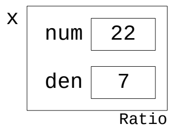
adding the new `Ratio` type to the collection of predefined numeric types `int`, `float`, *etc*. We can envision the object `x` like this:



Notice the use of the specifier `Ratio::` as a prefix to each function name. This is necessary for each member function definition that is given outside of its class definition. The *scope resolution operator* `::` is used to tie the function definition to the `Ratio` class. Without this specifier, the compiler would not know that the function being defined is a member function of the `Ratio` class. This can be avoided by including the function definitions within declaration, as shown below in Example 10.2.

When an object like the `Ratio` object `x` in Example 10.1 is declared, we say that the class has been *instantiated*, and we call the object an *instance* of the class. And just as we may have many variables of the same type, we may also have may instances of the same class:

```
Ratio x, y, z;
```

EXAMPLE 10.2 A Self-Contained Implementation of the `Ratio` Class

Here's the same `Ratio` class with the definitions of its member functions included within the class declaration:

```
class Ratio
{ public:
    void assign(int n, int d) { num = n; den = d; }
    double convert() { return double(num)/den; }
    void invert() { int temp = num; num = den; den = temp; }
    void print() { cout << num << '/' << den; }
  private:
    int num, den;
};
```

In most cases, the preferred style is to define the member functions outside of the class declaration, using the scope resolution operator as shown in Example 10.1. That format physically separates the function declarations from their definitions, consistent with the general principle of information hiding. In fact, the definitions are often put in a separate file and compiled separately. The point is that application programs that use the class need only know `what` the objects can do; they do not need to know how the objects do it. The function declarations tell what they do; the function definitions tell how they do it. This, of course, is how the predefined types (`int`, `double`, *etc*.) work: we know what the result should be when we divide one float by another, but we don't really know how the division is done (*i.e.*, what algorithm is implemented). More importantly, we don't want to know. Having to think about those details would distract us from the task at hand. This point of view is often called *information hiding* and is an important principle in object-oriented programming.

When the member function definitions are separated from the declarations, as in Example 10.1, the declaration section is called the *class interface*, and the section containing the member function definitions is called the *implementation*. The interface is the part of the class that the programmer needs to see in order to use the class. The implementation would normally be concealed in a separate file, thereby “hiding” that information that the user (*i.e.*, the programmer) does not need to know about. These class implementations are typically done by implementors who work independently of the programmers who will use the classes that they have implemented.

10.3 CONSTRUCTORS

The `Ratio` class defined in Example 10.1 uses the `assign()` function to initialize its objects. It would be more natural to have this initialization occur when the objects are declared. That’s how ordinary (predefined) types work:

```
int n = 22;
char* s = "Hello";
```

C++ allows this simpler style of initialization to be done for class objects using constructor functions.

A *constructor* is a member function that is invoked automatically when an object is declared. A constructor function must have the same name as the class itself, and it is declared without return type. The following example illustrates how we can replace the `assign()` function with a constructor.

EXAMPLE 10.3 A Constructor Function for the `Ratio` Class

```
class Ratio
{ public:
    Ratio(int n, int d) { num = n; den = d; }
    void print() { cout << num << ’/’ << den; }
  private:
    int num, den;
};

int main()
{ Ratio x(-1,3), y(22,7);
  cout << "x = ";
  x.print();
  cout << " and y = ";
  y.print();
}
```

```
    x = -1/3 and y = 22/7
```

The constructor function has the same effect as the `assign()` function had in Example 10.1: it initializes the object by assigning the specified values to its member data. When the declaration of `x` executes, the constructor is called automatically and the integers -1 and 3 are passed to its parameters `n` and `d`. The function then assigns these values to `x`’s `num` and `den` data members. So the declarations

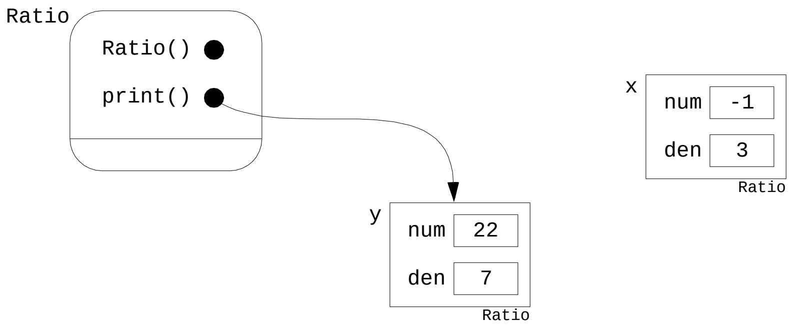
```
Ratio x(-1,3), y(22,7);
```

are equivalent to the three lines

```
Ratio x, y;
x.assign(-1,3);
y.assign(22,7);
```

A class's constructor "constructs" the class objects by allocating and initializing storage for the objects and by performing any other tasks that are programmed into the function. It literally creates a live object from a pile of unused bits.

We can visualize the relationships between the `Ratio` class itself and its instantiated objects like this:



The class itself is represented by a rounded box containing its member functions. Each function maintains a pointer, named "`this`", which points to the object that is calling it. The snapshot here represents the status during the execution of the last line of the program, when the object `y` is calling the `print()` function: `y.print()`. At that moment, the "`this`" pointer for the constructor is **`NULL`** because it is not being called.

A class may have several constructors. Like any other overloaded function, these are distinguished by their distinct parameter lists.

EXAMPLE 10.4 Adding More Constructors to the `Ratio` Class

```
class Ratio
{ public:
    Ratio() { num = 0; den = 1; }
    Ratio(int n) { num = n; den = 1; }
    Ratio(int n, int d) { num = n; den = d; }
    void print() { cout << num << '/' << den; }
  private:
    int num, den;
};

int main()
{ Ratio x, y(4), z(22,7);
  cout << "x = ";
  x.print();
  cout << "\ny = ";
  y.print();
  cout << "\nz = ";
  z.print();
}
```

```
x = 0/1
y = 4/1
z = 22/7
```

This version of the `Ratio` class has three constructors. The first has no parameters and initializes the declared object with the default values 0 and 1. The second constructor has one integer parameter and