Note that the keywords **`public`** and **`private`** are called access specifiers; they specify whether the members are accessible outside the class definition. The keyword **`protected`** is the third access specifier. It is described in Chapter 13.

10.7 THE COPY CONSTRUCTOR

Every class has at least two constructors. These are identified by their unique declarations:

```
X();            // default constructor
X(const X&);    // copy constructor
```

where `X` is the class identifier. For example, these two special constructors for a `Widget` class would be declared:

```
Widget();                // default constructor
Widget(const Widget&);   // copy constructor
```

The first of these two special constructors is called the *default constructor*; it is called automatically whenever an object is declared in the simplest form, like this:

```
Widget x;
```

The second of these two special constructors is called the *copy constructor*; it is called automatically whenever an object is copied (*i.e.*, duplicated), like this:

```
Widget y(x);
```

If either of these two constructors is not defined explicitly, then it is automatically defined implicitly by the system.

Note that the copy constructor takes one parameter: the object that it is going to copy. That object is passed by constant reference because it should not be changed.

When the copy constructor is called, it copies the complete state of an existing object into a new object of the same class. If the class definition does not explicitly include a copy constructor (as all the previous examples have not), then the system automatically creates one by default. The ability to write your own copy constructor gives you more control over your software.

EXAMPLE 10.9 Adding a Copy Constructor to the `Ratio` Class

```
class Ratio
{ public:
    Ratio(int n=0, int d=1) : num(n), den(d) { reduce(); }
    Ratio(const Ratio& r) : num(r.num), den(r.den) { }
    void print() { cout << num << '/' << den; }
  private:
    int num, den;
    void reduce();
};
int main()
{ Ratio x(100,360);
  Ratio y(x);
  cout << "x = ";
  x.print();
  cout << ", y = ";
  y.print();
}
```

```
x = 5/18, y = 5/18
```

The copy constructor copies the `num` and `den` fields of the parameter `r` into the object being constructed. When `y` is declared, it calls the copy constructor which copies `x` into `y`.

Note the required syntax for the copy constructor: it must have one parameter, which has the same class as that being declared, and it must be passed by constant reference: `const X&`.

The copy constructor is called automatically whenever

- an object is copied by means of a declaration initialization;
- an object is passed by value to a function;
- an object is returned by value from a function.

EXAMPLE 10.10 Tracing Calls to the Copy Constructor

```
class Ratio
{ public:
    Ratio(int n=0, int d=1) : num(n), den(d) { reduce(); }
    Ratio(const Ratio& r) : num(r.num), den(r.den)
    { cout << "COPY CONSTRUCTOR CALLED\n"; }
  private:
    int num, den;
    void reduce();
};

Ratio f(Ratio r)  // calls the copy constructor, copying ? to r
{ Ratio s = r;    // calls the copy constructor, copying r to s
  return s;       // calls the copy constructor, copying s to ?
}

int main()
{ Ratio x(22,7);
  Ratio y(x);  // calls the copy constructor, copying x to y
  f(y);
}
```

```
COPY CONSTRUCTOR CALLED
COPY CONSTRUCTOR CALLED
COPY CONSTRUCTOR CALLED
COPY CONSTRUCTOR CALLED
```

In this example, the copy constructor is called four times. It is called when `y` is declared, copying `x` to `y`; it is called when `y` is passed by value to the function `f`, copying `y` to `r`; it is called when `s` is declared, copying `r` to `s`; and it is called when the function `f` returns by value, even though nothing is copied there. Note that the initialization of `s` looks like an assignment. But as part of a declaration it calls the copy constructor just as the declaration of `y` does.

If you do not include a copy constructor in your class definition, then the compiler generates one automatically. This “default” copy constructor will simply copy objects bit-by-bit. In many cases, this is exactly what you would want. So in these cases, there is no need for an explicitly defined copy constructor.

However, in some important cases, a bit-by-bit copy will not be adequate. The **`string`** class, described in Chapter 9, is a prime example. In objects of that class, the relevant data member holds only a pointer to the actual string, so a bit-by-bit copy would only duplicate the pointer, not the string itself. In cases like this, it is essential that you define your own copy constructor.

10.8 THE CLASS DESTRUCTOR

When an object is created, a constructor is called automatically to manage its birth. Similarly, when an object comes to the end of its life, another special member function is called automatically to manage its death. This function is called a *destructor*.

Each class has exactly one destructor. If it is not defined explicitly in the class definition, then like the default constructor, the copy constructor, and the assignment operator, the destructor is created automatically.

EXAMPLE 10.11 Including a Destructor in the `Ratio` Class

```
class Ratio
{ public:
    Ratio() { cout << "OBJECT IS BORN.\n"; }
    ~Ratio()  { cout << "OBJECT DIES.\n"; }
  private:
    int num, den;
};

int main()
{ { Ratio x;                     // beginning of scope for x
    cout << "Now x is alive.\n";
  }                                   // end of scope for x
  cout << "Now between blocks.\n";
  { Ratio y;
    cout << "Now y is alive.\n";
  }
}
```

```
OBJECT IS BORN.
Now x is alive.
OBJECT DIES.
Now between blocks.
OBJECT IS BORN.
Now y is alive.
OBJECT DIES.
```

The output here shows when the constructor and the destructor are called.

The class destructor is called for an object when it reaches the end of its scope. For a local object, this will be at the end of the block within which it is declared. For a `static` object, it will be at the end of the `main()` function.

Although the system will provide them automatically, it is considered good programming practice always to define the copy constructor, the assignment operator, and the destructor within each class definition.