bool m_IsFaceUp;
};

Card::Card(rank r, suit s, bool ifu): m_Rank(r), m_Suit(s), m_IsFaceUp(ifu)
{}
```

```

int Card::GetValue() const
{
    //if a card is face down, its value is 0
    int value = 0;
    if (m_IsFaceUp)
    {
        //value is number showing on card
        value = m_Rank;
        //value is 10 for face cards
        if (value > 10)
        {
            value = 10;
        }
    }
    return value;
}

void Card::Flip()
{
    m_IsFaceUp = !(m_IsFaceUp);
}

```

I define two enumerations, `rank` and `suit`, to use as the types for the rank and suit data members of the class, `m_Rank` and `m_Suit`. This has two benefits. First, it makes the code more readable. A suit data member will have a value like `CLUBS` or `HEARTS` instead of 0 or 2. Second, it limits the values that these two data members can have. `m_Suit` can only store a value from `suit`, and `m_Rank` can only store a value from `rank`.

Next, I make the overloaded `operator<<()` function a friend of the class so I can display a card object on the screen.

`GetValue()` returns a value for a `Card` object, which can be between 0 and 11. Aces are valued at 11. (I deal with potentially counting them as 1 in the `Hand` class, based on the other cards in the hand.) A face-down card has a value of 0.

## The Hand Class

I define the `Hand` class for a collection of cards.

```

class Hand
{
public:
    Hand();

    virtual ~Hand();

    //adds a card to the hand
    void Add(Card* pCard);

```

```

    //clears hand of all cards
    void Clear();

    //gets hand total value, intelligently treats aces as 1 or 11
    int GetTotal() const;

protected:
    vector<Card*> m_Cards;
};

Hand::Hand()
{
    m_Cards.reserve(7);
}

Hand::~~Hand()
{
    Clear();
}

void Hand::Add(Card* pCard)
{
    m_Cards.push_back(pCard);
}

void Hand::Clear()
{
    //iterate through vector, freeing all memory on the heap
    vector<Card*>::iterator iter = m_Cards.begin();
    for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
    {
        delete *iter;
        *iter = 0;
    }
    //clear vector of pointers
    m_Cards.clear();
}

int Hand::GetTotal() const
{
    //if no cards in hand, return 0
    if (m_Cards.empty())
    {
        return 0;
    }
}

```



```

//if a first card has value of 0, then card is face down; return 0
if (m_Cards[0]->GetValue() == 0)
{
    return 0;
}

//add up card values, treat each ace as 1
int total = 0;
vector<Card*>::const_iterator iter;
for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
{
    total += (*iter)->GetValue();
}

//determine if hand contains an ace
bool containsAce = false;
for (iter = m_Cards.begin(); iter != m_Cards.end(); ++iter)
{
    if ((*iter)->GetValue() == Card::ACE)
    {
        containsAce = true;
    }
}

//if hand contains ace and total is low enough, treat ace as 11
if (containsAce && total <= 11)
{
    //add only 10 since we've already added 1 for the ace
    total += 10;
}

return total;
}

```

### Trap

---

The destructor of the class is virtual, but notice that I don't use the keyword `virtual` outside of the class when I actually define the destructor. You only use the keyword inside the class definition. Don't worry; the destructor is still virtual.

---

Although I've already covered this, I want to point it out again. All of the `Card` objects will exist on the heap. Any collection of cards, such as a `Hand` object, will have a vector of pointers to a group of those objects on the heap.

The `Clear()` member function has an important responsibility. It not only removes all of the pointers from the vector `m_Cards`, but it destroys the associated `Card` objects and frees the memory on the heap that they occupied. This is just like a real-world Blackjack game in which cards are discarded when a round is over. The virtual class destructor calls `Clear()`.

The `GetTotal()` member function returns the point total of the hand. If a hand contains an ace, it counts it as a 1 or an 11, whichever is best for the player. The program accomplishes this by checking to see whether the hand has at least one ace. If it does, it checks to see whether treating the ace as 11 will put the hand's point total over 21. If it won't, then the ace is treated as an 11. Otherwise, it's treated as a 1.

## The GenericPlayer Class

I define the `GenericPlayer` class for a generic Blackjack player. It doesn't represent a full player. Instead, it represents the common element of a human player and the computer player.

```
class GenericPlayer : public Hand
{
    friend ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer);

public:
    GenericPlayer(const string& name = "");
    virtual ~GenericPlayer();

    //indicates whether or not generic player wants to keep hitting
    virtual bool IsHitting() const = 0;

    //returns whether generic player has busted - has a total greater than 21
    bool IsBusted() const;

    //announces that the generic player busts
    void Bust() const;

protected:
    string m_Name;
};

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

GenericPlayer::~~GenericPlayer()
{}

```

```

bool GenericPlayer::IsBusted() const
{
    return (GetTotal() > 21);
}

void GenericPlayer::Bust() const
{
    cout << m_Name << " busts.\n";
}

```

I make the overloaded operator<<() function a friend of the class so I can display `GenericPlayer` objects on the screen. It accepts a reference to a `GenericPlayer` object, which means that it can accept a reference to a `Player` or `House` object, too.

The constructor accepts a string object for the name of the generic player. The destructor is automatically virtual because it inherits this trait from `Hand`.

The `IsHitting()` member function indicates whether a generic player wants another card. Because this member function doesn't have a real meaning for a generic player, I made it a pure virtual function. Therefore, `GenericPlayer` becomes an abstract class. This also means that both `Player` and `House` need to implement their own versions of this member function.

The `IsBusted()` member function indicates whether a generic player has busted. Because players and the house bust the same way—by having a total greater than 21—I put the definition in this class.

The `Bust()` member function announces that the generic player busts. Because busting is announced the same way for players and the house, I put the definition of the member function in this class.

## The Player Class

The `Player` class represents a human player. It's derived from `GenericPlayer`.

```

class Player : public GenericPlayer
{
public:
    Player(const string& name = "");
    virtual ~Player();
    //returns whether or not the player wants another hit
    virtual bool IsHitting() const;
}

```

```

    //announces that the player wins
    void Win() const;

    //announces that the player loses
    void Lose() const;

    //announces that the player pushes
    void Push() const;
};

Player::Player(const string& name):
    GenericPlayer(name)
{}
Player::~Player()
{}

bool Player::IsHitting() const
{
    cout << m_Name << ", do you want a hit? (Y/N): ";
    char response;
    cin >> response;
    return (response == 'y' || response == 'Y');
}

void Player::Win() const
{
    cout << m_Name << " wins.\n";
}

void Player::Lose() const
{
    cout << m_Name << " loses.\n";
}

void Player::Push() const
{
    cout << m_Name << " pushes.\n";
}

```

The class implements the `IsHitting()` member function that it inherits from `GenericPlayer`. Therefore, `Player` isn't abstract. The class implements the member function by asking the human whether he wants to keep hitting. If the human enters `y` or `Y` in response to the question, the member function returns `true`, indicating that the player is still hitting. If the human enters a different character, the member function returns `false`, indicating that the player is no longer hitting.

The `Win()`, `Lose()`, and `Push()` member functions simply announce that a player has won, lost, or pushed, respectively.

## The House Class

The `House` class represents the house. It's derived from `GenericPlayer`.

```
class House : public GenericPlayer
{
public:
    House(const string& name = "House");
    virtual ~House();

    //indicates whether house is hitting - will always hit on 16 or less
    virtual bool IsHitting() const;

    //flips over first card
    void FlipFirstCard();
};

House::House(const string& name):
    GenericPlayer(name)
{}

House::~~House()
{}

bool House::IsHitting() const
{
    return (GetTotal() <= 16);
}

void House::FlipFirstCard()
{
    if (!(m_Cards.empty()))
    {
        m_Cards[0]->Flip();
    }
    else
    {
        cout << "No card to flip!\n";
    }
}
```

The class implements the `IsHitting()` member function that it inherits from `GenericPlayer`. Therefore, `House` isn't abstract. The class implements the member function by calling `GetTotal()`. If the returned total value is less than or equal to 16, the member function returns `true`, indicating that the house is still hitting. Otherwise, it returns `false`, indicating that the house is no longer hitting.

`FlipFirstCard()` flips the house's first card. This member function is necessary because the house hides its first card at the beginning of the round and then reveals it after all of the players have taken all of their additional cards.

## The Deck Class

The `Deck` class represents a deck of cards. It's derived from `Hand`.

```
class Deck : public Hand
{
public:
    Deck();
    virtual ~Deck();

    //create a standard deck of 52 cards
    void Populate();

    //shuffle cards
    void Shuffle();

    //deal one card to a hand
    void Deal(Hand& aHand);

    //give additional cards to a generic player
    void AdditionalCards(GenericPlayer& aGenericPlayer);
};

Deck::Deck()
{
    m_Cards.reserve(52);
    Populate();
}

Deck::~~Deck()
{}

void Deck::Populate()
{
    Clear();
    //create standard deck
```

```

for (int s = Card::CLUBS; s <= Card::SPADES; ++s)
{
    for (int r = Card::ACE; r <= Card::KING; ++r)
    {
        Add(new Card(static_cast<Card::rank>(r), static_cast<Card::suit>(s)));
    }
}

void Deck::Shuffle()
{
    random_shuffle(m_Cards.begin(), m_Cards.end());
}

void Deck::Deal(Hand& aHand)
{
    if (!m_Cards.empty())
    {
        aHand.Add(m_Cards.back());
        m_Cards.pop_back();
    }
    else
    {
        cout << "Out of cards. Unable to deal.";
    }
}

void Deck::AdditionalCards(GenericPlayer& aGenericPlayer)
{
    cout << endl;
    //continue to deal a card as long as generic player isn't busted and
    //wants another hit
    while ( !(aGenericPlayer.IsBusted()) && aGenericPlayer.IsHitting() )
    {
        Deal(aGenericPlayer);
        cout << aGenericPlayer << endl;
        if (aGenericPlayer.IsBusted())
        {
            aGenericPlayer.Bust();
        }
    }
}

```

**Hint**

*Type casting* is a way of converting a value of one type to a value of another type. One way to do type casting is to use `static_cast`. You use `static_cast` to return a value of a new type from a value of another type by specifying the new type you want between `<` and `>`, followed by the value from which you want to get a new value between parentheses. Here's an example that returns the double value 5.0.

```
static_cast<double>(5);
```

`Populate()` creates a standard deck of 52 cards. The member function loops through all of the possible combinations of `Card::suit` and `Card::rank` values. It uses `static_cast` to cast the `int` loop variables to the proper enumerated types defined in `Card`.

`Shuffle()` shuffles the cards in the deck. It randomly rearranges the pointers in `m_Cards` with `random_shuffle()` from the Standard Template Library. This is the reason I include the `<algorithm>` header file.

`Deal()` deals one card from the deck to a hand. It adds a copy of the pointer to the back of `m_Cards` to the object through the object's `Add()` member function. Then, it removes the pointer at the back of `m_Cards`, effectively transferring the card. The powerful thing about `Deal()` is that it accepts a reference to a `Hand` object, which means it can work equally well with a `Player` or a `House` object. And through the magic of polymorphism, `Deal()` can call the object's `Add()` member function without knowing the exact object type.

`AdditionalCards()` gives additional cards to a generic player until the generic player either stops hitting or busts. The member function accepts reference to a `GenericPlayer` object so you can pass a `Player` or `House` object to it. Again, through the magic of polymorphism, `AdditionalCards()` doesn't have to know whether it's working with a `Player` or a `House` object. It can call the `IsBusted()` and `IsHitting()` member functions for the object without knowing the object's type, and the correct code will be executed.

## The Game Class

The `Game` class represents a game of Blackjack.

```
class Game
{
public:
    Game(const vector<string>& names);

    ~Game();

    //plays the game of blackjack
    void Play();
```



```

private:
    Deck m_Deck;
    House m_House;
    vector<Player> m_Players;
};

Game::Game(const vector<string>& names)
{
    //create a vector of players from a vector of names
    vector<string>::const_iterator pName;
    for (pName = names.begin(); pName != names.end(); ++pName)
    {
        m_Players.push_back(Player(*pName));
    }

    //seed the random number generator
    srand(static_cast<unsigned int>(time(0)));
    m_Deck.Populate();
    m_Deck.Shuffle();
}

Game::~~Game()
{}

void Game::Play()
{
    //deal initial 2 cards to everyone
    vector<Player>::iterator pPlayer;
    for (int i = 0; i < 2; ++i)
    {
        for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
        {
            m_Deck.Deal(*pPlayer);
        }
        m_Deck.Deal(m_House);
    }

    //hide house's first card
    m_House.FlipFirstCard();

    //display everyone's hand
    for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
    {
        cout << *pPlayer << endl;
    }
}

```

```

cout << m_House << endl;
//deal additional cards to players
for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
{
    m_Deck.AdditionalCards(*pPlayer);
}

//reveal house's first card
m_House.FlipFirstCard();
cout << endl << m_House;

//deal additional cards to house
m_Deck.AdditionalCards(m_House);

if (m_House.IsBusted())
{
    //everyone still playing wins
    for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
    {
        if ( !(pPlayer->IsBusted()) )
        {
            pPlayer->Win();
        }
    }
}
else
{
    //compare each player still playing to house
    for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
    {
        if ( !(pPlayer->IsBusted()) )
        {
            if (pPlayer->GetTotal() > m_House.GetTotal())
            {
                pPlayer->Win();
            }
            else if (pPlayer->GetTotal() < m_House.GetTotal())
            {
                pPlayer->Lose();
            }
        }
    }
}

```

```

        else
        {
            pPlayer->Push();
        }
    }
}

//remove everyone's cards
for (pPlayer = m_Players.begin(); pPlayer != m_Players.end(); ++pPlayer)
{
    pPlayer->Clear();
}
m_House.Clear();
}

```

The class constructor accepts a reference to a vector of `string` objects, which represent the names of the human players. The constructor instantiates a `Player` object with each name. Next, it seeds the random number generator, and then it populates and shuffles the deck. The `Play()` member function faithfully implements the pseudocode I wrote earlier about how a round of play should be implemented.

## The main() Function

After declaring the overloaded `operator<<()` functions, I write the program's `main()` function.

```

//function prototypes
ostream& operator<<(ostream& os, const Card& aCard);
ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer);

int main()
{
    cout << "\t\tWelcome to Blackjack!\n\n";
    int numPlayers = 0;
    while (numPlayers < 1 || numPlayers > 7)
    {
        cout << "How many players? (1 - 7): ";
        cin >> numPlayers;
    }

    vector<string> names;
    string name;
    for (int i = 0; i < numPlayers; ++i)

```

```

{
    cout << "Enter player name: ";
    cin >> name;
    names.push_back(name);
}
cout << endl;
//the game loop
Game aGame(names);
char again = 'y';
while (again != 'n' && again != 'N')
{
    aGame.Play();
    cout << "\nDo you want to play again? (Y/N): ";
    cin >> again;
}
return 0;
}

```

The `main()` function gets the names of all the players and puts them into a vector of `string` objects, and then instantiates a `Game` object, passing a reference to the vector. The `main()` function keeps calling the `Game` object's `Play()` member function until the players indicate that they don't want to play anymore.

## Overloading the operator<<( ) Function

The following function definition overloads the << operator so I can send a `Card` object to the standard output.

```

//overloads << operator so Card object can be sent to cout
ostream& operator<<(ostream& os, const Card& aCard)
{
    const string RANKS[] = {"0", "A", "2", "3", "4", "5", "6", "7", "8", "9",
                           "10", "J", "Q", "K"};
    const string SUITS[] = {"c", "d", "h", "s"};
    if (aCard.m_IsFaceUp)
    {
        os << RANKS[aCard.m_Rank] << SUITS[aCard.m_Suit];
    }
}

```

```

else
{
    os << "XX";
}

return os;
}

```

The function uses the rank and suit values of the object as array indices. I begin the array RANKS with "0" to compensate for the fact that the value for the rank enumeration defined in Card begins at 1.

The last function definition overloads the << operator so I can send a GenericPlayer object to the standard output.

```

//overloads << operator so a GenericPlayer object can be sent to cout
ostream& operator<<(ostream& os, const GenericPlayer& aGenericPlayer)
{
    os << aGenericPlayer.m_Name << ":\t";
    vector<Card*>::const_iterator pCard;
    if (!aGenericPlayer.m_Cards.empty())
    {
        for (pCard = aGenericPlayer.m_Cards.begin();
             pCard != aGenericPlayer.m_Cards.end();
             ++pCard)
        {
            os << *(*pCard) << "\t";
        }
        if (aGenericPlayer.GetTotal() != 0)
        {
            cout << "(" << aGenericPlayer.GetTotal() << ")";
        }
    }
    else
    {
        os << "<empty>";
    }
    return os;
}

```

The function displays the generic player's name and cards, along with the total value of the cards.

## SUMMARY

In this chapter, you learned the following concepts:

- One of the key elements of OOP is inheritance, which allows you to derive a new class from an existing one. The new class automatically inherits data members and member functions from the existing class.
- A derived class does not inherit constructors, copy constructors, destructors, or an overloaded assignment operator.
- Base class constructors are automatically called before the derived class constructor when a derived class object is instantiated.
- Base class destructors are automatically called after the derived class destructor when a derived class object is destroyed.
- Protected members are accessible only in their own class and certain derived classes, depending upon the derivation access level.
- Using public derivation means that public members in the base class become public members in the derived class, protected members in the base class become protected members in the derived class, and private members are (as always) inaccessible.
- You can override base class member functions by giving them new definitions in a derived class.
- You can explicitly call a base class member function from a derived class.
- You can explicitly call the base class constructor from a derived class constructor.
- Polymorphism is the quality whereby a member function will produce different results depending on the type of object for which it is called.
- Virtual functions allow for polymorphic behavior.
- Once a member function is defined as virtual, it's virtual in any derived class.
- A pure virtual function is a function to which you don't need to give a definition. You specify a pure virtual function by placing an equal sign and a zero at the end of the function header.
- An abstract class has at least one pure virtual member function.
- An abstract class can't be used to instantiate an object.

## QUESTIONS AND ANSWERS

**Q:** How many levels of inheritance can you have?

**A:** Theoretically, as many as you want. But as a beginning programmer, you should keep things simple and try not to go beyond a few levels.

**Q:** Is friendship inherited? That is, if a function is a friend of a base class, is it automatically a friend of a derived class?

**A:** No.

**Q:** Can a class have more than one direct base class?

**A:** Yes. This is called *multiple inheritance*. It's powerful, but creates its own set of thorny issues.

**Q:** Why would you want to call a base class constructor from a derived class constructor?

**A:** So you can control exactly how the base class constructor is called. For example, you might want to pass specific values to the base class constructor.

**Q:** Are there any dangers in overriding a base class function?

**A:** Yes. By overriding a base class member function, you hide the entire overloaded version of the function in the base class. However, you can still call a hidden base class member function explicitly by using the base class name and the scope resolution operator.

**Q:** How can I solve this problem of hiding base class functions?

**A:** One way is to override all of the overloaded version of the base class function.

**Q:** Why do you usually want to call the assignment operator member function of the base class from the assignment operator member function of a derived class?

**A:** So that any base class data members can be properly assigned.

**Q:** Why do you usually want to call the copy constructor of a base class from the copy constructor of a derived class?

**A:** So that any base class data members can be properly copied.

**Q:** Why can you lose access to an object's member functions when you point to it with a base class member?

**A:** Because non-virtual functions are called based on the pointer type and the object type.

**Q:** Why not make all member functions virtual, just in case you ever need polymorphic behavior from them?

**A:** Because there's a performance cost associated with making member functions virtual.

**Q:** So when should you make member functions virtual?

**A:** Whenever they may be inherited from a base class.

**Q:** When should you make a destructor virtual?

**A:** If you have any virtual member functions in a class, you should make the destructor virtual, too. However, some programmers say that to be safe, you should always make a destructor virtual.

**Q:** Can constructors be virtual?

**A:** No. This also means that copy constructors can't be declared as virtual either.

**Q:** In OOP, what is slicing?

**A:** Slicing is cutting off part of an object. Assigning an object of a derived class to a variable of a base class is legal, but you slice the object, losing the data members declared in the derived class and losing access to member functions of the derived class.

**Q:** What good are abstract classes if you can't instantiate any objects from them?

**A:** Abstract classes can be very useful. They can contain many common class members that other classes will inherit, which saves you the effort of defining those members over and over again.

## DISCUSSION QUESTIONS

1. What benefits does inheritance bring to game programming?
2. How does polymorphism expand the power of inheritance?
3. What kinds of game entities might it make sense to model through inheritance?
4. What kinds of game-related classes would be best implemented as abstract?
5. Why is it advantageous to be able to point to a derived class object with a base class pointer?

## EXERCISES

1. Improve the Simple Boss 2.0 program by adding a new class, `FinalBoss`, that is derived from the `Boss` class. The `FinalBoss` class should define a new method, `MegaAttack()`, that inflicts 10 times the amount of damage as the `SpecialAttack()` method does.
2. Improve the Blackjack game program by forcing the deck to repopulate before a round if the number of cards is running low.
3. Improve the Abstract Creature program by adding a new class, `OrcBoss`, which is derived from `Orc`. An `OrcBoss` object should start with 180 for its `health` data member. You should also override the virtual `Greet()` member function so that it displays:  
The orc boss growls hello.